400. Nth Digit

Medium Math Binary Search Leetcode Link

Problem Description

The problem asks us to return the `n^th` digit of the concatenation of all positive integers in order. The sequence starts with `1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...` and goes on infinitely. Each number in the sequence is laid out consecutively to form an infinitely long number string. For example, if `n` equals `3`, the function should return `3`, as `3` is the third digit of the sequence. If `n` equals `11`, we should return `0`, which is the 11th digit in the sequence.

Intuition

Going digit by digit would be quite inefficient for large `n`, as the numbers grow larger and the sequence quickly becomes very long. A better method is to calculate the position of the target digit. Let's break the problem into manageable parts:

1. Determine the length of the number `k` where the `n^th` digit lies. Numbers with the same number of digits are in the same block (e.g., the number 12 belongs to the block where each number has two digits – 10 to 99).

2. Calculate the actual number where our digit is present. Once we know the block where the digit is, we can compute the number itself.

3. Determine the exact digit within that number which is our answer.

Here is the step-by-step approach based on the intuition mentioned:

- We first find out the length `k` of the digits where `n` falls. Numbers with 1 digit are from 1 to 9, with 2 digits are from 10 to 99, and so on. We can figure out the count of all such numbers with `k` digits as `9 * 10^(k-1)`.

- We iterate from length `1` upwards, subtracting the count of all the `k`-digit numbers from `n` until `n` is less than the count of the next set of numbers with `k+1` digits. Each time we subtract, we're effectively skipping a whole block of digits.

- After finding the length `k` of numbers where our digit is, we identify the number itself. We do this by adding `10^(k-1)` to the result of `(n-1)//k` -- `10^(k-1)` gets the first number of the block and `(n-1)//k` finds how far from the first number the target one is.

- Finally, we find the `n^th` digit within the number we've identified. The index of our target digit in this number is `(n-1)%k`. We then convert the number to a string and retrieve our target digit using this index.

The given solution code follows these steps to find the answer in a time-efficient manner.

Solution Approach

The implementation in the solution code follows the steps outlined in the intuition to find the `n^th` digit efficiently.

1. **Variables Initialization:** `k` is initialized to `1`, representing the length of numbers we're currently dealing with, and `cnt` is set to `9` because there are 9 numbers of length 1.

2. **While Loop to Determine Block:**

   - The condition `k * cnt < n` is used to continue searching for the right block. If `n` is greater than the total number of digits in the current block (`k * cnt`), we advance to the next block of numbers with one more digit.
   - We subtract the number of digits in the current block from `n` with `n -= k * cnt`. This accounts for the digits we have skipped.
   - We then increase `k` by `1` to reflect the next block's digit length.
   - The count `cnt` is multiplied by 10 to reflect the count for the next size of numbers (`k+1` digits).
3. **Finding the Number:** Once we know the correct block:

   - We calculate the actual number using `num = 10 ** (k - 1) + (n - 1) // k`. `10 ** (k - 1)` gives us the starting number of the block. Then `(n - 1) // k` determines how many numbers into the block our target digit is.
4. **Determining the Exact Digit:**

   - We calculate the index within the number where the `n^th` digit lies using `idx = (n - 1) % k`.
   - We convert the number `num` to a string and retrieve the target digit using `return int(str(num)[idx])`. The conversion to a string allows us to easily access any digit by its index.

- **Algorithm Usage:** The implementation mainly involves arithmetic operations and string manipulation, which is an illustrative example of mathematical computing and iterative search techniques. It avoids brute force iteration by narrowing down the scope to the exact location of the desired digit.

- **Data Structures Used:** No complex data structures are needed for this solution; basic integer and string operations suffice.

- **Patterns Used:** The code employs a direct mathematical approach to determine the position rather than using search patterns or sorting methods.

The whole solution uses an efficient method to bypass a significant amount of unnecessary computation which would result from iterating through all the numbers in the sequence one by one. The core part of the solution hinges on understanding the pattern of digit lengths and being clever with arithmetic to pinpoint the exact location of the desired digit.

Example Walkthrough

Let's illustrate the solution approach with an example where `n = 15`. Our task is to determine the 15th digit in the concatenated sequence of positive integers.

Step 1: Determining the Length of Numbers

We initialize `k` to 1, which is the length of numbers we are currently considering, starting with numbers of length 1 (1 to 9). There are 9 such numbers, so `cnt = 9`.

- Since `n > k * cnt` (15 > 1 * 9), it implies that the 15th digit is not in the first block of single-digit numbers.
- Thus, we subtract the count of single-digit numbers from `n`: `n -= 1 * 9`, resulting in `n = 6`.
- We increase `k` to 2 for the next block of numbers, which are double-digit numbers (10 to 99), and update `cnt` to `9 * 10` to reflect the count for the two-digit numbers.

Step 2: While Loop to Determine Block

- Now `k` is 2 and `cnt` is 90. The condition `2 * 90 < 6` is not true, so we know that the 15th digit is within the block of two-digit numbers.

Step 3: Finding the Number

- We calculate the number containing the 15th digit: `num = 10 ** (2 - 1) + (6 - 1) // 2`.
- This gives us `num = 10 + 2`, because `10` is the first two-digit number and `2` is how many numbers into the two-digit block our target digit is. So, `num = 12`.

Step 4: Determining the Exact Digit

- We then calculate the index within the number where the 15th digit lies: `idx = (6 - 1) % 2`.
- The result is `idx = 1`, so we are looking for the second digit of the number `num`.
- We convert `num` into a string and retrieve the target digit: `int(str(num)[idx])`.
- This results in `int(str(12)[1])`, which evaluates to `2`.

So, the algorithm tells us that the 15th digit in the concatenated sequence of positive integers is `2`.

Python Solution

```python
class Solution:
    def find_nth_digit(self, n: int) -> int:
        # initialize variables
        # 'digit_length' represents the current digit length we are calculating (e.g., 1 for 0-9, 2 for 10-99, etc.)
        digit_length = 1

        # 'digit_count' represents the count of numbers that can be formed with the current 'digit_length'
        digit_count = 9

        # loop to find the correct digit length for the given 'n'
        while digit_length * digit_count < n:
            # subtract the total length covered so far
            n -= digit_length * digit_count

            # increment the digit length since we move on to numbers with more digits
            digit_length += 1

            # increase digit_count by a factor of 10 as we move to the next set of numbers
            digit_count *= 10

        # find the actual number where the result digit is located
        number = 10 ** (digit_length - 1) + (n - 1) // digit_length

        # find the index of the digit within 'number'
        index_within_number = (n - 1) % digit_length

        # get the digit at the calculated index of the number and return it
        return int(str(number)[index_within_number])

# Example usage:
# sol = Solution()
# result = sol.find_nth_digit(15)
# print(result) # Output will be 2, which is the 15th digit in the sequence of the number "123456789101112131415..."

```

Java Solution

```java
class Solution {
    public int findNthDigit(int n) {
        // Initialize digit length `k` for numbers of k digits
        // Initialize count `digitCount` for the count of numbers with `k` digits
        int digitLength = 1;
        int digitCount = 9;

        // Determine the range where the nth digit lies
        while ((long) digitLength * digitCount < n) {
            n -= digitLength * digitCount; // Reduce n by the number of positions we've covered
            digitLength++;                 // Move to next digit length
            digitCount *= 10;              // Increase the count for the next range of numbers
        }

        // Calculate the actual number where the nth digit is from
        int number = (int) Math.pow(10, digitLength - 1) + (n - 1) / digitLength;

        // Calculate the index within the number where the nth digit is located
        int digitIndex = (n - 1) % digitLength;

        // Extract and return the nth digit from number
        return String.valueOf(number).charAt(digitIndex) - '0';
    }
}

```

C++ Solution

```cpp
#include <cmath> // Include cmath library to use pow function
#include <string> // Include string library to convert number to string

class Solution {
public:
    int findNthDigit(int n) {
        // Define variables:
        // k represents the number of digits in the numbers we're currently looking at
        // digitCount represents the total number of digits for the current k digits wide numbers
        int numDigits = 1;
        long digitCount = 9;

        // Loop to find the range in which n falls
        // 1 * 9 digits for numbers with 1 digit
        // 2 * 90 digits for numbers with 2 digits
        // 3 * 900 digits for numbers with 3 digits and so on.
        while (1ll * numDigits * digitCount < n) {
            n -= numDigits * digitCount;
            ++numDigits;
            digitCount *= 10;
        }

        // Once the correct range is found, calculate the actual number where the nth digit is from
        int number = pow(10, numDigits - 1) + (n - 1) / numDigits;

        // Find the index within the number where the nth digit is located
        int indexInNumber = (n - 1) % numDigits;

        // Convert the number to a string to easily access any digit
        string numberStr = to_string(number);

        // Return the required digit converting it back to int
        return numberStr[indexInNumber] - '0';
    }
};

```

Typescript Solution

```typescript
/**
 * Find the nth digit of the infinite integer sequence.
 * @param {number} n - The position of the digit to find in the sequence.
 * @return {number} - The nth digit in the sequence.
 */
function findNthDigit(n: number): number {
    let digitLength = 1; // The current number of digits we are getting through (1 for 0-9, 2 for 10-99, etc.)
    let numberCount = 9; // The count of numbers that have digitLength digits (9 for one-digit numbers, 90 for two-digits, etc.)

    // Loop to find the digitLength in which the nth digit is located
    while (digitLength * numberCount < n) {
        n -= digitLength * numberCount; // Decrease n by the number of digits covered in this step
        digitLength += 1; // Increase the number length we are looking for
        numberCount *= 10; // Increase the count to match the next number length
    }

    // Calculate the actual number where the nth digit is found
    const startOfRange = Math.pow(10, digitLength - 1); // The first number with digitLength digits
    const number = startOfRange + Math.floor((n - 1) / digitLength); // Identify the exact number

    // Find the index within 'number' where the nth digit is located
    const digitIndex = (n - 1) % digitLength;

    // Convert the number to a string and get the digit at digitIndex
    return parseInt(number.toString()[digitIndex]);
}

```

Time and Space Complexity

The time complexity of the given code is `O(log n)` because the while loop runs proportional to the number of digits in `n`. Each increase in the number of digits results in a ten-fold increase in the number range, thus the loop iterates through each digit length once, which is related logarithmically to `n`.

The space complexity is `O(1)` because there are a fixed number of integer variables (`k`, `cnt`, `num`, `idx`) used that do not grow with the input size `n`. The conversion of `num` to a string does not significantly affect the space complexity as it is related to the current `k` value (number of digits), which is a small constant for practical purposes.