258. Add Digits

25 June 2024 Saturday
Kaushal Pratap

In this Leetcode problem, we have to keep summing the digits of a given integer until we arrive on a single digit.

For example, 38,
3 + 8 = 11
1 + 1 = 2

So, the answer will be 2.
In my one of the previous papers, I talked about a similar problem where we have to sum up the squares of the digits until we get 1 as answer. That time I used a Hash Table to design the algorithm.

Here's the link to that document - https://github.com/kaushal-pratap/code-portifolio/blob/main/Technical%20Documentation/HashMap.pdf

But in this problem, what I found interesting is you can actually solve it in O(1) time complexity as you don't need any loops or anything. I found an interesting theory or a new concept about this problem on Wikipedia.

Digital root

Digital root or repeated digital sum is a process which will give you the single digit after repetitive summation of digits of a given integer.

And digital root is nothing but the remainder with base 10.
If you want to find the digital root or the repetitive summation of digits until a single digit, you have to divide the given integer by 9. Because on the base 10, we're counting the numbers from 0-9.

For example:
Digital root of 12345 is 6, as 1+2+3+4+5 = 15

1+5 = 6
And round off 12345 % 9 = 6:

In short, if you want to get the single digit without loops and anything, you can just find the remainder after dividing the given integer with 9.

Wikipedia article - https://en.wikipedia.org/wiki/Digital_root

Image from Wikipedia

Let n be a natural number. For base $b > 1$, we define the **digit sum** $F_b : \mathbb{N} \to \mathbb{N}$ to be the following:

$$F_b(n) = \sum_{i=0}^{k-1} d_i$$

where $k = \lfloor \log_b n \rfloor + 1$ is the number of digits in the number in base b, and

$$d_i = \frac{n \bmod b^{i+1} - n \bmod b^i}{b^i}$$

is the value of each digit of the number. A natural number n is a **digital root** if it is a fixed point for F_b, which occurs if $F_b(n) = n$.

All natural numbers n are preperiodic points for F_b, regardless of the base. This is because if $n \geq b$, then

$$n = \sum_{i=0}^{k-1} d_i b^i$$

and therefore

$$F_b(n) = \sum_{i=0}^{k-1} d_i < \sum_{i=0}^{k-1} d_i b^i = n$$

because $b > 1$. If $n < b$, then trivially

$$F_b(n) = n$$

Therefore, the only possible digital roots are the natural numbers $0 \leq n < b$, and there are no cycles other than the fixed points of $0 \leq n < b$.

Let's look at the code for a better understanding. The code is very simple and easy but there's a special case or we can say an additional task after finding the remainder in one case that we will discuss next. Also, it's very important to learn these kinds of amazing theories.

Code-

```
5  class Solution:
6      def addDigits(self, num: int) -> int:
```

- We will begin by creating our class and function in line 5 and 6.
- We will take the integer input as *num*.

```
7               remainder = num % 9
8               if remainder == 0:
9                   return remainder + 9
10              return remainder
```

- Then, we will initialize the remainder in line 7.
- Now, in line 8, if remainder is zero, we will add 9 to it and return the function. And remainder + 9 will be our final answer in line 9.
  As we know, the remainder will only get zero when the integer is perfectly divisible by 9 like 9, 18, 27, 36 etc.
  But the fact is, if we return remainder as zero, that means zero is our final answer. And according to the problem statement, we will return the number which is left at last after all summations. But it's impossible to get zero after addition. So, that's why we will add 9 to the remainder.
- But why only 9?
- Cause all the multiples of 9 will eventually sum up to 9 and at the end, we'll be having our final answer as 9 only.
- But if the remainder is not zero, but it's absolutely single digit. So, we will simply return the remainder only in line 10.

- In short, just find the remainder by dividing the given integer by 9.

Time complexity analysis-

1. Creating our class and function in line 5 and 6, have a time complexity of O(1).
2. Calculating remainder in line 7 is also O(1).
3. Checking the conditions in *if statements* and returning the values in line 8 to 10 accounts for O(1).

So, the overall time complexity of this code is O(1).