2575. Find the Divisibility Array of a String

Medium Array Math String

Leetcode Link

Problem Description

In this problem, we have a string `word` which is made up of digit characters (0 to 9) and a positive integer `m`. The idea is to create a *divisibility array* `div` from this string. This array will have the same length as the `word` and each of its elements will either be a `1` or a `0`. The rule for the divisibility array is as follows: if the number represented by the substring of `word` from index `0` to index `i` can be divided by `m` without any remainder, then `div[i]` will be `1`. Otherwise, `div[i]` will be `0`. The task is to return this divisibility array.

An example for clarity: If our `word` is `"1234"` and `m=2`, our `div` array should be `[0, 1, 0, 0]` because only the substring `"12"` is divisible by `2`.

Intuition

The approach to solving this problem involves a consideration of how numbers in base 10 are constructed and how divisibility checks are performed. A key insight is that if we are to calculate whether various prefixes of a number are divisible by `m`, it's not efficient to compute the entire number from scratch at each step.

Instead, we employ a modular arithmetic property that allows us to update our current value by taking into account only the newly added digit. Specifically, if we want to shift a number `x` by one decimal place and then add a digit `d`, this can be expressed as `10*x + d`. However, since we're only interested in divisibility by `m`, we can work with `x` and `d` modulo `m` to keep the numbers small and manageable.

Respectively, with each new digit encountered in the `word`, we multiply our running tally `x` by 10 and add the numerical value of the current digit, always taking the modulo `m` after each operation. This keeps `x` as the remainder of the number composed of digits seen so far divided by `m`. If at any point `x` becomes `0`, the number composed of digits up to that point is divisible by `m`, and we add `1` to our answer array; otherwise, we add `0`.

We iterate over each character in the `word`, apply the process described above, and construct our divisibility array incrementally. This method is efficient and avoids redundant calculations, allowing us to get the answer in linear time with respect to the length of `word`.

Solution Approach

The provided solution uses a straightforward approach where no additional complex data structures or patterns are employed. The algorithm relies on basic arithmetic operations, specifically the modulus operation, and it follows the incremental construction philosophy:

1. Initialize an empty list `ans` which will eventually hold the resulting divisibility array.
2. Begin with a variable `x` set to `0`. This variable represents the current numeric value as we process each character of the input string `word`, considering the modulo `m`.
3. Iterate over each character `c` in the string `word`. For each character,
   - Convert the character to its integer value.
   - Update `x` by multiplying it by `10` (shifting the number to the left by one decimal place) and adding the integer value of the current character to include it in our numeric value.
   - Perform the modulus operation with `m` to update `x` to contain the remainder of the new number modulo `m`.
4. Check if `x` is `0` after the modulus operation. If it is, append `1` to the list `ans` since the number composed of all digits up to this point is divisible by `m`. If `x` is not `0`, append `0` to the list.
5. Proceed to the next character and repeat steps 3 and 4 until all characters are processed.
6. Once done, return the list `ans` as the final divisibility array.

The Python code provided efficiently implements this approach, using a loop to iterate over each character in `word` and modifying the variable `x` iteratively. The modulus operation is used to ensure that the numerical value considered at each step is within manageable bounds and directly corresponds to the divisibility condition.

It's important to note that this approach, while simple, takes advantage of the modulus operation's property that `(a * b) % m = ((a % m) * (b % m)) % m`. This property allows us to keep intermediate values small and perform continuous divisibility checks without having to compute or store large numbers, hence maintaining a constant space complexity with respect to the value of the numbers involved.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we have the string `word = "2034"` and the divisor `m = 3`. We want to generate a divisibility array `div` such that if the number represented by the substring of `word` from index `0` to index `i` is divisible by `3`, then `div[i]` is `1`; otherwise, it is `0`.

1. Initialize an empty list `ans` to hold the resulting divisibility array.
2. Let variable `x` be the running total, initialized to `0`.
3. We iterate over the characters in `word`. Initially, `word[0] = '2'`.
   - Convert character '2' to its integer value, which is 2.
   - Update `x` by calculating `(10 * x + 2) % 3 = (0 * 10 + 2) % 3 = 2 % 3 = 2`.
   - Since `x` is not `0`, append `0` to the list `ans`.
4. The list `ans` is now `[0]` and `x` is `2`.
5. The next character is '0'.
   - Convert '0' to integer 0.
   - Update `x` by calculating `(10 * x + 0) % 3 = (10 * 2 + 0) % 3 = 20 % 3 = 2`.
   - Since `x` is not `0`, append `0` to list `ans`.
6. The list `ans` is now `[0, 0]` and `x` is `2`.
7. The next character is '3'.
   - Convert '3' to integer 3.
   - Update `x` by calculating `(10 * x + 3) % 3 = (10 * 2 + 3) % 3 = 23 % 3 = 2`.
   - Since `x` is not `0`, append `0` to list `ans`.
8. The list `ans` is now `[0, 0, 0]` and `x` is `2`.
9. The last character is '4'.
   - Convert '4' to integer 4.
   - Update `x` by calculating `(10 * x + 4) % 3 = (10 * 2 + 4) % 3 = 24 % 3 = 0`.
   - Since `x` is `0`, append `1` to list `ans`.
10. The final divisibility array `ans` is `[0, 0, 0, 1]`, which corresponds to the fact that the substring "2034" is divisible by `3`, but the substrings "2", "20", and "203" are not.

In this example, the divisibility array `div` for `word = "2034"` and `m = 3` is `[0, 0, 0, 1]`.

Python Solution

```python
from typing import List

class Solution:
    def divisibilityArray(self, word: str, modulus: int) -> List[int]:
        # Initialize a list to hold the final answers.
        divisibility_checks = []
        # Initialize a variable for computing the running remainder.
        running_remainder = 0
        # Iterate over each character in the input word (assumed to be digits).
        for char in word:
            # Update the running remainder by incorporating the new digit
            # and taking the remainder with respect to modulus.
            running_remainder = (running_remainder * 10 + int(char)) % modulus
            # Append 1 to divisibility_checks if the current running_remainder is 0 (divisible by modulus)
            # Otherwise, append 0.
            divisibility_checks.append(1 if running_remainder == 0 else 0)
        # Return the list of divisibility checks.
        return divisibility_checks

```

Java Solution

```java
class Solution {
    public int[] divisibilityArray(String word, int m) {
        int wordLength = word.length(); // Get the length of the provided string
        int[] divisibility = new int[wordLength]; // Create an array to store divisibility results

        long numModM = 0; // Initialize variable to store the modulo m of the number formed so far

        // Iterate over each character in the string
        for (int i = 0; i < wordLength; ++i) {
            // Aggregate the number by shifting the previous number by one decimal place
            // and adding the new digit, then calculate modulo m of the new number
            numModM = (numModM * 10 + (word.charAt(i) - '0')) % m;

            // If the current aggregated number is divisible by m (modulo is 0)
            // then set the corresponding position in the result array to 1
            if (numModM == 0) {
                divisibility[i] = 1;
            }
        }

        return divisibility; // Return the populated divisibility result array
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to create a divisibility array from a string representation of a number
    vector<int> divisibilityArray(string word, int modulus) {
        // Initialize an empty vector to store the results
        vector<int> results;
        // Variable to keep track of the cumulative remainder
        long long currentRemainder = 0;

        // Iterate through each character of the string
        for (char& digitChar : word) {
            // Convert char to corresponding digit and update the cumulative remainder
            currentRemainder = (currentRemainder * 10 + digitChar - '0') % modulus;
            // Check if the current cumulative remainder is divisible by 'modulus' and add the result to 'results'
            results.push_back(currentRemainder == 0 ? 1 : 0);
        }
        // Return the final divisibility array
        return results;
    }
};

```

Typescript Solution

```typescript
/**
 * Computes an array indicating the divisibility of a number (constructed sequentially
 * from the input string "word") by the given divisor "m".
 *
 * @param {string} word - A string representing the digit sequence to form the number.
 * @param {number} m - The divisor to check divisibility against.
 * @returns {number[]} An array with binary values, 1 if divisible and 0 if not, at each step.
 */
function divisibilityArray(word: string, m: number): number[] {
    // Initialize the answer array to be returned
    const answer: number[] = [];

    // We'll use x to calculate the remainder on each step.
    let remainder = 0;

    // Iterate over each character in the input "word"
    for (const digit of word) {
        // Update the remainder: multiply by 10 (shift left in decimal)
        // then add the current digit value, and take modulus by "m"
        remainder = (remainder * 10 + Number(digit)) % m;

        // Add 1 to the answer array if divisible by "m", otherwise add 0
        answer.push(remainder === 0 ? 1 : 0);
    }

    // Return the populated answer array
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `word` string. This complexity arises because the code iterates over each character in the `word` string exactly once. Within the loop, it performs constant time operations: a multiplication, an addition, a modulo operation, and a conditional check. Since none of these operations depend on the size of the string, they don't add any additional factor to the complexity.

Space Complexity

The space complexity of the code is `O(n)`, with `n` being the length of the `word` string. The additional space is used to store the `ans` list, which contains an integer for each character in the `word`. No other significant space-consuming structures or recursive calls are involved, so the space complexity is linear with respect to the input size.