Exercise 1.26

Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the expmod procedure to use an explicit multiplication, rather than calling `square`:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (* (expmod base (/ exp 2) m)
             (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base
             (expmod base (- exp 1) m))
          m))))
```

"I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

The key to this problem is how the recursion works in both cases. With `square`, you'll only ever evaluate `expmod` once per function call. This is simple linear recursion, with order $\Theta(\log n)$ as proven in 1.2.4:

```
(expmod 2 16 3)
(square (expmod 2 8 3))
(square (square (expmod 2 4 3)))
...
(square (square (square .... (square (expmod 2 0 3)))))
|----- log(n) calls to square -----|
```

With `*`, `expmod` will be evaluated twice per function call. Since `expmod` calls itself, it's effectively calling `expmod` 2^n times, so even if `expmod`'s decomposition itself is $\log_2 n$, you'll still end up with a $2^{\log_2 n} = \Theta(n)$ function:

```
(expmod 2 16 3)
|-- (expmod 2 8 3)
    |-- (expmod 2 4 3)
    |-- (expmod 2 4 3)
        ...
        |-- (expmod 2 0 3)
        |-- (expmod 2 0 3)
|-- (expmod 2 8 3)
    |-- (expmod 2 4 3)
    |-- (expmod 2 4 3)
        ...
        |-- (expmod 2 0 3)
        |-- (expmod 2 0 3)
2^n calculations
```