728. Self Dividing Numbers

Easy Math

Leetcode Link

Problem Description

The problem provides the concept of a "self-dividing number." A self-dividing number is one where every digit of the number divides the number itself without leaving a remainder. For example, taking the number 128, each of its digits (1, 2, and 8) can divide 128 evenly, hence it is a self-dividing number. Importantly, self-dividing numbers may not contain the digit zero, as dividing by zero is undefined and would violate the self-dividing property.

The task is to write an algorithm that, given two integers `left` and `right`, returns a list of all the self-dividing numbers that lie within the inclusive range from `left` to `right`.

Intuition

The intuition behind the solution is to iterate through each number in the range from `left` to `right` inclusive, and for each number, check if all its digits divide the number itself evenly.

To do this, the following steps are needed:

1. Convert the number to a string so that each digit can be accessed individually.
2. Check each digit:
   - If the digit is '0', the number cannot be self-dividing by definition, since division by zero is not allowed.
   - If the digit is not '0', we then check if the number is divisible by this digit by converting it back to an integer and using the modulo operator `%`. If `num % int(digit)` equals zero, it means the number is evenly divisible by this digit.
3. If all digits pass this divisibility check, we include the number in the output list. If any digit fails (either being '0' or not dividing the number evenly), the number is not included as a self-dividing number.

The provided solution encompasses this reasoning effectively by employing list comprehension, which offers a concise and readable way to generate the list of self-dividing numbers.

Solution Approach

The solution approach for finding self-dividing numbers involves a straightforward algorithm that makes good use of Python's list comprehension and string manipulation capabilities. Here's a breakdown of the approach:

1. **Algorithm:**
   - We iterate over the numbers from `left` to `right` inclusive. This is done using the range function: `range(left, right + 1)`.
   - For each number, we convert the number to a string to easily iterate over each digit: `str(num)`.
   - We then use the `all()` function in Python which is a built-in function that returns `True` if all elements of the given iterable are `True`. If not, it returns `False`.
   - Inside the `all()` function, we have a generator expression that checks two conditions for every digit `i` in the string representation of number `num`. The conditions checked are:
     - The digit should not be '0'. If `i` is '0', it immediately returns `False` due to the first part of the `and` condition.
     - The number `num` should be divisible by the digit (converted back to an integer) without any remainder: `num % int(i) == 0`.
2. **Data Structures:**
   - No additional data structures are used besides the list that is being returned.
   - Strings are used transiently for digit-wise operations.
3. **Patterns:**
   - List Comprehension: This pattern allows us to build a new list by applying an expression to each item in an existing iterable (in our case, the range of numbers).
   - Generator Expression: Inside the `all()` function, we effectively use a generator expression, which is similar to a list comprehension but doesn't create the list in memory. This makes it more efficient, especially when dealing with large ranges of numbers.

Here's the implementation of the approach using Python code:

```python
class Solution:
    def selfDividingNumbers(self, left: int, right: int) -> List[int]:
        return [
            num
            for num in range(left, right + 1)
            if all(i != '0' and num % int(i) == 0 for i in str(num))
        ]
```

In this implementation:

- The list comprehension `[num for num in range(left, right + 1) if all(...)]` generates a list of numbers from `left` to `right` but only includes a number if it meets the condition specified in the `if all(...)` part.
- The condition `all(i != '0' and num % int(i) == 0 for i in str(num))` specifies that for a number to be included in the list, every digit of the number should neither be '0' nor should it divide the number leaving a remainder.

As a result, the function `selfDividingNumbers` returns a list containing all the self-dividing numbers within the provided range.

The approach is elegant due to its simplicity and efficient use of Python's language features to accomplish the task with minimal code.

Example Walkthrough

Let's consider a small range of numbers from `left = 1` to `right = 22` and walkthrough the algorithm to find self-dividing numbers within this range.

1. We start iterating from number `1` to `22`. The number `1` is a self-dividing number because it consists of only one digit which divides itself evenly.

2. Continuing the iteration, all single-digit numbers `2` to `9` are self-dividing because a non-zero single digit will always divide itself without a remainder.

3. When we reach two-digit numbers, we start checking each digit for the self-dividing property. For example, number `10` is not a self-dividing number because it contains the digit '0', which cannot be used for division.

4. Moving on to number `11`, both digits are '1', and since `11 % 1 == 0`, it is a self-dividing number.

5. Number `12` is not a self-dividing number because `12 % 2 == 0`, but `12 % 1 != 0`. Therefore, it fails the self-dividing test.

6. Skipping ahead slightly, the number `15` is also not a self-dividing number because even though `15 % 1 == 0`, we find that `15 % 5 != 0`.

7. For the number `21`, the digit '2' is fine since `21 % 2 == 0`, but the digit '1' fails because `21 % 1 != 0`. So, `21` is not a self-dividing number either.

8. Lastly, looking at `22`, it passes since `22 % 2 == 0` for both digits.

After completing the iteration, using the provided algorithm, we would obtain the list of self-dividing numbers for our range, which are `[1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]`.

The code provided in the solution approach does this process programmatically, using list comprehension to build the final list of self-dividing numbers. With the `all()` function and the generator expression inside it, the code is efficiently checking both conditions for each digit of each number in the specified range.

This example illustrates the step-by-step checks that the provided Python solution performs to return the correct list of self-dividing numbers, mirroring the careful consideration needed to assess each digit's divisibility for every number within the range.

Python Solution

```python
class Solution:
    def selfDividingNumbers(self, left: int, right: int) -> List[int]:
        # Initialize a list to store self-dividing numbers
        self_dividing_numbers = []

        # Iterate over the range from `left` to `right` (both inclusive)
        for num in range(left, right + 1):
            # Initialize a flag, assuming the number is self-dividing
            is_self_dividing = True

            # Iterate over each digit in `num`
            for digit_str in str(num):
                # Convert the digit from string to integer
                digit = int(digit_str)

                if digit == 0 or num % digit != 0:
                    # If the digit is zero or the `num` is not divisible by the digit,
                    # set the flag to False and break out of the loop
                    is_self_dividing = False
                    break

            # If the number is self-dividing, add it to the list
            if is_self_dividing:
                self_dividing_numbers.append(num)

        # Return the list of self-dividing numbers
        return self_dividing_numbers

```

Java Solution

```java
class Solution {
    // Method to find all self-dividing numbers within a given range, from 'left' to 'right'
    public List<Integer> selfDividingNumbers(int left, int right) {
        List<Integer> selfDividingNums = new ArrayList<>(); // List to store the self-dividing numbers
        for (int current = left; current <= right; ++current) { // Loop from 'left' to 'right'
            if (isSelfDividing(current)) { // Check if the current number is self-dividing
                selfDividingNums.add(current); // Add to the list if it is self-dividing
            }
        }
        return selfDividingNums; // Return the list of self-dividing numbers
    }

    // Helper method to check if a number is self-dividing
    private boolean isSelfDividing(int number) {
        for (int remaining = number; remaining != 0; remaining /= 10) { // Loop through digits of the number
            int digit = remaining % 10; // Get the last digit
            if (digit == 0 || number % digit != 0) { // Check if the digit is 0 or if it does not divide the number
                return false; // The number is not self-dividing if any digit is 0 or does not divide the number evenly
            }
        }
        return true; // Return true if all digits divide the number evenly
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to find all self-dividing numbers between the range left and right.
    std::vector<int> selfDividingNumbers(int left, int right) {
        std::vector<int> result; // Vector to store the self-dividing numbers.
        for (int num = left; num <= right; ++num) {
            if (isSelfDividing(num)) {
                result.push_back(num); // Add number to the result if it's self-dividing.
            }
        }
        return result;
    }

private:
    // Helper function to check if a number is self-dividing.
    bool isSelfDividing(int num) {
        for (int t = num; t > 0; t /= 10) { // Iterate over each digit.
            int digit = t % 10;
            // If the digit is 0 or does not divide num, then num is not self-dividing.
            if (digit == 0 || num % digit != 0) {
                return false;
            }
        }
        // If all digits divide num, then num is self-dividing.
        return true;
    }
};

```

Typescript Solution

```typescript
// Type definition for storing an array of numbers.
type NumberArray = number[];

/**
 * Function to find all self-dividing numbers between the range left and right.
 * @param left The start of the range.
 * @param right The end of the range.
 * @returns An array of self-dividing numbers.
 */
function selfDividingNumbers(left: number, right: number): NumberArray {
    const result: NumberArray = []; // Array to store the self-dividing numbers.
    for (let num = left; num <= right; ++num) {
        if (isSelfDividing(num)) {
            result.push(num); // Add number to the result if it's self-dividing.
        }
    }
    return result;
}

/**
 * Helper function to check if a number is self-dividing.
 * A self-dividing number is a number that is divisible by every digit it contains.
 * @param num The number to check.
 * @returns True if the number is self-dividing, or false otherwise.
 */
function isSelfDividing(num: number): boolean {
    for (let t = num; t > 0; t = Math.floor(t / 10)) { // Iterate over each digit.
        const digit: number = t % 10;
        // If the digit is 0 or does not divide num, then num is not self-dividing.
        if (digit === 0 || num % digit !== 0) {
            return false;
        }
    }
    // If all digits divide num, then num is self-dividing.
    return true;
}

```

Time and Space Complexity

The time complexity of the provided code can be analyzed as follows:

- The code iterates through all numbers from `left` to `right` inclusive, which gives us a range of `O(n)` where `n` is the number of integers in the range.
- For each number, it converts the number to a string. The conversion operation is done in constant time per digit. Let's say `k` is the average number of digits in the numbers.
- For each digit in the string representation of the number, it performs a modulo operation. There are `k` digits, so we perform `k` modulo operations for each number.

Therefore, the time complexity is `O(n*k)`, where `n` is the number of integers in the range `[left, right]` and `k` is the average number of digits in those numbers.

The space complexity of the provided code can be analyzed as follows:

- The main additional space usage in the code comes from storing the output list, which at most contains `n` numbers, where `n` is the number of integers in the range `[left, right]`. So, the space needed for the list is `O(n)`.
- Additionally, for each number, a temporary string representation is created, and this string has `k` characters, where `k` is the number of digits in the number. However, since these strings are not stored together and only exist during the execution of the modulo check, we do not count this as additional space that scales with the input.

Thus, the space complexity is `O(n)`, where `n` is the number of integers in the output list.