Exercise 1.16. Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product $a\,b^n$ is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

Answer. Well, this problem is not really easy to tackle. In order to capture the intuition of how an iterative exponentiation process evolved, we first have to degenerate the problem into three cases in terms of n, that is, 0, even and odd. Further more, as is mentioned by the hint: the answer is finally given through a, and we see that this is the simplest case where n is equal to 0. Therefore, we can immediately draw out the profile of our `expt-iter` procedure as the following:

```
(define (expt-iter b n a)
  (cond ((= n 0) a)
        ((even? n) (<??>))
        (else (<??>))))
```

Inspired by the recursive `fast-expt` procedure describe in seciton 1.2.4, we can take advantage of successive squaring in computing exponentials in the case where n is an even number if we use the rule:

$$(b^{n/2})^2 = (b^2)^{n/2}$$

which is also mentioned by the hint. Thus, we can express this as consequence in the conditional expression `((even? n) (<??>))`

```
(expt-iter (square b) (/ n 2) a)
```

Now, let's come to handle the last case where the value of n is an odd number. Notice what stated by the hint in terms of *invariant quantity*: to define the state transformation in such a way that the product $a\,b^n$ is unchanged from state to state, yet, this indicates that the value of b^n should be transfer to a without changing their product. On the other hand, in the former case where n is even, we've seen that successive squaring is a fast way to obtain the value of an exponent. Thus, when it come to the case where n is odd, our procedure should be able to devolve the value between b^n and a and immediately switch back to the case of even. Therefore, we can put down the consequence of the alternative case as the following:

```
(expt-iter b (- n 1) (* a b))
```

So, what we have is the complete implementation of the `expt-iter` procedure:

```
(define (fast-expt b n)
  (expt-iter b n 1))

(define (expt-iter b n a)
  (cond ((= n 0) a)
        ((even? n) (expt-iter (square b) (/ n 2) a))
        (else (expt-iter b (- n 1) (* a b)))))
```