```
#|
Name: Elise Harvey
Kerb: erharvey
6.5151 ps00
|#


(load "Desktop/6.5151/ps00/code/p0utils.scm")


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 1       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: Try applying each operator to some integers.

(modulo 13 8) ; = 5
(remainder 13 8) ; = 5
(modulo -13 8) ; = 3
(remainder -13 8) ; = -5
(modulo -13 -8) ; = -5
(remainder -13 -8) ; = -5


;;; Q: What is the difference between remainder and modulo? Which one
is the best choice for implementing modular arithmetic as described
above?

#|
Remainder and modulo both return the "remainder" of division, but they
tend to behave differently for negative numbers. For remainder, the
output will follow the sign of the first number. On the other hand,
modulo will follow the sign of the second number.

In this case, we need n>0 and want the output to be in the range [0,
n-1]. With this in mind, it would make more sense to use modulo as our
function for implementing modular arithmetic.
|#


;;; Q:  Write procedures for addition, subtraction, and multiplication
modulo n.

; addition
(define +mod
  (lambda (a b n)
    (modulo (+ a b) n)))

; subtraction
(define -mod
```

```
  (lambda (a b n)
    (modulo (- a b) n)))

; multiplication
(define *mod
  (lambda (a b n)
    (modulo (* a b) n)))

;; test cases

; addition
(+mod 7 5 8) ; confirmed = 4
(+mod 10 10 3) ; confirmed = 2
(+mod 99 99 100) ; = 98

; subtraction
(-mod 5 12 2) ; confirmed = 1

; multiplication
(*mod 6 6 9) ; confirmed = 0
(*mod 50 -3 100) ; = 50


;;; Q: Fill in the following code fragment to complete the procedure
modular that allows us to make the binary modular operators. modular
should take the modulus and an ordinary arithmetic procedure and
produce a modular arithmetic procedure.

(define modular
  (lambda (modulus op) ; modulus is n and op is the operation we are
performing before taking the modulo
    (lambda (a1 a2)
      (modulo (op a1 a2) modulus))))

;; test cases
((modular 17 +) 13 11) ; confirmed = 7
((modular 17 -) 13 11) ; confirmed = 2
((modular 17 *) 13 11) ; confirmed = 7



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 2       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: What is the order of growth in tome of slow-exptmod? What is
its order of growth in space? Does slow-exptmod use an iterative
algorithm or a recursive algorithm?

(define (slow-exptmod n)
```

```
  (let ((*mod (modular n *)))
    (define (em a b)
      (if (= b 0)
          1
          (*mod a (em a (- b 1)))))
    em))

#|
We can see that the code is recursive as it has pending operations
when the next level is called. Namely, we need to recurse all the way
down to compute the base case and then go back up the levels to
compute (*mod a (em a (-b 1))).

Given that it stops the recursion when b=0, we see that order of
growth in time is O(b). Since this recursion has pending computations
at each level, it also has an order of growth of O(b) in space.
|#


;;; Q: Fill in the details of the procedure exptmod that computes
a^b(mod n) using repeated squaring.

(define (exptmod p)
  (let ((mod* (modular p *)))
    (define (square x)
      (mod* x x))
    (define (em base exponent)
      (cond
       ((= exponent 0) 1)
       ((even? exponent) (square (em base (/ exponent 2))))
       (else (mod* base (em base (- exponent 1))))))
    em))

;; test cases
((exptmod 10) 2 0) ; confirmed =  1
((exptmod 10) 2 3) ; confirmed = 8
((exptmod 10) 3 4) ; confirmed = 1
((exptmod 100) 2 15) ; confirmed = 68
((exptmod 100) -5 3) ; confirmed = 75


;;; Q: What is the order of growth in time of your implementation of
exptmod? What is its order of growth in space? Does your exptmod use
an iterative algorithm or a recursive algorithm?

#|
This function is still recursive as we call em inside of itself with
pending computations. It still needs to go to the base-case before
recursing back up to perform computations. BUT this procedure is
better than slow-exptmod.
```

```
If the exponent is 0, then it is the base case and we recurse up. If
the exponent is even, then we square and recurse with exponent/2.
Finally, if the exponent is odd we mutliply the base with the next
level. At every other level, we divide the exponent by 2, giving this
a total runtime of O(log exponent). Similar to above, we have to keep
track of each memory stack so we are using O(log exponent) in terms of
space.
|#



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 3       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: Start by writing a procedure random-k-digit-number that takes
an integer k>0 and returns a random k-digit number.

(define (random-k-digit-number k)
  (define (random-k-digit-helper random-number i) ; i tracks factor of
10 the number is on
    (if (= i k) ; if i = k, stop
        random-number
        (random-k-digit-helper (+ (* random-number 10) (random 10)) (+
i 1))))
  (random-k-digit-helper 0 0))

;; test cases
(random-k-digit-number 1) ; confirmed randomly generates one-digit
numbers
(random-k-digit-number 3) ; confirmed randomly generates three-digit
numbers
(random-k-digit-number 4) ; confirmed randomly generates four-digit
numbers


;;; Q: Write a procedure count-digits that takes an integer n>0 and
returns the number of digits in its decimal representation.

(define (count-digits n)
  (define (count-digits-helper i value)
    (if (< value 1) ; if value now a fraction
        i
        (count-digits-helper (+ i 1) (/ value 10))))
  (count-digits-helper 0 n))

;; test cases
(count-digits 3) ; confirmed = 1
(count-digits 2007) ; confirmed = 4
```

```
(count-digits 123456789) ; confirmed = 9


;;; Q: Use this approach to write a procedure big-random that takes an
integer n>0 and returns a random number from 0 to n-1.

(define (big-random n)
  (define (big-random-helper n)
    (let* ((k (count-digits n))
           (random-number (random-k-digit-number k)))
      (if (< random-number n)
          random-number
          (big-random-helper n))))
  (big-random-helper n))

;; test cases
(big-random 100) ; confirmed gets random values in [0, 99]
(big-random 1) ; confirmed always 0
(big-random (expt 10 40)) ; got
5070305110987601166565978816531780177O6,
12250944868457923767744381609091744733 71, ...



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 4       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: What is the order of growth in time of slow-prime? What is its
order of growth in space? Does slow-prime? use an iterative algorithm
or a recursive algorithm?

(define (slow-prime? n)
  (define (test-factors n k)
    (cond ((>= k n) #t)
          ((= (remainder n k) 0) #f)
          (else (test-factors n (+ k 1)))))
  (if (< n 2)
      #f
      (test-factors n 2)))

#|
This procedure is recursive and will make n-2 recursive calls. At each
call, it operates in O(1). Thus, its order of growth in time is O(n).
As there are no pending computations at each level, I would expect the
order of growth in space to be O(1).
|#


;;; Q: Proposal 1. "We only have to check factors less than or equal
```

to (sqrt n)" How would this affect the order of growth in time?

#|
This would reduce our order of growth to O((sqrt n)) as we would only check up to (sqrt n).
|#

;;; Q: Proposal 2. "We only have to check odd factors (and 2, as a special case)." How would this affect the order of growth in time?

#|
In this case, we only check every other number, or (n/2) - 2. This would still result in a O(n) order of growth in time.
|#

;;; Q: Test Fermat's Little Theorem using your exptmod procedure and a few suitable choices of a and p.

```
(define (test-fermats? a p)
  (let ((left ((exptmod p) a p))
        (right (modulo a p)))
    (if (= left right)
        #t
        #f)))

;; test cases
(test-fermats? 7 3) ; 3 is prime, expect #t, Value: #t
(test-fermats? 7 997) ; 997 is the largest 3-digit prime, Value: #t
(test-fermats? 10 997) ; 997 is a prime, Value: #t
```

;;; Q: Write a procedure prime? that uses this technique to test whether its parameter p is prime.

```
(define prime-test-iterations 20)

(define (prime? p)
  (define (prime?-helper i)
    (if (>= i prime-test-iterations)
        #t
        (let ((a (big-random p))) ; randomly generate a
          (if (not (test-fermats? a p))
              #f
              (prime?-helper (+ i 1))))))
  (if (= p 0)
      #f
      (prime?-helper 0)))
```

```
;; test cases
(prime? 2) ; confirmed = #t
(prime? 4) ; confirmed = #f
(prime? 1) ; confirmed = #t
(prime? 0) ; confirmed = #f
(prime? 200) ; Value: #f
(prime? 199) ; Value: #t


;;; Q: What is the order of growth in time of your implementation of
prime? What is its order of growth in space? Does prime? use an
iterative algorithm or a recursive algorithm?

#|
First, we are running this prime-test-iterations times--which is a
constant--so the runtime depends on the complexity of one iteration.

We generate a random number using p. Big random then relies on the
number of digits in p. We can see that the number of digits in p is
roughly log_{10}(p)+1 or O(log p). Then, the  prime? procedure relies
on test-fermats? which calls exptmod--also O(log p). Thus, at each
iteration there are a constant number of calls of O(log p) operations,
making the overall runtime O(log p). As there aren't any pending
computations that need to be tracked at each call, this procedure's
order of growth in space will be O(1).
|#



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 5       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: Write a procedure random-k-digit-prime that returns a random
prime number with about k digits.

(define random-k-digit-prime
  (lambda (k)
    (define (random-k-digit-prime-helper i)
      (if (> i (* 10 k)) ; only try up to 10k times, then will exit
          'failed
          (let ((p (random-k-digit-number k)))
            (if (prime? p)
                p
                (random-k-digit-prime-helper (+ i 1))))))
    (random-k-digit-prime-helper 0)))

;; test cases
;(random-k-digit-prime 1) ; generates numbers in  [1, 2, 3, 5, 7]
;(random-k-digit-prime 2) ; gives two digit primes
```

```
;(random-k-digit-prime 10) ; tested a few numbers in google, all
numbers returned were primes
;(count-digits (random-k-digit-prime 100)) ; takes a few seconds to
run, normally returns 100
;(count-digits (random-k-digit-prime 100))


;;; Q: In what ways can your random-prime procedure fail?

#|
I have my code set up such that it stops after a certain number of
attempts. This is helpful for generating really large k-digit primes.
It may enter an infinite loop of continually trying without me
noticing. Thus, this kind of failure is helpful. This procedure could
also fail by giving a non-prime number. In problem 4, we saw
Carmichael numbers may pass our prime test. This is an issue if we
select one of these numbers, as our effective Diffie-Hellman requires
that p is a true prime. If the procedure fails and gives a Carmichael
(i.e. not prime number), then our protocol will not work.
|#



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 6       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: Write a procedure ax+by=1 that solves for x and y using the
approach outlined above

(define ax+by=1
  (lambda (a b) ; note: we assume a>0 and b>0
    (cond
     ((not (= (gcd a b) 1)) 'failed) ; won't work if the gcd does not
equal 1
     ((= (remainder a b) 1) (list 1 (* -1 (quotient a b)))) ; base
case, return (1, -q)
     (else
      (let* ((q (quotient a b))
             (r (remainder a b))
             (result (ax+by=1 b r)))
        (list (cadr result) (- (car result) (* q (cadr
result))))))))) ; solve for (x, y) when result is giving (x', y')

;; test cases
(ax+by=1 17 13) ; confirmed = (-3 4)
(ax+by=1 7 3) ; confirmed = (1 -2)
(ax+by=1 10 27) ; confirmed = (-8 3)
(ax+by=1 10 20) ; testing returns failed gcd(a, b) != 1
```

```
;;; Q: Now write a procedure inversemod that finds the multiplicative
inverse of e modulo n, using ax+by=1.

(define (inversemod n)
  (lambda (e)
    (let ((result (ax+by=1 e n)))
      (if (eq? result 'failed)
          'failed ; if gcd from above != 1, this won't work
          (let ((d (car result)))
            (if (< d 0) ; we want the answer to be in [0, n-1], so add
n if result is negative to get correct mod
                (+ d n)
                d))))))

;; test cases
((inversemod 11) 5) ; confirmed = 9
((inversemod 11) 9) ; confirmed = 5
((inversemod 11) 7) ; confirmed = 8
((inversemod 12) 5) ; confirmed = 5
((inversemod 12) 8) ; confirmed failure (gcd)
((inversemod 101) (random-k-digit-prime 2)) ; returns different values
(random numbers)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 7       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Q: Write a procedure eg-send-message that takes a message string
and a receiver and calls the receiver's decryption procedure,
encrypted with the receiver's public key.

(define (eg-send-message message receiver)
  (let* ((public-key (eg-receiver-public-key receiver)) ; get values
from dh-system with utils
         (dh-system (eg-public-key-system public-key)) ; get values
from dh-system with utils
         (little-p (dh-system-prime dh-system)) ; using little-p and
big-P as errors happened with p and P
         (a (dh-system-primitive-root dh-system)) ; get values from
dh-system with utils
         (big-P (eg-public-key-number public-key)) ; get values from
dh-system with utils
         (T (big-random little-p)) ; after some research, it is best
if T<p
         (mod-expt (exptmod little-p)) ; define math we need to use
(for x, y)
         (mod-* (modular little-p *)) ; define math we need to use
```

```
(for y)
         (m (string->integer message)) ; turn message to an integer
         (x (mod-expt a T)) ; x=a^T(mod p)
         (y (mod-* m (mod-expt big-P T))) ; y=mP^T (mod p)
         (ciphertext (eg-make-ciphertext x y))) ; make ciphertext
    ((eg-receiver-decryption-procedure receiver) ciphertext))) ; pass
generated ciphertext

;; test cases (demonstrate your eg-send-message with a few short
strings sent from Ben to Alyssa)
(define dh-system (public-dh-system 100))
(define Alyssa (eg-receiver dh-system))
(eg-send-message "Hi there." Alyssa)
(eg-send-message "12345678901234567890123456789012345678901" Alyssa) ;
this works!
(eg-send-message "1234567890123456789012345678901234567890123"
Alyssa) ; this DOES NOT work!


;;; Q: What is the longest string you can send that will be correctly
decrypted with a 100 digit system? You will find that it is not too
long!

#|
In the last test case above, I used a string of numbers to find that a
100 digit system can only decrypt strings up to 41 characters (42 does
not work)!
|#



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;          PROBLEM 8       ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; Q: How can Eve make trouble in Ben and Alyssa's network?

(define (Eve receiver)
  (let ((receiver-public-key
         (eg-receiver-public-key receiver))
        (receiver-decryption-procedure
         (eg-receiver-decryption-procedure receiver)))
    (let ((my-spying-procedure
           (lambda (ciphertext)
             (write ciphertext)
             (newline)
             (receiver-decryption-procedure ciphertext))))
      (eg-make-receiver receiver-public-key
                        my-spying-procedure))))
```

```
;; test spying
(define Alyssa (Eve Alyssa))
(eg-send-message "Hi there." Alyssa) ; Eve can see the ciphertext!

#|
Even though Eve cannot decrypt the message, Eve could make trouble by
messing with the data being sent. Eve could intercept the message and
edit the ciphertext! This would lead to gibberish when being decrypted
and whatever data Ben and Alyssa are sharing would be lost/not
received. To do this, we can modify the ciphertext that is
intercepted.
|#


;;; Q: Modify the Eve program to make it possible for Eve to make
trouble in the relationship. Explain and demonstrate your nasty trick.

(define (Eve receiver)
  (let ((receiver-public-key
         (eg-receiver-public-key receiver))
        (receiver-decryption-procedure
         (eg-receiver-decryption-procedure receiver)))
    (let ((my-spying-procedure
           (lambda (ciphertext)
             (write ciphertext)
             (newline)
             (receiver-decryption-procedure (cons (+ (car ciphertext)
(random 100)) (+ (cdr ciphertext) (random 100)))))))) ; CHANGE IS HERE:
changed the ciphertext slightly to mess up the data when decrypted
using random numbers
      (eg-make-receiver receiver-public-key
                        my-spying-procedure))))

;; test attack method
(define Alyssa (Eve Alyssa))
(eg-send-message "Hi there." Alyssa) ; gibberish now: "\x87;!
Ùã3ëw\x9f;4±¨E\x86;\x10;®\x94;\x91;\x7f;a£R\x1;½Ð7:ïÄ÷\x12;E{\xc;
£É\x92;Úý|÷e\r"
```