2979. Most Expensive Item That Can Not Be Bought

Medium   Math   Dynamic Programming   Number Theory

Problem Description

In this problem, Alice is shopping for gifts to give to Bob at a market with an unlimited number of items, each at a unique price point that corresponds to every positive integer. Alice has an endless supply of coins in two denominations, which are distinct prime numbers: `primeOne` and `primeTwo`. The objective is to figure out the most expensive gift that Alice is unable to buy for Bob using any combination of her coins in the two prime denominations. Essentially, we're searching for the largest price `x` that can't be formed by summing multiples of `primeOne` and `primeTwo`.

Intuition

To solve this problem, we can apply the Chicken McNugget Theorem. This theorem states that for any two coprime numbers (which any two distinct primes are by definition), the greatest integer that cannot be expressed as the sum of multiples of these two numbers is obtained by the formula `a * b - a - b`, where `a` and `b` are the numbers in question. In the context of our problem, since `primeOne` and `primeTwo` are distinct primes, they are coprime (meaning they have no common divisors other than 1). Therefore, the solution becomes the product of `primeOne` and `primeTwo` minus each of these primes.

The implementation of this in the solution is straightforward, we simply do the calculation as described by the theorem. We are guaranteed that any number greater than the result we calculate can be made by some combination of `primeOne` and `primeTwo` coins, while no larger number less than the result can be formed, making it the most expensive item Alice cannot purchase.

Solution Approach

The implementation of the given problem's solution is very straightforward, thanks to the applicable mathematical theorem, which has already been referenced in the intuition section: the Chicken McNugget Theorem. This theorem eliminates the need for a more complex algorithm or data structure and simplifies the solution to a simple mathematical expression.

Here's a step by step breakdown of the one-line implementation that is based directly on this theorem:

1. The `mostExpensiveItem` function is defined to take two integers, `primeOne` and `primeTwo`.
2. The function computes the expression `primeOne * primeTwo - primeOne - primeTwo`.
   - Here, we take the product of the two primes, which would represent the greatest number that *could* be formed if you were allowed to use both denominations together for a single price.
   - From this product, we subtract each of the prime denominations themselves. This step is per the Chicken McNugget Theorem, which considers that the actual greatest unattainable price is just below this artificial 'combined denomination'.
3. The computed expression provides the answer to the problem, so it is directly returned as the result of the function.

This one-liner easily resolves the problem with:

- **Efficiency**: It operates in constant time, meaning the time taken doesn't depend on the size of the input—it's a direct calculation.
- **Simplicity**: No loops, conditionals, or auxiliary data structures are necessary.
- **Mathematical Elegance**: It leverages an existing mathematical principle, which is both educational and efficient for coding.

In short, the approach relies on the fact that, aside from `1`, prime numbers have no common factors, and by utilizing our understanding of the Chicken McNugget Theorem, we can quickly determine the greatest price not attainable using any combination of the available prime denomination coins. The elegant solution thus becomes a simple arithmetic operation applied to `primeOne` and `primeTwo`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose Alice has coins of only two prime denominations: `3` and `5` (these are our `primeOne` and `primeTwo` respectively). We want to figure out the most expensive gift that Alice cannot buy using any combination of these coins.

Applying the Chicken McNugget Theorem, which we will use to solve this problem, requires us to calculate `primeOne * primeTwo - primeOne - primeTwo`. In our example, this translates to `3 * 5 - 3 - 5`.

So let's do the math:

1. First, we multiply the two prime denominations: `3 * 5`, which gives us `15`.
2. Next, we subtract both prime denominations from the product: `15 - 3 - 5`, leading to `7`.

Hence, according to the theorem, `7` is the highest-priced gift Alice cannot purchase with any combination of coins of denominations `3` and `5`.

To illustrate that all prices above `7` can indeed be formed using combinations of `3` and `5`:

- For the price of `8`, Alice can use `1 * 3` and `1 * 5` (3 + 5).
- For the price of `9`, Alice can use `3 * 3` (3 + 3 + 3).
- For the price of `10`, Alice can use `2 * 5` (5 + 5).

And so on. It's evident that every number greater than `7` can be represented as a sum of the coins in denominations `3` and `5`. Hence, our example demonstrates and confirms the solution approach and the effectiveness of the Chicken McNugget Theorem in such scenarios.

Solution Implementation

Python

```python
class Solution:
    def mostExpensiveItem(self, prime_one: int, prime_two: int) -> int:
        # Multiply two prime numbers and subtract the sum of the same prime numbers from the product.
        # This calculation reflects some specific formula or business logic.
        return prime_one * prime_two - prime_one - prime_two
        # If prime_one and prime_two are indeed prime numbers as the parameter names suggest,
        # this calculation will never yield the highest possible value for any given pair of primes
        # since it subtracts more than it adds (subtracting each prime once).
```

Java

```java
// Class name should be meaningful, reflecting its purpose or the context in which it is used.
// Assuming we are dealing with a solution for finding the most expensive item based on two prime factors.
class Solution {

    // Java method names should be in camelCase, so 'mostExpensiveItem' is already correctly named.
    // Comments have been added for clarity.

    /**
     * Calculates the cost of the most expensive item based on two prime factors.
     *
     * @param primeOne The first prime factor.
     * @param primeTwo The second prime factor.
     * @return The cost of the most expensive item derived from the two prime factors.
     */
    public int mostExpensiveItem(int primeOne, int primeTwo) {
        // The mathematical operation is assumed to be purposeful,
        // representing some specific business logic or formula.
        // Therefore, no changes are made to the actual expression.

        // Return the result of the formula
        return primeOne * primeTwo - primeOne - primeTwo;
    }
}
```

C++

```cpp
// Class to encapsulate the solution
class Solution {
public:
    // Function to calculate the most expensive item given two prime costs
    int mostExpensiveItem(int primeCostOne, int primeCostTwo) {
        // The calculation assumes a simple formula based on the two prime costs
        // It multiplies the two prime costs and then subtracts each one from the result
        // This could represent a specific pricing strategy or cost calculation
        return primeCostOne * primeCostTwo - primeCostOne - primeCostTwo;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the most expensive item cost based on the input prime numbers.
 *
 * @param {number} primeOne - The first prime number.
 * @param {number} primeTwo - The second prime number.
 * @returns {number} The calculated cost of the most expensive item.
 */
function mostExpensiveItem(primeOne: number, primeTwo: number): number {
    // The cost of the most expensive item is the product of the two prime numbers,
    // subtracting each of the prime numbers from the product.
    const cost: number = (primeOne * primeTwo) - primeOne - primeTwo;

    return cost;
}
```

```python
class Solution:
    def mostExpensiveItem(self, prime_one: int, prime_two: int) -> int:
        # Multiply two prime numbers and subtract the sum of the same prime numbers from the product.
        # This calculation reflects some specific formula or business logic.
        return prime_one * prime_two - prime_one - prime_two
        # If prime_one and prime_two are indeed prime numbers as the parameter names suggest,
        # this calculation will never yield the highest possible value for any given pair of primes
        # since it subtracts more than it adds (subtracting each prime once).
```

Time and Space Complexity

The time complexity of the `mostExpensiveItem` method is `O(1)`. This is because there are no loops or recursive calls in the function, and the mathematical operations of multiplication and subtraction can be performed in constant time regardless of the input size.

Similarly, the space complexity of the method is also `O(1)`. This function does not utilize any additional data structures that grow with the input size. It only calculates a value based on the input parameters and returns it, using a fixed amount of space.