625. Minimum Factorization

Medium Greedy Math

Problem Description

The problem is to find the smallest positive integer (`x`) that can be formed such that the product of all its digits equals a given positive integer (`num`). For instance, if `num` is 18, `x` could be 29 (since 2 * 9 = 18). We want to find the smallest such `x` if it exists. However, there are a couple of constraints: if `x` does not exist or it is not a 32-bit signed integer (which means `x` must be less than 2^31), the function should return 0.

Intuition

To find the smallest integer `x` meeting our criteria, we need to consider a couple of key observations:

1. To minimize `x`, we should try to use the largest digits possible (except for 0 and 1 since they don't change the product). Hence, we should start checking from 9 down to 2.
2. Digits must multiply to `num`, so we'll repeatedly divide `num` by these digits, ensuring divisibility at each step.
3. We build `x` by appending digits to its right, meaning we first find the highest place-value digit and then move towards the lower ones.

The approach works by iterating from 9 to 2 and checking if `num` can be evenly divided by these digits. When it can, the digit divides `num`, and itself is added to what will become `x`. This process repeats until `num` is reduced to 1 (if it can't be reduced to 1, `x` is not possible under the problem constraints). After each division, we scale `x` up by a factor of 10 (to push previously added digits leftward) before adding the new digit. In the end, `x` must be within the 32-bit signed integer range, or else we return 0.

Solution Approach

The implementation follows the intuition closely and uses a straightforward iterative method, with primary focus on the following steps:

1. Early return for numbers less than 2: Given that our smallest possible positive integer that is not `1` must consist of multiple digits, the cases where `num` is `0` or `1` are special. The function returns the `num` itself since no multiplication is needed.

```
if num < 2:
    return num
```

2. Initializing variables: `ans` is initialized to `0` - it will hold the answer. `mul` is set to `1` and is our multiplying factor which helps in building the integer `x` from its least significant digit to the most significant digit.

```
ans, mul = 0, 1
```

3. Iterating from 9 down to 2: The loop runs backwards from 9 to 2, checking at each step if `num` can be divided by `i` without remainder (using the modulus operator `%`).

```
for i in range(9, 1, -1):
```

4. Dividing `num` by the digit if possible: When `num` is divisible by `i`, we divide `num` by `i` using integer division `//=` and update `ans`. The new digit is placed in its correct position by the current value of `mul`. `mul` is then increased by a factor of 10 to make space for the next digit.

```
while num % i == 0:
    num //= i
    ans = mul * i + ans
    mul *= 10
```

5. Checking the final conditions: Once we break out of the loop, we check if `num` has been reduced to `1`. If it's not, it means we could not fully factorize `num` using digits `2-9`, so `x` cannot exist under our constraints. Moreover, we verify that the answer fits into a `32-bit` signed integer by comparing it to `2**31 - 1`. If either condition is not met, we return `0`.

```
return ans if num < 2 and ans <= 2**31 - 1 else 0
```

The algorithm does not use any complex data structures, but it effectively leverages arithmetic operations and a simple for-loop to achieve the goal. This approach is efficient because it processes each digit in `num` at most once and avoids unnecessary computations or storage.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the number `num = 26`. We are tasked with finding the smallest positive integer `x` such that the product of its digits equals 26.

Firstly, the function would check if `num` is less than `2` and would return `num` itself if that's the case. Since `26` is greater than `1`, this early return doesn't get triggered.

```
if num < 2:
    return num
```

Then, the variables `ans` and `mul` are initialized:

```
ans, mul = 0, 1
```

Here, `ans` will be built up to form our final number `x`, and `mul` is the multiplying factor which helps place the found factors into their correct positions within `x`.

The loop starts at `9` and checks if `num` is divisible by any number from `9` down to `2`. It proceeds as follows:

- `9` does not divide `26`, so it moves to the next digit.
- `8` does not divide `26`, the algorithm continues to iterate.
- ...

Until we reach the digit `2`:

```
for i in range(9, 1, -1):
```

- `2` does divide `26`, leaving us with a quotient of `13` (`26 // 2 = 13`).

The current state of our variables is now:

- `num` becomes `13`
- `ans` is updated to `mul * i + ans`, which is `1 * 2 + 0 = 2`
- `mul` is increased, multiplying by `10`, becoming `10`.

Now with `num` as `13`, we continue to iterate. Our range is now exhausted since no digit between `2` to `9` divides `13`. The loop terminates.

Finally, we check:

- Is `num` now `1`? No, it's `13`, so we can't fully factorize `num` using digits `2-9`.
- Would `ans` be within a 32-bit signed integer range if `num` was `1`? We can't check this since the first condition has already failed.

Since we couldn't reduce `num` to `1` by dividing by digits from `2` to `9`, `x` does not exist within the problem constraints. Hence the function should return `0`.

```
return ans if num < 2 and ans <= 2**31 - 1 else 0
```

This example showed that the number `26` cannot be factorized into a product of digits between `2` and `9`, which means there is no such `x` that would satisfy the problem's conditions.

Solution Implementation

Python

```python
class Solution:
    def smallestFactorization(self, num: int) -> int:
        # If the number is less than 2, return the number itself as it's the smallest factorization
        if num < 2:
            return num

        # Initialize the answer and the multiplier for the place value (ones, tens, etc.)
        answer, multiplier = 0, 1

        # Iterate over the digits from 9 to 2 since we want the smallest possible number after factorization
        for i in range(9, 1, -1):
            # While the current digit i is a factor of num
            while num % i == 0:
                # Divide num by i to remove this factor from num
                num //= i
                # Add the current digit to the answer with the correct place value
                answer = multiplier * i + answer
                # Increase the multiplier for the next place value (move to the left in the answer)
                multiplier *= 10

        # If num is fully factorized to 1 and the answer is within the 32-bit signed integer range, return answer
        # Else, return 0 because a valid factorization is not possible or the answer is too big
        return answer if num == 1 and answer <= 2**31 - 1 else 0
```

Java

```java
class Solution {

    // Method to find the smallest integer that has the exact same set of digits as the input number when multiplied.
    public int smallestFactorization(int num) {
        // If the number is less than 2, return it as the smallest factorization of numbers 0 and 1 is themselves.
        if (num < 2) {
            return num;
        }

        // Initialize result as a long type to avoid integer overflow.
        long result = 0;
        // Multiplier to place the digit at the correct position as we build the result number.
        long multiplier = 1;

        // Iterating from 9 to 2 which are the possible digits of the result.
        for (int i = 9; i >= 2; --i) {
            // If the current digit divides the number, we can use it in the factorization.
            while (num % i == 0) {
                // If so, divide the number by the digit to remove this factor from number.
                num /= i;
                // Append the digit to result, which is constructed from right to left.
                result = multiplier * i + result;
                // Increase the multiplier for the next digit.
                multiplier *= 10;
            }
        }

        // After we have tried all digits, num should be 1 if it was possible to factorize completely.
        // Also, the result should fit into an integer.
        // If these conditions hold, cast the result to integer and return it. Otherwise, return 0.
        return num == 1 && result <= Integer.MAX_VALUE ? (int) result : 0;
    }
}
```

C++

```cpp
class Solution {
public:
    // This function returns the smallest integer by recombining the factors of the input 'num'.
    int smallestFactorization(int num) {
        // If the number is less than 2, it is already the smallest factorization.
        if (num < 2) {
            return num;
        }

        // 'result' holds the smallest integer possible from the factorization.
        // 'multiplier' is used to construct the 'result' from digits.
        long long result = 0, multiplier = 1;

        // Iterate from 9 to 2 to check for factors.
        // We start from 9 because we want the smallest possible integer after factorization.
        for (int i = 9; i >= 2; --i) {
            // While 'i' is a factor of 'num', factor it out and build the result.
            while (num % i == 0) {
                num /= i;

                // Add the factor to the result, adjusting the position by 'multiplier'.
                result = multiplier * i + result;

                // Increase the multiplier by 10 for the next digit.
                multiplier *= 10;
            }
        }

        // Check if remaining 'num' is less than 2 (fully factored into 2-9) and result fits into an int.
        // If these conditions are not met, return 0 as specified.
        return num < 2 && result <= INT_MAX ? static_cast<int>(result) : 0;
    }
};
```

TypeScript

```typescript
function smallestFactorization(num: number): number {
    // If the number is less than 2, it is already the smallest factorization.
    if (num < 2) {
        return num;
    }

    // 'result' holds the smallest integer possible from the factorization.
    // 'multiplier' is used to construct the 'result' from digits.
    let result: number = 0;
    let multiplier: number = 1;

    // Iterate from 9 to 2 to check for factors.
    // We start from 9 because we want the smallest possible integer after factorization.
    for (let i: number = 9; i >= 2; --i) {
        // While 'i' is a factor of 'num', factor it out and build the result.
        while (num % i === 0) {
            num /= i;

            // Add the factor to the result, adjusting the position by 'multiplier'.
            result = multiplier * i + result;

            // Check if the result is growing beyond the range of a 32-bit integer
            if (result > Number.MAX_SAFE_INTEGER) {
                return 0;
            }

            // Increase the multiplier by 10 for the next digit.
            multiplier *= 10;
        }
    }

    // Check if remaining 'num' is less than 2 (fully factored into 2-9) and result fits within a 32-bit signed integer.
    // If these conditions are not met, return 0 as specified.
    return num < 2 && result <= 2**31 - 1 ? result : 0;
}
```


```python
class Solution:
    def smallestFactorization(self, num: int) -> int:
        # If the number is less than 2, return the number itself as it's the smallest factorization
        if num < 2:
            return num

        # Initialize the answer and the multiplier for the place value (ones, tens, etc.)
        answer, multiplier = 0, 1

        # Iterate over the digits from 9 to 2 since we want the smallest possible number after factorization
        for i in range(9, 1, -1):
            # While the current digit i is a factor of num
            while num % i == 0:
                # Divide num by i to remove this factor from num
                num //= i
                # Add the current digit to the answer with the correct place value
                answer = multiplier * i + answer
                # Increase the multiplier for the next place value (move to the left in the answer)
                multiplier *= 10

        # If num is fully factorized to 1 and the answer is within the 32-bit signed integer range, return answer
        # Else, return 0 because a valid factorization is not possible or the answer is too big
        return answer if num == 1 and answer <= 2**31 - 1 else 0
```


Time and Space Complexity

The time complexity of the given code is `O(log(num))`. This is because the while loop that reduces `num` by a factor of `i` will run at most `O(log(num))` times. This is similar to how division works in terms of complexity while reducing the number digit by digit. Since the outer for loop is constant and only runs 8 times (from 9 to 2), it does not affect the time complexity significantly.

The space complexity of the code is `O(1)`. No additional space that grows with the input size `num` is used. We use a fixed number of variables (`ans`, `mul`, and `i`) that do not depend on the size of the input `num`.