National University of Singapore
School of Computing
CS1101S: Programming Methodology (Scheme)
Semester I, 2011/2012

Mission 8: **Tome Theory**

Start date: 2 September 2011
Due: 9 September 2011, 23:59

Readings:

- SICP, Section 2.1 & 2.2
- Concrete Abstractions, Chap. 7

As you were training in the temple, the Grandmaster suddenly made an urgent announcement for some members of the Jedi to gather. It turns out that JFDI has intercepted a message destined for Darth, our arch-enemy. Unfortunately, this message is encrypted. While unable to read the message content, the Grandmaster has at least determined that the encryption technique used is an ancient algorithm known as the RSA.

To break the encryption, the Grandmaster needs all the help he can get. Your mission, therefore, is to get up to speed to assist him in this endeavor.

Turning to the library for reference, you come across the following in a preserved tome:

The RSA System

People have been using secret codes for thousands of years; for this reason it is surprising that in 1976, Whitfield Diffie and Martin Hellman at Stanford University discovered a major new conceptual approach to encryption and decryption: *public-key cryptography.*[1]

Cryptography systems are typically based on the notion of using *keys* for encryption and decryption. An *encryption key* specifies the method for converting the original message into an encoded form. A corresponding *decryption key* describes how to undo the encoding. In traditional cryptographic systems, the decryption key is identical to the encryption key, or can be readily derived from it. As a consequence, if you know how to *encrypt* messages with a particular key then you can easily *decrypt* messages that were encrypted with that key.

Diffie and Hellman's insight was to realize that there are cryptographic systems for which knowing the encryption key gives no help in decrypting messages; that is, for which there is no practical way to derive the decryption key

[1]W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, IT-22:6, 1976, pp 644–654.

from the encryption key. This is of immense practical importance. In traditional cryptographic systems, someone can send you coded messages only if the two of you share a secret key. Since anyone who learns that key would be able to decrypt the messages, keys must be carefully guarded and transmitted only under tight security. In Diffie and Hellman's system, you can tell your *encryption* key to anyone who wants to send you messages, and not worry about key security at all. For even if everyone in the world knew your encryption key, no one could decrypt messages sent to you without knowing your *decryption key*, which you keep private to yourself. Diffie and Hellman called such a system a *public-key* cryptography system.

A few months after Diffie and Hellman announced their idea, Ronald Rivest, Adi Shamir, and Leonard Adelman at MIT discovered a workable method for implementing it. This *RSA cryptography system* has remained the most popular technique for public-key cryptography.

The theory behind RSA

RSA uses integers to represent groups of characters[2] and uses special functions that transform integers to integers.

In the RSA scheme, you select two large prime numbers, p and q. You then define

$$\begin{aligned} n &= pq && (1) \\ m &= (p-1)(q-1). && (2) \end{aligned}$$

You also select a number e, such that $\gcd(e, m) = 1$. Your *public key*, which you can advertise to the world, is the pair of numbers n and e. Anyone who wants to send you a message s (represented by an integer) encrypts it using the following *RSA transformation* defined by n and e:

encrypted message = s to the power of e, modulo n

or

$$S = (s^e) \bmod n.$$

If you receive an encrypted message S, you decrypt it by performing another RSA transformation with n and a special number d:

s' = encrypted message to the power of d, modulo n

or

$$s' = (S^d) \bmod n.$$

The number d is chosen to have the property that $s = s'$ for every message s,[3] namely,

$$s = (s^e)^d \bmod n.$$

[2]For example, the ASCII standard representation of a character is a 7-bit integer. In this problem set we will represent a block of four characters as a 28-bit integer ($0 \leq s < 2^{28}$) by concatenating the ASCII codes of the four characters.

[3]Actually, this is true only if $\gcd(s, n) = 1$. If n is the product of two large primes, then almost all messages $s < n$ will satisfy this.

It can be shown that the number d that has this property is the one for which

$$de \equiv 1 \bmod m \tag{3}$$

that is, for which d is the *multiplicative inverse* of e modulo m.[4] It turns out that it is easy to compute d efficiently if you know e and $m = (p-1)(q-1)$.

Thus, to generate a pair of RSA keys, you choose prime numbers p and q, compute $n = pq$, choose e, and use this to compute d. You publish the pair n and e as your public key, but keep d secret to yourself. People send you encrypted messages using the pair (n, e). You decrypt these messages using the pair (n, d).

The security of the RSA system is based on the fact that even if someone knows e and n, the most efficient way known for them to decrypt a message is to factor n to find p and q, then use these to compute m, then use e and m to compute d.

That is to say, cracking an RSA code is, as far as anyone knows, as difficult a computational problem as factoring n into its prime factors p times q. And although there has been a tremendous amount of research on factoring, factoring arbitrary large numbers is not a computationally feasible task. For example, factoring $n = pq$ where p and q are each 200-digit primes, even with the today's best factoring algorithms, would require running for more than 100 years on today's fastest supercomputers.[5]

Digital signatures; Encrypting and signing

In their 1976 paper, Diffie and Hellman suggested applying public-key encryption to solving another important problem of secure communication. The problem is this: suppose you want to send a message by electronic mail. How can people who receive the message be sure that it really comes from you—that it is not a forgery? What is required is some scheme for marking a message in a way that cannot be forged. Such a mark is called a *digital signature*.

Diffie and Hellman's suggestion was to proceed as follows: take the message and apply a publicly agreed upon *compression function* (also called a *hash function*) that transforms the message to a single, relatively small number. In general, there will be many messages that produce the same hash value. Now transform the hash value using your private key. The transformed hash value is your digital signature, which you transmit along with the message. Anyone who receives a message can authenticate the signature by transforming it using your *public key* and checking that this gives the same result as applying the compression function to the message.

The reason this scheme works is that anyone who wants to forge a message claiming to be from you must produce a number that, when transformed by

[4]This is a basic result in number theory , we'll just ask you to take it on faith.

[5]No one has actually proved that cracking an RSA code is as difficult a problem as factoring, but no other method for cracking these codes has been discovered. In addition, some computer scientists believe that it may be possible to prove that there can be no fast (e.g., logarithmic time) algorithms for factoring. Given the popularity of RSA, the discovery of such an algorithm would result in a massive security breakdown for banks, businesses, and other organizations that use RSA.

your public key, matches the hash value. Anyone can compute the hash value, since the compression function is assumed to be public. But since you are assumed to be the only one who knows your private key, only you can produce the number which is transformed to the hash value by your public key. Trying to forge a digital signature is essentially the same task as trying to crack a public-key encrypted message.

An even cuter idea works as follows: Suppose Barbara wants to send George a signed message that only George will be able to read. She encrypts the message using George's public key. Then she signs the encrypted result using her own private key. When George receives a message that is supposed to be from Barbara, he first uses Barbara's public key to authenticate the signature, then decrypts the message using his own private key. Figure 1 gives an overview of the method.

Notice what this accomplishes: George can be sure that only someone with Barbara's private key could have sent the message. Barbara can be sure that only someone with George's private key can read the message. This is accomplished without exchanging any secret information between George and Barbara. It's this capacity for achieving secure communication without having to worry about exchanging secret keys that makes public-key cryptography such an important technique.

Implementing RSA

The primary thing we need in order to implement RSA is the fast exponentiation algorithm from section 1.2.6 of the SICP:

```
(define (expmod b e m)
  (cond ((zero? e) 1)
        ((even? e)
         (remainder (square (expmod b (/ e 2) m)) m))
        (else (remainder (* b (expmod b (- e 1) m)) m))))
```

We'll assume that an RSA key is represented as a pair—modulus and exponent:

```
(define make-key cons)
(define key-modulus car)
(define key-exponent cdr)
```

The basic RSA transformation is then

```
(define (RSA-transform number key)
  (expmod number (key-exponent key) (key-modulus key)))
```

Generating prime numbers

To generate RSA keys, we first of all need a way to generate primes. The most straightforward way is to pick a random number in some desired range and start testing successive numbers from there until we find a prime. The following procedure starts searching at a randomly chosen integer between `start` and `start` + `range`:

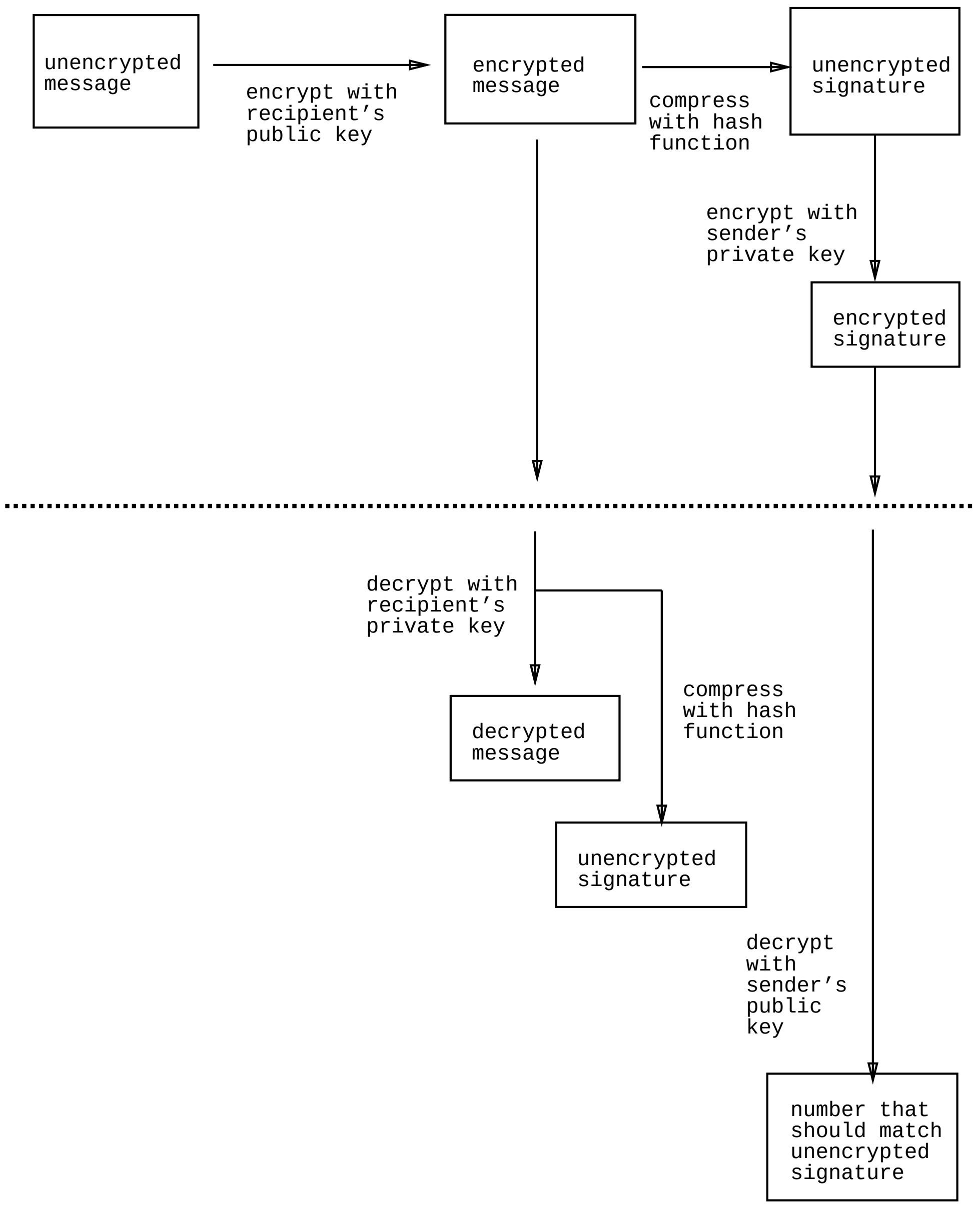
unencrypted message
encrypt with recipient's public key
encrypted message
compress with hash function
unencrypted signature
encrypt with sender's private key
encrypted signature
decrypt with recipient's private key
decrypted message
compress with hash function
unencrypted signature
decrypt with sender's public key
number that should match unencrypted signature

Figure 1: Encryption with digital signature.

```
(define (choose-prime smallest range)
  (let ((start (+ smallest (choose-random range))))
    (search-for-prime (if (even? start) (+ start 1) start))))

(define (search-for-prime guess)
  (if (fast-prime? guess 2)
      guess
      (search-for-prime (+ guess 2))))

(define choose-random
  ;; restriction of Scheme RANDOM primitive
  (let ((max-random-number (- (expt 2 31) 1)))
    (lambda (n)
      (random (floor (min n max-random-number))))))
```

The test for primality is the Fermat test, described in section 1.2.6 of the SICP:

```
(define (fermat-test n)
    (let ((a (choose-random n)))
      (= (expmod a n n) a)))

(define (fast-prime? n times)
    (cond ((zero? times) true)
          ((fermat-test n) (fast-prime? n (- times 1)))
          (else false)))
```

Generating RSA key pairs

Now we can generate a public RSA key and matching private key. We'll represent these as a pair:

```
(define make-key-pair cons)
(define key-pair-public car)
(define key-pair-private cdr)
```

The following procedure generates an RSA key pair. It picks primes p and q that are in the range from 2^{14} to 2^{15} so that $n = pq$ will be in the range 2^{28} to 2^{30}, which is large enough to encode four characters per number.[6] After picking the primes, it computes n and m according to equations (1) and (2). It then chooses an exponent e and finds a number d that satisfies equation (3).

```
(define (generate-RSA-key-pair)
  (let ((size (expt 2 14)))
    (let ((p (choose-prime size size))
          (q (choose-prime size size)))
      (if (= p q)                 ;check that we haven't chosen the same
prime twice
          (generate-RSA-key-pair) ;(VERY unlikely)
          (let ((n (* p q))
                (m (* (- p 1) (- q 1))))
            (let ((e (select-exponent m)))
              (let ((d (invert-modulo e m)))
                (make-key-pair (make-key n e) (make-key n d)))))))))
```

The exponent e can be any random number $0 < e < m$ with $\gcd(e, m) = 1$. The `gcd` procedure is given in section 1.2.5 of the notes, but is actually a Scheme primitive.

[6]We're using such small values of n for this problem set because we want you to play around with cracking an RSA system. By starting with larger random numbers, you can use the same method to produce a system that really is secure.

```
(define (select-exponent m)
  (let ((try (choose-random m)))
    (if (= (gcd try m) 1)       ;if gcd is not 1, then try again
        try
        (select-exponent m))))
```

Computing the multiplicative inverse

The number d required for the RSA key must satisfy

$$de \equiv 1 \bmod m$$

Using the definition of equality modulo m, this means that d must satisfy

$$km + de = 1$$

where k is a (negative) integer. One can show that a solution to this equation exists if and only if $\mathrm{gcd}(e, m) = 1$. The following procedure generates the required value of d, assuming that we have another procedure available which, given two integers a and b, returns a pair of integers (x, y) such that $ax+by = 1$.[7]

```
(define (invert-modulo e m)
  (if (= (gcd e m) 1)
      (let ((y (cdr (solve-ax+by=1 m e))))
        (modulo y m))   ;take y modulo m, in case y was negative
      (error "gcd not 1" e m)))
```

Solving $ax + by = 1$ can be accomplished by a nice recursive trick that is closely related to the recursive GCD algorithm in section 1.2.5 of the text. Let q be the quotient of a by b, and let r be the remainder of a by b, so that

$$a = bq + r$$

Now (recursively) solve the equation

$$b\bar{x} + r\bar{y} = 1$$

and use $\bar{x}$ and $\bar{y}$ to generate x and y.

Encrypting and decrypting messages

Finally, to use RSA, we need a way to transform between strings of characters and numbers. The code for this problem set includes procedures `string->intlist` and `intlist->string` that convert between character strings and lists of integers. Each integer (between 0 and 2^{28}) encodes 4 successive characters from the message. If the number of characters is not a multiple of 4, the message is padded by appending spaces:

```
(string->intlist "This is a string.")
;Value: (242906196 69006496 245157985 217822450 67637294)

(intlist->string '(242906196 69006496 245157985 217822450 67637294))
;Value: "This is a string.   "
```

[7]The Scheme primitive `modulo`, which we use to insure a positive result, is the same as `remainder`, except on negative arguments: `(remainder -12 7)` is -5, while `(modulo -12 7)` is 2. In general, `(modulo a b)` always has the same sign as `b`, while `(remainder a b)` always has the same sign as `a`.

The code for these two procedures is included with the problem set code, but you are not responsible for it. You may want to look at it if you are interested in how character strings can be manipulated in Scheme.

To encrypt a message, we transform the message into a list of numbers and convert the list of numbers using the RSA process together with one key in the key pair.

```
(define (RSA-encrypt string key1)
  (RSA-convert-list (string->intlist string) key1))
```

You might guess that the right way to encode the list of numbers would be to encode each number in the list separately. But this doesn't work well. (See homework exercise 1 below.) Instead, we encrypt the first number, subtract that from the second number (modulo n) and encrypt the result, subtract that from the next number and encrypt the result, and so on, so that each number in the resulting encrypted list will depend upon all the previous numbers:

```
(define (RSA-convert-list intlist key)
  (let ((n (key-modulus key)))
    (define (convert lst sum)
      (if (null? lst)
          '()
          (let ((x (RSA-transform (modulo (- (car lst) sum) n)
                                  key)))
            (cons x (convert (cdr lst) x)))))
    (convert intlist 0)))
```

We'll leave it to you to implement the analogous `RSA-unconvert-list` procedure that undoes this transformation using the other key in the key pair. Then we have

```
(define (RSA-decrypt intlist key2)
  (intlist->string (RSA-unconvert-list intlist key2)))
```

Finally, to generate digital signatures for encrypted messages, we need a standard compression function. In this problem set, we'll simply add the integers modulo 2^{28}.[8]

```
(define (compress intlist)
  (define (add-loop l)
    (if (null? l)
        0
        (+ (car l) (add-loop (cdr l)))))
  (modulo (add-loop intlist) (expt 2 28)))
```

Your mission consists of **two** tasks.[9] Use the tome's inscription above for reference.

[8]In practice, people use more complicated compression schemes than this. You might want to think about why.

[9]This problem set was designed in 1987 by Ruth Shyu and Eric Grimson and revised in 1992 by David LaMacchia and Hal Abelson.

For your convenience, the template file **`mission08-template.rkt`** contains a line to load the Scheme source file **`rsa-mission08.rkt`**. For completeness, all definitions mentioned in the tome's inscription have been included, although you will find that much of it is unneeded for this mission.

To test the code, evaluate

```
(define test-public-key1 (key-pair-public test-key-pair1))
(define result1 (RSA-encrypt "This is a test message." test-public-key1))
```

`result1` should be the list

```
(19021802 316443328 154393319 244434258 143923032 217578826)
```

`test-key-pair1` is a sample RSA key pair that we have generated for you to test your code with. Keep in mind that punctuation and upper vs. lower case are significant in the test string.

Important Note: Your solutions to the following exercises are to be entered into mission08-template.rkt. For exercises that require you to test your solution, the output of these tests should be commented. Prior to submission, ensure that your answers are written such that should the entire file be copied to the code submission box, and it will run without errors and that the printouts from your solutions, if any, should be clearly displayed in the interactions screen. Failure to comply will result in a penalty to our time and your grades.

Task 1:

The procedure `RSA-encrypt` would be much simpler if we were to encrypt each number in the list separately:

```
(define (RSA-encrypt string public-key)
    (map (lambda (int) (RSA-transform int public-key))
         (string->intlist string)))
```

What would be the analogous `RSA-decrypt` procedure? (We do not mean the one defined in the file.) Why is this simple scheme inadequate for secure encryption?

Important Note: There should be no printouts to the interactions screen for this exercise.

Task 2:

As the tome's inscription reads in the section "Encrypting and decrypting messages", it is left to you to implement `RSA-unconvert-list`, which is required to decrypt the encrypted messages. Implement this procedure, which takes as arguments a list of integers to decode and a decoding key, and returns a list of integers, undoing the transformation implemented by `RSA-convert-list`.

Hint: This procedure is very similar in form to `RSA-convert-list`. If you find yourself doing something much more complicated, then you are barking up the wrong tree—ask for help if necessary.

To test your procedure, try

```
(define test-private-key1 (key-pair-private test-key-pair1))

(RSA-unconvert-list result1 test-private-key1)
```

You should obtain the result

```
(242906196 69006496 213717089 229128819 205322725 67875559)
```

If that works correctly, then you should be able to evaluate

```
(RSA-decrypt result1 test-private-key1)
```

to obtain the original test message (except for some trailing whitespaces). We've also supplied a second key pair for you to work with, which you can obtain by evaluating

```
(define test-public-key2 (key-pair-public test-key-pair2))
(define test-private-key2 (key-pair-private test-key-pair2))
```

Submit the code for your procedure, the sample encryption and decryption of the above test message, and a sample encryption and decryption (using `test-key-pair1` and `test-key-pair2`) of several messages of your choice (please include the Scheme procedure calls—e.g. `(RSA-encrypt "This is a test message." test-private-key1)`—that you used to generate your sample and comment out the output of these calls and the resulting samples).

Important Note: The printouts for this exercise should consist of several sets of 2 lines, each set having the list of numbers representing the converted message and the decrypted message.