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.

Answer. In order to find out what cause that distinction between our original `expmod` procedure and that of Louis, we'd better trace the process generated by both procedures.

First, let's take a look at the process generated by our original `expmod` procedure, for example, `(expmod 3 4 5)`:

```
(expmod 3 4 5)
(remainder (square (expmod 3 2 5))
           5)
(remainder (square (remainder (square (expmod 3 1 5))
                              5))
           5)
(remainder (square (remainder (square (remainder (* 3
                                                    (expmod 3 0 5))
                                                 5))
                              5))
           5)
(remainder (square (remainder (square (remainder (* 3 1)
                                                 5))
                              5))
           5)
(remainder (square (remainder (square (remainder 3 5))
                              5))
           5)
(remainder (square (remainder (square 3)
                              5))
           5)
(remainder (square (remainder 9 5))
           5)
(remainder (square 4)
           5)
(remainder 16 5)
1
```

Obviously, this is a linear recursive process. If I double the size of `exp` in the initial expression, say `(expmod 3 4 5)`, by means of successive squaring, what the interpreter has to do additionally is to add one more step both on the top and on the bottom of the process. So the number of steps grows logarithmically with the exponent, that is, $\Theta(\log n)$.

On the other hand, by breaking `square` into an explicit multiplication in this case, Louis's procedure generates a tree recursion, as is shown in Figure 1. Note that this is a binary balanced tree with a depth of $\log_2 n$, where n represents the scale of `exp` in Louis's `expmod` procedure. The amount of computation here will have to be multiplied by itself, even if I have only put an increment on `exp`. Thus, the number of steps increases exponently, that is $\Theta(2^d)$, where d is the depth of the tree in Figure 1.[1] Further more, we can express it in terms of n.

$$\begin{aligned}\Theta(2^d) &= \Theta(2^{\log_2 n})\\ &= \Theta(n)\end{aligned}$$

1. The number of steps can also be calculated by the amount of nodes in the tree, except for those who are leaves. Therefore, in a binary balanced tree whose depth is $\log_2 n$, it takes up approximately $\Theta(2^d)$ steps for computation.

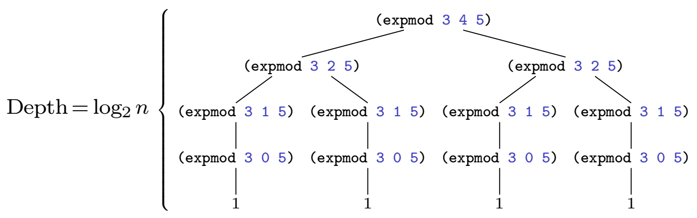

Figure 1. The Tree-Recursive Process Generated in Computing Louis's `expmod` Procedure

Hence, the order of growth here in the process generated by Louis's procedure is $\Theta(n)$.