Alyssa P. Hacker's solution would simply fast exponentiate a^n and then check modulo m, whereas as written the original `expmod` would either compute the remainder of $a^{\frac{n}{2}} \bmod m$ squared (if n is even), or compute $a \cdot a^{n-1} \bmod m$ (if n is odd) and then take the result modulo m again. The difference is that Alyssa's method would require checking the remainder of a much larger value (at the end of the process), whereas `expmod` as written would calculate the remainder at each step (since `expmod` calls itself, which invokes another remainder call). Alyssa's method would result in only one remainder check, whereas the original method would result in many, but with values that never will be much bigger than m.

This is a specific method that was chosen because you can compute the remainder of x times y modulo m by calculating separetly x modulo m and y modulo m and then multiplying them together and calculating the remainder of that result modulo m.
For example, let $x = 10, y = 5, m = 7$. We know that $xy \bmod m = (x \bmod m)(y \bmod m) \bmod m$ so we get $50 \bmod 7 = (10 \bmod 7)(5 \bmod 7) \bmod 7 = 3(5) \bmod 7 = 15 \bmod 7 = 1$.
This means when n in `expmod` is even, we can split up $a^n \bmod m$ into $a^{\frac{n}{2}} \bmod m$ twice, and multiply them together (which we can achieve by squaring), and then take the resulting value modulo m.
When `expmod` is odd, we can split up $a^n \bmod m$ into $a^{n-1} \bmod m$ and $a \bmod m$ (which is just a) and then multiply the two and take the resulting value modulo m. This is exactly how the original version of `expmod` is written.

Original version:

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

Alyssa's version:

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

Let's Fermat test $n = 5$, with $a = 2$, for illustrative purposes.

```
(define (fermat-test n a)
  (= (expmod a n n) a))
```

First, using the original `expmod`. Substitution of `even?` is omitted for brevity and simplicity.

```
(fermat-test 5 2)
(= (expmod 2 5 5) 2)
(= (cond ((= 5 0) 1)
         ((even? 5)
          (remainder
           (square (expmod 2 (/ 5 2) 5))
           5))
         (else
```

```
          (remainder
           (* 2 (expmod 2 (- 5 1) 5))
           5)))
   2)
(= (remainder
    (* 2 (expmod 2 4 5))
    5)
   2)
(= (remainder
    (* 2 (cond ((= 4 0) 1)
               ((even? 4)
                (remainder
                 (square (expmod 2 (/ 4 2) 5))
                 5))
               (else
                (remainder
                 (* 2 (expmod 2 (- 4 1) 5))
                 5))))
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square (expmod 2 2 5)))
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square (cond ((= 2 0) 1)
                        ((even? 2)
                         (remainder
                          (square (expmod 2 (/ 2 2) 5))
                          5))
                        (else
                         (remainder
                          (* 2 (expmod 2 (- 2 1) 5))
                          5))))
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square
           (remainder
            (square (expmod 2 1 5))
            5))
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square
           (remainder
            (square (cond ((= 1 0) 1)
                          ((even? 1)
                           (remainder
```

```
                          (square (expmod 2 (/ 1 2) 5))
                          5))
                         (else
                          (remainder
                           (* 2 (expmod 2 (- 1 1) 5))
                           5))))
            5))
          5)
     5)
    2)
(= (remainder
     (* 2 (remainder
           (square
            (remainder
             (square (remainder
                      (* 2 (expmod 2 0 5))
                      5))
             5))
           5)
     5)
    2)
(= (remainder
     (* 2 (remainder
           (square
            (remainder
             (square (remainder
                      (* 2 (cond ((= 0 0) 1)
                                 ((even? 0)
                                  (remainder
                                   (square (expmod 2 (/ 0 2) 5))
                                   5))
                                 (else
                                  (remainder
                                   (* 2 (expmod 2 (- 0 1) 5))
                                   5)))))
                      5))
             5))
           5)
     5)
    2)
(= (remainder
     (* 2 (remainder
           (square
            (remainder
             (square (remainder
                      (* 2 1)
                      5))
             5))
           5)
     5)
    2)
(= (remainder
     (* 2 (remainder
           (square
            (remainder
             (square (remainder
```

```
                    2
                    5))
           5))
         5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square
           (remainder
            (square 2)
            5))
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square
           (remainder
            4
            5))
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          (square
           4)
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          16
          5)
    5)
   2)
(= (remainder
    (* 2 (remainder
          16
          5)
    5)
   2)
(= (remainder
    (* 2 1)
    5)
   2)
(= (remainder
    2
    5)
   2)
(= 2 2)
```

true

Now let's try the same for Alyssa P. Hacker's `expmod`.

```
(fermat-test 5 2)
(= (expmod 2 5 5) 2)
(= (remainder (fast-exp 2 5) 5)
   2)
(= (remainder
    (cond ((= 5 0) 1)
          ((even? 5) (square (fast-exp 2 (/ 5 2))))
          (else (* 2 (fast-exp 2 (- 5 1)))))
    5)
   2)
(= (remainder
    (* 2
       (fast-exp 2 4))
    5)
   2)
(= (remainder
    (* 2
       (cond ((= 4 0) 1)
             ((even? 4) (square (fast-exp 2 (/ 4 2))))
             (else (* 2 (fast-exp 2 (- 4 1))))))
    5)
   2)
(= (remainder
    (* 2
       (square (fast-exp 2 2)))
    5)
   2)
(= (remainder
    (* 2
       (square (fast-exp 2 2)))
    5)
   2)
(= (remainder
    (* 2
       (square (cond ((= 2 0) 1)
                     ((even? 2) (square (fast-exp 2 (/ 2 2))))
                     (else (* 2 (fast-exp 2 (- 2 1))))))
    5)
   2)
(= (remainder
    (* 2
       (square (square (fast-exp 2 1)))
    5)
   2)
(= (remainder
    (* 2
       (square (square (cond ((= 1 0) 1)
                             ((even? 1) (square (fast-exp 2 (/ 1 2))))
                             (else (* 2 (fast-exp 2 (- 1 1)))))))
    5)
   2)
(= (remainder
    (* 2
       (square (square (* 2 (fast-exp 2 0))))
    5)
   2)
```

```
(= (remainder
    (* 2
       (square (square (* 2 (cond ((= 0 0) 1)
                                  ((even? 1) (square (fast-exp 2 (/ 0 2)))
                                  (else (* 2 (fast-exp 2 (- 0 1)))))))))
    5)
   2)
(= (remainder
    (* 2
       (square (square (* 2 1)))
    5)
   2)
(= (remainder
    (* 2
       (square (square 2))
    5)
   2)
(= (remainder
    (* 2
       (square 4)
    5)
   2)
(= (remainder
    (* 2
       16)
    5)
   2)
(= (remainder
    32
    5)
   2)
(= 2
   2)
```

true

As you can see, the final `remainder` calculation of Alyssa's method computes 32 modulo 5, whereas the original method's largest computation is only 16 modulo 5. This doesn't seem like a big difference for small numbers, but when scaled up, Alyssa's method will have operations like `remainder` be called on significantly larger numbers.

So which one is faster? For very small numbers, like what is illustrated above, we wouldn't expect the speed difference to be noticable. But what about for larger numbers? It turns out that swapping the original `expmod` for Alyssa's results in the interpreter not being able to find any primes larger than 10^6, even when only one a is being tested. Even for smaller primes ($> 10^3$), it is considerably slower than the original `expmod`. It turns out that large modulo operations must be very computationally costly! Therefore, we can conclude that Alyssa is not correct and this procedure would not serve well for our fast prime tester.