2578. Split With Minimum Sum

Greedy Math Sorting

Problem Description

concatenated string, they form a permutation of the original number num. The meaning of a permutation here is that the frequency of each digit in num1 plus num2 should match the frequency of each digit in num. Additionally, num1 and num2 can start with zeros, and the goal is to minimize the sum of num1 and num2.

Given a positive integer num, the task is to find two non-negative integers num1 and num2 such that when combined as a

Intuition The key to solving this problem is understanding what permutations and minimum sums involve. To achieve the minimum sum, smaller digits should contribute more to the less significant places (rightmost) of the numbers, and we should distribute the digits

One approach could be using a counting method. Counting each digit's occurrences in num can help us ensure that we distribute the digits correctly between num1 and num2. By starting with the smallest digits and alternating between num1 and num2, we can

between num1 and num2 as evenly as possible, especially the smallest non-zero digit to avoid a larger sum.

spread out the digits evenly and keep the sums low. Here's how we can think through the problem: • Count the occurrences of each digit (0-9) in num. • Starting from the smallest digit, assign digits to num1 and num2 alternately.

Finally, compute the total by summing up num1 and num2.

require us to modify num and is efficient in terms of time complexity.

• Ensure that the least significant digit of num (the rightmost) goes to the least significant place in num1 and num2 to keep their sums to a minimum. Loop through the digits until all counted digits are assigned.

This approach leads us to distribute the digits in a way that balances the sum. The counting method is handy since it doesn't

Solution Approach

used up (i.e., its count in the counter is not zero).

digits evenly and contributing to the smaller sum.

num1 and num2 that are permutations of num.

• First digit: Assign '1' to num1, so num1 = 1.

• Fourth digit: Assign '2' to num2, so num2 = 21.

Variable to keep track of the length of the number

Initialize an array to hold our two new numbers

Iterate over the length of the original number

digit counter[current digit] -= 1

Return the sum of the two new numbers

digit counter[num % 10] += 1

Split the number into digits and populate the counter

Variable to track the current digit we are considering

// Create an array to count the occurrences of each digit

// Variable to hold the total number of digits in num

Decrease the count for the current digit as we're now using it

split_numbers[index % 2] = split_numbers[index % 2] * 10 + current_digit

Depending on the index being even/odd, add the digit to the corresponding number

// This method takes an integer and splits it into two numbers such that the sum of the two is minimized

of unique digits in num, which is less than or equal to 10 (since these are decimal digits).

Alternate Construction: Starting from the least significant place, we proceed like this:

applied in the solution using the given Python code as a reference: Initialize a Counter: A counter (from Python's collections standard library) is used to track the frequency of each digit in

the input number num. This helps us to make sure that we use each digit exactly as many times as it appears in num.

To implement the solution, we follow a count and greedy algorithm. Below are the steps that elaborate on how this approach is

Count Digits: We start by calculating the frequency of each digit and the total number of digits n in num. Every digit is

reduced and its occurrences are recorded until num becomes zero. Create an Answer Array: An answer array ans with two elements initializes both num1 and num2 to zero. As we proceed

- through the digits by increasing order, we will build up these two numbers. **Greedy Assignment**: We greedily assign the smallest available digits first while decrementing their count in the counter. We iterate through a range defined by the number of digits n and for each digit, find the smallest digit that has not been fully
- Alternate Construction: For each iteration, we alternate between appending the chosen digit to num1 and num2. This is handled by the bit manipulation i & 1, which alternates between 0 and 1 with each increment of i, therefore distributing the
- Sum Calculation: After the alternating assignment, we calculate the sum of the two generated numbers num1 and num2 and return the result as the minimum possible sum. Since we're concatenating smaller digits first and evenly splitting the digits, this sum would be the smallest possible.

The algorithm's time complexity is reported as O(n), where n is the number of digits in num, because we scan through the input

number once and iterate through the counter based on the number of digits. The space complexity is O(C), where C is the count

The approach leverages a balanced distribution of digits to both num1 and num2 with a focus on placing lower digits in the less

significant positions to achieve the minimum sum. This is done effectively using direct counting and iteration without the need for

sorting, which makes it a neat and efficient solution. **Example Walkthrough**

Let's take the number num = 1122 and walk through the solution approach to find the minimum sum of two non-negative integers

Initialize a Counter: We use a counter to keep the frequency of digits in 1122. Here, the digits '1' and '2' each occur twice.

Count Digits: We count the occurrences of each digit and note that the total number of digits n is 4. The frequencies are {'1':

2, '2': 2}. Create an Answer Array: We initialize ans with two elements, both set to 0, which eventually will represent num1 and num2. Greedy Assignment: We want to place smaller digits first to minimize the sum. Since we only have '1' and '2', we start with '1'.

Second digit: Assign '1' to num2, so num2 = 1. • Third digit: Assign '2' to num1, so num1 = 21.

Sum Calculation: After assignment, we have num1 = 21 and num2 = 21. The minimum sum is 21 + 21 = 42. By following the steps outlined in the solution approach, we assigned the digits to num1 and num2 in an alternating fashion,

starting with the smallest digits. This ensured that we got the minimum sum of all possible permutations of the integer 1122.

- Solution Implementation
- **Python** from collections import Counter
- class Solution: def splitNum(self, num: int) -> int: # Create a counter to keep track of the frequency of each digit digit_counter = Counter()

for index in range(num length): # Find the next smallest digit available by checking the counter while digit counter[current_digit] == 0: current digit += 1

num length = 0

while num > 0:

num //= 10

current_digit = 0

num_length += 1

split numbers = [0] * 2

return sum(split_numbers)

public int splitNum(int num) {

int totalDigits = 0;

int[] digitCount = new int[10];

++digitCounts[num % 10];

// Iterating through the number of digits.

// Array to store the two numbers formed from the digits.

for (int i = 0, digitIndex = 0; i < digitTotal; ++i) {</pre>

// Add the found digit to one of the two numbers.

splitNumbers[i % 2] = splitNumbers[i % 2] * 10 + digitIndex;

// Initialize an array to store the two new numbers formed from the digits.

// Use the current smallest digit and update the appropriate number.

// Reconstruct the two numbers by alternating digits starting from the smallest.

++digitCount[num % 10]; // Increment count for the current least significant digit.

// Find the next non-zero frequency digit.

while (digitCounts[digitIndex] == 0) {

// Decrease the count of found digit.

// Return the sum of the two split numbers.

// Initialize a count array with 10 zeroes for digits 0-9.

++digitTotal; // Increment total number of digits.

// Variable to keep track of the total number of digits.

return splitNumbers[0] + splitNumbers[1];

const digitCount: number[] = Array(10).fill(0);

const splitNumbers: number[] = Array(2).fill(0);

while (digitCount[digitIndex] === 0) {

++digitIndex;

--digitCount[digitIndex]:

for (let i = 0, digitIndex = 0; i < digitTotal; ++i) {</pre>

// Find the smallest digit that has not been used up.

++digitTotal;

int splitNumbers[2] = {0};

++digitIndex;

function splitNum(num: number): number {

let digitTotal = 0;

--digitCounts[digitIndex];

Java

class Solution {

```
// Count the occurrences of each digit in the input number
       while (num > 0) {
            int digit = num % 10; // Extract the last digit of num
            digitCount[digit]++; // Increment its count in the array
                                 // Remove the last digit from num
            num /= 10;
            totalDigits++;
                                  // Increment the total number of digits
        // Array to hold the two numbers formed from splitting
        int[] parts = new int[2];
        // We will all ever go up to the total number of digits in the original number
        for (int i = 0, digitIndex = 0; i < totalDigits; ++i) {</pre>
            // Find the smallest non-zero count digit
            while (digitCount[digitIndex] == 0) {
               digitIndex++;
           // Decrease the count of the chosen digit
            digitCount[digitIndex]--;
            // Construct the split numbers digit by digit, alternating between the two numbers
            parts[i % 2] = parts[i % 2] * 10 + digitIndex;
        // Return the sum of the two constructed numbers
        return parts[0] + parts[1];
C++
class Solution {
public:
    // Method to split a number and sum the parts.
    int splitNum(int num) {
        // Arrav to count frequency of each digit in 'num'.
        int digitCounts[10] = {0};
       // Total number of digits in 'num'.
        int digitTotal = 0;
        // Counting digit frequencies and the total number of digits.
        for (; num > 0; num /= 10) {
```

// Count digit occurrences and total digits until num is greater than 0. for (; num > 0; num = Math.floor(num / 10)) {

};

TypeScript

```
splitNumbers[i & 1] = splitNumbers[i & 1] * 10 + digitIndex;
   // Return the sum of the two constructed numbers.
   return splitNumbers[0] + splitNumbers[1];
from collections import Counter
class Solution:
   def splitNum(self, num: int) -> int:
       # Create a counter to keep track of the frequency of each digit
       digit_counter = Counter()
       # Variable to keep track of the length of the number
       num length = 0
       # Split the number into digits and populate the counter
       while num > 0:
           digit counter[num % 10] += 1
           num //= 10
           num_length += 1
       # Initialize an array to hold our two new numbers
       split numbers = [0] * 2
       # Variable to track the current digit we are considering
       current_digit = 0
       # Iterate over the length of the original number
        for index in range(num length):
           # Find the next smallest digit available by checking the counter
           while digit counter[current_digit] == 0:
                current digit += 1
           # Decrease the count for the current digit as we're now using it
           digit counter[current digit] -= 1
```

Depending on the index being even/odd, add the digit to the corresponding number

split_numbers[index % 2] = split_numbers[index % 2] * 10 + current_digit

Inside the loop, the modulo and division operations are 0(1) operations but they're repeated for each digit, contributing again 0(n).

Time Complexity

After the initial while loop, there's a nested loop where the outer loop runs n times (for i in range(n)), and the inner while loop may run up to 10 times (as there are 10 possible digits 0-9). However, on average, if digits are evenly distributed,

The time complexity of the code can be analyzed as follows:

Return the sum of the two new numbers

return sum(split_numbers)

Time and Space Complexity

the inner loop will run a constant number of times; thus, the nested loop contributes O(n). The multiplication and addition inside the nested loop are 0(1) operations.

The while loop runs as long as 'num' has digits, contributing O(n) time complexity where n is the number of digits in num.

- Considering all the above, the overall time complexity of the splitNum function is O(n) since none of the steps depend on the number of digits logarithmically or have a more significant impact than linear in terms of digits.
- **Space Complexity** The space complexity of the code can be analyzed as follows:

Variables n, j, and other loop variables use O(1) space.

digits (0-9) regardless of the size of num.

An array ans of size 2 is used, requiring 0(1) space.

A Counter object is used to store the frequency of each digit, which requires at most 0(1) space because there are only 10

The reference answer suggests 0(n) for both time and space complexities. However, the given code has a time complexity of O(n) and a space complexity of O(1), assuming that n is the number of digits in num. There seems to be a discrepancy in the

In total, the space complexity is 0(1) because the space required does not change with the size of the input num.

analysis regarding space complexity.