

1689. Partitioning Into Minimum Number Of Deci-Binary Numbers

MediumGreedyString

Problem Description

This LeetCode problem defines a term "deci-binary" which refers to a number that comprises only `0` and `1` digits. A deci-binary number does not contain any digits other than `0` and `1` and should not have leading zeros. The task is to determine the minimum quantity of deci-binary numbers that can be summed together to equal a given positive decimal number `n`. The input `n` is provided as a string representation of a decimal integer, and the expected output is an integer representing the minimum number of deci-binary numbers required to sum up to `n`.

Intuition

The intuition behind the solution is that the minimum number of deci-binary numbers required is determined by the largest digit in the original number `n`. Since deci-binary numbers can only have the digits `0` or `1`, and no leading zeroes are allowed, each deci-binary number can only contribute a `1` towards a single decimal place in the sum.

To build the original number `n` using deci-binary numbers, you would start by placing a `1` in the position of the largest digit of `n`, and then subtract `1` from this digit. You would repeat this process, placing a `1` in the position of the current largest digit after subtraction, until all the digits in the original number are reduced to `0`.

For example, if `n = "123"`, we would need a `1` in the hundreds place, tens place, and ones place to start, and then continue placing `1`s in the tens and ones place until the number is exhausted. It would take three deci-binary numbers to reduce the hundreds place, two to reduce the tens, and three to reduce the ones place, resulting in a total of three deci-binary numbers (`3` being the largest digit in `n`).

Therefore, the solution is as simple as finding the maximum digit in the string `n`, as this will dictate how many times a `1` needs to be placed in the position of this digit across all deci-binary numbers. The Python code `int(max(n))` efficiently finds the largest digit and converts it to an integer, which is the minimum number of deci-binary numbers needed.

Solution Approach

The implementation of the solution is remarkably straightforward and doesn't require complex algorithms or data structures. It also doesn't involve a pattern recognition exercise typical for many combinatorial problems; rather, it simply relies on a characteristic of the given decimal number—the value of its highest digit.

The core of the solution approach lies in understanding that, since we can only use deci-binary numbers (numbers with digits `0` or `1` only), the maximum number one such number can contribute to any digit place of the original number `n` is `1`. That's because having any digit greater than `1` would violate the definition of deci-binary.

The algorithm is as follows:

- Iterate through each character in the string `n`. This represents each digit of the decimal number.
- Convert each character to an integer.
- Keep track of the maximum integer value found during the iteration.

Since we are given the number in the form of a string, no conversion to a number is needed to find the maximum digit value. This is because the characters `'0'` to `'9'` have increasing ASCII values, so comparing them as characters yields the same result as comparing their numerical values.

In terms of the solution code provided:

```
class Solution:
    def minPartitions(self, n: str) -> int:
        return int(max(n))
```

`max(n)` gets the character with the highest ASCII value in the string `n`, which corresponds to the highest digit in the decimal number. Converting this character back to an integer with `int()` gives us the minimum number of deci-binary numbers needed. The reason for this conversion is that the `max()` function would give us the maximum character (i.e., digit in the form of a string), but the problem requires us to return an integer.

No additional data structures are required because we only need to determine the maximum digit, which is a single value, and no additional computation or storage is necessary. The algorithm's time complexity is $O(m)$, where m is the number of digits in the string representation of the number `n`, because it requires a single pass through all the digits to find the maximum one. The space complexity is $O(1)$ as it only uses a constant amount of additional memory.

Example Walkthrough

Let's illustrate the solution approach using the example `n = "82734"`. We want to find the minimum number of deci-binary numbers that sum up to this number.

- The first step is to examine each digit of `n`. We check `8`, `2`, `7`, `3`, and `4`.
- We find that the largest digit in this number is `8`.
- Knowing that a deci-binary number only consists of `0`s and `1`s, the largest number it can contribute to any single digit is `1`.
- Therefore, we would need at least eight deci-binary numbers to contribute to the digit `8` in the thousands place.
- No other digit in the number `82734` is larger than `8`, so no more than eight deci-binary numbers will be needed for any other digit.
- Hence, the answer is `8`, which indicates that eight deci-binary numbers would be enough to add up to `82734`.

To visualize this, we can represent the deci-binary numbers and their sum like so:

```
Deci-binary numbers that sum to 82734:
11111
11111
11111
11111
11111
11111
11111
11111
10000
-----
82734
```

As you can see, each deci-binary number contributes a `1` to every position but in the last deci-binary number, we only need to contribute to the thousands place to make the sum equal to `82734`. It takes eight such numbers to match the maximum digit `8` in the original number.

By executing the line `int(max(n))`, we take the maximum digit character, which is `'8'`, and convert it into an integer, giving us the final answer, `8`.

Solution Implementation

Python

```
class Solution:
    def minPartitions(self, n: str) -> int:
        # Find the maximum digit in the string representation of the number
        max_digit = max(n)

        # Convert the maximum digit from string to integer
        min_partitions = int(max_digit)

        # Return the minimum number of partitions required
        return min_partitions

# Explanation: The provided method minPartitions is used to find the minimum number of
# decimal digits needed to write down the number n in such a way that each digit is
# used only once. The input is a string representation of a non-negative integer n,
# and the method returns an integer representing the answer. The logic simply finds
# the maximum digit in the string, as this will be the minimum number needed. For
# example, for input '82734', the maximum digit is '8', so you would need at least
# 8 partitions (since the number must be decomposable into a sum of numbers containing
# each digit at most once).
```

Java

```
class Solution {
    // Method to find the minimum number of partitions required
    public int minPartitions(String n) {
        // Initialize the variable to store the maximum digit in the string
        int maxDigit = 0;

        // Loop through each character of the string representing the number
        for (int i = 0; i < n.length(); ++i) {
            // Find the numeric value of the current digit
            int currentDigit = n.charAt(i) - '0';

            // Update maxDigit if the current digit is greater than the maxDigit so far
            maxDigit = Math.max(maxDigit, currentDigit);

            // If the maximum digit is 9, we can return it immediately as it's the highest possible digit
            if (maxDigit == 9) {
                break;
            }
        }

        // Return the maximum digit as the minimum number of partitions required
        return maxDigit;
    }
}
```

C++

```
class Solution {
public:
    // Function to calculate the minimum number of decimal digits
    // one must add to decompose the given string of digits
    int minPartitions(string n) {
        // Initialize the answer to zero
        int maxDigit = 0;

        // Iterate over each character in the string
        for (const char& digitChar : n) {
            // Convert the character to the corresponding integer digit
            int digit = digitChar - '0';

            // Update the maximum digit found so far
            maxDigit = std::max(maxDigit, digit);
        }

        // Return the highest digit as the minimum number of partitions needed
        return maxDigit;
    }
};
```

TypeScript

```
// Function to find the minimum number of deci-binary numbers
// needed such that they sum up to the string n.
// A deci-binary number is a number base-10 that each of its digits
// is either 0 or 1 without any leading zeros.
function minPartitions(n: string): number {
    // Split the input string into an array of its characters,
    // then map each character to its integer representation
    let digits = n.split('').map(digit => parseInt(digit));

    // Find the maximum digit in the array, as this will be the
    // minimum number of deci-binary numbers needed
    let maxDigit = Math.max(...digits);

    // Return the maximum digit, which represents the answer
    return maxDigit;
}
```

```
class Solution:
    def minPartitions(self, n: str) -> int:
        # Find the maximum digit in the string representation of the number
        max_digit = max(n)

        # Convert the maximum digit from string to integer
        min_partitions = int(max_digit)

        # Return the minimum number of partitions required
        return min_partitions

# Explanation: The provided method minPartitions is used to find the minimum number of
# decimal digits needed to write down the number n in such a way that each digit is
# used only once. The input is a string representation of a non-negative integer n,
# and the method returns an integer representing the answer. The logic simply finds
# the maximum digit in the string, as this will be the minimum number needed. For
# example, for input '82734', the maximum digit is '8', so you would need at least
# 8 partitions (since the number must be decomposable into a sum of numbers containing
# each digit at most once).
```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by finding the maximum digit in the string `n`. Since the `max()` function iterates through each character in the string once, the time complexity is $O(d)$, where `d` is the length of the input string `n`.

Space Complexity

The space complexity of the function is $O(1)$. This is because it only uses a fixed amount of additional space to store the maximum digit found in the string, regardless of the length of the input.