1015. Smallest Integer Divisible by K

Medium Hash Table Math

Problem Description

This problem requires finding the smallest positive integer `n` which consists entirely of the digit `1` and is divisible by a given positive integer `k`. We are interested in the length of such an integer `n`, rather than the number itself. The challenge is that `n` could be very large, potentially exceeding the limits of standard data types used in programming (like a 64-bit signed integer).

Here are the key points to understand:

- `n` must be made up of the digit `1` repeated one or more times. Some examples of such numbers are `1`, `11`, `111`, `1111`, and so on.
- We should find the smallest such number `n` that is divisible by the given `k`.
- The length we are after is the count of the digit `1` in this smallest number `n`.
- If there is no such `n` that can be divisible by `k`, we should return `-1`.

The main challenge in solving this problem is to handle the potentially large size of `n` without running into overflow issues.

Intuition

To arrive at the solution for this problem, we need a way to check if a number consisting only of the digit `1` is divisible by `k` without having to actually construct the potentially very large number. The insight here is to use the properties of modular arithmetic.

Here's the intuition behind the solution:

- We start by checking if `1 % k` is `0`, which means `1` is divisible by `k`. If it is, we return `1` since this is the smallest possible length.
- If not, then we iteratively build the number `n` by appending the digit `1` to it and simultaneously calculating `n % k` for each new `n`. This operation is equivalent to `(n * 10 + 1) % k`.
- This process continues until we either find that `n % k == 0` (meaning `n` is divisible by `k`) or we have iterated `k` times without finding such an `n`.
- We iterate up to `k` times because of a pigeonhole principle - if `n` is not found to be divisible by `k` after `k` iterations, it will never be, and we return `-1`.

Essentially, the solution relies on the fact that a number that only contains the digit `1` can be built iteratively, and at each step, we can check divisibility using modular arithmetic, avoiding the need to deal with very large numbers directly. If the remainder is ever zero, we've found our number `n` and we return its length, corresponding to the number of iterations.

Solution Approach

The solution provided is an implementation of a direct simulation, leveraging modular arithmetic. There is no need for any complex data structures or algorithms beyond basic arithmetic operations and a for-loop to control the iterations. Here is how the implementation works:

1. Initialize `n` to `1 % k`. This is because we start by analyzing the smallest number made up only of the digit `1`, which is the number `1` itself.
2. Loop from `1` to `k` using a for-loop, which represents the number of digits in `n` and not the number itself. The reason for this restriction is the pigeonhole principle mentioned earlier, which suggests that within `k` steps, a remainder should repeat, leading to a cycle without finding a divisible number if it has not already been found.
3. Inside the loop, we first check if the current remainder `n` is `0`. If this is the case, the number represented by the current length `i` (number of digits in `n`) is divisible by `k`, and we return `i` as the result.
4. If `n` is not zero, we update `n` by trying to append a digit `1` to our number, which is mathematically represented by `n = (n * 10 + 1) % k`. The multiplication by `10` shifts the digits left, adding a new zero to the end, and then we add `1` to include the new digit `1` at the lowest place value. We use modulus `k` to keep `n` within a manageable size, which is also the remainder we are interested in.
5. If we complete the for-loop without finding a number `n` that is divisible by `k` (i.e., without the remainder becoming zero), we return `-1`. This implies that no number consisting solely of `1`s up to the length of `k` is divisible by `k`, and by the pigeonhole principle, no such number exists.

In summary, the solution uses a smart brute-force method to simulate the actual division operation on numbers consisting of the digit `1` only, while calculating the remainders using modular arithmetic, thus avoiding dealing with very large numbers that could lead to overflow issues.

Example Walkthrough

Let's take a simple example where `k = 3` to illustrate how the solution approach works in practice:

1. We start by initializing `n` with the value of `1 % 3`. Upon calculation, `1 % 3` equals `1` since `1` divided by `3` leaves a remainder of `1`.
2. We then begin our for-loop, which will iterate from `1` to `k` (in this case, `3`). For each iteration `i`, we perform the following steps:

   a. On the first iteration (`i = 1`), we check if `n` is `0`. It is not (`n = 1`), so we continue.

   b. Since `n` is not `0`, we then update `n` as `(n * 10 + 1) % k`. For the first iteration: `(1 * 10 + 1) % 3` equals `11 % 3`, which gives us a remainder of `2`. We then proceed to the next iteration with `n` now equal to `2`.
3. In the second iteration (`i = 2`), we find that `n` is not `0`, so we perform the update step again: `(n * 10 + 1) % k` becomes `(2 * 10 + 1) % 3`, which simplifies to `21 % 3`. This gives us a remainder of `0`.
4. As soon as we get a remainder of `0`, we know that the number `21` (which consists of `2` digits of `1`) is divisible by `3`. Thus, we return the current iteration number `i`, which in this case is `2`.

In conclusion, for this example, the smallest number consisting entirely of the digit `1` that is divisible by `3` has a length of `2` (the number is `11`). Therefore, the function would return `2`.

Solution Implementation

Python

```python
class Solution:
    def smallestRepunitDivByK(self, k: int) -> int:
        # Initialize remainder of the first repunit which is 1
        remainder = 1 % k

        # Iterate through each length from 1 to k
        for length in range(1, k + 1):
            # If the remainder is 0, we have found the smallest length
            if remainder == 0:
                return length
            # Update the remainder by appending a '1' to the current number and taking mod
            remainder = (remainder * 10 + 1) % k

        # If no such length was found, return -1 indicating no solution exists
        return -1
```

Java

```java
class Solution {
    public int smallestRepunitDivByK(int k) {
        // Initialize remainder with 1 modulo k, this corresponds to the repunit '1'
        int remainder = 1 % k;

        // Loop through numbers from 1 to k to find the smallest repunit
        for (int length = 1; length <= k; ++length) {
            // If the remainder is 0, then we have found a repunit of length 'length' that is divisible by k
            if (remainder == 0) {
                return length;
            }
            // Update the remainder for the next iteration which corresponds to appending another '1' to the repunit
            // This is equivalent to shifting the current number left by one digit (multiply by 10) and adding another '1'
            remainder = (remainder * 10 + 1) % k;
        }

        // If no such repunit is found that is divisible by k within the first k numbers, return -1
        return -1;
    }
}
```

C++

```cpp
class Solution {
public:
    /**
     * Returns the length of the smallest positive integer that is
     * composed solely of ones and is divisible by 'k'.
     *
     * @param k The divisor to check against.
     * @return The length of the smallest repunit divisible by 'k',
     *         or -1 if none exists within 'k' digits.
     */
    int smallestRepunitDivByK(int k) {
        // Initialize remainder 'currentRemainder' with the remainder of 1 divided by 'k'.
        int currentRemainder = 1 % k;

        // Iterate up to 'k' times to find the smallest repunit.
        for (int i = 1; i <= k; ++i) {
            // If the current remainder is 0, we have found a repunit divisible by 'k'.
            if (currentRemainder == 0) {
                return i; // Return the current length 'i'.
            }

            // Calculate the next remainder when 'i+1' ones are considered.
            // This is akin to appending another '1' to the end of the repunit.
            // The % k ensures we work with remainders reducing the overall number size.
            currentRemainder = (currentRemainder * 10 + 1) % k;
        }

        // If we have not returned within the loop, no repunit of length <= 'k' is divisible by 'k'.
        // Therefore, return -1 to indicate failure.
        return -1;
    }
};
```

TypeScript

```typescript
/**
 * Finds the smallest length of a positive integer number consisting only of ones ('1')
 * that is divisible by the given integer `k`.
 * If there is no such number, the function returns -1.
 *
 * @param {number} k - An integer by which the repunit (a number consisting only of ones) has to be divisible.
 * @return {number} - The length of the smallest repunit divisible by `k`, or -1 if no such repunit exists.
 */
function smallestRepunitDivByK(k: number): number {
    // Initialize the remainder when dividing '1' by 'k'.
    let remainder = 1 % k;
    // Loop up to 'k' times to find the smallest repunit divisible by 'k'.
    for (let length = 1; length <= k; ++length) {
        // If the current remainder is 0, a repunit number that is divisible by 'k' has been found.
        if (remainder === 0) {
            return length;
        }
        // Calculate the next remainder when adding another '1' to the repunit and taking the modulus with 'k'.
        remainder = (remainder * 10 + 1) % k;
    }
    // If no repunit divisible by 'k' has been found after 'k' iterations, return -1.
    return -1;
}
```

```python
class Solution:
    def smallestRepunitDivByK(self, k: int) -> int:
        # Initialize remainder of the first repunit which is 1
        remainder = 1 % k

        # Iterate through each length from 1 to k
        for length in range(1, k + 1):
            # If the remainder is 0, we have found the smallest length
            if remainder == 0:
                return length
            # Update the remainder by appending a '1' to the current number and taking mod
            remainder = (remainder * 10 + 1) % k

        # If no such length was found, return -1 indicating no solution exists
        return -1
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(K)`. This is because there is a for-loop that iterates up to `k` in the worst case scenario. At each iteration, the computational work done is constant, meaning it does not depend on the size of `k`. Therefore, the upper bound of iterations is `k`, leading to a linear time complexity relative to the input value `k`.

Space Complexity

The space complexity of the code is `O(1)`. The code uses a fixed amount of additional memory space regardless of the input size `k`, which includes variables `n` and `i`. Since the memory used does not scale with `k`, the space complexity is constant.