6.945 Problem Set 0: Diffie-Hellman

Richard Futrell
futrell@mit.edu

February 11, 2014

1 Problem 1

These test cases distinguish the procedure `modulo` from the procedure `remainder`:

```
(modulo 13 8)
;Value: 5

(remainder 13 8)
;Value: 5

(modulo -13 8)
;Value: 3

(remainder -13 8)
;Value: -5

(modulo -13 -8)
;Value: -5

(remainder -13 -8)
;Value: -5
```

`modulo` and `remainder` differ in their value for -13, 8; `modulo` is 3 while `remainder` is -5. `remainder` produces negative values for `(remainder -13 8)`, so it cannot be the `modulo` function we want, which must produce a value b such that $0 \leq b \leq modulus$. On the other hand, both `(modulo -13`

`-8)` and `(remainder -13 -8)` produce -5; the negative value for `modulo` is correct because it can be the value of b in $-13 = -8a + b$ for $a = 1$.

Now we will make `+mod`, `-mod`, etc., by creating a function which takes a procedure and makes it modular in the last argument. Then we make `+mod` by passing `+` to that function.

```
(define (proc-mod proc)
  (lambda (a b n)
    (modulo (proc a b) n)))

(define +mod (proc-mod +))
(define -mod (proc-mod -))
(define *mod (proc-mod *))

(+mod 7 5 8)
;Value: 4

(+mod 10 10 3)
;Value: 2

(-mod 5 12 2)
;Value: 1

(*mod 6 6 9)
;Value 0

(+mod 99 99 100)
;Value 98

(*mod 50 -3 100)
;Value 50

(+mod 0 0 1)
;Value: 0
```

Alternatively we can make a procedure `modular` which will take a procedure and make it into that procedure modulo p.

```
(define modular
  (lambda (modulus op)
    (lambda (a1 a2)
      (modulo (op a1 a2) modulus))))

#| Tests

((modular 8 +) 7 5)
;Value: 4

((modular 17 +) 13 11)
;Value: 7

((modular 17 -) 13 11)
;Value: 2

((modular 17 *) 13 11)
;Value: 7

((modular 3 /) 70 10)
;Value: 1

((modular 1 + ) 1 1)
;Value: 0
```

2 Problem 2

The procedure `slow-exptmod` which we are given is a linear recursive algorithm which takes $O(N)$ time and $O(N)$ space for an exponent N. We can instead do modular exponentiation by repeated squaring. This algorithm is recursive and takes $O(logN)$ space time and space.

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

#| Tests

((exptmod 10) 2 0)
;Value: 1

((exptmod 10) 2 3)
;Value: 8

((exptmod 10) 3 4)
;Value: 1

((exptmod 100) 2 15)
;Value: 68

((exptmod 100) -5 3)
;Value: 75

((exptmod 10) 2 5)
;Value: 2

|#
```

3 Problem 3

Here is the procedure to generate a random k (or less) digit number.

```
(define (random-k-digit-number k)
  (define (iter number-so-far k)
    (if (<= k 0) number-so-far
        (iter (+ (random 10) (* 10 number-so-far))
              (- k 1))))
  (if (and (integer? k) (> k 0))
      (iter 0 k)
      (error ``k must be an integer >= 1''))))

#| Tests

(random-k-digit-number 1)
;Value: 3

(random-k-digit-number 3)
;Value: 238

(random-k-digit-number 3)
;Value 358

(random-k-digit-number 50)
;Value: 57169176749835956664726017294555066353372987885812

;;; More extensive tests will come later once we have some auxiliary
;;; functions.

|#
```

Here is a procedure to count digits:

```
;;; Here is a procedure to count digits:
(define (count-digits integer)
  (define (iter integer count)
    (let ((integer-divided (/ integer 10)))
      (if (< integer-divided 1)
          count
          (iter integer-divided (+ count 1)))))
  (if (and (integer? integer) (>= integer 0))
      (iter integer 1)
      ;; Don't know how to count digits of negatives or non-integers:
      ;; should the decimal point and minus sign count? So give an
error.
      (error "The argument of count-digits must be an integer >=0.")))

#| Tests

(count-digits 3)
;Value: 1

(count-digits 2007)
;Value: 4

(count-digits 123456789)
;Value: 9

(count-digits 0)
;Value: 1

|#
```

Now that we have `count-digits`, we can do a more rigorous test of `random-k-digit-number`:

```
(define (test-random-k-digit-number k n)
  ;; Return whether random-k-digit-number is working correctly.
  (define (iter n-so-far truth-so-far)
    (if (= n-so-far n)
        truth-so-far
        (and (<= (count-digits (random-k-digit-number k))
                 k)
             (+ n-so-far 1))))
  (iter n #t))

#| Test

(test-random-k-digit-number 100 10000)
;Value: #t

|#
```

With these functions in hand, we can now make our big random number generator:

```
(define (big-random n)
  (let ((result (random-k-digit-number (count-digits n))))
    (if (< result n)
        result
        (big-random n))))

;;; Also some functions that will come handy in testing it:
(define (repeat-n-times proc n)
  ;; Get a list of results of a function called n times.
  (if (<= n 0)
      '()
      (cons (proc) (repeat-n-times proc (- n 1)))))

(define (all xs)
  ;; Check whether all elements of a list are true.
  (if (null? xs)
       #t
       (if (car xs)
           (all (cdr xs))
           #f)))

#| Tests

(big-random 100)
;Value: 81

(big-random 100)
;Value: 53

(big-random 1)
;Value: 0

(big-random 1)
;Value: 0
```

```
(all '(#t #t #t))
;Value: #t

(all '(#t #t #f))
;Value: #f

(length (repeat-n-times (lambda () 1) 10))
;Value: 10

(all
 (repeat-n-times
  (lambda ()
    (eq? (big-random 1) 0))
  1000))
;Value: #t

(big-random (expt 10 40))
;Value: 1775734406229631205199583597404723076053

|#
```

4 Problem 4

We are given the procedure `slow-prime` which iterates through all factors 2 through N and checks if N is divisible by them, so it takes $O(N)$ time. It is an iterative algorithm that only takes $O(1)$ space. Ben Bitdiddle's first optimization, sweeping only though potential factors 2 through the square root of N, reduces growth in time to $O(\sqrt{N})$. His second optimization, only checking odd factors, means we take half as much time, which is a constant factor so it does not change the asymptotic growth of $O(\sqrt{N})$.

We can detect primality instead using Fermat's Little Theorem. Here is a procedure to test Fermat's little theorem for integer a and potential prime p:

```
(define (test-fermats-little-theorem a p)
  (= ((exptmod p) a p) (modulo a p)))
```

```
#| Tests

(test-fermats-little-theorem 3 2)
;Value: #t

(test-fermats-little-theorem 2 2)
;Value: #t

(test-fermats-little-theorem 100 41)
;Value: #t

(test-fermats-little-theorem 100 40)
;Value: #f

(test-fermats-little-theorem 2 42)
;Value: #f

(test-fermats-little-theorem 25 43)
;Value: #t

(test-fermats-little-theorem 1152523 45)
;Value: #t

(test-fermats-little-theorem -1152523 47)
;Value: #t

|#
```

With that in hand, we can now make the procedure `prime?`:

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

(define (prime? p)
  (if (and (integer? p) (> p 1))
      (all (repeat-n-times
            (lambda () (test-fermats-little-theorem
                         (big-random (- p 1))
```

```
                        p))
            prime-test-iterations))
      #f))

#| Tests
(prime? 2)
;Value: #t

(prime? 4)
;Value: #f

(prime? 1)
;Value: #f

(prime? 0)
;Value: #f

(prime? 200)
;Value: #f

(prime? 199)
;Value: #t

(prime? -3)
;Value: #f

(prime? "hello")
;Value: #f

(prime? 561) ; A Carmichael number
;Value: #t
```

The procedure `prime?` carries out a constant number of iterations of `test-fermats-little-theorem`, which itself performs an `exptmod`, which we previously determined is recursive and takes $O(logN)$ time and space. So `prime?` takes $O(MlogN)$ steps where M is the number of calls to `test-fermats-little-theorem`. It is iterative but makes calls to a recursive algorithm.

5 Problem 5

We are now ready to make a procedure to produce a random k-digit prime. It is a generate-and-test algorithm.

```
(define random-k-digit-prime
  (lambda (k)
    (let ((number (random-k-digit-number k)))
      (if (prime? number)
          number
          (random-k-digit-prime k)))))

#| Tests

(random-k-digit-prime 1)
;Value: 2

(random-k-digit-prime 1)
;Value: 5

(random-k-digit-prime 2)
;Value: 5

(random-k-digit-prime 2)
;Value: 41

(count-digits (random-k-digit-prime 100))
;Value: 100

(count-digits (random-k-digit-prime 100))
;Value: 100

(all
 (repeat-n-times
  (lambda ()
    (let ((k (random-k-digit-number 2)))
      (<= (count-digits (random-k-digit-prime k)) k)))
  20))
```

```
;Value: #t

|#
```

The algorithm can fail by generating a non-prime number, since the test prime? is probabilistic and Fermat's Little Theorem might introduce false positives for the randomly selected values of a, especially for Carmichael numbers, though these occurences will be rare. It can also generate number which is less than k digits, because the leftmost digit might have been generated as 0. This is made more likely by the fact that primes are denser for smaller values of k.

6 Problem 6

Now we want to calculate a modular multiplicative inverse, an integer d such that $ed = 1 \pmod{n}$. We do this by solving $ed + nk = 1$ for e and n, and throwing out n. We solve this equation recursively: if a is divisible by b, then $x = 1$ and $y = -\frac{a}{b}$. Otherwise solve $bx + ry = 1$, where r is the remainder of a/b, and use the x' and y' from the solution to return y' and $x' - qy'$, where q is the quotient of a/b.

```
(define (ax+by=1 a b)
  ;; Get an ordered pair of values for x and y such that ax+by=1.
  (if (= (gcd a b) 1)
      (let ((q (quotient a b)) (r (remainder a b)))
        (if (= r 1)
            (list 1 (- q))
            (let ((xy (ax+by=1 b r)))
              (let ((x-prime (car xy)) (y-prime (cadr xy)))
                (list y-prime (- x-prime (* q y-prime)))))))
      (error "a and b must have gcd 1")))

;;; To facilitate testing, here is a function to verify the output of
;;; ax+by=1.

(define (test-ax+by=1 a b)
  ;; Return whether the output of ax+by=1 is correct.
  (let ((results (ax+by=1 a b)))
    (let ((x (car results))
          (y (cadr results)))
      (= 1 (+ (* a x) (* b y))))))

#| Tests
(ax+by=1 17 13)
;Value: (-3 4)

(ax+by=1 7 3)
;Value: (1 -2)

(ax+by=1 10 27)
;Value: (-8 3)

(test-ax+by=1 16 17)
;Value: #t

(test-ax+by=1 50 51)
;Value: #t

|#
```

Now with `ax+by=1` in hand, we can get the modular inverse by solving $ax + by = 1$ and taking x (modulo b) and throwing out y. It is necessary to take the result modulo b because `ax+by=1` produces negative integer results.

```
(define (inversemod n)
  (lambda (e)
    (if (= (gcd e n) 1)
        (let ((result (car (ax+by=1 e n))))
          (modulo result n))
        (error ‘‘a and b must have gcd 1’’))))

(define (test-inversemod n x)
  (let ((result ((inversemod n) x)))
    (= 1 ((modular n *) x result))))

#| Tests
```

```
((inversemod 11) 5)
;Value: 9

((inversemod 11) 9)
;Value: 5

((inversemod 11) 7)
;Value: 8

((inversemod 12) 5)
;Value: 5

;((inversemod 12) 8)
;error

(test-inversemod 101 (random-k-digit-prime 2))
;Value: #t

;(test-inversemod (random-k-digit-number 2) (random-k-digit-prime 3))
;Value: #t
; or error
```

7 Problem 7

Here is the ElGamal message sending procedure:

```
(define (eg-send-message message receiver)
  (let ((receiver-public-key (eg-receiver-public-key receiver))
        (receiver-decryption-procedure
         (eg-receiver-decryption-procedure receiver)))
    (let ((receiver-system (eg-public-key-system receiver-public-key))
          (receiver-number (eg-public-key-number
                            receiver-public-key)))
      (let ((k (dh-system-size receiver-system))
            (p (dh-system-prime receiver-system))
            (a (dh-system-primitive-root receiver-system)))
        (let ((my-secret (random-k-digit-number k))
               (mod-expt (exptmod p))
               (mod-* (modular p *)))
           (let ((x (mod-expt a my-secret))
                 (y (mod-* (string->integer message)
                           (mod-expt receiver-number my-secret))))
             (let ((ciphertext (eg-make-ciphertext x y)))
               (receiver-decryption-procedure ciphertext)))))))

;;; And here is a test procedure.
(define (test-eg-send-message message receiver)
  (= message (eg-send-message message receiver)))

#| Tests

(define dh-system (public-dh-system 100))
(define Alyssa (eg-receiver dh-system))
(eg-send-message "Hi there." Alyssa)
;Value: "Hi there."

(eg-send-message "" Alyssa)
;Value: ""

(eg-send-message "12345678901234567890123456789012345678901" Alyssa)
;Value: "12345678901234567890123456789012345678901"

(eg-send-message "123456789012345678901234567890123456789012" Alyssa)
;Value:
"Px\365\"\360J4\r\263\235\251\200\331\000\216\251v49Dc\214\375c'N\334E\254\251\001
0o@\237S\004"

|#
```

As the last test shows, the longest string that can be sent is 41 characters.

8 Problem 8

To do a man-in-the-middle attack, I define Effective-Eve as a fake receiver that purports to be Alyssa, but actually intercepts messages, decrypts them, and then re-encrypts them and forwards them to the real Alyssa. Ben thinks he is sending messages to Alyssa and Alyssa thinks she is getting messages from Ben but in fact both are communicating with Effective Eve.

Of course, this only works if Effective Eve binds to Alyssa before Ben knows Alyssa's public key. Effective Eve replaces Alyssa's public key with her own. Similarly if there were an authentication step whereby Alyssa can tell that messages are really coming from Ben, then Effective Eve would be hosed.

```
(define (Effective-Eve receiver)
  ;; This is a fake receiver whose decryption procedure is to
  ;; decrypt the message, print it, then reencrypt it and pass it on
  ;; to the real receiver. This is possible since there is no
  ;; authentication.
  (let ((receiver-public-key (eg-receiver-public-key receiver)))
    (let ((receiver-system (eg-public-key-system receiver-public-key))
          (receiver-number (eg-public-key-number
                            receiver-public-key)))
      (let ((k (dh-system-size receiver-system))
            (p (dh-system-prime receiver-system))
            (a (dh-system-primitive-root receiver-system)))
        (let ((my-secret (random-k-digit-number k))
              (mod-expt (exptmod p))
              (mod-* (modular p *))
              (mod-inv (inversemod p)))
          (let ((advertised-number
                 (mod-expt (dh-system-primitive-root dh-system)
                           my-secret)))
            (let ((fake-public-key
                   (eg-make-public-key dh-system advertised-number))
                  (sneaky-decryption-procedure
                   (lambda (ciphertext)
                     (let ((x (eg-ciphertext-x ciphertext))
                           (y (eg-ciphertext-y ciphertext)))
                       (let ((m (mod-* y (mod-inv (mod-expt x
                                                            my-secret)))))
                         (let ((cleartext (integer->string m)))
                           (write cleartext)
```

```
                                (newline)
                                (eg-send-message cleartext receiver)))))))
                 (eg-make-receiver fake-public-key sneaky-decryption-procedure))))))

#| Tests

(define dh-system (public-dh-system 100))
(define Alyssa (eg-receiver dh-system))
(define Alyssa (Effective-Eve Alyssa))
(eg-send-message "Don't let Eve see this!" Alyssa)
"Don't let Eve see this!"
;Value: "Don't let Eve see this!"

|#
```