343. Integer Break

Medium Math Dynamic Programming

Problem Description

The task is to take a given integer `n` and break it down into a sum of `k` positive integers, where `k >= 2`. The objective is to find the way of breaking down `n` that results in the maximum possible product of these `k` integers. For instance, if `n` is 8, we can break it down into 3 + 3 + 2, and the product of these numbers (3 * 3 * 2) is 18, which is the maximum possible product for any breakdown of 8 into at least two positive integers. The problem asks for a function that computes the highest product possible for any number `n`.

Intuition

The intuition behind the solution comes from mathematical proof that the optimal way to decompose `n` into a combination of integers for the maximum product is by using as many 3's as possible, with the possible addition of a 2 or 4 but never higher than 4, because breaking 5 into 2 + 3 results in a higher product, as 2 * 3 = 6 > 5.

Here's how we arrive at the solution:

- For `n` less than 4, the maximum product is always `n-1`, as the only way to break the number down is into 1's and another integer (since k has to be >= 2), and 1 does not contribute to a larger product.
- If `n` is divisible by 3 perfectly, the maximum product is simply 3 raised to the power of `n/3`.
- If `n` gives a remainder of 1 when divided by 3, we then subtract 4 from `n` and break the rest into 3's. By subtracting 4 instead of 1, we add another factor of 2 in the product, whereas subtracting 1 would leave us with a factor of 1, which is not helpful. Thus, we return 3 raised to the power of `(n // 3 - 1)` and multiply by 4.
- If `n` leaves a remainder of 2 when divided by 3, we can simply multiply the power of 3's with 2 to get the maximum product.

Therefore, the `integerBreak` function efficiently calculates the maximum product by identifying these patterns and applying the correct mathematical operation based on the remainder of `n` when divided by 3.

Solution Approach

The implementation of the solution follows a straightforward application of mathematical patterns and does not rely on complex data structures or elaborate algorithms like dynamic programming. The reference to dynamic programming in the previous context could be misleading since the provided solution is not using a classic dynamic programming approach but rather direct calculations based on mathematical insights.

Here is a step-by-step explanation of the solution's implementation:

1. Check if `n` is less than 4. If yes, return `n - 1` as the solution. This works because, for numbers 2 and 3, the maximum product is just themselves minus 1 (as we have to break them down into at least two numbers).

```
if n < 4:
    return n - 1
```

2. If `n` is divisible by three (remainder is zero), return `3` raised to the power of `n // 3`. This step leverages the fact that the product of the sum will be maximized if we can divide the number into as many 3's as possible since 3 is the most efficient factor for this purpose.

```
if n % 3 == 0:
    return pow(3, n // 3)
```

3. If `n` gives a remainder of one when divided by three, subtract 4 from `n` to leave a number that's a multiple of three. Then, multiply `3` raised to the power of `(n // 3) - 1` by 4. This is based on the insight that having fours instead of a combination of ones and threes yields a better product.

```
if n % 3 == 1:
    return pow(3, n // 3 - 1) * 4
```

4. If `n` leaves a remainder of two when divided by three, multiply `3` raised to the power of `n // 3` by 2. Here, a single two can be paired with the maximum number of threes for an optimal product.

```
if n % 3 == 2:
    return pow(3, n // 3) * 2
```

The `pow` function in Python efficiently computes powers and is used here because large powers might be involved. It is more efficient than multiplying 3 or any other number in a loop for the number of times specified.

This solution utilizes a mathematical deduction rather than a dynamic programming approach used in some other maximum/minimum value problems. The method exploits the mathematical certainty that multiplying by 3 as much as possible gives the largest product, except in certain cases where the remainder imposes an adjustment to include a 2 or a 4 for optimization.

Example Walkthrough

Let's illustrate the solution approach with an example where `n = 10`.

1. First, we check if `n` is less than 4. Since it is not (n = 10), we move to the next step.
2. Then, we check if `n` is perfectly divisible by 3. In our case, `n % 3` is 1, because when 10 is divided by 3, it leaves a remainder of 1. This means we do not return `3` raised to the power of `n // 3` but proceed to the next step.
3. Now, we deal with cases where the remainder is 1. When 1 is subtracted from `n` (which makes it 9), it is perfectly divisible by 3. But instead of subtracting by 1, we subtract by 4 to make use of this remainder. So we are left with 10 - 4 = 6, which is divisible by 3. Hence, we return `3` raised to the power of `(n // 3) - 1` times 4. In practice, this means calculating `3^(10 // 3 - 1) * 4` which simplifies to `3^(10 // 3 - 1) * 4 = 3^2 * 4 = 9 * 4 = 36`.
4. The case where `n % 3` is 2 does not apply here because our remainder is 1.

So, the highest product we can achieve by breaking 10 into a sum of at least two positive integers is 36. This walkthrough correlates with the solution approach by leveraging mathematical properties and patterns to find the solution without employing iterative or recursive methods that dynamic programming might use.

Solution Implementation

Python

```python
class Solution:
    def integerBreak(self, n: int) -> int:
        # If n is less than 4, the maximum product is always n - 1
        if n < 4:
            return n - 1

        # If n is a multiple of 3, the maximum product is 3 raised to the power of n divided by 3
        if n % 3 == 0:
            return 3 ** (n // 3)

        # If the remainder when n is divided by 3 is 1,
        # the maximum product is 4 times 3 raised to the power of (n // 3 - 1)
        if n % 3 == 1:
            return 4 * (3 ** (n // 3 - 1))

        # If the remainder when n is divided by 3 is 2,
        # the maximum product is 2 times 3 raised to the power of (n // 3)
        return 2 * (3 ** (n // 3))
```

Java

```java
class Solution {

    // This method is used to get the maximum product of integers that sum up to n
    public int integerBreak(int n) {
        // If n is less than 4, the maximum product is n - 1
        if (n < 4) {
            return n - 1;
        }

        // If n is a multiple of 3, the maximum product is 3 raised to the power of (n divided by 3)
        if (n % 3 == 0) {
            return (int) Math.pow(3, n / 3);
        }

        // If n modulo 3 gives remainder 1, we use one 4 (as 2*2) and the rest as 3s
        if (n % 3 == 1) {
            return (int) Math.pow(3, (n / 3) - 1) * 4;
        }

        // If n modulo 3 gives remainder 2, we use one 2 and the rest as 3s
        return (int) Math.pow(3, n / 3) * 2;
    }
}
```

C++

```cpp
#include <cmath> // Include cmath header for the pow function

class Solution {
public:
    // Function to break an integer into a product of maximum sum
    int integerBreak(int n) {
        // When n is less than 4, the maximum product is n - 1
        if (n < 4) {
            return n - 1;
        }

        // When n is a multiple of 3, the maximum product is 3^(n/3)
        if (n % 3 == 0) {
            return std::pow(3, n / 3);
        }

        // When n is more than 4 and gives a remainder of 1 when divided by 3,
        // we can break it as a product of 4 and 3^(n/3 - 1),
        // since 2*2 > 3*1 and we replace a 3 with two 2's
        if (n % 3 == 1) {
            return std::pow(3, n / 3 - 1) * 4;
        }

        // If n gives a remainder of 2 when divided by 3, we simply multiply
        // 3^(n/3) by 2
        return std::pow(3, n / 3) * 2;
    }
};
```

TypeScript

```typescript
function integerBreak(n: number): number {
    // If the input number is less than 4, the maximum product is always 'n - 1'.
    if (n < 4) {
        return n - 1;
    }

    // Calculate the count of '3's that can be multiplied to form the maximum product.
    const countOfThrees = Math.floor(n / 3);

    // If 'n' is divisible by 3, the maximum product is 3 raised to the power of how many times 3 fits into 'n'.
    if (n % 3 === 0) {
        return 3 ** countOfThrees;
    }

    // If the remainder is 1 after division by 3, then taking one '3' out and using a '4' yields a larger product.
    // This is because 2 * 2 is greater than 3 * 1.
    if (n % 3 === 1) {
        return 3 ** (countOfThrees - 1) * 4;
    }

    // If the remainder is 2, simply multiply it with the product of '3's.
    return 3 ** countOfThrees * 2;
}
```

```python
class Solution:
    def integerBreak(self, n: int) -> int:
        # If n is less than 4, the maximum product is always n - 1
        if n < 4:
            return n - 1

        # If n is a multiple of 3, the maximum product is 3 raised to the power of n divided by 3
        if n % 3 == 0:
            return 3 ** (n // 3)

        # If the remainder when n is divided by 3 is 1,
        # the maximum product is 4 times 3 raised to the power of (n // 3 - 1)
        if n % 3 == 1:
            return 4 * (3 ** (n // 3 - 1))

        # If the remainder when n is divided by 3 is 2,
        # the maximum product is 2 times 3 raised to the power of (n // 3)
        return 2 * (3 ** (n // 3))
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(1)`. This is because the calculation performed by the function consists of a constant number of arithmetic operations (addition, subtraction, division, and multiplication) and calls to `pow`, which runs in constant time for fixed exponents. The input size `n` only affects the calculation through division and modulo operations, which are independent of the magnitude of `n` in terms of time complexity.

Space Complexity

The space complexity of the code is also `O(1)`. There are no data structures used that grow with the input size. The function only uses a fixed amount of space for intermediate calculations and to store the final result before returning it.