MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001--Structure and Interpretation of Computer Programs
Fall Semester, 2002

Project I

1. **Issued: On Week 3 / Day 2**
2. **Part I due Week 4 / Day 5**
3. **Part II due Week 5 / Day 5**

Code for Part I – secret_fall02.scm, substitution_fall02.scm, for Part II – public_fall02.scm

Note – if you downloaded a standard version of Scheme, it should automatically include a version of the Blowfish encryption system. If your version of Scheme does not include Blowfish, please contact one of the Lab Assistants for information on how to download it.

You should begin working on the assignment once you receive it. It is to your advantage to get work done early, rather than waiting until the night before it is due. You should also read over and think through each part of the assignment for that week (as well as any project code) before you sit down at the computer. It is generally much more efficient to test, debug, and run a program that you have thought about beforehand, rather than doing the planning ``online." Diving into program development without a clear idea of what you plan to do generally ensures that the assignments will take much longer than necessary. We have separated this project into parts, however, you are encouraged to start subsequent parts as soon as you have finished an earlier one, provided you have also seen the appropriate lectures covering needed material, rather than waiting for the deadlines.

You should submit your solutions as a Scheme file, electronically to your TA. You can use the templates provided in the code you load for this project as a starting point, and include within that file your documented code, and examples of the usage of your code on test cases. Actual submission is to be done using the "Submit Project Files" button on the tutor, and selecting the appropriate part (1.1 or 1.2) for submission. Note that **late work will not be accepted.**

The purpose of this project is to familiarize you with procedural and data abstractions by writing Scheme code within a cryptographic context. There is a lot to read and understand; we recommend that you first skim through the week's part of the project to familiarize yourself with the format, before tackling problems.

In **PART I** of this project, you will use procedural abstractions to deal with a particular style of cryptographic system, and you will build on your experience to create more complex procedures for use in cryptographic systems. In **PART II** of this project, you will use procedural and data abstractions to create a system for manipulating messages received in a secure fashion.

Don't be intimidated! This looks like a lot of reading, but remember that this is a two part project, spread out over 3 weeks, with different sections to be read and answered for each part.

Part I

1. Introduction to Cryptography

The desire to send secret messages that only an intended recipient can understand (even if the message is intercepted by a foe) is a very ancient one. In this part of the project, you will explore some basic ideas in cryptography. You will consider various mathematical functions that *encrypt* a numeric representation of a textual message to disguise its contents so that only an intended recipient can *decrypt* and read it. Later in the project, you will also consider mathematical functions that do not encrypt the message, but rather operate on the message to produce a number that can be used later to identify if the message is *authentic*.

1.1 Numeric Representations of Textual Messages

If we want to send a message, it is likely that the message will be composed of English words and sentences. Thus we need some way of representing concatenations of words in Scheme. We will see throughout the term that there are many ways of doing this, but for the purposes of this project, we are going to use **strings.** A string is literally that – a string or sequence of characters in a particular order. In Scheme, we create a string by using double quotes to delimit the beginning and end of the string, for example:

```
"This is a string"

"This is another string, both are made from sequences of characters!"
```

Scheme has a number of built-in procedures for manipulating strings, including ways of creating them, ways of comparing them, and ways of getting out the individual characters in a string. You should look at the online Scheme manual for details.

In order to explore some of these mathematical manipulations of messages, we need a way to convert a simple textual (or string) version of a message into numbers o which our mathematical functions can operate. Consider a *plain text* message `"ab"`. One of the simplest ways to encode the message is to assign a unique number to each unique character in the message. For example, we might assign the number 1 to `a`, 2 to `b`, and so on. A plain text message `"ab"` would thus be converted to a character encoding (which we'll call ``charcodes'') that reads `"1 2"`. Of course, we'd like our computer to handle this method for encoding text messages into numbers for us.

1.1.1 ``charcodes'' - character by character encoding

The Scheme procedures for this part of the project are in the file `secret_fall02.scm`. You should read through `secret_fall02.scm`, although there are some procedures in the file that you are not likely to understand (as they use concepts we won't cover for a few lectures).

In `secret_fall02.scm`, we have provided you with some basic procedures that perform conversions between a string message (e.g. `"cat"`) and a ``charcodes'' representation (e.g. `" 3 1 20"`) of that same message, in which a numeric character code is presented for each successive character in the message. The procedure `message->charcodes` takes a plain text message (represented as a string) and returns a charcodes string using a simple substitution of the number 1 for the character `a`, 2 for `b`, etc. The

procedure `charcodes->message` does the reverse: it takes charcodes (a string with the successive numbers for each character in the message) and returns back the corresponding string text message:

```
(message->charcodes "cat")
;Value: " 3 1 20"

(charcodes->message " 3 1 20")
;Value: "cat"
```

1.1.2 “intcode” - full message encoding

In some cases we will find it convenient to work with a different conversion of a textual message into a numeric value. While ``charcodes" consist of a separate number for each individual character in a text message, an ``intcode" is one single large integer corresponding to the full message all at once.

In this project, we provide different procedures that convert the message text into an intcode by simply abutting together the (three digit) number codes for each character.[1] The file `secret_fall02.scm` provides the procedures `message->intcode` and `intcode->message` that perform this simple transformation between messages and intcode representations of those messages. For comparison with the charcodes, we see:

```
(message->charcodes "ab")
;Value " 1 2" (or 001 followed by 002).

(message->intcode "ab")
;Value 1002 (or 001 abutted with 002).

(intcode->message 1002)
;Value "ab"
```

Having these two different ways of representing a message will enable us to explore two different approaches to encrypting a message. In the case of ``charcodes", we will explore character-by-character encryption whereby we mathematically operate on each individual character code in succession. In the case of the ``intcode", we will look at mathematical operations on the entire message at once for both encryption and authentication.

1.2 Character by Character Encryption

So how might we send a ``secret" version of a plain text message? One might be tempted to simply transmit the charcodes or intcode version of a message. But clearly this is not very secure: if a foe knows we are doing standard number for letter substitution (i.e. we always start with 1 for `a`) and he or she intercepts our charcodes, then it is trivial for our foe to decode the message.

In this section, we want to consider some possible mathematical manipulations of our charcodes to disguise or encode them in such a fashion that only our intended recipient can easily decode them back to

[1] If you are used to programming in other languages, this approach might concern you – what if we run out of “room” in a fixed size integer? Fortunately, Scheme supports arbitrary precision integers: we can have integers with as many digits as we need – there is no “maximum” size integer (subject to overall machine memory limitations).

their original values. We would like to be able to send messages even if everyone knows what method we're using for sending the message. It is common to perform encryption using a publicly known *algorithm* in combination with a secret *key* to encrypt a plain text message into a *cipher text*. The goal is that (hopefully) only the other holder of the key can ``unlock" the cipher text and decrypt the message. In this project we will restrict ourselves to symmetric ciphers where both the sender and the receiver use the same key to encrypt or decrypt the message. In the following, we will look at two character-by-character encodings; in this part of the project you will experiment with these and also create some encodings of your own.

1.2.1 The Caesar Code

The ``classic" Caesar code uses an integer key indicating a rotation of the start (and end) of the alphabet. This is illustrated pictorially in the left of Fig. 1, where the outer ring indicates a character in the plaintext message, and the inner ring indicates the encoded version of the character assuming a key of 4 (a rotation of the inner ring by four places). Thus the plain text character `"b"` is encoded into the character `"F"`.

The Caesar code corresponds to a simple addition of the key to the character code for a message character in our programming system. For example, the character `"b"` corresponds to a charcode of 2; for a key equal to 4 this is Caesar encoded into the number 6. If we do this for every character in a plain text message, we produce the cipher text. For example the Caesar encryption of `"cat"` results in a secret cipher consisting of the charcodes `" 7 5 24"`. (Since we can just as well transmit numbers as strings we don't bother changing this back into characters before transmitting).

Note that in Figure 1, the numeric shift is shown as a ring. In that figure, the alphabet is 26 characters long, so a shift of 4 for the letter "z" (number 26) becomes the letter "D" (number 4). This is a variant of modular arithmetic where numbers ``wrap around" to limit the numbers to a given range. Modular arithmetic plays a fundamental role in cryptography: you will need to understand modular arithmetic for this problem set, and should study the Scheme procedures `quotient`, `remainder`, and `modulo` (see the Scheme manuals for what they do).

The Caesar code thus corresponds to a simple mathematical manipulation of the number representing each individual message character. We have defined a procedure `caesar-encode` that takes the character number (`num`) and the `key`, and performs this computation to output a new number (an integer) for that character:

```
(define (caesar-encode num key)
  (modulo (+ num key) 256))

(define (caesar-decode num key)
  (modulo (- num key) 256))
```

In this case, we are assuming that our alphabet consists of 256 possible characters rather than 26, so that we can use mixed upper and lower case and other typographical characters in our messages.

These individual encoder and decoder procedures are intended to work in conjunction with a procedure `encrypt-to-charcodes` that accomplishes the encryption of a complete text message. It essentially works by taking the `encoder` procedure, using it to encode each successive character in the plain text message, and then aggregating the results into a charcodes output. The procedure has been provided for you; you can look at its implementation in `secret_fall02.scm`. The encoder is used by the

encryption procedure, together with the plain text message and the key (equal to 77 in this case), as in the following example:

```
(define cipher-1
  (encrypt-to-charcodes "The Ides of March" caesar-encode 77))

cipher-1
;Value: " 65 85 82 13 54 81 82 96 13 92 83 13 58 78 95 80 85"

(decrypt-from-charcodes cipher-1 caesar-decode 77)
;Value: "The Ides of March"
```

The example also illustrates the use of `caesar-decode` in conjunction with the general decryption procedure `decrypt-from-charcodes` to recover the message from the cipher text (assuming we have the key!).

Note that the `encrypt-to-charcodes` procedure takes another procedure as an argument. This is an example of ``higher order procedures'' as discussed in Section 1.3 of the text, and in Lecture 6 on September 19.

1.2.2 The ``NSA'' Code

The procedures `encrypt-to-charcodes` and `decrypt-from-charcodes` are general purpose, in that they can take different encoder and decoder procedures. Consider the `nsa-encode` procedure below, which corresponds roughly to the code ring that appears as part of the official National Security Agency seal (pictured to the right in Fig. 1).

```
(define (nsa-encode num key)
  (modulo (+ (* -1 num) key) 256))

(encrypt-to-charcodes "bc" nsa-encode 1)
;Value: " 255 254"

(decrypt-from-charcodes
    (encrypt-to-charcodes "bc" nsa-encode 1) nsa-decode 1)
;Value: "bc"
```

Note that the ``flip'' of the inner code ring is accomplished by `(* -1 num)`, followed by a rotation of the ring by a number of characters given by the key. One difference between our `nsa-encode` procedure and the code ring in the NSA seal is that we have 256 rather than 26 characters in our alphabet.

In the Problems, you will be asked to provide the corresponding `nsa-decode` decoder. You will also have the opportunity to try cracking the Caesar and NSA codes.

1.3 Full Message (single intcode) Encryption

While the Caesar and NSA encoders work on a character by character basis (by repeatedly using an `encoder` procedure on each character), we are also interested in an alternative approach whereby we

simply use a single mathematical function to encrypt the message all at once. In this case, we want encryption methods that work on the ``intcode'' representation (the message as a single integer).

To support encryption into an intcode, `secret_fall02.scm` provides for you corresponding encryption and decryption procedures:

```
(define (encrypt-to-intcode msg encoder key)
  (encoder (message->intcode msg) key))

(define (decrypt-from-intcode cipher decoder key)
  (intcode->message (decoder cipher key)))
```

Consider the following simple intcode encoder and decoder procedures that work with these integer encryption and decryption procedures:

```
(define (encode-add int key) (+ int key))
(define (decode-add int key) (- int key))

;; Simple example
(define add-key 8)

(message->intcode "ab")
;Value: 1002

(encrypt-to-intcode "ab" encode-add add-key)
;Value: 1010

;; Bigger example
(define add-key 9234827538182374765023)
(define cipher-3 (encrypt-to-intcode "This is TOP SECRET" encode-add
add-key))

cipher-3
;Value: 24400800901919200901919224423924020147805676542460400921 6

(decrypt-from-intcode cipher-3 decode-add add-key)
;Value: "This is TOP SECRET"

;; Note what happens if we try this
(decrypt-from-intcode cipher-3 decode-add (- add-key 1))
;Value: "This is TOP)>\230]\b\274i1"
```

1.4 Breaking the code

Many approaches can be taken to attempt to break a code. With character based encryption (like the Caesar code), for example, knowledge about the frequency of characters in typical English usage can give clues about what codes relate to which characters. In other cases, the encryption algorithm itself may be very weak, so that plain text can be recovered through analysis of a cipher text. In still other cases, one may not only want to crack a particular message, but rather one might want to discover the secret key

being used in association with a given algorithm. If one has cracked the key, then any transmission can be quickly decrypted. In the Problems for this part of the project, you will experiment with creating some different encryption procedures, and will also try to crack the key for some messages we have found lying around.

2. Programming assignment

Load the first part of the code for project 1, that is, load the file `secret_fall02.scm` into a Scheme environment. This code file is part of the code package that can be downloaded from the 6.001 web site.

Preparing material to "hand in"

We are asking you to prepare your solutions as a Scheme file (that is, a file that has `.scm` as the second part of its name). Since this is a Scheme file, you will need to ensure that any text you include (and you should!! – comments and other documentation, examples of code that you used to run test cases, and so on) is commented out (using the character ; at the beginning of each such line). We suggest that you use the format provided in `secret_fall02.scm` as a basis for your solutions.

Ideally, your solutions should follow something like the following format. Note that by copying the portion of `secret_fall02.scm` starting with Problem 1, you can easily create a template for your solutions.

```
;; Problem 1 (please be sure to label each section of your work, so that
;; your TA can find it easily)

;; Here are some examples of using the Caesar encoder/decoder

;(define example1
;   (encrypt-to-charcodes "this is an example of some text"
;                         Caesar-encode
;                         12345))

; shown here should be the result of your evaluation of this expression,
; which you can clip out of your transcript buffer or out of your Scheme
; buffer, including any comments you want to add to explain why you
; used this particular example.

;; When you get to pieces of code that you have written, include them
;; with the requested procedure name, directly into the file, such as

(define (here-is-my-procedure some-argument)
     some-body)
```

When you have completed your project, you should submit this Scheme file electronically to your TA.

So now you are ready to start with the actual problems!!

Problem 1: Encryption with charcodes

Gain some experience using the `encrypt-to-charcodes` and `decrypt-from-charcodes` procedures, using the Caesar encoder and decoder. You should be able to encrypt and decrypt several messages of your own, using keys of your own. Show your examples.

The `cipher-2` text in `secret_fall02.scm` was produced with a Caesar code (`caesar-encode`) using 125 as a key. What does it decrypt to?

As previously described, the procedure `(encrypt-to-charcodes msg encoder key)` takes a plain text `msg`, a particular encoding procedure `encoder`, and an integer `key`. The `nsa-encode` procedure that can act as one of these encoders is also defined for you. However, the corresponding `nsa-decode` procedure is missing. Your task is to write this procedure that ``undoes'' the work of `nsa-encode` and which can be used with `decrypt-from-charcodes` to recover an nsa encrypted message. The *decoder* procedure should take two arguments: an integer `num` (corresponding to an encoded single character in the cipher text) and an integer `key`, and return an integer such that `(nsa-decode (nsa-encode num key) key)` returns the value of `num` for all positive integer keys.

Write the `nsa-decode` procedure that undoes the work of `nsa-encode`. Show that it works on test cases of your own.

The `cipher-4` message in `secret_fall02.scm` was encrypted using `nsa-encode` with key 007. Verify that your `nsa-decode` procedure works by decrypting `cipher-4`.

Note: you might even be able to break `cipher-4` by hand, particularly if you recognize the most frequent letter in English usage.

Problem 2: More encryption with charcodes

Write one or two new encoder procedures of your own, and the corresponding decoders. They do not have to involve modular arithmetic operations as in the two examples shown above. Key properties to keep in mind are: the encoding function should be one-to-one, meaning that it maps at most one input argument to any output argument; the encoding function and decoding function which must be inverses of one another, so that applying the decoder to the result of the encoder should return the value with which you started; and ideally both the encoder and decoder should be easy things to compute (in part 2 we will get to the issue of having the decoder be much harder than the encoder).

Show that your encoders and decoders work in a ``stand-alone'' fashion (on a single numeric character code) by testing them given a single number and a key. It is always good practice to work through some test cases to verify that any individual procedure works correctly before testing it within a larger system (in this case, within the `encrypt-to-charcodes` and `decrypt-from-charcodes` procedures).

Once you have your new separate encoder and decoder procedures working, show that they work with `encrypt-to-charcodes` and `decrypt-from-charcodes` by encrypting and decrypting some full messages.

Problem 3: Breaking the Caesar code

We know that Al and Cap1 are using a Caesar code, but we don't know the key they have agreed upon. However, at one of their recent crime scenes, we found both plain text and cipher text for one of their exchanges (they think that since no one else has their key any future message will remain secret). Your

mission, should you choose to accept it: figure out the key they used. Write a procedure `crack-caesar` that takes a plain text message and the corresponding cipher string that was encoded with `caesar-encode` and some unknown key, and which returns that secret key. Hint: you may find the built-in Scheme procedure `string=?,` which compares two strings for equality, to be helpful.

Hint: think carefully about how many possible keys there are given the modular arithmetic used in `caesar-encode`.

Verify that your `crack-caeser` works by discovering the key corresponding to `al-cap1-plaintext` and `al-cap1-ciphertext` defined for you in `secret_fall02.scm`.

Problem 4: Breaking any Caesar-like code

Al and Cap1 hear that you've broken their key, and no longer trust the Caesar code. They switch to the (presumably) more secure NSA encoder. Again, they hit another target but leave their plain text and enciphered transmission behind (in `better-al-cap1-plaintext` and `better-al-cap1-ciphertext`).

In the previous problem, your procedure assumed that the message had been encrypted with `caesar-encode`. To stay one step ahead of Al and Cap1, we want to generalize our cracker so that given any encoder procedure (whatever it may be) in addition to the plain and cipher texts, the key can be discovered. To make things somewhat bounded, we will also assume that keys are less than or equal to a given number of digits (e.g. a 3 digit key will be a positive integer less than `(expt 10 3) -> 1000)`. Your procedure should take arguments as in `(crack-string-encryption encoder plaintext cipher-string max-key-digits)` and again produce the key. If no key is found you may either generate an error using `(error "some error message")` or return `-1` which we will hold as an invalid key.

Once you have your procedure working, crack the key Al and Cap1 used to nsa-encrypt their new message (given `better-al-cap1-plaintext` and `better-al-cap1-ciphertext`). Also show that your solution still works for Problem 3.

```
(crack-string-encryption nsa-encode
                 better-al-cap1-plaintext
                 better-al-cap1-ciphertext
                 3)
;Value: ????
```

If *n* is the number of possible digits in the key, how long (how many encryption comparison attempts) will it take in the worst case to break a general Caesar-like key?

Problem 5: Encryption with an intcode

We now switch gears and consider encryption to and from the ``intcode'' representation of our message text (rather than the character by character encoders discussed above) as described earlier.

Consider the simple `encode-add` and `decode-add` procedures below (also described in Section 1 of this handout) that work with the `encrypt-to-intcode` and `decrypt-from-intcode` procedures, and which can take a positive integer key up to, say, 30 digits long).

```
(define (encode-add int key) (+ int key))
(define (decode-add int key) (- int key))

(define add-key 1043252235123512351235)
(define cipher-3  (encrypt-to-intcode "This is TOP SECRET" encode-add add-
key))
(decrypt-from-intcode cipher-3 decode-add add-key)
;Value: "This is TOP SECRET"
```

Given a plain text message and its corresponding encrypted intcode, we want to think about how long it would take to discover the secret key given a known plain text and cipher text pair. Consider first a ``brute force" procedure similar to that used in the previous exercises, which tries every key until it finds the one used to encrypt the message. What would be the order of growth in computation time as a function of the number of digits *n* in the key?

Fortunately, however, we can do much better for the special case of `encode-add`. You should be able to write an extremely efficient procedure `(quick-crack-add plaintext cipherint)` that returns the secret key used to encrypt `cipherint` using `encode-add`. Hint: you should have a procedure that runs in constant time! (Demonstrating that public knowledge of the encryption procedure in this case is as good as knowing the key.) If you can't think of a constant time method for the special case of `encode-add`, you can write a general purpose but less efficient version.

Try to crack the key for two transmissions encoded with `encode-add`. The first is in `add-plain-1` and `add-cipher-1`, and the second is in `add-plain-2` and `add-cipher-2` (both of these are defined for you in `secret_fall02.scm`.) Caution: if you have an inefficient cracker, it may take a long time to discover the key for `add-cipher-2` (and you may need to know how to terminate a long Scheme computation in the middle using `C-c C-c`).

Problem 6: More encryption with an intcode

Write intcode `encode` and `decode` procedures of your own that work with `encrypt-to-intcode` and `decrypt-from-intcode`, and show that they work.

We would like to have a ``standard" encode and decode procedure (e.g. so they could be coded up and embedded in everyone's email program) that would allow people to have secret keys but wouldn't require everyone to write their own unique secret algorithm. If someone knows your `encode` procedure, is it easy or hard for them to crack your key?

3. Substitution codes

Simple codes, like the Caesar code of the previous part, are often easy to break. Even with 256 characters in our alphabet, there are only 256 possible Caesar encryptions, and decryption is symmetric with encryption. Hence decryption is easy; just write procedures that try out all possible decryptions until the message is converted into readable form. A harder encryption scheme uses a substitution code. Here, the idea is to randomly select a (unique) new letter for

each letter in the alphabet. For example, we might choose to replace “a” with “f”, “b” with “w”, “c” with “z”, “d” with “a” and so on. Note that we must have a one-to-one mapping between characters (that is, two letters in the alphabet cannot be replaced with the same letter in the encryption). Clearly, there are many more possible encryption schemes. In fact, if our alphabet has 256 characters, there are 256! possible different encryptions, or on the order of different choices! Clearly trying out all possible encryptions in order to break a substitution code is not possible. Even if we restrict ourselves to messages composed of the 26 letters of the English alphabet, plus a space, there are 27! (or roughly) different possibilities. In this part of the project, we are going to consider substitution codes, and possible ways of breaking them.

In this section of Part I of this project, we are going to explore ways to decrypt substitution ciphers. We will make some simplifying assumptions in this project. In particular, we restrict ourselves to messages composed only of the 26 lower case letters of the English alphabet (a, b, …, z) plus a space to indicate separation between words. Thus, in the system we will be building, we define

```
(define alphabet
   (list #\a #\b #\c #\d #\e #\f #\g #\h #\i #\j #\k #\l #\m
    #\n #\o #\p #\q #\r #\s #\t #\u #\v #\w #\x #\y #\z #\ ))
```

Note that the elements of this list are “characters”, that is, self-evaluating data objects representing the letter or character described. Note as well that the last element of this list is the character representing a space, which can also be represented by `#\Space`. **Also note that in this part of the project, we are going to be using the data structure of a list; you may find it useful to listen to Lecture 7 before starting this part of the project!**

An example message is just a string of characters chosen from this alphabet, such as

```
"the cia says it does not spy domestically but you can draw your own
conclusions"
```

Note that since this is a string, it is enclosed in the double quotation marks “..”, and a string is constructed from a sequence of characters.

3.1 Substitution code encrypting

To create a code for encrypting such messages we just need to decide on a substitution for the characters in the alphabet. Here is one way of doing this (you should just look over this code to ensure you understand how it works; your goal in the project will focus on decryption):

```
;;;
;;; code for creating an encryption

(define (create-substitution-code alphabet)
  ;; code to create a substitution code for the 26 lower case letters,
  ;; plus a space
  ;; idea is to create a random ordering of the numbers 0 to 26, which
  ;; can then be used to select a match for the current letters
  ;; thus the output is a random ordering of integers (e.g. (i j k ..)
  ;; Hence the 0th letter (a) would be replaced by the i'th letter in
  ;; the alphabet, the 1st letter (b) would be replaced by the j'th letter,
  ;; and so on
```

```
  (define (loopit used total)
    (if (= (length used) total)
        used
        (let ((trial (random total)))
          (if (memq trial used)
              (loopit used total)
              (loopit (cons trial used) total)))))
  (loopit '() (length alphabet)))

;;; create an example encoding
(define my-code (create-substitution-code alphabet))
```

The idea is to associate a number with each letter, representing its place in the list of characters; thus "a" would correspond to "0", "b" to "1" and so on. Then, we create a new list of random numbers, uniquely selected from the set 0, 1, …, n (where n-1 is the total number of characters in the alphabet). By creating such a list, the k'th element in the list can be used to represent the number of the new character to be substituted for the k'th letter in the original alphabet. Thus, if the initial number in the new list is "7", this says that "a" would be replaced by "H"; and if the second number in the new list is "0", then "b" would be replaced by "A", and so on. Note that we will use UPPER CASE characters as our substitutions.

Given such an encoding scheme (e.g., `my-code`), which we will want to keep secret, we can then encode any message by simply substituting the new character for each character in the original message:

```
(define (string-lookup char sequence n)
  (if (null? sequence)
      (error "char not in set" char sequence)
      (if (char=? char (list-ref sequence 0))
          n
          (string-lookup char (cdr sequence) (+ n 1)))))

(define (convert-message-substitute msg code)
  ;; msg is a string of characters, code is a list of integers
  ;; output will be encoded string of characters
  (define (aux done todo)
    (if (null? todo)
        (list->string done)   ; return result as a string
        (let ((next (car todo)))
          (let ((index (string-lookup next alphabet 0)))
            (let ((new-index (list-ref code index)))
              (let ((new-char (list-ref alphabet new-index)))
                (aux (append done (list new-char))
                     (cdr todo)))))))))
  (let ((msg-as-list (string->list msg)))
     (aux '() msg-as-list)))

;;; here is an example

(define simple-test-msg
  (convert-message-substitute "we see everyone and we need to know everything"
                              my-code))
;Value: "pyg yygyxybvdtygqtugpygtyyugndgjtdpgyxybvnostr"
```

3.2. Breaking a substitution code

Now, given that we intercept an encrypted message, how can we break it? Well, the simplest approach is to try to guess the encoding scheme, but with 27! (or) choices, this is clearly impractical. Fortunately, we can take advantage of the structure of the English language. In particular, not all letters in normal English use appear with the same frequency. Here is the frequency of letters based on an analysis of a large body of English text:

Frequency distribution of letters in English text

Source: H. Beker and F. Piper, Cipher Systems, Wiley-Interscience, 1982.

A 8.167
B 1.492
C 2.782
D 4.253
E 12.702
F 2.228
G 2.015
H 6.094
I 6.996
J 0.153
K 0.772
L 4.025
M 2.406
N 6.749
O 7.507
P 1.929
Q 0.095
R 5.987
S 6.327
T 9.056
U 2.758
V 0.978
W 2.360
X 0.150
Y 1.974
Z 0.074

Thus, as you might expect, the most common letter in English is an "E", followed by "T, A, O, I, N…" and here is the list of characters ordered by frequency of use.

```
(define legal-alphabet
  (list #\_ #\E #\T #\A #\O #\I #\N #\S #\H #\R #\D #\L #\C #\U
        #\M #\W #\F #\G #\Y #\P #\B #\V #\K #\J #\X #\Q #\Z))
```

Note that we are going to use upper case letters as our target alphabet for reasons that will be clear shortly, and we use #_ in place of #\Space for similar reasons.

Here is our rough plan. Given an intercepted message, we need to determine the frequency with which characters appear. We can do this by building a data abstraction called a histogram. A histogram is a collection of bins (or buckets) each of which has a key (in our case a character from the alphabet) and a count (in our case the number of times that character has appeared in the message). In the code for this part of the project `(substitution_fall02.scm)` we have provided a partial implementation of a histogram. Once we have built such a histogram, we can use it to determine relative frequencies of characters in the message. If we are lucky, this relative frequency will match up with the data above, i.e., the most common character in the secret message will correspond to the character "e", the next most common will correspond to "t" and so on. Thus, we need to create code to let us find histograms of character frequencies, and then we need to use that information to help us decode the message.

Problem 7: You need to create abstractions for a histogram. A histogram is constructed from bins: create a constructor `make-bin` that takes a key and a count as arguments, and constructs a representation of this coupled information. Associated with this constructor should be two selectors: `bin-key` and `bin-count` for retrieving the associated parts of the data structure.

A histogram is also a data structure. For this, you will need to provide a constructor for making a blank histogram, called `create-hist` which should be a procedure of no arguments. The constructor `extend-hist` should take a bin (as a data abstraction) and a histogram, and return a new histogram, with the bin attached to the original histogram. The corresponding selectors are `next-bin` which takes a histogram as argument and returns the first bin in that histogram, and `rest-bins` which takes a histogram as argument and returns a histogram consisting of all but the first bin of the original histogram. Finally you should create a predicate `empty-hist?` which returns true if the supplied argument is an empty histogram, and otherwise returns false.

Complete the implementation of a histogram. Demonstrate that your abstraction satisfies the associated contract.

Problem 8: To count the frequency of characters in a string, we need to build a histogram of the occurrences of characters. You should do this in stages. First, create a procedure `increment-bin,` which takes a bin as input, and returns a new version of that bin (using your selectors and constructors!) with the count portion increased by 1.

Second, create a procedure `increment-hist` which takes a character and a histogram as input, and returns a new histogram with the following properties. If the original histogram contains a bin with a key matching the character, then the new histogram will have the count portion of the corresponding bin increased by 1, using `increment-bin`. If the original histogram does not contain a bin with a key matching the character, then the new histogram will include a new bin with that character as key, and with a count of 1 (since this is the first time we have seen this character). Note that you will find it useful to use `char=?` to compare two characters.

Third, create a procedure `update-histogram` which takes as input a string (representing an encrypted message) and a histogram. It returns a new histogram, wherein each bin's count value has been increased by the number of times the bin's character appears in the string. For

simplicity, we will convert the string to a list of characters (see the template in the file).

Show examples of your procedure working correctly.

Here is an example of the expected behavior:

```
(define test-hist (update-histogram simple-test-msg (create-hist)))

test-hist
;Value: ((#\p 3) {#\y 11) (#\g 8) (#\Space 1) ... (#\s 1) (#\r 1))
```

Problem 9: Now that you have completed the procedures for creating histograms, you can try it out. Use `update-histogram` to find the frequency of characters in the message `long-test-msg`. What are the values of the bins in the histogram for this message?

3.2.1 Sorting a histogram

In the previous problem, we were able to measure the occurrence of characters in a test message, but to put this into a useful form, we would like to sort the result, so that the most frequent character is first, then the second most frequent, and so on. To do this, we are going to build a simple sorting procedure, built around the following idea.

Start with a sequence of elements (such as the sequence of bins in a histogram, or more simply such as a list of integers). Imagine walking down this sequence in order, keeping track of the best element seen so far (e.g. the largest), a sequence of all the other elements you have looked at so far, in arbitrary order, and the remaining elements of the sequence to be examined. At each step, you can compare the best element to date against the next element in the sequence to be examined, and continue with a new "best" element, a new sequence of things seen so far, and a new sequence of things still to be examined. When you get to the end of the sequence, you should place the "best" element at the beginning of an output sequence, the remainder of which you will get by repeating this process on all the other elements. Of course, we have some freedom in what kinds of elements we are sorting (e.g. a list of histogram bins) and some associated freedom in how we compare those elements to decide which is larger. Though this is a slow process, it will get the job done.

Problem 10: Implement `sort-histogram`, using this idea as applied to a histogram data structure. Demonstrate it working on the histogram you built for `long-test-msg`. An example of the expected behavior is

```
(define sorted-test-hist (sort-histogram test-hist))

sorted-test-hist
;Value: ((#\y 11) (#\g 8) (#\t 5) (#\p 3) … (#\o 1) (#\j 1))
```

The idea of sorting a histogram can be generalized. If we want to sort any sequence of elements, we can use the same general idea, but simply add two new pieces of information: the procedure needed to extract the information to be used in sorting from each element of the sequence (in the

case of our histogram, we used the histogram value) and the procedure to be used to compare two values to decide which comes first.

Problem 11: Write a procedure called `general-sort-histogram`, which takes as arguments a histogram, a means of extracting a value from each entry and a means of comparing two values to decide which comes first.

1. Demonstrate the use of this procedure to sort the histogram as in problem 10.
2. Demonstrate the use of this procedure to sort the histogram by key, rather than by value. For this, you may find the predicate `char>?` to be useful.

Problem 12: Suppose we want to further generalize this idea of sorting, so that we can create different sorters for different purposes. We would like to write a procedure, called `make-sort`, which takes as input a means of selecting values from entries, and a means of comparing those values, and returns a procedure that can then be applied to any histogram to sort it. For example, we would like to do:

```
(define histogram-sorter-by-count (make-sort hist-value >))

(histogram-sorter-by-count test-hist)
```

and have this accomplish the same task as in Problem 10.

Write the procedure `make-sort`, and demonstrate its use in each of the cases of Problem 11.

3.2.2 Cracking a substitution coded message

Now that you have a histogram of the frequency of characters in the message, you are set to try cracking the code. You know that the normal frequency of characters is given by the order shown in `legal-alphabet`. So the most obvious thing to do is to replace the first character in your sorted histogram with the new character for a space #_, then replace the second character in your sorted histogram with the new character #\E, and so on. Note that we are going to use upper case characters to represent replacement characters, to distinguish them from the original characters in the message. Think about why we do this? Why not just use lower case characters?

Problem 13: As a first step in accomplishing this substitution, complete the procedure `substitute`. This procedure should take a string as input, together with an old character and a new character, and replace each instance of the old character with the new. For simplicity, we will convert the input string into a list of characters, and when you have completed the substitution, we will convert the result back into a string. Show examples of your code working, such as

```
(substitute simple-test-msg #\g #\_)
;Value: "py_ yy_yxybvdty_qtu_py_tyyu_nd_jtdp_yxybvnostr"
```

Problem 14: Now take a shot at breaking the message. Complete the definition for `multi-substitute`, so that it takes as input a string representing a message, and a list of substitutions. That is, the second argument should be a list of replacements, where each replacement indicates a character from the coded message, and the character with which to replace it. You should design your own abstraction for a replacement, using the constructor `make-replacement` and selectors `old-char new-char`. Then, using the information about the most common frequency of characters in English and the frequency of characters that you find in `long-test-msg`, try decoding this message. If the most obvious choice (i.e. matching the order of normal frequency to the order you found) doesn't work, try some variations. Show your trials as well as your code for `multi-substitute`.

4. A more clever crack at substitution codes

Clearly even with information about frequency of characters it is going to take too long to crack a code. So how can we do better? The problem is that knowledge of frequency helps, but it is not guaranteed to exactly match the frequency of letters in general. So we would like to have some way to search for alternative orders of frequency.

Imagine the following idea. Suppose that the most common character in our message is an "a", followed by "d" then "f", and so on. We can represent all possible replacements for "a" as a set of nodes in a tree, as shown below.

Thus all the nodes at the first level of this tree represent all possible replacements for the most common character, "a", and they are ordered by the most common occurrences of characters in general (i.e., the space character, _, then E, then T, and so on). If we walk down the leftmost branch of this tree, we reach the node representing the assignment of "a" to "_". Now suppose we consider all the branches below this node. They correspond to assignments of the second most common character in our message (in this case "d") to possible characters from the replacement alphabet, in order of frequency, but excluding the assignment already made (i.e. we don't include "_"). Similarly as we move to the next lower node down the left most branch. On the other hand, if we take the second branch from the top, we assign "a" to the second most common character "E". If we look at the choices below this node, we see that "d" can be assigned to any character except "E", since we have already used that one.

Now one possibility is to create a procedure that searches this tree of possibilities, by walking down the left most branch at each node until we reach the bottom of the tree (where all characters have been assigned a replacement), doing the substitution at each stage. When we reach a leaf of the tree, we can see if the substitution results in a clear message. If not, we can back up one level, go down the next branch, and try again. Of course, if we explore this entire tree, we are back to where we started, as we will explore all possible substitutions. The trick is to note that as we try initial substitutions, we can be a bit more clever.

In particular, suppose we have already substituted the space character "_" for a character in our coded message, and we have substituted a few more characters. Suppose as a consequence that in the new version of the message string, we have a sequence of characters, all of which have been substituted, and which are separated by "_" at the beginning and end, e.g.

```
"ay_WE_sEWEs_sjdhw_FRZD_adnmsyd"
```

Here, we see that the word "WE" occurs in the string, as well as a partial word, "sEWEs" and a second word, "FRZD". Before we go any further in the substitution of characters, we could check whether these words are real. Given that we have a dictionary of possible words, we could see if "WE" and "FRZD" are legal words. If they both are, then we can continue with the search, i.e. going to the next level in the tree. If they are not (and as far as we know, "FRZD" is not a word), then we can stop at this stage and backtrack. That is, there is no sense in considering further substitutions, since the current substitutions lead to garbage, so we can go back to our last choice, and try the next option. In this way, we hope to explore only a small part of this tree of possibilities.

Problem 15: We have provided some machinery to help you implement this idea. The procedure `message-passes-dictionary`, and its associated procedures, will take as input a message, represented as a string using "_" to represent spaces, and using upper case letters to represent translated characters; as well as a starting point, and ending point, and a dictionary (such as `my-dictionary`). This procedure returns true if all the words between the starting point and the ending point in the message string are in the dictionary.

Using this idea, you are to complete the definition for `decode-with-dictionary`. We have provided part of the code for you. This procedures takes several arguments as input: `msg` is a string to be decoded; `histogram` is a histogram of characters whose order is the order in which to try substitution; `tried` represents the set of characters already tried as substitutions for the first element in the histogram; `todo` represents the remaining choices, and `dictionary` is the set of known words. For example, if `new-hist` is a sorted histogram of character frequency in the message, and `my-dictionary` is a list of legal words, then we would evaluate:

```
(decode-with-dictionary long-test-msg new-hist '()
         legal-alphabet my-dictionary)
```

Note how the procedure stops if either the histogram is empty or the value of a bin is 0, since there are no more choices worth considering. Also notice how the procedure returns false if there are no more choices for substitution. Finally, notice how we return a labeled list of the tag "done" and the decoded message when we get a solution, so that at other stages of the search, we can simply return this value.

Your job is to implement the recursive step. Think carefully about this stage. The local variable `next` represents the next stage in the search, in which the next most frequent character in the message has been replaced by the current choice. If this substitution is legal, then you want to continue the search, otherwise you want to try the next option at this stage. Also note that if continuing with the search does not result in a decryption of the message, then you want to back up to the previous stage, and try the next option there.

Show your code, and use it to decode the message `long-test-msg`.

Problem 16: So how much does this help? To demonstrate this, we have provided you with a means of measuring the amount of time taken by a procedure. The last expression in the file

`substitution_fall02.scm` uses a special form called with-timings. Evaluating this form will cause the first procedure to be evaluated and its value returned (in this case it will decode our message for us). The form will also print out the amount of time spent processing the expression, the amount of time spent in garbage collection, and the total amount of time spent. Copy this expression into your Scheme buffer and try running it several times, replacing histogram with whatever variable you use to hold your information about frequency of characters. What are the average times for processing, garbage collection and total time?

Now replace `legal-alphabet` with `legal-unsorted-alphabet`, and do the same thing. How much longer does it take if we don't use the normal frequency of letters to guide our search? Can you even get this to run to completion before you either run out of space in your machine, or you get so bored you kill the process?

Addendum:

Please identify any collaborators on this project, and indicate to what portions they contributed.

Part II

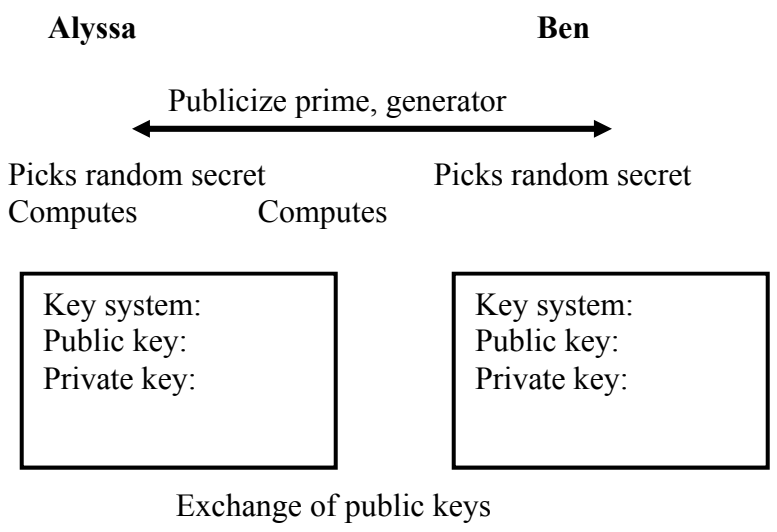
1. Background on public-key cryptography

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]

As we saw in Part 1, 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. 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. Even if everyone in the world knew your public key, no one could decrypt messages sent to you without knowing some additional secret information, which you keep private. Diffie and Hellman called this a *public-key* cryptography system.

1.1 Diffie-Hellman key agreement

The public-key system we will investigate is based on the method of *Diffie-Hellman key agreement*. 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. Here is the basic idea, as illustrated below



[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.

Computes Computes

If Alyssa and Ben wish to communicate 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, taken modulo p, produce all the integers, in some order.)

Remember that modular arithmetic simply means using the remainders of values. For example, given any integer n and any integer p, the value of n mod p is simply the remainder of n after division by p. For example, 10 mod 3 = 1, while 10 mod 7 = 3. Thus, in modular arithmetic with a base of p, only the values 0, 1, …, p-1 will be used, because all other numbers are equivalent to these.

Now, having agreed on p and g, Alyssa picks a secret number x_a and computes.[3] Ben picks a secret number x_b and computes. Alyssa sends y_a to Ben, and Ben sends y_b to Alyssa (alternatively, they both simply publish these numbers; in either case, they can be assumed to be publicly known values). Having obtained y_b, Alyssa now computes and similarly, Ben, having received y_a, computes. But these are the same number because

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 during this protocol, *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_a, and y_b. If p is a large prime, there is no efficient way to use these to compute K.

1.2 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 key exchange scheme above.

Just as above, Alyssa picks a prime p, a generator g, and a secret number x_a, and she computes. She keeps x_a secret to herself and publishes the values (p,g,y_a). 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_a. Next he picks his own secret number x_b. From this, he computes, and he also computes. Ben uses K as the key for encrypting the message to Alyssa using some symmetric algorithm (such as those explored in Part I of the project). He sends the encrypted text to Alyssa, along with y_b.

When Alyssa receives an encrypted message, she takes the y_b part that came with it, and computes. (Remember: Alyssa, and only Alyssa, knows x_a.) She now uses K as the key for decrypting the message.

---

[3] 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, that is, that r and s have the same remainder modulo p.

Other people who see the message can't decrypt it: They know p, g, y_a and y_b, but they can't compute K from this without knowing either x_a or x_b.

This method of public-key encryption is known as *ElGamal key agreement*[4]. 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.

1.3 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 strongly suggest that you read that section of the textbook, to be sure you understand the ideas behind the following code.

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, which gives us a way of finding prime numbers:

```
(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.

```
(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)
```

[4] This method, and the digital signature method discussed below, are based on work during the early 80s by cryptographer Taher ElGamal.

```
  (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 project.

1.3.1 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 $p=2q+1$ where q is also prime. The following theorem lets us compute generators for safe primes:[5]

> If $p=2q+1$ is a safe prime, then for any number either, or, or g is a generator for p.

We can use these ideas as follows:

1. 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.
2. To find a generator for a safe prime $p=2q+1$, we choose a random number g such that 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.[6]

1.3.2 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:

```
(define (generate-key-system digits)
  (let ((p (choose-safe-prime digits))) ; p will be public
    (let ((g (find-generator p)))       ; g will be public
      (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

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

[6] 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.

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.

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`.

1.3.3 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 project, 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-intcode. Given the cipher-intcode and the same key, `symmetric-decrypt` will recover the message text. We could implement these using an encoder and decoder from part 1 of this project. As you discovered, coming up with a good symmetric private key encryption algorithm is itself somewhat tricky. In order to provide you with an ``industrial grade'' encryption system, we will use a particular symmetric cipher called *Blowfish*, which was invented by cryptographer Bruce Schneier.[7] If this is not already part of your system, instructions on how to download Blowfish may be found in the section of the course web site on downloading Scheme.

For example,

```
(symmetric-encrypt "My little secret" 87)
;Value:
217056195070110126076206155091206150169205088225245023075107233145109225

(symmetric-decrypt
217056195070110126076206155091206150169205088225245023075107233145109225 87)
;Value: "My little secret"
```

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 x_b (not to be confused with the x_a of Alyssa's key system, which only Alyssa knows), use Alyssa's y_a raised to the x_bth power modulo p as the shared key for the symmetric cipher, and send the result together with.

```
(define (encrypt message-text public-key)
```

[7] We wont' show you the code for Blowfish, 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.

```
(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-intcode`.

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 text (provided that the key system is the correct one).

1.4 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[8]. 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. This is called the *discrete log problem*.

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, i.e., grows as the number of digits of p. But to scan all the numbers less than p has order of growth . 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 a much smaller amount. Thus, you can quickly ``overpower'' anyone trying to crack your codes by simply making your primes larger.

Are there better algorithms than brute force search? Yes, but they don't help much. There's a method called *Pohlig's rho algorithm,* with order of growth, but this is still exponential in the number of digits in p. The fastest algorithm known is called the *number-field sieve*, and has order of growth

which is not quite exponential in the number of digits, but still grows pretty damned fast.[9]

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 is authentic -- that it really comes from you

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

[9] As a consequence of this, the encryption you are implementing as part of this project is really 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.

and is not a forgery? As we discussed earlier, 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. The first step (the hashing) is performed by a procedure `message-digest`, which takes an arbitrary string and returns a number between 0 and 2^{128} as its result.[10] The second step is to perform a clever computation on the hashed value using our key system.

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

1. The message was signed by the person who knows the secret x (who is presumably the person who distributed the public key).
2. 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, and the secret x_a in her key system as follows:

1. Pick a random integer k between 2 and p-2 such that, and compute the *inverse* of k modulo p-1, i.e., find the number d such that.
2. Compute.
3. Compute
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_a:

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. If so, the signature is good. If not, the signature is bad.[11]

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.

---

[10] 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 in message-digest, called MD5, was invented by Ron Rivest of MIT.

[11] Note that when checking this condition we can compute each term modulo p before multiplying and comparing, because $ab \mod p$ is congruent to $(a \mod p)(b \mod p) \mod p$.

2.1 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. It's easy: just keep picking values for *k* until you find a good one:

```
(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. 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.

2.2 Modular inverses

The number *d* required for the signature must satisfy, 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. Put another way, if we can subtract off some integral number of *m*'s to get to 1. One can show that a solution to this equation exists if and only if (that is to say, *k* and *m* have no common factors). 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)))
```

2.3 Solving $ax + by = 1$

The hard part about `invert-modulo` above is writing the procedure `solve-ax+by=1`. Given a and b, where we require that, how do we find (x,y) that solves $ax + by = 1$? (The requirement on a and b that is crucial.)

Let's consider a simple case first to get a feel for this equation. What if b=0? Then we have to solve $ax + 0y = 1$. But the requirement means we can't be solving the general equation ax=1 for any a, because if b=0 the only way is if a is 1 (because if a was anything else we could ``cancel'' the common factors in a and 0). So in the simple case where our input b=0, our solution is simply $(x,y) = (1,0)$. What if b is not zero? Then a and b have some relationship $a = bq + r$, where we know that r is non-zero (otherwise a and b have the common factor q, which is not allowed). Substituting for a in $ax + by = 1$, we get $(bq + r)x + by = 1$. Gathering all the terms involving b together, we have $b(qx + y) + rx = 1$. But this looks just like our original equation: if we let and, we must now solve. We've transformed our problem into another problem that looks just like it... but with one big difference. The new problem is *smaller* or simpler than the original problem, because the remainder r has to be smaller than the divisor b. This neat recursive trick, whereby we keep reducing the problem until we get back to our trivial problem (where our inputs are a=1 and b=0) is closely related to the recursive GCD algorithm in section 1.2.5 of the text.

You might need one more trick in order to write `solve-ax+by=1`. We need to return *two* values: x and y. Your `solve-ax+by=1` procedure may use 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. If you need to return two values, you can put them together using `(list a b)`, and can then later get the first item back out of it using `car`, and the second item out using `cadr`.

We'll leave to you the details of how to write the actual procedure. This is not easy: you should study closely the discussion above, as well as the integer GCD algorithm in the book to get a feel for how `solve-ax+by=1` might work, and then think very carefully about the appropriate base case and recursive reductions.

3. Programming assignment

Begin by loading the code for this part of the project, namely `public_fall02.scm`.

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.

Problem 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.

Problem 2:

Write the procedures `choose-safe-prime` and `find-generator`. `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. 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.

Problem 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.

Problem 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.

Problem 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. 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=8699, g=4469, y=3222.

Problem 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 3. 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. Now try some public keys of size 4 and of size 5. 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.

Problem 7:

It was a dark and stormy night.[12] 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 Charles Vest. -- A friend

41204

170146241058085224019033029160020215198243090246204123199024174125183067250132
123120206195119048126098008240027021028134242001037050221050140135222225209175
098080195152012185127051110185055028188210219100024100176078085009174094207186
157251051194202119228122016038058254168106017088032166041108234169006237029203
068231130179096143006015065211172082244068153234002022184084051101069167229141
122190027027188153076023064228216083021158144245086202227037169148152014014143
207161182105008077026234224007158092121027213130009229132194254239076235153096
255209041030116112180058033001132225254020110242049243040060108174087155189042
044082119068160133022231118152188245005026209090113010182003088057050170060221
252061047151061135209113093148139212143158037100185076220023217165022015167079
148165081055111189243220106236086151121162182052063141115240088180059247141095
071229204121042182025094066238246187163245190160028004098173087137106131114096
209140004003178083123139204232014057123102178056068234031175199018077188248167
127139002028042060124089024130124032093119039199023213069038221078050254247023
120136135023234230026028228240030145221256249188092203192127155013253178126051
081040172097176008094249119133031047036076022221117089202016226245060136004242
022048153256013232100250251175246215068250050080102172252076086192002069132166
111087057209239082009148107029020010092089196115044143093133095012250199243135
211060150043117031096028130002119102188141001037037155053170226078116242114076
040225185151168056189019011086146091194244118089145048026188130160251049207211
089129122192056186085143101197036220166251030155157118122040219253113058015164
031128046205080079102050162173073058136237017099248201220124170156074236223201
177180129034066087119034170102112070030090202081051196176161055170079134198182
081130014148218133091214023246008068204084239234105001123136226131046143184182
106160138128076159125087068175226048226131027194062200135121122013242215069084
024197115190080202096104055131232036220195005240039212208129082159153243054052
079080144067188204044027164005128150006092005102244171160092126144124092158019
196114190140065032180011069189019133067079173056035225128049120048106021028190
093194
```

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

The MIT President's office is working on better security, but due to reengineering, they have been unable to issue public keys of size greater than 5. Vest's public key and the above cipher-intcode are defined for you as `pk-ex-vest` and `message-ex-vest`. Crack the key and decrypt the message. It sounds exciting, 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.)

Addendum:

Please identify any collaborators on this project, and indicate what portions they contributed to, in accordance with the course policy on collaboration.

[13][1] If you are used to programming in other languages, this approach might concern you – what if we run out of "room" in a fixed size integer? Fortunately, Scheme supports arbitrary precision integers: we can have integers with as many digits as we need – there is no "maximum" size integer (subject to overall machine memory limitations).
[14][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.
[15][3] 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, that is, that r and s have the same remainder modulo p.
[16][4] This method, and the digital signature method discussed below, are based on work during the early 80s by cryptographer Taher ElGamal.
[17][5] We won't include the proof here, since this is not a number theory class. Just take the result on faith.
[18][6] 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.
[19][7] We wont' show you the code for Blowfish, 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.
[20][8] Typical someones who come to mind here are certain government agencies, international crime rings, and MIT undergraduates.
[21][9] As a consequence of this, the encryption you are implementing as part of this project is really 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.
[22][10] 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 in message-digest, called MD5, was invented by Ron Rivest of MIT.

[23][11] Note that when checking this condition we can compute each term modulo p before multiplying and comparing, because $ab \mod p$ is congruent to $(a \mod p)(b \mod p) \mod p$.
[24][12] …with apologies to Baron Bulwer-Lytton and to Radia Perlman, Michael Speciner, and Charles Kaufman, who begin their excellent book *Network Security* like this.