2847. Smallest Number With Given Digit Product

Medium Greedy Math

Leetcode Link

Problem Description

Given a **positive** integer `n`, the task is to find out the smallest positive integer whose digits multiply to give `n`. This problem boils down to figuring out what digits are needed and in what order they should be arranged to get the smallest possible number. If there is no such number that satisfies the condition, we return `-1`.

The important detail to note is that we're dealing with the multiplication of digits, so we need to use the factors of the given number `n` to construct the required smallest number. We also have to keep in mind that we aim to find the smallest such number, so we have to arrange the digits in ascending order (which is why we are checking divisibility from larger factors to smaller).

Intuition

The intuition behind the solution involves breaking down the given number into its prime factors, but with a twist. We'll need to factorize `n` in such a way that the factors are digits from `1` to `9`.

We can leverage the fact that any positive integer can be factorized into prime factors. However, since we need the factors to be single digits, we use digits 2 to 9 (which are the only single-digit prime and composite numbers that can be factors of `n`).

The solution approach goes through the numbers 9 to 2 (in decreasing order) and checks whether they are factors of `n`. If any number `i` is a factor of `n`, then it's accounted for in the count array `cnt`, and `n` is divided by this factor `i`. This process is repeated until `n` is no longer divisible by `i`.

If at the end of the process, `n` is greater than 1, it means `n` had a factor that is not a digit (1 to 9), which implies no such number of single digits exists whose product is `n`. In this case, we return `-1`.

Otherwise, we construct the answer string by concatenating the factors in ascending order as many times as counted in `cnt`. The resultant string is the smallest number by digits whose product equals the original `n`. If no factors were used and `n` is reduced to `1`, we simply return `"1"` as per the problem condition since 1 has no factors and it is the only case where the input number itself is the answer.

This solution ensures we always use the largest possible factors first, as this minimizes the total number of digits in the end result (since smaller digits need to be used more frequently to reach the same product).

Solution Approach

The solution follows these steps:

1. **Initialize a count array `cnt` of size 10**: Elements of this array will represent how many times each digit from 2 to 9 divides `n`. We use a size 10 array for convenient indexing, even though indices 0 and 1 are not used.

```
cnt = [0] * 10
```

2. **Factorization Loop**: Begin a loop from `i = 9` down to `i = 2`. For each `i`, check if `n` is divisible by `i`. If it is, divide `n` by `i`, decrement `n`, and increment the count of `i` in `cnt`. Continue this process until `n` is no longer divisible by `i`.

```
for i in range(9, 1, -1):
    while n % i == 0:
        n //= i
        cnt[i] += 1
```

3. **Final Check for Remaining `n`**:

   - If `n` **greater than 1**: If after the factorization loop, `n` is still greater than 1, then `n` had a prime factor greater than 9, or another factor that is not a single digit, hence no such single-digit positive integer exists whose product is equal to `n`.

```
if n > 1:
    return "-1"
```

   - If `n` **equals 1**: Construct the smallest number possible by concatenating the factors. String concatenation is used to repeatedly add each factor (`i`) to the result string (`ans`) the number of times it appears in `cnt`. To ensure the smallest possible number is formed, concatenation follows the ascending numerical order (from digit 2 to 9).

```
ans = "".join(str(i) * cnt[i] for i in range(2, 10))
```

4. **Return Result**:

   - If `ans` is not empty, return it. It means we have found factors in the range 2 to 9 that multiply to `n`.
   - Otherwise, if `ans` is empty, it implies `n` was 1 to begin with since no factors were found in the loop. In this special case, the smallest number that can be formed is `"1"`.

```
return ans if ans else "1"
```

The approach leverages **digits as potential prime and composite factors** and takes advantage of the **properties of divisibility** to decompose the number `n` into these factors. By iterating from higher to lower factors, it ensures that larger factors are used up first, minimizing the length of the final answer (as using more small factors would make the number longer). The data structure used is an array to keep track of the count of each factor, and the chosen algorithm is a backwards iteration combined with a greedy strategy for digit selection.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Assume we are given the positive integer `n = 36`.

1. **Initialize the count array `cnt`**: We start by initializing an array `cnt` of size 10 with all elements set to 0:

```
cnt = [0] * 10  # cnt = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

2. **Factorization Loop**: We begin a loop from `i = 9` down to `i = 2`. For the given `n = 36`:

   - Check `i = 9`: `36` is not divisible by `9`, so we move to the next `i`.
   - Check `i = 8`: `36` is not divisible by `8`, so we move to the next `i`.
   - Check `i = 7`: `36` is not divisible by `7`, so we move to the next `i`.
   - Check `i = 6`: `36` is divisible by `6`, so we divide `n` by `6` and increment `cnt[6]`. Now, `n` is `6`, and `cnt` becomes `[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]`.
   - Now `n` is `6`, we check again from `i = 6` (which is a loop invariant condition), we find that `6` is divisible by `6`, so we divide `n` by `6` and increment `cnt[6]` again. Now, `n` is `1`, and `cnt` becomes `[0, 0, 0, 0, 0, 0, 2, 0, 0, 0]`.

3. **Final Check for Remaining `n`**:

   - If `n` **greater than 1**: This is not the case here; after factorization, `n` has been reduced to `1`, so we move to the next step.
   - If `n` **equals 1**: We proceed to construct the smallest possible number by concatenating the factors. Our `cnt` array indicates that the digit `6` should appear twice.

```
ans = "".join(str(i) * cnt[i] for i in range(2, 10))  # ans = "66"
```

4. **Return Result**: Since `ans` is not empty (it is "66"), this is the result we return.

The smallest positive integer whose digits multiply to give `n = 36` is `66`. Note how we used the larger factor (`6`) instead of smaller factors like `2` and `3` multiple times. This gave us the smallest number by digit count satisfying the required condition.

Python Solution

```python
class Solution:
    def smallestNumber(self, num: int) -> str:
        # Initialize a list to keep track of the count of digits (2 to 9)
        digit_count = [0] * 10

        # Check for all prime factors of num from 9 to 2
        # If the current digit divides num, keep dividing and increment the respective count
        for digit in range(9, 1, -1):
            while num % digit == 0:
                num //= digit
                digit_count[digit] += 1

        # If there's a leftover value in num greater than 1, it means num
        # cannot be factored into the digits 2-9, so return "-1"
        if num > 1:
            return "-1"

        # Assemble the result string, comprising each digit multiplied by its count,
        # sorted in ascending order to get the smallest number
        result = "".join(str(i) * digit_count[i] for i in range(2, 10))

        # If result is empty (all counts are zero), return "1" since the smallest number
        # that can be divided by these is 1
        return result if result else "1"
```

Java Solution

```java
class Solution {

    // Method to calculate the smallest number from the product of digits equal to n
    public String smallestNumber(long n) {
        // Array to count the occurrences of each digit from 2 to 9
        int[] digitCount = new int[10];

        // Iterate from digit 9 to 2
        for (int i = 9; i > 1; --i) {
            // Factor out the current digit from n as long as it divides n completely
            while (n % i == 0) {
                // Increment the count for the digit i
                ++digitCount[i];
                // Divide n by the digit i
                n /= i;
            }
        }

        // If n is greater than 1 at this point, it means n had a prime factor greater than 9
        // which cannot be represented as a digit, hence return "-1"
        if (n > 1) {
            return "-1";
        }

        // StringBuilder to construct the smallest number
        StringBuilder sb = new StringBuilder();

        // Iterate over all digits from 2 to 9
        for (int i = 2; i < 10; ++i) {
            // Append each digit to the StringBuilder the number of times it appeared in the earlier step
            while (digitCount[i] > 0) {
                sb.append(i);
                --digitCount[i];
            }
        }

        // Obtain the final string answer from StringBuilder
        String answer = sb.toString();

        // If the answer is empty, then n was either 1 or 0, both cases where '1' is the answer
        return answer.isEmpty() ? "1" : answer;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the smallest number with the given multiplicative factors
    string smallestNumber(long long n) {
        int digitCounts[10] = {}; // Array to store counts of each digit from 2 to 9

        // Factorizes the number n by the digits from 9 to 2
        for (int i = 9; i > 1; --i) {
            while (n % i == 0) { // Check if i is a factor
                n /= i; // Divide n by the factor i
                ++digitCounts[i]; // Increment the count of the digit i in the result
            }
        }

        // If after factorizing there is a remainder greater than 1,
        // it means the number cannot be factorized into the digits 2-9
        if (n > 1) {
            return "-1"; // Return "-1" indicating it's not possible
        }

        string result; // String to store the result
        // Construct the result from the digit counts
        for (int i = 2; i < 10; ++i) {
            // Append the digit i, digitCounts[i] number of times to the result
            result += string(digitCounts[i], '0' + i);
        }

        // If the result is still an empty string, it means n was originally 1
        // In this case, return "1"
        return result.empty() ? "1" : result;
    }
};
```

Typescript Solution

```typescript
// Function to find the smallest number with the given multiplicative factors
function smallestNumber(n: bigint): string {
    // Array to store counts of each digit from 2 to 9
    let digitCounts: number[] = new Array(10).fill(0);

    // Factorizes the number n by the digits from 9 to 2
    for (let i = 9; i > 1; --i) {
        while (n % BigInt(i) === BigInt(0)) { // Check if i is a factor
            n /= BigInt(i); // Divide n by the factor i
            digitCounts[i]++; // Increment the count of the digit i in the result
        }
    }

    // If after factorizing there is a remainder greater than 1,
    // it means the number cannot be factorized into the digits 2-9
    if (n > BigInt(1)) {
        // Return a string indicating it's not possible
        return "-1";
    }

    // String to store the result
    let result = '';
    // Construct the result from the digit counts
    for (let i = 2; i < 10; ++i) {
        // Append the digit i, digitCounts[i] times to the result
        result += i.toString().repeat(digitCounts[i]);
    }

    // If the result is still an empty string, it means n was originally 1
    // In this case, return "1"
    return result || "1";
}
```

Time and Space Complexity

Time Complexity: The time complexity of the code is determined by the while loops that check for divisibility by numbers from 9 to 2. This process can occur at most $O(\log(n))$ times for each divisor because after each iteration, `n` is divided by a number between 2 and 9, which reduces its size by at least a factor of 2. Since there are 8 possible divisors (from 9 to 2), the overall time complexity can be considered $O(8 * \log(n))$, which simplifies to $O(\log(n))$.

Space Complexity: The space complexity of the function is $O(1)$. This is because the array `cnt` is of fixed size 10 (not dependent on the input size `n`), and the rest of the variables are of constant size.