Problem Set 2: RSA Encryption

Written Exercises

Proof by Induction

RSA encryption relies on modular exponentiation, which can be implemented in Scheme with the following code:

```
(define exptmod
  (lambda ((b <integer>) (e <integer>) (m <integer>))
    (cond ((zero? e) 1)
          ((even? e)
           (modulo (square (exptmod b (quotient e 2) m)) m))
          (else
           (modulo (* b (exptmod b (- e 1) m)) m)))))
```

Using mathematical induction and the Substitution Model, prove that:

`(exptmod b e m)` = modulo(b^e, m)

1) Claim

Let P(e) be the claim that (exptmod b e m) = modulo(b^e, m), where *b, e,* and *m* are integers and *m*>1.

2) Inductive Hypothesis

Assume that there exists a fixed *e,* where 0<*k*<*e*, (exptmod b k m) = modulo(b^k, m).

3) Base Case

For the base-case (exptmod b 0 m), use the substitution model to show that P(0) is true.

(exptmod b 0 m)
[{PROC} {b} {0} {m}]
(zero? {0})
Evaluates to #t, so evaluate second argument
{1}

Because (exptmod b 0 m) = modulo(b^0, m)=1, the base case holds true, and P(0) is true.

4) Inductive Step

By IH, **NOS P(e) is true,** where (exptmod b e m) = modulo(b^e, m).
There are two cases for e: e is even, or e is odd.
For the even case:

(exptmod b e m)
[{PROC} {b} {e} {m}]

(zero? {e})
Evaluates to #f, as e>0
(even? {e})
Evaluates to #t, as e is assumed to be even in this case
(modulo (square (exptmod {b} (quotient {e} 2) {m})) {m})
(exptmod {b} (quotient {e} 2) {m})
(exptmod {b} {e/2} {m})
By IH, evaluates to {modulo ($b^{e/2}$, m)}
(modulo (square {modulo ($b^{e/2}$, m)}) {m})
(modulo (* {modulo ($b^{e/2}$, m)} {modulo ($b^{e/2}$, m)}) {m})
(modulo (* {$b^{e/2}$} {$b^{e/2}$}) {m})
(* {$b^{e/2}$} {$b^{e/2}$})
Evaluates to b^e
[{modulo} {b^e} {m}]
= modulo(b^e, m)

For the odd case:
(exptmod b e m)
[{PROC} {b} {e} {m}]
(zero? {e})
Evaluates to #f, as e>0
(even? {e})
Evaluates to #f, as e is odd. Move to else statement evaluation
(modulo (* b (exptmod b (- e 1) m)) m)
(exptmod {b} (- e 1) {m})
(exptmod {b} {e-1} {m})
By IH, evaluates to modulo(b^{e-1}, m)
(modulo (* {b} {modulo(b^{e-1}, m)}) {m})
[modulo (* {b} {modulo(b^{e-1}, m)}) {m}]
Because modulo(q * modulo(p, m), m) = modulo(p * q, m), make same changes to above.
[{modulo} {b*b^{e-1}} {m}]
= modulo(b^e, m)
In both cases, (exptmod b e m) = modulo(b^e, m)

QED.

Programming Exercises

Problem 1: Unconverting a List

Procedure rsa-unconvert-list

```
(define rsa-unconvert-list
    (lambda ((lst <list>) (key <key>))
      (let ((n (key-modulus key)))
        (let loop ((rst lst) (prev 0))
          (cond ((null? rst) rst)
                (else
                 (let ((cur (- (rsa-transform (car rst) key) prev))
                       (sub (car rst)))
            (cons cur
                       (loop (cdr rst) sub)))))))))
```

OUTPUT for test-key-1

```
(rsa-unconvert-list result-1 (key-pair-private test-key-1))
```

(14949 14836 4205 13043 14817 13285)

```
(rsa-decrypt result-1 (key-pair-private test-key-1))
```

"test message"

OUTPUT for test-key-2

```
(define result-2
     (rsa-encrypt "test message" (key-pair-public test-key-2)))
(rsa-decrypt result-2 (key-pair-private test-key-2))
```

"test message"

OUTPUT for sample-1

```
(define sample-key-1 (generate-rsa-key-pair 16))
(define sample-1
     (rsa-encrypt "nathan kim" (key-pair-public sample-key-1)))
(rsa-decrypt sample-1 (key-pair-private sample-key-1))
```

"nathan kim"

OUTPUT for sample-2

```
(define sample-key-2 (generate-rsa-key-pair 16))
(define sample-2
     (rsa-encrypt "eddy liiiiiiiiiiiiiiiiiin yayyyyyyyyyyyyyyy"
(key-pair-public sample-key-2)))
(rsa-decrypt sample-2 (key-pair-private sample-key-2))
```

"eddy liiiiiiiiiiiiiiiiiin yayyyyyyyyyyyyyyy"

Problem 2: Digitally Signed Messages

Procedure encrypt-and-sign

```
(define encrypt-and-sign
  (lambda ((str <string>) (rpub <key>) (spri <key>))
    (let ((msg (rsa-encrypt str rpub)))
      (let ((sig (rsa-transform (compress msg spri) spri)))
    (make-signed-message msg sig)))))
```

OUTPUT for encrypt-and-sign test

```
(define result-2
     (encrypt-and-sign "Test message from User 1 to User 2"
                       (key-pair-public test-key-2)
                       (key-pair-private test-key-1)))
(signed-message-body result-2)
```

(141024 48674 204177 339709 317361 307215 225252 330769 134147 215314 540102 3551 396788 42331 536293 214006 171543)

```
(signed-message-signature result-2)
```

163298

Procedure decrypt-and-verify

```
(define decrypt-and-verify
  (lambda ((msg <signed-message>) (rpri <key>) (spub <key>))
    (let ((Dh (signed-message-signature msg))
          (h (compress (signed-message-body msg) spub)))
      (if (= (rsa-transform Dh spub) h)
          (rsa-decrypt (signed-message-body msg) rpri)
          #f))))
```

OUTPUT for decrypt-and-verify test

```
(decrypt-and-verify result-2
                    (key-pair-private test-key-2)
                    (key-pair-public test-key-1))
```

"Test message from User 1 to User 2"

```
(decrypt-and-verify result-2
                    (key-pair-private test-key-1)
                    (key-pair-public test-key-2))
```

#f

Problem 3: Cracking RSA

Procedure crack-rsa

```
(define crack-rsa
  (lambda ((pubkey <key>))
    (let* ((n (key-modulus pubkey))
           (e (key-exponent pubkey))
           (p (smallest-divisor n))
           (q (/ n p))
           (m (* (- p 1) (- q 1)))
           (a (car (euclid e m)))
           (d (modulo a m)))
      (make-key d n)
      )))
```

Output of the following calls:

```
(crack-rsa (make-key 1763 132731))
```

(42827 . 132731)

```
(crack-rsa (make-key 19883 159197))
```

(81347 . 159197)

```
(crack-rsa (make-key 25639 210281))
```

(168151 . 210281)

Problem 4: Identifying the Sender

Define check-sender-recipient procedure

```
(define check-sender-recipient
  (lambda ((msg <signed-message>) (lst <pair>))
    (let ((rpub (car lst))
          (spub (cdr lst)))
      (decrypt-and-verify msg (crack-rsa rpub) spub))))
```

From Vinit to Joyce

```
(check-sender-recipient secret-message (make-key-pair Vinit Joyce))
```

"If cryptography is outlawed, only outlaws will have cryptography."

Problem 5: Factoring Larger Moduli

Define pollard procedure

```
(define pollard
  (lambda ((n <integer>) (base <integer>) (max <integer>))
    (let loop ((k 1) (last-base base))
      (cond ((> k max) #f)
            (else
             (let* ((r (exptmod last-base k n))
                    (d (car (cdr (cdr (euclid (- r 1) n))))))
               (cond ((= d 1) (loop (+ k 1) r))
                     ((= d n) (loop (+ k 1) r))
                     (else d)))))))))
```

BIG Pollard OUTPUTS

```
(pollard 39173679876983836729 3 3000)
```

6699424463

```
(pollard 2278570691794489592002651 2 27000)
```

1274508504281

Extra Credit

Define crack-rsa-2 procedure

```
(define crack-rsa-2
  (lambda ((pubkey <key>) (base <integer>) (max <integer>))
    (let* ((n (key-modulus pubkey))
           (e (key-exponent pubkey))
           (p (pollard n base max)))
      (if (not p) #f
          (let*
              ((q (/ n p))
               (m (* (- p 1) (- q 1)))
               (a (car (euclid e m)))
               (d (modulo a m)))
            (make-key d n))))))
```

Procedure crack-rsa-2 OUTPUTS

```
(crack-rsa-2 (make-key 37 9709644929798597233) 2 15000)
```

(3673919700704867653 . 9709644929798597233)

```
(crack-rsa-2 (make-key 3841 15873420455170035847) 2 10000)
```

(1735703355329074561 . 15873420455170035847)

```
(crack-rsa-2 (make-key 17333 47126568369600710999) 2 10000)
```

#f

Code Appendix

```
;;; Set Language to Full Swindle
;;; ps2.scm
(require racket/base) ;;This allows the type system to work.
(require rnrs/mutable-pairs-6)
(define square
  (lambda ((n <number>))
    (* n n)))

;; ----- Data type definitions -----

;; Keys are implemented as simple pairs
(define <key> <pair>)

;; Constructors and accessors for RSA keys:
(define make-key
  (lambda ((exponent <integer>) (modulus <integer>))
    (cons exponent modulus)))

(define key-exponent car)
(define key-modulus  cdr)

;; A key pair is just a pair whose head is the public key and
;; whose tail is the private key
(define <key-pair> <pair>)

;; Constructor and accessors for key pairs
(define make-key-pair
  (lambda ((public <key>) (private <key>))
    (cons public private)))

(define key-pair-public  car)
(define key-pair-private cdr)

;; A message is a list of integers, the blocks of the message
(define <message> <list>)

;; A signed message consists of a message + a signature
(define <signed-message> <pair>)

;; Constructor and accessors for signed messages
(define make-signed-message
  (lambda ((message <message>) (signature <integer>))
```

```
    (cons message signature)))

(define signed-message-body car)
(define signed-message-signature cdr)

;; ----- The RSA transformation -----

(define rsa-transform
  (lambda ((number <integer>) (key <key>))
    (exptmod number
           (key-exponent key)
           (key-modulus key))))

(define rsa-convert-list
  (lambda ((lst <list>) (key <key>))
    (let ((n (key-modulus key)))
      (let loop ((rst lst) (prev 0))
     (cond ((null? rst) rst)
           (else
            (let ((cur (rsa-transform (modulo (+ (car rst) prev) n)
key)))
            (cons cur
                  (loop (cdr rst) cur)))))))))


(define rsa-encrypt
  (lambda ((msg <string>) (key <key>))
    (rsa-convert-list (string->intlist msg) key)))

(define rsa-decrypt
  (lambda ((msg <message>) (key <key>))
    (intlist->string (rsa-unconvert-list msg key))))

;; ----- RSA key generation functions -----

;; (choose-prime smallest range)
;; Find an odd prime by sequential search, beginning at a random
;; point within 'range' of 'smallest'

(define choose-prime
  (lambda ((smallest <integer>) (range <integer>))
    (let ((start (+ smallest (bigrand range))))
      (search-prime
```

```
       (if (even? start)
        (+ start 1)
        start)))))

;; Sequentially search for an odd prime beginning at 'guess'
;; Assumes 'guess' is odd
(define search-prime
  (lambda ((guess <integer>))
    (cond ((divisible-by-primes? guess)
         (search-prime (+ guess 2)))
        ((fast-prime? guess)
           guess)
          (else
           (search-prime (+ guess 2))))))

;; Uses Fermat's theorem to quickly test if a number is composite
;; Returns #f if 'candidate' is definitely composite, #t if it may
;; be prime.
(define fermat-test
  (lambda ((candidate <integer>) (witness <integer>))
    (= (exptmod witness candidate candidate) witness)))

;; Uses fermat-test to test whether a candidate is prime
;; Returns #f if candidate is definitely composite, #t if it may
;; be prime.
(define fast-prime?
  (lambda ((candidate <integer>))
    (and (fermat-test candidate 2)
         (fermat-test candidate 3)
         (fermat-test candidate 5)
         (fermat-test candidate 7))))

;; Test whether candidate is divisible by some small known primes.
;; This eliminates a lot of candidates quickly, and avoids the more
;; expensive fermat-test (this is important when you are generating
;; big keys!  It also helps avoid Carmichael numbers.

(define *small-primes* (list 2 3 5 7 11 13 17 23))

(define divisible-by-primes?
  (lambda ((candidate <integer>))
    (let loop ((primes *small-primes*))
      (cond ((null? primes) #f)
```

```
          ((divides? (car primes) candidate) (car primes))
          (else
           (loop (cdr primes)))))))

;; Generate an RSA key pair with a modulus of 'bits' significant bits.
(define generate-rsa-key-pair
  (lambda ((bits <integer>))
    (let* ((base (expt 2 (quotient bits 2)))
           (p (choose-prime base base))
           (q (choose-prime base base)))
      (if (= p q)   ; Try again if we choose the same prime twice
          (generate-rsa-key-pair bits)
          (let* ((n (* p q))        ; key modulus
                 (m (- n p q -1))   ; m = (p - 1)(q - 1) = n - p - q +
1
                 (z (select-exponents m)) ; choose e and d
                 (e (car z))
                 (d (cdr z)))
            (make-key-pair (make-key e n)
                           (make-key d n)))))))

;; Choose a random encryption exponent and compute the corresponding
;; decryption exponent, given phi(n) = (p - 1)(q - 1)
(define select-exponents
  (lambda ((m <integer>))
    (let* ((e (random (expt 2 15)))
           (z (euclid e m))
           (g (caddr z)))
      (if (= g 1)
          (cons e (modulo (car z) m))
          (select-exponents m)))))  ; try again if e doesn't work

;; ----- Encoding and decoding messages -----
;;
;; These procedures convert between strings and lists of integers in
;; the range 0 .. 2^14.  You don't need to study this code; you can
;; just use it as is.

(define (make-string-padder size)
  (lambda ((str <string>))
    (let ((rem (modulo (string-length str) size)))
      (if (zero? rem)
       str
```

```
        (string-append str (make-string (- size rem)))))))

(define (make-string-to-intlist-func bits)
  (let* ((size (quotient (+ bits 6) 7))
         (pad (make-string-padder size)))
    (lambda ((str <string>))
      (let loop ((in (map char->integer (string->list (pad str))))
            (out '()))
     (cond ((null? in) (reverse out))
           (else
            (let-values (((firstk rest) (first-k in size)))
             (let ((value
                  (let accum ((tot 0) (lst firstk))
                    (cond ((null? lst) tot)
                        (else
                         (accum (+ (* tot 128) (car lst))
                              (cdr lst)))))))
               (loop rest (cons value out)))))))))))

(define (make-intlist-to-string-func bits)
  (define (split-val val size)
    (cond ((zero? size) '())
        (else
         (cons (modulo val 128)
             (split-val (quotient val 128)
                      (- size 1))))))
  (let* ((size (quotient (+ bits 6) 7)))
    (lambda ((lst <list>))
      (let loop ((in lst) (out '()))
     (cond ((null? in)
            (list->string
           (map integer->char
                (filter (lambda (c) (not (zero? c))) (reverse out)))))
           (else
            (loop (cdr in)
                (append (split-val (car in) size) out)))))))))

;; Convert a string to a list of numbers
(define string->intlist (make-string-to-intlist-func 14))

;; Convert a list of numbers to a string
(define intlist->string (make-intlist-to-string-func 14))
```

```
;; Returns two values, the first k elements of the list, and the rest
;; of the list
(define (first-k lst k)
  (let loop ((n k) (out '()) (rst lst))
    (cond ((or (null? rst)
               (zero? n))
           (values (reverse out) rst))
          (else
           (loop (- n 1)
              (cons (car rst) out)
              (cdr rst))))))

;; ----- Useful mathematical functions -----

;; (bigrand value) - returns a pseudorandom number in the range
;; 0 .. value-1.
(define bigrand
  (lambda ((value <integer>))
    (let loop ((cur 0))
      (cond ((> cur value)
             (modulo cur value))
            (else
             (loop (+ (* cur 256) (random 256)))))))))

;; (divides? a m)
;; Returns true iff a divides m
(define divides?
  (lambda ((a <integer>) (m <integer>))
    (zero? (modulo m a))))

;; (exptmod b e m)
;; Compute b to the e power, modulo m (i.e., b^e (mod m))
(define exptmod
  (lambda ((b <integer>) (e <integer>) (m <integer>))
    (cond ((zero? e) 1)
          ((even? e)
           (modulo (square (exptmod b (quotient e 2) m)) m))
          (else
           (modulo (* b (exptmod b (- e 1) m)) m)))))

;; (euclid a m)
;; Computes the greatest common divisor of a and m using
;; an extension of Euclid's algorithm.  Returns a list
```

```
;; (s t g) where g = (a, m) and s, t are constants satisfying
;; Bezout's identity (sa + tm = g)
(define euclid
  (lambda ((a <integer>) (m <integer>))
    (cond ((zero? m) (list 1 0 a))
          (else
           (let* ((q (quotient a m))
                  (r (modulo a m))
                  (z (euclid m r))
                  (u (car z))
                  (v (cadr z))
                  (g (caddr z)))
             (list v (- u (* q v)) g))))))

;; (smallest-divisor n)
;; Finds the smallest divisor d > 1 of n by trial division

(define smallest-divisor
  (lambda ((n <integer>))
    (if (even? n)
     2
     (let ((sqrt-n (floor (sqrt n))))
       (let loop ((try 3))
         (cond ((> try sqrt-n) n)        ; stop if we get to sqrt(n)
            ((divides? try n) try)    ; stop if we find a divisor
            (else
             (loop (+ try 2)))))))))  ; try the next odd divisor

;; ----- Compression (hash) function -----

;; Returns a compression function for values up to the specified size
;;
;; Each value is squared and multiplied by the index of its position
;; in the sequence, modulo a suitable prime.  The output is the
;; modular sum of these over all elements of the sequence.

(define (make-compression-func bits)
  ;; Find the first prime less than or equal to start (assumed odd)
  (define (find-modulus start)
    (cond ((or (divisible-by-primes? start)
               (not (fast-prime? start)))
           (find-modulus (- start 2)))
          (else start)))
```

```
  (let* ((start (- (expt 2 bits) 1))
       (modulus (find-modulus start)))

    (lambda ((lst <list>) (key <key>))
      (define (nexthash prev x i m)
      (modulo (+ prev (* x x i) i) m))

      (let loop ((index 1) (hash 0) (rst lst))
      (cond ((null? rst) (modulo hash (key-modulus key)))
            (else
             (loop (+ index 1)
                 (nexthash hash (car rst) index modulus)
                 (cdr rst))))))))

;; A compression function suitable for 18-bit integers
(define compress (make-compression-func 18));


;; ----- Test data and sample keys -----

;; Public keys
(define Anna       (make-key 32455 108371))
(define Graham     (make-key 30529 216221))
(define Vinit      (make-key 28859 135659))
(define Joyce      (make-key 28559 171967))
(define Caspar     (make-key 27089 89701))

;; Key pairs for testing purposes
(define test-key-1
  (make-key-pair
   (make-key 2587 573193)
   (make-key 28947 573193)))

(define test-key-2
  (make-key-pair
   (make-key 13889 557983)
   (make-key 269249 557983)))

(define big-key-1
  (make-key-pair
   (make-key 19307 43565950551638075623)
   (make-key 35440349052136018387 43565950551638075623)))
```

```
(define big-key-2
  (make-key-pair
   (make-key 22059 30199096579027135423)
   (make-key 22743442190663818579 30199096579027135423)))

(define secret-message
  (make-signed-message
   '(51962 51267 90 7772 107084 58028 83572 123964
         112290 9595 44541 124077 116589 35134 106672
         39495 135016 97906 100462 129026 118420 7092
         6383 89215 133924 59099 123979 17539 31512 82548
         31291 88212 29978)
   6242))

;;
------------------------------------------------------------------------
--

;; Test messages
(define result-1
     (rsa-encrypt "test message" (key-pair-public test-key-1)))

;; here there be answers

;; Problem 1
(define rsa-unconvert-list
    (lambda ((lst <list>) (key <key>))
      (let ((n (key-modulus key)))
        (let loop ((rst lst) (prev 0))
          (cond ((null? rst) rst)
                (else
                 (let ((cur (modulo (- (rsa-transform (car rst) key)
prev) n))
                       (sub (car rst)))
            (cons cur
                       (loop (cdr rst) sub)))))))))

;; testing
(rsa-unconvert-list result-1 (key-pair-private test-key-1))
; (14949 14836 4205 13043 14817 13285)
"uncovert OUTPUT-------------------------------"
(rsa-decrypt result-1 (key-pair-private test-key-1))
```

```
; "test message"

(define sample-key-1 (generate-rsa-key-pair 16))
(define sample-1
     (rsa-encrypt "nathan kim!!!!!!!" (key-pair-public sample-key-1)))
(rsa-decrypt sample-1 (key-pair-private sample-key-1))

(define sample-key-2 (generate-rsa-key-pair 16))
(define sample-2
     (rsa-encrypt "eddy liiiiiiiiiiiiiiiiiin yayyyyyyyyyyyyyyy"
(key-pair-public sample-key-2)))
(rsa-decrypt sample-2 (key-pair-private sample-key-2))

;; Problem 2
(define encrypt-and-sign
  (lambda ((str <string>) (rpub <key>) (spri <key>))
    (let ((msg (rsa-encrypt str rpub)))
      (let ((sig (rsa-transform (compress msg rpub) spri)))
    (make-signed-message msg sig)))))

"signed message OUTPUT--------------------------------"
(define result-2
     (encrypt-and-sign "Test message from User 1 to User 2"
                       (key-pair-public test-key-2)
                       (key-pair-private test-key-1)))
(signed-message-body result-2)
(signed-message-signature result-2)

(define decrypt-and-verify
  (lambda ((msg <signed-message>) (rpri <key>) (spub <key>))
    (let ((Dh (signed-message-signature msg))
          (h (compress (signed-message-body msg) spub)))
      (if (= (rsa-transform Dh spub) h)
          (rsa-decrypt (signed-message-body msg) rpri)
          #f))))

"decrypt-and-verify OUTPUT--------------------------------"
(decrypt-and-verify result-2
                    (key-pair-private test-key-2)
                    (key-pair-public test-key-1))

(decrypt-and-verify result-2
                    (key-pair-private test-key-1)
```

```
                    (key-pair-public test-key-2))

;; Problem 3
(define crack-rsa
  (lambda ((pubkey <key>))
    (let* ((n (key-modulus pubkey))
           (e (key-exponent pubkey))
           (p (smallest-divisor n))
           (q (/ n p))
           (m (* (- p 1) (- q 1)))
           (a (car (euclid e m)))
           (d (modulo a m)))
      (make-key d n)
      )))

"crack-rsa OUTPUT------------------------------"
(crack-rsa (key-pair-public test-key-1))
(crack-rsa (key-pair-public test-key-2))

(crack-rsa (make-key 1763 132731))
(crack-rsa (make-key 19883 159197))
(crack-rsa (make-key 25639 210281))

;; Problem 4
(define check-sender-recipient
  (lambda ((msg <signed-message>) (lst <pair>))
    (let ((rpub (car lst))
           (spub (cdr lst)))
      (decrypt-and-verify msg (crack-rsa rpub) spub))))
#|
;; Test check-sender-recipient procedure
(check-sender-recipient result-2
                    (make-key-pair (key-pair-public test-key-2)
                    (key-pair-public test-key-1)))

"Anna"
(check-sender-recipient secret-message (make-key-pair Anna Graham))
(check-sender-recipient secret-message (make-key-pair Anna Vinit))
(check-sender-recipient secret-message (make-key-pair Anna Joyce))
(check-sender-recipient secret-message (make-key-pair Anna Caspar))

"Vinit"
(check-sender-recipient secret-message (make-key-pair Vinit Anna))
```

```
(check-sender-recipient secret-message (make-key-pair Vinit Graham))
(check-sender-recipient secret-message (make-key-pair Vinit Joyce))
(check-sender-recipient secret-message (make-key-pair Vinit Caspar))

"Joyce"
(check-sender-recipient secret-message (make-key-pair Joyce Anna))
(check-sender-recipient secret-message (make-key-pair Joyce Graham))
(check-sender-recipient secret-message (make-key-pair Joyce Vinit))
(check-sender-recipient secret-message (make-key-pair Joyce Caspar))

"Caspar"
(check-sender-recipient secret-message (make-key-pair Caspar Anna))
(check-sender-recipient secret-message (make-key-pair Caspar Graham))
(check-sender-recipient secret-message (make-key-pair Caspar Vinit))
(check-sender-recipient secret-message (make-key-pair Caspar Joyce))

|#
"Secret message from Vinit to Joyce
OUTPUT--------------------------------"
(check-sender-recipient secret-message (make-key-pair Vinit Joyce))

;; (identify-sender secret-message (list Anna Graham Vinit Joyce
Caspar))

;; Problem 5
(define pollard
  (lambda ((n <integer>) (base <integer>) (max <integer>))
    (let loop ((k 1) (last-base base))
      (cond ((> k max) #f)
            (else
             (let* ((r (exptmod last-base k n))
                    (d (car (cdr (cdr (euclid (- r 1) n))))))
               (cond ((= d 1) (loop (+ k 1) r))
                     ((= d n) (loop (+ k 1) r))
                     (else d)))))))))

#|
"Pollard test OUTPUT--------------------------------"
(pollard 29958343 2 20)
(pollard 13549648109 3 150)
(pollard 5157437 2 7)
(pollard 5157437 2 9)
(pollard 34297190193172563733 3 35000)
```

```
|#

"BIG Pollard OUTPUTS"
(pollard 39173679876983836729 3 3000)
(pollard 22785706917944895920026 51 2 27000)

;; Extra Credit
(define crack-rsa-2
  (lambda ((pubkey <key>) (base <integer>) (max <integer>))
    (let* ((n (key-modulus pubkey))
           (e (key-exponent pubkey))
           (p (pollard n base max)))
      (if (not p) #f
          (let*
              ((q (/ n p))
               (m (* (- p 1) (- q 1)))
               (a (car (euclid e m)))
               (d (modulo a m)))
            (make-key d n))))))
#|
"crack-rsa-2 test OUTPUT--------------------------------"
(crack-rsa-2 (key-pair-public big-key-1) 2 10000)
(crack-rsa-2 (key-pair-public big-key-2) 2 10000)
(crack-rsa-2 (key-pair-public big-key-2) 2 20000)
|#

"crack-rsa-2 OUTPUT--------------------------------"
(crack-rsa-2 (make-key 37 9709644929798597233) 2 15000)
(crack-rsa-2 (make-key 3841 15873420455170035847) 2 10000)
(crack-rsa-2 (make-key 17333 47126568369600710999) 2 10000)
```