2269. Find the K-Beauty of a Number

Easy Math String Sliding Window Leetcode Link

Problem Description

The problem at hand is about finding the *k-beauty* of an integer `num`. The *k-beauty* of a number is defined by how many of its substrings, with a specified length `k`, are divisors of the number itself when the number is interpreted as a string. Here are a few keypoints to understand the problem:

- A *substring* is a section of a string that is taken continuously from it. For example, "234" is a substring of "12345".
- The *substring* in question must be `k` characters long.
- It must be a divisor of `num`, meaning that `num` divided by this substring should leave no remainder.
- Leading zeros in a substring are allowed, so "01" could be considered a valid substring of "1001" for calculating its k-beauty.
- `0` is not considered a divisor, so any substring that is "00" or leading to an integer value of 0 will not be counted for k-beauty.

The objective is to calculate and return the total count of such divisors found within a given number under these conditions.

Intuition

To solve this problem, the idea is to traverse the number as a string, and check every possible substring of length `k`. For each substring, we check if it is a non-zero number and if it divides `num` without a remainder. If it does, we increment our answer. Here's the step-by-step intuition:

1. Convert the integer `num` to a string because we need to manipulate it as a sequence of characters to extract substrings.
2. Traverse the string representation of `num` and look at each possible substring that has a length of `k`. This is done by running a loop from the start of the string to the point where we have room to take a k-length slice.
3. Convert each substring back to an integer and check two conditions:
   - The number formed by the substring should not be zero, as zero is not a valid divisor.
   - The number `num` should be divisible by this integer representation of the substring, which means `num % t` should be zero (`t` being the integer form of the substring).
4. If both conditions are satisfied, we increment a counter.
5. After checking all possible substrings, we return the counter value which represents the k-beauty of `num`.

Following this approach ensures that we check all possibilities in a single pass over the string representation of `num` leading to an efficient and comprehensive solution.

Solution Approach

The solution involves a straightforward implementation of the intuition behind the problem. No sophisticated data structures or complex patterns are necessary. Instead, the algorithm makes use of simple string manipulation and arithmetic operations to determine the k-beauty. Here is a closer look at the implementation steps:

1. **Convert to String**: The first step involves converting the given integer `num` into its string representation. This is a crucial step since we need to work with substrings which are inherently a sequence of characters.

   ```
   s = str(num)
   ```

2. **Initialize a Counter**: Before we enter the main loop of the algorithm, we initialize a counter called `ans` to zero. This variable keeps track of the number of valid k-length substrings that are divisors of `num`.

   ```
   ans = 0
   ```

3. **Traverse Substrings**: We create a loop that starts from index 0 and goes all the way to `len(s) - k`, which ensures that we can always extract a substring of length `k` without going out of bounds.

   ```
   for i in range(len(s) - k + 1):
   ```

4. **Extract and Convert Substrings**: Within the loop, for each index `i`, we extract the substring starting from index `i` and ending at `i + k`, both inclusive. This is then converted back into an integer.

   ```
   t = int(s[i : i + k])
   ```

5. **Check Divisibility and Non-Zero Condition**: We then check if `t` is a non-zero number and if `num` is divisible by `t`. If so, it means the substring meets the required conditions and should be counted toward the k-beauty.

   ```
   if t and num % t == 0:
   ```

6. **Increment the Counter**: Every time we find a valid substring, we increment our counter by one.

   ```
   ans += 1
   ```

7. **Return the Result**: Once the loop completes, we have checked every possible k-length substring, and the result is the value of `ans`, which gets returned.

   ```
   return ans
   ```

This approach has a time complexity of O(n*k), where n is the number of digits in `num`, because we need to inspect each of the `n - k + 1` possible substrings and, for each, perform an operation that costs O(k) (substring extraction and conversion to integer). It also has a space complexity of O(k) for storing the extracted substring; note that it's effectively O(1) if we consider the substring extraction space as part of the input space.

By methodically examining each step and knowing the complexities, we're able to ensure the algorithm's correctness and performance for solving the problem.

Example Walkthrough

Let's walk through an example to illustrate the solution using the given problem description and intuition. To make it simple, let's take `num = 120` and `k = 2`.

1. **Convert to String**: The number `num` is converted to a string.

   ```
   s = str(120)  # s = "120"
   ```

2. **Initialize a Counter**: We initialize our counter `ans` to zero.

   ```
   ans = 0
   ```

3. **Traverse Substrings**: With `k = 2`, we need to create a loop that runs from index `0` through `len(s) - k` or `3 - 2`, which equals `1`. This will allow us to inspect every possible 2-length substring.

   ```
   for i in range(len(s) - 1):  # i ranges from 0 to 1
   ```

4. **Extract and Convert Substrings**: We extract and convert every substring of length `k` into an integer for each `i`. For `i = 0`, the substring is "12". For `i = 1`, the substring is "20".

   ```
   t = int(s[i : i + 2])  # t is 12 when i = 0 and 20 when i = 1
   ```

5. **Check Divisibility and Non-Zero Condition**: We check if the extracted number `t` is not zero and if `num` is divisible by `t`.

   For `i = 0`, `t = 12`:

   ```
   if 12 and 120 % 12 == 0:  # True, since 120 is divisible by 12
       ans += 1  # ans is now 1
   ```

   For `i = 1`, `t = 20`:

   ```
   if 20 and 120 % 20 == 0:  # True, since 120 is divisible by 20
       ans += 1  # ans is now 2
   ```

6. **Increment the Counter**: The counter has been incremented at each step when the conditions were met, so `ans` is now `2`.

7. **Return the Result**: Having checked all possible 2-length substrings of "120", we now return the final result.

   ```
   return ans  # Returns 2
   ```

This example demonstrates that "120" has 2 substrings of length 2, "12" and "20", both of which are divisors of 120. Thus, the k-beauty of 120 with k = 2 is 2.

Python Solution

```python
class Solution:
    def divisorSubstrings(self, num: int, k: int) -> int:
        # Initialize the count of valid substrings
        valid_substring_count = 0

        # Convert number to string for easy substring manipulation
        num_str = str(num)

        # Loop through the string representation of 'num'
        # only until a point where a substring of length 'k' can be obtained
        for i in range(len(num_str) - k + 1):
            # Extract a substring of length 'k' starting from index 'i'
            substring = int(num_str[i : i + k])

            # Check if the substring (now an int) is a non-zero number
            # and if 'num' is divisible by this number
            # to avoid division by zero error and to ensure it is a valid divisor
            if substring != 0 and num % substring == 0:
                # If conditions are met, increment the valid substring count
                valid_substring_count += 1

        # Return the final count of valid divisors found as substrings
        return valid_substring_count

```

Java Solution

```java
class Solution {

    // This method counts the number of k-digit long substrings within the number num that are divisors of num.
    public int divisorSubstrings(int num, int k) {
        int countDivisors = 0; // This will be the count of divisors
        String numberAsString = Integer.toString(num); // Convert the number to a string

        // Loop through the string representation of num to extract substrings of length k
        for (int i = 0; i <= numberAsString.length() - k; ++i) {
            // Extract the substring of length k starting at index i
            int substringAsInt = Integer.parseInt(numberAsString.substring(i, i + k));

            // Check for both non-zero substrings and if num is divisible by the substring
            if (substringAsInt != 0 && num % substringAsInt == 0) {
                countDivisors++; // If it's a divisor, increment the count
            }
        }
        return countDivisors; // Return the count of k-digit divisors of num
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int divisorSubstrings(int num, int k) {
        // Initialize the count of substrings that are divisors to 0
        int divisorCount = 0;

        // Convert the integer 'num' to a string for easy slicing
        string numStr = to_string(num);

        // Loop over the string to extract all possible substrings of length 'k'
        for (int i = 0; i <= numStr.size() - k; ++i) {
            // Extract a substring of length 'k' starting at position 'i'
            int substringValue = stoi(numStr.substr(i, k));

            // Check if the extracted substring is not zero (to avoid division by zero)
            // and if 'num' is divisible by the integer value of the substring
            // Increment the divisor count if both conditions are true
            divisorCount += (substringValue != 0) && (num % substringValue == 0);
        }

        // Return the total count of divisor substrings
        return divisorCount;
    }
};

```

Typescript Solution

```typescript
function divisorSubstrings(num: number, k: number): number {
    let count = 0; // Initialize a counter to keep track of valid substrings
    const numStr = num.toString(); // Convert the given number to a string for easy manipulation

    // Loop through the string with a window of size 'k'
    for (let i = 0; i <= numStr.length - k; ++i) {
        const substring = numStr.substring(i, i + k); // Extract a substring of length 'k'
        const substringValue = parseInt(substring); // Convert the substring back to an integer

        // Check if the substring is not zero and is a divisor of 'num'
        if (substringValue !== 0 && num % substringValue === 0) {
            count++; // If it's a valid divisor, increment the counter
        }
    }

    return count; // Return the total count of valid divisors
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the given code is `O(n * k)` where `n` is the number of digits in the input `num` and `k` is the length of each substring being checked for divisibility. This is because the code iterates through each possible substring that can be formed using `k` consecutive digits of `num`. Since there are `(n - k + 1)` such substrings and each one takes `O(k)` time to process (due to the slicing operation and the integer conversion), the overall time complexity is `O(n * k)`.

Space Complexity:

The space complexity of the given code is `O(k)` because the only additional space used is for the substring `t` which is a substring of `k` characters. However, because `k` is the length of the substring and does not scale with the size of the input number `num`, it could also be considered `O(1)` if `k` is considered to be a constant. If we treat `k` as a variable, the space complexity accounting for `t` would be `O(k)` due to the string slice operation to create substrings; otherwise, it's `O(1)` for constant-size variables and the output variable `ans`.