2160. Minimum Sum of Four Digit Number After Splitting Digits

Easy Greedy Math Sorting

Leetcode Link

Problem Description

In this problem, we are given a four-digit positive integer `num`. Our task is to split this integer into exactly two new integers, `new1` and `new2`. To create these two new integers, we have to distribute the digits of `num` between them. Some important points to note about this task are:

- A digit can be placed in `new1` or `new2` without restriction, except that all digits from `num` must be used.
- Leading zeros are allowed in `new1` and `new2`.
- We want to minimize the sum of `new1` and `new2`.

As an example, consider `num = 2932`. We can create several pairs from these digits, such as `[22, 93]`, `[23, 92]`, `[223, 9]`, and `[2, 329]`. However, our goal is to find the pair where the sum of `new1` and `new2` is the smallest. This requires a strategic approach to pairing the digits into two numbers where their combined value is minimized.

Intuition

The intuition behind solving this problem lies in understanding that to minimize the sum of two numbers created from the digits of `num`, we need to consider digit positions carefully. The smallest sum is obtained when the least significant digits (the ones and tens place) are as small as possible in both new numbers.

One effective strategy is to:

1. Sort the digits of the original number in ascending order so that the smallest digits are placed in positions that contribute the least to the sum.
2. Assign digits to `new1` and `new2` strategically to ensure that the least significant digits of both numbers are as small as possible. This is because adding a smaller digit in the tens place will reduce the sum more significantly than adding it in the ones place due to the positional value in decimal notation.

Here's how we can do it:

- After sorting the digits, the smallest digits will be at the front of the array. Manage these small digits so that they contribute to tens place of both the numbers.
- Specifically, we take the two smallest digits and place them into the tens places of `new1` and `new2`. This is done by multiplying the sum of these two smallest digits by 10.
- Then we add the remaining two digits to the ones places.

Thus, the approach to arriving at the solution relies on sorting the four digits and then constructing two new numbers such that their sum is minimized by careful placement of the least significant digits.

Solution Approach

The solution is implemented in Python and uses fundamental programming constructs such as a list and sorting algorithm. Below is the step-by-step approach taken in the provided code:

1. First, we initialize an empty list, `nums`, which will hold the individual digits of the given `num`.

2. We then extract each digit from `num` using a `while` loop that continues until `num` becomes zero. We use the modulus operator `%` to get the least significant digit and append it to the list `nums`. Then, we use floor division `//` by 10 to remove the least significant digit from `num`.

```
while num:
    nums.append(num % 10)
    num //= 10
```

3. The resulting `nums` list contains the individual digits in reverse order. We sort this list to arrange the digits in ascending order. Sorting is crucial to place the smallest digits in the tens places of `new1` and `new2` to minimize the sum.

4. The sorted `nums` list now has the smallest digit at index `0` and the second smallest at index `1`. By adding the smallest two digits and multiplying by 10, we achieve two numbers with the smallest digits in the tens place:

```
return 10 * (nums[0] + nums[1]) + nums[2] + nums[3]
```

5. To complete the numbers `new1` and `new2`, we add the third smallest digit (`nums[2]`) and the largest digit (`nums[3]`) to the sum (which will occupy the ones places).

6. The final result returned is the minimum possible sum of `new1` and `new2` constructed from the digits of the original number, `num`.

The algorithm used is simple yet effective because it directly leverages the property of positional notation in decimal numbers. The choice of the list as the data structure enables easy manipulation of digits and the use of Python's built-in sort method streamlines digit organization.

Example Walkthrough

Let's walk through the solution approach with a small example. Suppose the given number is `num = 4723`.

Following the approach described, we perform these steps:

1. We start by initializing an empty list to store the digits. We call this list `nums`.

2. We then extract each digit from `num` and append it to `nums`. Processing the number `4723` gives us:

```
nums = [3, 2, 7, 4]  # Note that digits are appended in reverse order
```

3. Next, we sort the `nums` list in ascending order:

```
nums.sort()  # After sorting, nums = [2, 3, 4, 7]
```

4. Once the list is sorted, we identify the two smallest digits, which now are at indices `0` and `1`. In our case, they are `2` and `3`. According to our strategy, they will go into the tens places of `new1` and `new2`. So, by adding them together and multiplying by `10`, we obtain the tens place for both:

```
Tens place sum = 10 * (2 + 3) = 50
```

5. Now, we take the remaining digits, which are `4` and `7`, to fill in the ones places. Adding them to our tens place sum will give us the final value:

```
Total sum = 50 + 4 + 7 = 61
```

6. Therefore, the smallest possible summed pair of numbers created from `4723` is $new1 = 23 and $new2 = 47, with a sum of `61`.

This example demonstrates that by sorting the digits first and then strategically placing the smaller digits in the more significant tens positions across `new1` and `new2`, we achieve the minimum sum as per the problem's requirement.

Python Solution

```python
class Solution:
    def minimumSum(self, num: int) -> int:
        # Initialize a list to store the digits of the input number.
        digits = []

        # Extract the digits from the input number and append them to the list.
        while num:
            # Get the last digit of the number and append it to the digits list.
            digits.append(num % 10)
            # Remove the last digit from the number by integer division by 10.
            num //= 10

        # Sort the digits in ascending order.
        digits.sort()

        # Generate the minimum sum by combining the digits.
        # The minimal sum is obtained by adding the two smallest digits after
        # placing them in the tens place of two separate numbers, then adding
        # the two larger digits in the units place.
        min_sum = 10 * (digits[0] + digits[1]) + digits[2] + digits[3]

        # Return the calculated minimum sum.
        return min_sum
```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for the sort method

class Solution {
    public int minimumSum(int num) {
        // Initialize an array to store the individual digits of the number
        int[] digits = new int[4];

        // Extract the digits from the number and store them in the array
        for (int i = 0; num != 0; ++i) {
            digits[i] = num % 10; // Get the last digit of the number
            num /= 10; // Remove the last digit from the number
        }

        // Sort the array of digits in ascending order
        Arrays.sort(digits);

        // Reconstruct the two minimum possible numbers by pairing the smallest digits
        // with the next smallest digits (10s place and 1s place respectively)
        return 10 * (digits[0] + digits[1]) + digits[2] + digits[3];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the minimum sum of two numbers formed from digits of the input.
    int minimumSum(int num) {
        // Create a vector to store individual digits of the number
        std::vector<int> digits;

        // Extract the digits and put them into the vector
        while (num) {
            digits.push_back(num % 10); // Get the last digit of 'num'
            num /= 10;                  // Remove the last digit from 'num'
        }

        // Sort the digits
        std::sort(digits.begin(), digits.end());

        // The smallest two-digit numbers are formed by taking the two smallest digits
        // and two largest digits and arranging them as shown below:
        // For example, digits = {1, 2, 3, 5} forms 12 and 35 which is the minimum sum possible
        return 10 * (digits[0] + digits[1]) + digits[2] + digits[3];
    }
};
```

Typescript Solution

```typescript
/**
 * Calculate the minimum sum of two numbers formed from digits of the input number.
 *
 * @param num - A four-digit number.
 * @returns The minimum sum of two new numbers formed by rearranging the digits.
 */
function minimumSum(num: number): number {
    // Create an array to hold the individual digits of the input number
    const digits: number[] = new Array(4).fill(0);

    // Extract the digits from the input number and fill the digits array
    for (let i = 0; i < 4; ++i) {
        digits[i] = num % 10; // Get the last digit of the current num
        num = Math.floor(num / 10); // Remove the last digit from num
    }

    // Sort the array of digits in ascending order
    digits.sort((a, b) => a - b);

    // Form two new numbers:
    // The first new number is formed by adding the two smallest digits, then multiplying by 10
    // The second new number is formed from the remaining two digits
    // This ensures the smallest possible sum
    const newNumber1 = 10 * (digits[0] + digits[1]);
    const newNumber2 = digits[2] + digits[3];

    // Return the sum of the two new numbers
    return newNumber1 + newNumber2;
}
```

Time and Space Complexity

Time Complexity:

The provided function consists of a while loop that iterates through the digits of the input number (`num`). This loop will run a number of times equal to the number of digits in `num`. For a 32-bit integer, the maximum number of digits is 10 (`2147483647` is the largest 32-bit integer). Therefore, the while loop has a constant runtime with respect to the size of the input and does not depend on the value of `num`. Sorting the `nums` list which contains at most 4 digits using a basic sorting algorithm (like Timsort, which is used by Python's `sort()` method) has a constant time complexity because the number of digits is fixed and small. The final return statement is a constant time operation. Hence, overall, the time complexity is $O(1)$, a constant time complexity.

Space Complexity:

Space is allocated for the `nums` list which will always have at most 4 elements since the maximum number of digits for an integer is 4 in this problem context. Hence, the space complexity of the code is $O(1)$, a constant space complexity, since it does not grow with the size of the input number.