258. Add Digits

Easy Math Number Theory Simulation

Problem Description

Given an integer `num`, the task is to reduce this number to a single digit by adding its constituent digits together. If the sum is still not a single digit, the process should be repeated with the new sum until only one digit remains. For example, if the input is `num = 38`, the process would be:

- `3 + 8 = 11`
- `1 + 1 = 2` Since `2` is a single digit, `2` is the final result. This process is often referred to as digital root or repeated digital sum.

Intuition

The solution to this problem uses a number theory concept known as digital root. The digital root of a non-negative integer is the position it holds relative to the largest multiple of 9 less than or equal to the number. It can be observed that the digital root of any number is a pattern that repeats every 9 numbers and is related to the modulo operation.

For any number `n > 9`, the digital root is equivalent to `1 + ((n - 1) % 9)`. This is because numbers 1 through 9 have a digital root which is the number itself, and the pattern then repeats from 10 onwards, with 10 having a digital root of 1 again, 11 having a digital root of 2, and so on.

Therefore, the solution directly calculates this value without the need to iteratively add the digits of the number. It handles the edge case of `0` separately, as its digital root should be `0`. The given code uses this principle to return the digital root in a single line:

- If `num` is `0`, return `0`.
- If `num` is not `0`, apply the digital root formula: `(num - 1) % 9 + 1`.

Solution Approach

The implementation of the solution utilizes a mathematical shortcut to find the digital root of an integer, eliminating the need for iteration or recursion. This solution does not use any complex data structures or algorithms but simply applies a mathematical formula to accomplish the task in constant time complexity O(1). Below is an explanation of the code provided in the reference solution:

```
class Solution:
    def addDigits(self, num: int) -> int:
        return 0 if num == 0 else (num - 1) % 9 + 1
```

The algorithm follows these steps:

1. Check if `num` is `0`. If it is, the digital root is also `0`. This case is handled separately because the modulo operation does not apply to `0` as we want the digital root of `0` to be `0`.
2. If `num` is not `0`, apply the digital root formula `(num - 1) % 9 + 1`.

The method is based on the property that for any number `n` the sum of its digits until a single digit is achieved (its digital root) is equal to that number modulo 9, except for multiples of 9 where the result is `9` and not `0`. The formula `(num - 1) % 9 + 1` effectively transforms the range of `num % 9`, which is `0` to `8`, to `1` to `9`, catering to the digital root definition.

This concise approach makes the implementation very efficient since it does not involve any loops, recursion, or the use of additional data structures, hence the constant time complexity.

Example Walkthrough

To demonstrate the solution approach, let's take `num = 94` as our example.

- Normally, we would add the digits `9 + 4 = 13`.
- Next, we would add `1 + 3` to get `4`.

Using our one-liner formula, we can arrive at the same result without doing the two-step summation process:

1. We first check if the number is `0`, which it is not.
2. We apply the formula `(num - 1) % 9 + 1`:
   - `(94 - 1) % 9 + 1`
   - `93 % 9 + 1`
   - As 93 is a multiple of 9, `93 % 9` is `0`.
   - Finally, `0 + 1` is `1`.

The result is `1` for our input `94`, but we need a single digit other than `0` (since we took `94`, not a multiple of `9`), we add `1` to our `0` result to correct the range, thus our single digit is `1`.

This simple example illustrates the efficiency of using the digital root formula, bypassing the need for multiple summing steps.

Solution Implementation

Python

```
class Solution:
    def add_digits(self, num: int) -> int:
        """
        This method uses the digital root concept, which in
        this case can be computed using the modulo operation.
        The digital root is a single-digit value obtained by
        an iterative process of summing digits, on each iteration
        using the result from the previous iteration to compute the sum
        of digits until a single-digit number is obtained.

        :param num: The number from which we need to find the digital root.
        :return: The digital root of the given number.
        """
        # If the number is 0, the digital root is also 0.
        if num == 0:
            return 0
        # For all other numbers, compute the digital root using the formula:
        # (num - 1) % 9 + 1. This works because it maps a number to its
        # digital root, which is the remainder when divided by 9,
        # except when the number is a multiple of 9, then the result
        # is 9 instead of 0, hence the '-1' before modulo and '+1' after.
        else:
            return (num - 1) % 9 + 1

# Example usage:
# sol = Solution()
# print(sol.add_digits(38))  # Output would be 2
```

Java

```
class Solution {
    // Method to add the digits of an integer until the sum is a single digit
    public int addDigits(int num) {
        // This algorithm is based on the digital root concept, which can be computed
        // using the formula below. The result is the same as summing the digits
        // until a single digit is reached.
        // The expression '(num - 1) % 9' gives a result in the range 0-8 for all
        // positive integers, which is then offset by adding 1 to get a range from 1-9.
        // It handles the case when num is a multiple of 9 correctly, which should return 9.
        return (num - 1) % 9 + 1;
    }
}
```

C++

```
class Solution {
public:
    // Function to add the digits of a number until the result is a single digit.
    int addDigits(int num) {
        // Using Digital Root formula which is a well-known and proven method for this problem.
        // If the number is 0, the digital root is 0.
        if (num == 0) {
            return 0;
        }
        // For all other numbers, the digital root is the remainder when the number
        // is divided by 9, with a special case when num is a multiple of 9.
        else {
            // Subtract 1 from num to handle the multiple of 9 scenario, then
            // mod by 9 to get the remainder, and add 1 to adjust for the initial subtraction.
            return (num - 1) % 9 + 1;
        }
    }
};
```

TypeScript

```
// Function to add the digits of a number until the result is a single digit.
function addDigits(num: number): number {
    // If the number is 0, the digital root is 0.
    if (num === 0) {
        return 0;
    }
    // For other numbers, use the Digital Root formula, which is a subtraction of 1,
    // modulus operator (to find the remainder when diving by 9), and then an addition of 1.
    // This handles the case where num is a multiple of 9, and provides the correct digital root.
    else {
        // Subtract 1 from num before applying modulus operator
        // Then add 1 after modulus operation to obtain the correct digital root
        return ((num - 1) % 9) + 1;
    }
}


class Solution:
    def add_digits(self, num: int) -> int:
        """
        This method uses the digital root concept, which in
        this case can be computed using the modulo operation.
        The digital root is a single-digit value obtained by
        an iterative process of summing digits, on each iteration
        using the result from the previous iteration to compute the sum
        of digits until a single-digit number is obtained.

        :param num: The number from which we need to find the digital root.
        :return: The digital root of the given number.
        """
        # If the number is 0, the digital root is also 0.
        if num == 0:
            return 0
        # For all other numbers, compute the digital root using the formula:
        # (num - 1) % 9 + 1. This works because it maps a number to its
        # digital root, which is the remainder when divided by 9,
        # except when the number is a multiple of 9, then the result
        # is 9 instead of 0, hence the '-1' before modulo and '+1' after.
        else:
            return (num - 1) % 9 + 1

# Example usage:
# sol = Solution()
# print(sol.add_digits(38))  # Output would be 2
```

Time and Space Complexity

The time complexity of the given code is `O(1)`. This is because the calculation `num - 1) % 9 + 1` is a constant-time operation, unrelated to the size of the input `num`. The function performs a single arithmetic operation regardless of the value of `num`, so the time it takes to run does not change with larger inputs.

The space complexity of the code is also `O(1)`. The algorithm uses a fixed amount of space for the computations; it does not allocate any additional space that grows with the input size. There are no data structures used that would scale with the size of `num`, only a few variables for the calculation.