MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 2

- Issued: Tuesday, February 10
- Tutorial preparation for: Week of February 16. Next week, MIT will hold Monday's classes on Tuesday. Monday's tutorials will be held on Tuesday, February 17. Your tutor will announce arrangements for making up the Tuesday tutorials.
- Written solutions due: Friday, February 20 in recitation
- Reading: Finish chapter 1 before lecture on February 12. Read sections 2.1 and 2.2.1 before lecture on February 19.

Public-Key Cryptography[1]

1. Background on public-key cryptography

People have been using secret codes for thousands of years. So 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.*[2]

Cryptographic systems are typically based on *ciphers* (i.e., codes) that use *keys* for encryption and decryption. In traditional ciphers, the so-called *symmetric ciphers*, the key used to encrypt a message is also the key that decrypts the message. 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. This is of immense 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, or *public 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 public key, no

[1] This problem set is based on an earlier 6.001 problem set originally written in 1987 by Ruth Shyu and Eric Grimson and revised in 1992 by David LaMacchia and Hal Abelson. There was a major revision by Hal Abelson for 1998 (changing the method of encryption from RSA to ElGamal).

[2] The first publication of this idea appeared in W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, IT-22:6, 1976, pp 644–654, and Diffie and Hellman have been credited with the invention. In December 1997, however, the British Communications Electronics Security Group (part of British intelligence), published a memo describing how the technique was invented by CESG's Malcolm Williamson in 1973, but kept secret. See James Ellis, "The Story of Non-Secret Encryption," http://www.cesg.gov.uk/ellisint.htm.

one could decrypt messages sent to you without knowing some additional secret information, which you keep private to yourself. Diffie and Hellman called such a system a *public-key* cryptography system.

Diffie-Hellman key agreement

The public-key system we will investigate in this problem set is based on the method of *Diffie-Hellman key agreement*, which Hal described in lecture on February 10. This is a method by which two people can construct a shared secret that will be known only to the two of them, even though all the communication between them is public.

Recall the basic idea from lecture: If Alyssa and Ben wish to communicate[3] they agree (in public) on a large prime number p and a number g which is a generator for p. (For g to be a generator means that the powers $g, g^2, g^3, \ldots, g^{p-1}$, taken modulo p, produce all the integers $1, 2, 3, \ldots, p-1$, in some order.)

Alyssa picks a secret number x and computes $y \equiv g^x \bmod p$.[4] Ben picks a secret number $\hat{x}$ and computes $\hat{y} \equiv g^{\hat{x}} \bmod p$. Alyssa sends Ben y, and Ben sends Alyssa $\hat{y}$. Alyssa now computes $\hat{y}^x \bmod p$ and Ben computes $y^{\hat{x}} \bmod p$. But these are the same number because

$$\hat{y}^x \equiv (g^{\hat{x}})^x \equiv g^{\hat{x}x} \equiv g^{x\hat{x}} \equiv (g^x)^{\hat{x}} \equiv y^{\hat{x}} \bmod p$$

Now that Alyssa and Ben have this shared number, call it K, they can use K as a key for sending and receiving messages using some ordinary symmetric cipher.

The essential point is that *all* communications between Alyssa and Ben could be public, and an eavesdropper would still not know K and so could not decrypt the messages. All the eavesdropper would know is p, g, y, and $\hat{y}$. If p is a large prime, there is no efficient way to use these to compute K.

ElGamal Encryption and Decryption

Suppose Alyssa wants to set up a system that allows anyone in the world to send her an encrypted message that only she can decrypt. She can do this with a small variation of the secret-sharing scheme above.

Just as above, Alyssa picks a prime p, a generator g, and a secret number x, and she computes $y = g^x \bmod p$. She keeps x secret to herself and publishes the values (p, g, y). These published values form Alyssa's public key.

Suppose now that Ben (or anyone) wants to send Alyssa an encrypted message. He gets Alyssa's public key, which has the values of p, g, and y. Next he picks his own number $\hat{x}$. From this, he

[3] Note for those people familiar with the literature on cryptography: Alyssa P. Hacker and Ben Bitdiddle hold the patent on imaginary characters in 6.001. The management has accordingly notified Alice and Bob that they may not participate in this problem set.

[4] The notation $r \equiv s \bmod p$ (read "r is congruent to s modulo p) means that r and s produce the same value when they are reduced modulo p, i.e., that r and s have the same reminder modulo p.

computes $\hat{y} = g^{\hat{x}} \bmod p$, and he also computes $K = y^{\hat{x}} \bmod p$. Ben uses K as the key for encrypting the message to Alyssa using some symmetric algorithm. He sends the encrypted text to Alyssa, along with $\hat{y}$.

When Alyssa receives an encrypted message, she takes the $\hat{y}$ part that came with it, and computes $K = \hat{y}^x \bmod p$. (Remember: Alyssa, and only Alyssa, knows x.) She now uses K as the key for decrypting the message. Other people who see the message can't decrypt it: They know p, g, y and $\hat{y}$, but they can't compute K from this without knowing either x or $\hat{x}$.

This method of public-key encryption is known as *ElGamal key agreement*[5] The method is also sometimes called *half-certified Diffie-Hellman.* "Half-certified" here refers to the fact that Ben can be sure that he is sending a message to Alyssa (or to whomever published that key). Alyssa, however, has no idea who really sent her the message, since anyone in the world can see her public key.

Implementing encryption and decryption

Our main tools for implementing encryption and decryption are computing primes and doing fast modular exponentiation as in section 1.2.6 of the textbook.

We have the procedure that computes a power of a number modulo another number:

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

We also have the Fermat test:

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

(define (fast-prime? n times)
    (cond ((= times 0) true)
          ((fermat-test n) (fast-prime? n (- times 1)))
          (else false)))
```

To generate a prime, we pick a random value with a specified number of digits and start testing successive odd numbers from there until we find a prime. We'll consider a number to be prime if it passes two rounds of the Fermat test.

[5]This method, and the digital signature method discussed below, are based on work during the early 80s by cryptographer Taher ElGamal. ElGamal currently works as a security expert for Netscape.

```
(define (choose-prime digits)
  (let ((range (expt 10 (- digits 1))))
    ;;start with some number between range and 10*range
    (let ((start (+ range (choose-random (* 9 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))))
```

The procedure `choose-random` used here takes an arbitrary integer n and returns a number chosen at random between 2 and $n-2$, inclusive. Picking random numbers in this range will be useful for several of the procedures in this problem set.

Finding generators: safe primes

Unfortunately, it's not enough just to find a prime. We also have to find a generator for the prime. Finding a generator for an arbitrary prime can be complicated, but there is a certain kind of prime for which it is easy. These are so-called *safe primes*. A safe prime is a prime number p of the form $2q+1$ where q is also prime. The following theorem lets us compute generators for safe primes:[6]

> If $p = 2q+1$ is a safe prime, then for any number $2 \leq g \leq p-2$ either $g^2 \equiv 1 \bmod p$, or $g^q \equiv 1 \bmod p$, or g is a generator for p.

We can use these ideas as follows:

- To find a safe prime, we search for a prime q and test if $p = 2q+1$ is also prime. If it is, we've found a safe prime. If it's not, we keep searching.
- To find a generator for a safe prime $p = 2q+1$, we choose a random number g such that $2 \leq g \leq p-2$ and check whether g^2 and g^q are both different modulo p from 1. If so, g is a generator. If not, we try again with a new guess for g.[7]

Key systems and public keys

Given the above procedures, arranging to do encryption and decryption is straightforward. Let's call the list of four numbers—p, g, x, and y—a *key system*. The public part of this, namely, p, g, and y, we'll call a *public key*.

The following procedure constructs a key system according to the method described above: pick a safe prime p, pick a generator g, pick a random number x, and compute $y = g^x$ modulo p:

[6] We won't include the proof here, since this is not a number theory class. Just take the result on faith.

[7] There's a bit of number theory we've sloughed over that guarantees that these algorithms are reasonable. Given a safe prime, the odds that a randomly picked g is a generator are about 1 in 2. Given a prime q, the odds that $2q+1$ is also prime are about 1 in $\ln q$. So trying random guesses is likely to succeed without too long a wait.

```
(define (generate-key-system digits)
  (let ((p (choose-safe-prime digits)))
    (let ((g (find-generator p)))
      (let ((x (choose-random p)))      ;x will be secret
        (let ((y (expmod g x p)))       ;y will be public
          (make-key-system p g x y))))))
```

The argument here specifies the number of digits (minus 1) for the prime.

The procedure `make-key-system` in the final line of the procedure is an example of a *data constructor*. All it does is package the four numbers together into a structure called a *list*. Once a key system has been constructed, we can select the individual pieces using *data selectors*: `key-system-p`, `key-system-g`, `key-system-x`, and `key-system-y`. These constructors and selectors are all very simple procedures. In this problem set we've defined all necessary constructors and selectors for you, so you don't have to think about them much. We'll discuss constructors and selectors in lecture on February 19th.

Once we have a key system, we can extract the public parts to form the corresponding public key.

```
(define (key-system->public-key key-system)
  (make-public-key (key-system-p key-system)
                   (key-system-g key-system)
                   (key-system-y key-system)))
```

Here `make-public-key` is another data abstraction we have provided for you, with selectors `public-key-p`, `public-key-g`, and `public-key-y`.

Encrypting and decrypting

The final element we need in order to implement ElGamal public-key encryption is a symmetric cipher, which will use the shared key to encrypt and decrypt. For this problem set, we've provided procedures `symmetric-encrypt` and `symmetric-decrypt`. `Symmetric-encrypt` takes a text string message and a numeric key, and encrypts the message using the key to produce a cipher-text. Given the cipher-text and the same key, `symmetric-decrypt` will recover the message text.[8]

For example,

```
(symmetric-encrypt "My little secret" 87)
;Value: "\234b\bJ6\025\205\r\253\271'\031/\213\264\v"

(symmetric-decrypt "\234b\bJ6\025\205\r\253\271'\031/\213\264\v" 87)
;Value: "My little secret"
```

[8]The particular symmetric cipher used here, *Blowfish*, was invented by cryptographer Bruce Schneier. We won't show you the code for this, which is buried in the guts of the Scheme system. For a description of the algorithm see Schneier's book *Applied Cryptography*, second edition, Wiley, 1996.

The weird backslash numbers in the cipher-text are Scheme's way of printing character codes that do not correspond directly to printable characters. (Our cipher produces 8-bit bytes, not all of which are printable characters.)

Putting this all together, given a message together with Alyssa's public key, we encrypt the message as described above. Namely, we pick a random value for $\hat{x}$ (not to be confused with the x of Alyssa's key system, which only Alyssa knows), use Alyssa's y raised to the $\hat{x}$th power modulo p as the shared key for the symmetric cipher, and send the result together with $y = g^{\hat{x}} \bmod p$.

```
(define (encrypt message-text public-key)
  (let ((p (public-key-p public-key))
        (g (public-key-g public-key))
        (y (public-key-y public-key)))
    (let ((my-x (choose-random p)))
      (make-encrypted-message
       (expmod g my-x p)
       (symmetric-encrypt message-text (expmod y my-x p))))))
```

The procedure `make-encrypted-message` is another data constructor. The associated selectors that retrieve the two pieces are `encrypted-message-y` and `encrypted-message-cipher-text`.

We'll leave it to you to implement the corresponding `decrypt` procedure, which takes an encrypted message and a key system, and returns the decrypted ciphertext (provided that the key system is the correct one).

Cracking the system; The discrete logarithm problem

Suppose someone wants to decrypt a message not intended for them. Suppose this someone is the kind of someone who happens to have massive computational power available to devote to the problem.[9] They have to start with the public key of the recipient and recover the secret number x. That is, given p, g, and y they must find the number x such that $y \equiv g^x \bmod p$. This is called the *discrete log problem.*[10]

How does one compute discrete logs? One way is simply brute-force search: Try all the values for x between 2 and $p - 2$ until you find the one that works. Unfortunately for our "someone," the computational burden here is vast. Remember that if we use successive squaring, the time required to raise a number to a power up to p has order of growth $\Theta(\log p)$, i.e., grows as the number of digits of p. But to scan all the numbers less than p has order of growth $\Theta(p)$. Each time you add one more digit to your prime, you increase the computation for cracking discrete logs by a factor of 10, while you increase the computation required to do encryption and decryption by only a little bit.

[9] Typical someones who come to mind here are certain government agencies, international crime rings, and MIT undergraduates.

[10] Actually, if you're reading very closely, you'll see that we've made an unwarranted assumption. What you really need is to solve the so-called *Diffie-Hellman problem*: given $g^x \bmod p$ and $g^{\hat{x}} \bmod p$, compute $g^{x\hat{x}} \bmod p$. As far as anyone knows, the only way to do this is to compute discrete logs, and for many classes of primes the discrete log problem and the Diffie-Hellman problem have been proved equivalent.

Are there better algorithms than brute force search? Yes, but they don't help a whole lot. There's a method called *Pohlig's rho algorithm* which has order of growth $\Theta(\sqrt{p})$, but this is still exponential in the number of digits in p. The fastest algorithm known is called the *number-field sieve*, and it has order of growth

$$e^{(1.923+o(1))(\ln p)^{1/3}(\ln(\ln p))^{2/3}}$$

which is not quite exponential in the number of digits, but still grows pretty damned fast.[11]

2. Digital signatures

In their seminal 1976 paper, Diffie and Hellman suggested applying public-key cryptography to solving another important problem of secure communication: 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 can be easily verified, but cannot be forged. Such a mark is called a *digital signature*.

In order to perform digital signatures, one generally makes use of some standard *message-digest function* (also called a *hash function*) that transforms an arbitrary length string into a single number of uniform length. We've provided a procedure `message-digest` that performs this operation. It takes an arbitrary string and returns a number between 0 and 2^{128} as its result.[12]

A digital signature scheme consists of two procedures, one that signs messages and one that verifies signatures. Signing a message uses the secret information in a key system. Verification uses the corresponding public key. The idea is that anyone can have the public key and thus verify the signature, but only the person who knows the secret value x in the key system could have produced the signature. This is like the opposite of public-key encryption, where anyone can encrypt the message, but only the person with the secret can decrypt the message. A verified signature attests to the facts that

- The message was signed by the person who knows the secret x (who is presumably the person who distributed the public key).
- The message that was received is the authentic message that was signed (i.e., it was not tampered with).

There are many (in fact, an infinite number) of signature schemes. The one we present, called *ElGamal signatures*, is closely related to ElGamal encryption as described above.

To sign a message M (which may itself be encrypted or not) Alyssa first applies the message digest function to M and reduces that modulo p to produce a number h. Then she uses h together with the values p, g, x in her key system as follows:

[11] As a consequence of this, the encryption you are implementing in this problem set really is a high-grade security method. For instance, the encryption system PGP 5.0 uses ElGamal encryption with a suggested prime size of around 300 digits. If anyone has the ability to crack something like that, they aren't talking.

[12] There are tremendous subtleties in designing a good message-digest function. One property it should have is that it should be computationally infeasible to find two strings that hash to the same value. The particular function used here, called MD5, was invented by Ron Rivest of MIT.

1. Pick a random integer k between 2 and $p-2$ such that $\gcd(k, p-1) = 1$, and compute the *inverse* of k modulo $p-1$, i.e., find the number d such that $kd \equiv 1 \bmod (p-1)$.
2. Compute $r = g^k \bmod p$.
3. Compute $s = d(h - xr) \bmod (p-1)$.
4. The signature is the pair of numbers r and s, which are transmitted along with the message.

To verify a signed message M, r, s, using Alyssa's public key p, g, y:

1. Check that $0 < r < p$. Otherwise, the signature is bad.
2. Compute h by applying the digest function to M and reducing modulo p.
3. Check whether $y^r r^s \bmod p = g^h \bmod p$. If so, the signature is good. If not, the signature is bad.[13]

The algebra that shows why this works is a bit messy, but it is straightforward to check. (Trust us.) The main point to remember is that anyone can verify a signature, but only the person with the secret x information from the key system can produce the signature. Notice that if we can crack someone's public key, we can then forge that person's digital signature.

Implementing signing and verifying

Here is the procedure that signs a message, using the information in a key system to implement the steps above.

```
(define (sign message key-system)
  (let ((p (key-system-p key-system))
        (g (key-system-g key-system))
        (x (key-system-x key-system)))
    (let ((h (modulo (message-digest message) p))
          (k (good-k p)))
      (let ((r (expmod g k p))
            (d (invert-modulo k (- p 1))))
        (let ((s (modulo (* d (- h (* x r))) (- p 1))))
          (make-signature r s))))))
```

`Make-signature` here is another data constructor, which combines the two parts into a structure. You get the parts back using the selectors `signature-r` and `signature-s`.

`Good-k` finds a random number k with $\gcd(k, p-1) = 1$. It's easy: just keep picking values for k until you find a good one:

[13] Note that when checking this condition we can compute each term modulo p before multiplying and comparing, because $ab \bmod p \equiv (a \bmod p)(b \bmod p) \bmod p$.

```
(define (good-k p)
  (let ((k (choose-random p)))
    (if (= (gcd k (- p 1)) 1)
        k
        (good-k p))))
```

The only hard part is `invert-modulo`, which finds a number d such that $dk = 1 \bmod m$. We'll discuss that below.

We'll also leave it to you to implement the corresponding `verify` procedure, which takes the message (a string), a signature (i.e., a pair r and s), and a public key, and checks the signature.

Modular inverses

The number d required for the signature must satisfy $dk = 1 \bmod m$, where $m = p - 1$. Using the definition of equality modulo m, this means that d must satisfy the equation $em + dk = 1$ where e is a (negative) integer. One can show that a solution to this equation exists if and only if $\gcd(k, m) = 1$. 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 $ax + by = 1$.

```
(define (invert-modulo k m)
  (if (= (gcd k m) 1)
      (let ((y (cadr (solve-ax+by=1 m k))))
        (modulo y m))                    ;just in case y was negative
      (error "gcd not 1" k m)))
```

This procedure uses the Scheme primitive `list` data constructor. Given two (or more) items, `list` combines them into a single structure called a *list*. The selector `car` returns the first item in the list, and the selector `cadr` returns the second item. List structure will be formally introduced in lecture on February 19th, but the simple explanation here is all you need now for this problem set.

Solving $ax + by = 1$ can be accomplished by a neat recursive trick that is closely related to the recursive GCD algorithm in section 1.2.5 of the text. Let q be the integer 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. We'll leave to you the details of how to write the actual procedure. (Ask in recitation.)

3. Tutorial exercises

You should prepare these exercises for oral presentation in tutorial.

Tutorial exercise 1: Write the procedures `choose-safe-prime` and `find-generator`, which you will need for completing the computer part of this assignment. `Choose-safe-prime` should take a `digits` input (as does `choose-prime`) and return a safe prime. `Find-generator` should take a safe prime and return a generator for the prime. Bring your written answers to your tutor, who will use them to help debug your coding style and suggest alternatives.

Tutorial exercise 2: For each of the cases below, give an expression involving `foo` whose evaluation will return the value 2.

```
(define foo
  (lambda (x) (* 2 x)))

(define foo
  (lambda (x) (x 1)))

(define foo
  (lambda (x) (lambda (y) (* x y))))

(define (foo x)
  (x (lambda (y) (* y 2))))
```

Tutorial exercise 3: Do exercise 1.26 of the text.

Tutorial exercise 4: Do exercises 1.42 and 1.43 of the text.

4. Programming assignment

Begin by loading the code for problem set 2, using the Edwin command `M-x load problem set`.

Note on debugging procedures that use randomness: Many of the procedures you will be working with this assignment depend on selecting random numbers, and so will give different answers each time you run them. Such procedures can be confusing to debug, since it's hard to tell whether things are changing due to your modifications or just due to selecting different random numbers. To help you in debugging, we've provided a procedure `reset-random!` (which takes no arguments). Whenever you run `(reset-random!)` the random number generator will be returned to its initial state. This permits you to do repeatable experiments.

Computer exercise 1: Implement the `decrypt` procedure, which takes as arguments an encrypted message and a key system and produces the unencrypted message (assuming the key system is the correct one for decrypting the message). For testing your procedure, we've provided a key-system `ks-ex-1` and a test encrypted message, `enc-message-ex-1`. Turn in a listing of your procedure and demonstrate that you can decrypt the message.

Computer exercise 2: Define the procedures `find-generator` and `choose-safe-prime`, which you wrote for tutorial exercise 1. Test them by finding a safe prime (use size equal to 5) and a generator for it. Once you think this is working, you should be able to run the procedure `generate-key-system`. Generate a key system and the corresponding public key. Demonstrate that you can use the public key to encrypt messages and then use the key system to decrypt them.

Computer exercise 3: Implement the `verify` procedure for ElGamal digital signatures. To test your code, we've pre-defined a public key `pk-ex-3` and two signed messages: `m1-ex-3` with signature `s1-ex-3`, and `m2-ex-3` with signature `s2-ex-3`. Determine which message is authentic.

Computer exercise 4: Define the procedure `solve-ax+by=1`. It takes two non-negative integer arguments a and b whose GCD is assumed to be 1 and returns the list consisting of integers x and y. Demonstrate that your procedure works by finding integers x and y that satisfy the equation:

$$1915954701x + 2019374789y = 1$$

Don't forget to check your answer! Once this is working, you should be able to generate a key system and use this to sign a message with the `sign` procedure. Demonstrate that you can sign messages and then verify the signatures with the corresponding public key.

Computer exercise 5: Write a procedure `find-discrete-log` that solves the discrete log problem by brute-force search. It should take as arguments values for y, g, and p, and return an x such that $y \equiv g^x \bmod p$. Once this has been defined, you should be able to run the supplied procedure `crack-public-key` that cracks a public key, returning the original key system. Test your procedure by cracking the public key `pk-ex-5` of size 4, which is predefined in this problem set to be $p = 19079, g = 362, y = 6843$.

Computer exercise 6: How long does it take to crack the ElGamal system? To help you determine this, we've included a procedure called `timed`, which, if you place at the beginning of a combination, will evaluate the rest of the combination (as if the `timed` weren't there) and return the value along with the time required for the evaluation. Thus, evaluating

```
(timed crack-public-key pk)
```

will apply the procedure `crack-public-key` to the argument `pk` (or whatever argument you want to use) and return the result, together with the time (in seconds) required to do the computation. Crack some public keys of size 4. Remember that all the random choices will result in a wide range of results, but you should be able to get some sort of average time. How do you expect the time to crack the key to grow as the size of the prime increases? Suppose you had a million times the computational resources that you are now using in working on this problem set, and that all these resources could be dedicated to cracking a single key. How long would it take to crack a key of size 50? Of size 100? Give your answer in seconds, minutes, days, or years, whichever seems most appropriate.

Computer exercise 7: It was a dark and stormy night.[14] A 6.001 lecturer sat hunched over his terminal. Suddenly there was a crash as a rock flew through the window. Attached to the rock was a note:

```
Our network sniffer has just intercepted the following e-mail
addressed to Dean Williams. -- A friend
  13998
  \232\211\351\216\3509m\025\327\234V\203v\354=F\tw\026\004~\335\264\234
  \234\233bU\223b\206/\310\276\356\331\240\256I\262\360\317\266I\002\373
  J\221eq<\001\301\236\303\375\372\373OFj\331\000\325c&z\306x\216\243\b
  \341\250\006\026\367X\r\302\343b~\t\271\222rkE\030C\325\033\372z\3245n
  \3506\025{[)\313+p\350x;%u\006\262\225\331\237\272K\202u1\257\331\305
  \235e5(\327,\231\255\306\223\3345\370\262h\337\242.\211|\325\3736c\004
  D84iOH\322eLl\000\267\036\371\313V(\376H\312\270\001\234\203\220b\330
  \265\354\020k$= X&\250? \372
```

The Dean's office is working on better security, but due to reengineering, they have been unable to issue public keys of size greater than 5. William's public key and the above cipher-text are defined for you as `pk-ex-7` and `secret-ex-7`. Crack the key and decrypt the message. It sounds ominous, doesn't it? (Be patient in cracking the key, which may take a minute or two. Note that this is a longer key than the ones you cracked in exercise 6.)

Turn in answers to the following questions along with your answers to the questions in the problem set:

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).
   - If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the text (such as some of the archives),

[14]...with apologies to Baron Bulwer-Lytton and to Radia Perlman, Michael Speciner, and Charles Kaufman, who begin their outstanding book *Network Security* like this.

please indicate your consultants' names and your references. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

- Otherwise, write "I worked alone using only the reference materials," and sign your statement.