MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.5150/6.5151—Large-Scale Symbolic Systems
Spring 2024

Pset 0

Issued: 7 February 2024 Due: 16 February 2024

Reading:
http://groups.csail.mit.edu/mac/users/gjs/6.5150/dont-panic/
Software Design for Flexibility Prologue; Appendix on Scheme
SICP Chapter 1; SICP Section 2.1; `p0utils.scm`

General Instructions

A problem set for this class generally asks you to do some programming. We usually give you a program to read and extend. You should turn in your extensions, in the form of a paper illustrated with clearly annotated code and executions that demonstrate its effectiveness. We may also ask for short essays explaining an idea. Your answers to these questions must be clear and concise: good logic expressed in good English is required. Thank you.

The purpose of this problem set is to help you get started. Since we will mostly use MIT/GNU Scheme you should install MIT/GNU Scheme on your favorite computer. Scheme is available at `http://www.gnu.org/software/mit-scheme/`. For a bit of help look at the "Don't Panic" memo, referred to by URL in the reading. You will also need to get experience with EDWIN, the EMACS-derived editor that is included in MIT/GNU Scheme. Unless you are very familiar with EMACS we highly recommend that you go through the tutorial that comes with the editor. (A small amount of boring work that will save you enormous amounts of time later.) The exercises in this problem set are designed to help you gain experience with writing and testing procedures and data abstractions in Scheme.

For each problem below, include your code (with identification of the problem number being solved), as well as comments and explanations of your code, and demonstrate your code's functionality against a set of test cases. On occasion, we may provide some example test cases, but you should **always** create and include your own additional, meaningful test cases to ensure that your code works not only on typical inputs, but also on more difficult cases. Get in the habit of writing and running these test cases after **every** procedure you write—no matter how trivial the procedure may seem to you.

Read the entire description before you start working. In this problem set **every sentence in boldface** describes something that you should include in your paper—e.g., procedures to write, test cases to run, or questions to answer. Usually the problem sets will be less formal than this one, but we will hold your hand here to help you get started.

For this problem set, you need to make a directory (folder, in modern usage) on your computer for the problem set (I would personally call it `"~/6.5150/ps00/"`) and a subdirectory called `"code"` (so you would have the directory `"~/6.5150/ps00/code/"`. Download the problem set text into `"~/6.5150/ps00/"` and the code into `"~/6.5150/ps00/code/"`. There will be a file in the code directory named `"load.scm"`. You can load the code you need into the Scheme system

by pointing the Scheme at the code directory, with executing `(cd "~/6.5150/ps00/code/")` and then executing `(load "load")`. The code files you load will contain some extra material, not in the text, that support the problem set.

For example, this problem set includes some provided code in the file `"p0utils.scm"`.

For later problem sets we will be using a more sophisiticated software manager system for getting the code that you need and loading it into the Scheme system. This manager is introduced in Appendix A of the textbook (Software Design for Flexibility). You will not need it for this problem set.

Note: Loading of the support files for this problem set by loading `"load.scm"` reinitializes the top-level environment of Scheme, so you will lose any definitions you have made in that environment. But this will not lose buffers in your EMACS or EDWIN, so your code is not lost. Of course, you should write out the files you are building regularly, probably in your directory `"~/6.5150/ps00/"`.

Introduction

One of the longstanding problems in using encryption to encode messages is that the recipient of the message needs to know the key in order to decrypt the message. Clearly we somehow have to get the key to the participants so they can use it. We can't send the key to them without encrypting *it*, or someone might eavesdrop and get it. But this puts us in an infinite loop: the person getting the key will need to know what second key was used to encrypt the first key, etc.

How can we break this cycle? One amazingly clever idea is called the Diffie-Hellman key agreement protocol.[1] This protocol enables two agents to share a secret, using completely open communication: Alyssa and Ben,[2] who have never interacted before, have an open conversation—anyone, including Eve, the eavesdropper, may hear the complete conversation. At the end of the conversation Alyssa and Ben will possess a shared secret that no one else, including Eve, can know. How is this possible?

The idea is very simple. It is amazing that it had not been thought of hundreds of years ago. Here is the idea:

- Everyone knows two publicly advertised numbers a, p.[3]
- Alyssa chooses a secret number, S_a, that she remembers.
- Ben chooses a secret number, S_b, that he remembers.
- Alyssa computes $P_a = a^{S_a} \pmod{p}$ and announces P_a.
- Ben computes $P_b = a^{S_b} \pmod{p}$ and announces P_b.
- Alyssa computes $x = {P_b}^{S_a} \pmod{p} = a^{S_b S_a} \pmod{p}$.
- Ben computes $y = {P_a}^{S_b} \pmod{p} = a^{S_a S_b} \pmod{p}$.
- But $x = y$, so Alyssa and Ben share a secret.

The secret shared by Alyssa and Ben is the number $x = y$. The reason why Eve cannot obtain the same secret is that for a sufficiently large prime number p, and sufficiently large secret numbers S_a and S_b, given the public information, a, p, $a^{S_a} \pmod{p}$, and $a^{S_b} \pmod{p}$, there is no efficient algorithm known to get $a^{S_a S_b} \pmod{p}$.[4] The secret shared by Alyssa and Ben can then be used as a key for a cryptographic system to encrypt and decrypt messages that cannot be read by Eve.

---

[1] This discovery was the greatest advance in cryptography in 2000 years! See the breakthrough paper: Whitfield Diffie and Martin E. Hellman, "New Directions in Cryptography," in *IEEE Transactions on Information Theory*, vol. **IT-22**, pp 644-654, November, 1976. Apparently, this idea had been thought of a bit earlier, by members of the British intelligence service, but they did not publish it. Tough!

[2] Traditionally, in cryptographic literature, the participants are named Alice and Bob, but in our class they are named Alyssa and Ben.

[3] For this to work p must be a prime number. Systems based on some primes are harder to crack than others. The primes that are usually chosen are of the form $p = 2q + 1$, where q is itself a prime. Such a p is called a *safe prime*. The value of the "primitive root", a, is usually 2 or 5.

[4] The only known algorithm for cracking this depends on the *discrete logarithm problem*. As far as we know, if p, S_a, and S_b are each only a few thousand decimal digits long, the time to crack this problem with classical computers, however fast, is longer than the lifetime of the Earth. But, if quantum computers can be built, cryptosystems like this one will be cracked easily. This was shown by Peter Shor in his landmark paper "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," which appeared in the *SIAM Journal of Computing*, **26**, pp. 1484-1509 (1997).

In this project you will implement a Diffie-Hellman key agreement protocol and a very simple cryptosystem based on the Diffie-Hellman idea. Along the way, you will have to create procedural and data abstractions for solving some subproblems, including:

- modular arithmetic (addition, subtraction, and multiplication mod n)
- fast exponentiation (computing a^n when n is very large)
- generating large random numbers
- testing whether a large number is prime
- picking a large prime number at random
- finding multiplicative inverses mod n

Problem 1: Modular Arithmetic

We will need operators for *modular arithmetic*. Modular arithmetic is arithmetic on a reduced set of integers, in the range 0 to $n-1$ for some fixed integer $n > 0$. The integer n is called the *modulus*. To indicate that we are doing modular arithmetic, we write (mod n) after an expression: for example, $5 + 8 \pmod{12}$.

Addition, subtraction, and multiplication work mostly the same way as in integer arithmetic, except that the result must always be in the range $[0, n-1]$. We guarantee this by taking the remainder of the result after dividing by n. For example, $5 + 8 = 13$ in integer arithmetic, but in mod-12 arithmetic, we take the remainder after dividing 13 by 12, which is 1. Here are some other examples:

$$\begin{aligned} 5+8 &= 1 \pmod{12} \\ 2+3 &= 5 \pmod{12} \\ 6*5 &= 6 \pmod{12} \\ 9-18 &= 3 \pmod{12} \end{aligned}$$

The last example may be somewhat mysterious, since $9 - 18 = -9$ in ordinary integer arithmetic. To determine the correct value of $-9 \pmod{12}$, we need to add or subtract a multiple of 12 that produces a result in the desired range $[0, 11]$. More formally, we need to find integers a and b such that $-9 = 12a + b$, where $0 \leq b \leq 11$. By choosing $a = -1$, we have $-9 = -12 + b$ which solves for $b = 3$.

Scheme has two operators for computing remainders after division: `remainder` and `modulo`. **Try applying each operator to some integers.** For example:

```
(modulo 13 8)       ; -> ?
(remainder 13 8)    ; -> ?
(modulo -13 8)      ; -> ?
(remainder -13 8)   ; -> ?
(modulo -13 -8)     ; -> ?
(remainder -13 -8)  ; -> ?
```

What is the difference between `remainder` and `modulo`? Which one is the best choice for implementing modular arithmetic as described above? Include your test results and your answers to these questions in a comment in your solution.

Write procedures for addition, subtraction, and multiplication modulo n. Each procedure should take three parameters: the values to combine, a and b, and the modulus n. For example, the expression `(+mod 7 5 8)` should compute $7 + 5 \pmod 8$. Here are three skeleton procedures to get you started:

```
(define +mod
  (lambda (a b n)
    YOUR-CODE-HERE))
```

```
(define -mod
  (lambda (a b n)
    YOUR-CODE-HERE))
```

```
(define *mod
  (lambda (a b n)
    YOUR-CODE-HERE))
```

Test your code for at least the following cases.

```
(+mod 7 5 8)     ; -> 4
(+mod 10 10 3)   ; -> 2
(-mod 5 12 2)    ; -> 1
(*mod 6 6 9)     ; -> 0
(+mod 99 99 100) ; -> ?
(*mod 50 -3 100) ; -> ?
```

Notice that the procedures you wrote for modular addition, subtraction, and multiplication are almost all the same. Time for abstraction! Also, when working in modular arithmetic we often do most of our work in a particular modulus. Thus it is helpful to be able to define particular operators for each modulus. For example, we might define addition, subtraction and multiplication modulo 12 as follows:

```
(define +m12 (modular 12 +))
(define -m12 (modular 12 -))
(define *m12 (modular 12 *))
```

And we would be able to use these operators:

```
(-m12 (*m12 (+m12 5 8) 3) 7) ; -> 8
```

Fill in the following code fragment to complete the procedure `modular` that allows us to make the binary modular operators. `modular` should take a modulus and an ordinary arithmetic procedure and produce a modular arithmetic procedure. For example, `((modular 8 +) 7 5)` should compute $(7 + 5) \pmod 8 = 4$.

```
(define modular
  (lambda (modulus op)
    (lambda (a1 a2)
      YOUR-CODE-HERE)))
```

Test your code for at least the following cases.

```
((modular 17 +) 13 11)   ; -> 7
((modular 17 -) 13 11)   ; -> 2
((modular 17 *) 13 11)   ; -> 7
```

Quite often we will use an alternative syntax for defining procedures. The following definitions are equivalent:

```
(define modular
  (lambda (modulus op)
    (lambda (a1 a2)
      BODY OF THE PROCEDURE)))

(define (modular modulus op)
  (lambda (a1 a2)
    BODY OF THE PROCEDURE))

(define (modular modulus op)
  (define (the-operator a1 a2)
    BODY OF THE PROCEDURE)
  the-operator)

(define ((modular modulus op) a1 a2)
  BODY OF THE PROCEDURE)
```

The choice of definition syntax is a matter of style. You may want to emphasize some particular aspect for the reader to notice. After all, much of the value of computer language is communicating to human readers.

Note about preparing your submission

You should run the tutorial for the editor so you understand how to work in the MIT/GNU Scheme environment. As you are working on your project, you may want to plan ahead for the document that you will submit as your work. We assume that you will start a file for this problem set (ps00) with some name like `ps00.scm` by typing the command `C-x C-f ps00.scm` to EDWIN. This will make an editing buffer that will be in Scheme mode, helping you with balancing parentheses and adjusting indentation. When you save the buffer (with `C-x C-s`) the buffer will be written out as the file with name `ps00.scm` in your directory. You will also have a buffer named *scheme*, which is running a read-eval-print loop. (You can show both buffers simultaneously as two windows and move between them easily.) You will compose your answers to the problems in the `ps00.scm` directory. You can comment out results of evaluation, explanations, and discussion with comment brackets `#|` and `|#`. When you save the file, you get something you can read back and work on again, or print and submit for us to read. If you comment out only the results and the commentary text then you can execute the contents of the file directly. The Scheme interpreter will ignore the commented material.

We will help you to get started.

Problem 2: Raising a Number to a Power

This problem is easy if you've read Chapter 1 in SICP.

Recall that the basic operation in Diffie-Hellman key agreement is raising a number to a power (modulo n).

Here's a simple procedure that computes $a^b \pmod{n}$ by multiplying a by itself b times. Note that it uses modular arithmetic operations, namely `*mod`, rather than `*`:

```
(define (slow-exptmod n)
  (let ((*mod (modular n *)))
    (define (em a b)
      (if (= b 0)
          1
          (*mod a (em a (- b 1)))))
    em))
```

Answer these questions in comments in your file: **What is the order of growth in time of `slow-exptmod`? What is its order of growth in space? Does `slow-exptmod` use an iterative algorithm or a recursive algorithm?** Measure time and space the same way we did in lecture: time by counting the number of primitive operations that the computation uses, and space by counting the maximum number of pending operations.

As its name suggests, `slow-exptmod` isn't going to be fast enough for our purposes. We can make a faster procedure using the trick of *repeated squaring*.[5] Compare these two ways of computing 3^8. The left column shows how `slow-exptmod` would do it, and the right column uses repeated squaring:

```
3^0 = 1            3^0 = 1
3^1 = 3^0*3 = 3    3^1 = 1 * 3 = 3
3^2 = 3^1*3 = 9    3^2 = (3^1)*(3^1) = 9
3^3 = 3^2*3 = 27   3^4 = (3^2)*(3^2) = 81
3^4 = 3^3*3 = 81   3^8 = (3^4)*(3^4) = 6561
3^5 = 3^4*3 = 243
3^6 = 3^5*3 = 729
3^7 = 3^6*3 = 2187
3^8 = 3^7*3 = 6561
```

[5] This technique is discussed in section 1.2.4 of SICP.

Fill in the details of the procedure `exptmod` that computes $a^b \pmod{n}$ using repeated squaring. You should use your modular arithmetic operations in your solution. Do not use `expt` or `slow-exptmod` in your solution.

```
(define (exptmod p)
  (let ((mod* (modular p *)))
    (define (square x)
      (mod* x x))
    (define (em base exponent)
      YOUR-CODE-HERE)
    em))
```

Test your code for at least the following cases:

```
((exptmod 10) 2 0)   ; -> 1
((exptmod 10) 2 3)   ; -> 8
((exptmod 10) 3 4)   ; -> 1
((exptmod 100) 2 15) ; -> 68
((exptmod 100) -5 3) ; -> 75
```

Answer these questions in comments in your file: **What is the order of growth in time of your implementation of `exptmod`? What is its order of growth in space? Does your `exptmod` use an iterative algorithm or a recursive algorithm?**

Problem 3: Large Random Numbers

We will need a source of random numbers for the modulus and for the individual secrets. Scheme has a builtin procedure `random` that takes a single integer $n > 0$ and returns a random integer in the range $[0, n-1]$. For example, here are the results of a few calls to `random`:

```
(random 10) ; -> 1
(random 10) ; -> 6
(random 10) ; -> 6
(random 10) ; -> 0
(random 10) ; -> 7
```

Unfortunately, the implementation of `random` in some versions of Scheme is not sufficient for our purposes, because its parameter n can be no larger than $2^{31} - 1$, which is only a couple billion. (The random number generator in MIT/GNU Scheme does not have this limitation.) But we're going to want random numbers at least as large as 2^{128}, if not larger.

Just for practice, let's proceed as if we did not have a fully competent Scheme. So our goal for this problem is to make a procedure `big-random` that behaves like `random`, taking a parameter $n > 0$ and returning a random number in $[0, n-1]$, but that doesn't have any limit on the size of n.

Start by writing a procedure `random-k-digit-number` that takes an integer $k > 0$ and returns a random k-digit number. You should choose each digit using the builtin procedure

`random`, then construct the k-digit number by putting those digits together. For example, if you generate two digits a and b, then you can form a two-digit number by computing $10a + b$.

Test your procedure on at least the following test cases:

```
(random-k-digit-number 1)  ; -> ?        (1 digit)
(random-k-digit-number 3)  ; -> ???      (1-3 digits)
(random-k-digit-number 3)  ; -> ???      (is it different?)
(random-k-digit-number 50) ; -> ???...   (1-50 digits)
```

Note that `random-k-digit-number` may return a number shorter than k digits, since the leading digits of the number may turn out to be 0. The result will be a random number in the range $[0, 10^k - 1]$.

With `random-k-digit-number`, we can now generate arbitrarily large random numbers. But we want `big-random` to take a maximum n, not a digit count. So we need a way to use `random-k-digit-number` to generate random numbers in the range $[0, n - 1]$. We'll do this by first generating a random number with the same number of digits as n, then ensuring that this number is less than n.

Write a procedure `count-digits` that takes an integer $n > 0$ and returns the number of digits in its decimal representation. One way to do this is to count digits by repeatedly dividing by 10. **Test your procedure on at least the following test cases:**

```
(count-digits 3)         ; -> 1
(count-digits 2007)      ; -> 4
(count-digits 123456789) ; -> 9
```

We're almost ready to write `big-random`, but there's one more problem. Suppose somebody calls `(big-random 500)`, expecting to get back a number between 0 and 499. We use `count-digits` to determine that 500 has 3 digits, and then use `random-k-digit-number` to generate a random 3-digit number. If that number is less than 500, then great, we can return it as the result of `big-random`. But what if the number is greater than or equal to 500? Then we just pick another random 3-digit number. We repeat this process until we get a number that's in the range we want.

This is a simple example of a *probabilistic* algorithm—an algorithm that depends on random chance. A probabilistic algorithm isn't *guaranteed* to succeed, but its probability of success can be made as high as we need it to be. In this case, it's possible for the algorithm to have a really bad string of luck, and repeatedly pick 3-digit numbers higher than 500. But the chance of this happening, say, 1000 times in a row is the same as the chance of flipping heads on a coin 1000 times in a row, which is less than the probability that cosmic rays will cause your computer to make an error in running your Scheme code. So, in practice, as long as we keep picking random numbers (and assuming `random-k-digit-number` really is random), this probabilistic algorithm is just as likely to succeed as a deterministic algorithm.

Use this approach to write a procedure `big-random` that takes an integer $n > 0$ and returns a random number from 0 to $n - 1$. Your procedure should handle arbitrarily large n.

Since your procedure needs to generate a random number, test it for a property, and then return it if it satisfies that property.

Test `big-random` on at least these test cases:

```
(big-random 100)          ; -> ??  (1-2 digit number)
(big-random 100)          ; -> ??  (is it different?)
(big-random 1)            ; -> 0
(big-random 1)            ; -> 0     (should be always 0)
(big-random (expt 10 40)) ; -> ????...  (roughly 40-digit number)
```

Problem 4: Prime Numbers

This problem is easy if you've read Chapter 1 in SICP.

In the Diffie-Hellman key agreement protocol we will need a very large prime number for the modulus p. Let's go!

We'll start by developing a test for whether a number is prime. By definition, a prime number is not divisible by any integer other than itself and 1. This leads directly to a simple way to test whether n is prime, by testing every number less than n to see if it's a factor of n:

```
(define (slow-prime? n)
  (define (test-factors n k)
    (cond ((>= k n) #t)
          ((= (remainder n k) 0) #f)
          (else (test-factors n (+ k 1)))))
  (if (< n 2)
      #f
      (test-factors n 2)))
```

Answer these questions in comments in your file: **What is the order of growth in time of `slow-prime?` What is its order of growth in space? Does `slow-prime?` use an iterative algorithm or a recursive algorithm?**

Unfortunately `slow-prime?` is too slow. Ben Bitdiddle proposes two optimizations:

- "We only have to check factors less than or equal to $\sqrt{n}$." **How would this affect the order of growth in time?** Note that we're not asking you to write Scheme code implementing Ben's suggestion; just think about it and answer this question as a comment in your file.
- "We only have to check odd factors (and 2, as a special case)." **How would this affect the order of growth in time?**

Ben's improvements won't be enough for us to test very large numbers for primality; we need a completely different algorithm. For faster prime number testing, we turn to a beautiful result about modular arithmetic. Fermat's Little Theorem states that if p is prime, then $a^p = a \pmod{p}$ for all a. In other words, if p is prime, then we can take any integer a, raise it to the power p, take the remainder after dividing by p, and we'll get a back again (modulo p). **Test Fermat's Little Theorem using your `exptmod` procedure and a few suitable choices of a and p.** Include your tests in your answer file.

The converse of the theorem doesn't hold, unfortunately; if p is composite (not prime), then it isn't always true that $a^p \neq a \pmod{p}$. But it's true often enough that we can use this theorem as the basis for a *probabilistic* algorithm that tests whether a number p is prime:

1. Pick a random integer a in the range $[0, p-1]$, using your `big-random` procedure.
2. Test whether $a^p = a \pmod{p}$.
3. If *not*, then p is definitely not prime, by Fermat's Little Theorem.
4. If so, then p may or may not be prime. Repeat the test with a new random integer a.

If you pick enough random numbers a, and all of them pass the test of Fermat's Little Theorem, then you have strong confidence that p is prime.[6]

Write a procedure `prime?` that uses this technique to test whether its parameter p is prime. Your procedure should test at least 20 random values of a before assuming that p is prime. In fact, it's good practice to define a name for this constant, `prime-test-iterations`, since it may need to be adjusted later.

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

(define prime?
  (lambda (p)
    YOUR-CODE-HERE))
```

Test `prime?` on at least the following test cases:

```
(prime? 2) ; -> #t
(prime? 4) ; -> #f
(prime? 1) ; -> #f
(prime? 0) ; -> #f
(prime? 200) ; -> ?
(prime? 199) ; -> ?
```

Answer these questions in comments in your file: **What is the order of growth in time of your implementation of `prime?` What is its order of growth in space?** Be sure to take the calls to `exptmod` into account when answering these questions. **Does `prime?` use an iterative algorithm or a recursive algorithm?**

[6] But not certainty. Some composite numbers p, called Carmichael numbers, pass the test for almost all a. Fortunately Carmichael numbers are rare. See `http://mathworld.wolfram.com/CarmichaelNumber.html` for more information.

Problem 5: Random Primes

Fortunately, prime numbers are fairly common,[7] so we can find them by a probabilistic *generate and test* strategy. We'll guess a number at random, and then test whether it's prime. If not, we'll pick another random number, and repeat.

Write a procedure `random-k-digit-prime` that returns a random prime number with about k digits.

```
(define random-k-digit-prime
  (lambda (k)
    YOUR-CODE-HERE))
```

Note that your procedure is *probabilistic* – i.e., most of the time it successfully returns a prime number, but sometimes it may fail. **In what ways can your random-prime procedure fail?** Answer this question in a comment. Some kinds of failure are better than others.

Test `random-k-digit-prime` on at least the test cases below.

```
(random-k-digit-prime 1)
(random-k-digit-prime 2)
(random-k-digit-prime 10)
(count-digits (random-k-digit-prime 100))   ; Not always 100.
(count-digits (random-k-digit-prime 100))
```

Problem 6: Multiplicative Inverses

We will also need a way to find *multiplicative inverses* in modular arithmetic. Given an integer e and a modulus n, we want to find d such that $ed = 1 \pmod{n}$. In rational or real arithmetic, d would be $1/e$, but we want an integer. The multiplicative inverse of e exists if and only if e and n have no common factors; in other words, only if the greatest common divisor (GCD) of e and n is 1. (The GCD algorithm is described in section 1.2.5 of the text, but you can use the Scheme builtin procedure `gcd` for this project.)

Here's how we find the multiplicative inverse d. We want $ed = 1 \pmod{n}$, which means that $ed + nk = 1$ for some integer k. So we'll write a procedure that solves the general equation $ax + by = 1$, where a and b are given, x and y are variables, and all of these values are integers. We'll use this procedure to solve $ed + nk = 1$ for d and k. Then we can throw away k and simply return d.

So we've reduced the problem to solving $ax + by = 1$ for x and y, given a and b. We assume that all of these terms are integers, and that a and b are greater than 0. Let q be the quotient of dividing a by b, and let r be the remainder. (Scheme has builtin procedures `quotient` and `remainder` for this purpose.) Then $a = qb + r$. Now consider the special case when $r = 1$: then $a = qb + 1$, which means $a \cdot 1 + b(-q) = 1$, so we have our solution; $x = 1$ and $y = -q$. Otherwise, if $r \neq 1$, recursively

[7] Another famous result, the Prime Number Theorem, holds that the number of primes less than n is roughly $n/\ln n$. For example, if we pick a random 40-digit number, then the probability that it's prime is roughly $1/\ln 10^{40}$, or 1/92. See `http://mathworld.wolfram.com/PrimeNumberTheorem.html`.

solve the equation $bx' + ry' = 1$, and use the solution (x', y') to find the solution to the original equation $ax + by = 1$:

$$1 = bx' + ry' = bx' + (a - qb)y' = ay' + b(x' - qy')$$

Write a procedure `ax+by=1` that solves for x and y using the approach outlined above. Your procedure should return (x, y) as a list.

```
(define ax+by=1
  (lambda (a b)
    YOUR-CODE-HERE))
```

Test `ax+by=1` on at least the test cases below. Note that it will only succeed if $\gcd(a, b) = 1$, so don't expect it to work otherwise.

```
(ax+by=1 17 13) ; -> (-3 4)    17*-3 + 13*4 = 1
(ax+by=1 7 3)   ; -> (1 -2)     7*1  + 3*-2 = 1
(ax+by=1 10 27) ; -> (-8 3)    10*-8 + 3*27 = 1
```

Now write a procedure `inversemod` that finds the multiplicative inverse of e modulo n, using `ax+by=1`. Note that before trying to invert e, your procedure should ensure that $\gcd(e, n) = 1$. You can use the builtin Scheme procedure `gcd` to test this, and the Scheme builtin procedure `error` to signal an error if the test fails.

```
(define (inversemod n)
  (lambda (e)
    YOUR-CODE-HERE))
```

Test `inversemod` on at least the test cases below.

```
((inversemod 11) 5) ; -> 9            5*9 = 45 = 1 (mod 11)
((inversemod 11) 9) ; -> 5
((inversemod 11) 7) ; -> 8            7*8 = 56 = 1 (mod 11)
((inversemod 12) 5) ; -> 5            5*5 = 25 = 1 (mod 12)
((inversemod 12) 8) ; -> error        gcd(8,12)=4, so no inverse exists
((inversemod 101) (random-k-digit-prime 2))
         -> ? (test your answer with *mod)
```

Problem 7: The ElGamal Public-Key Cryptosystem

The ElGamal public-key cryptosystem[8] is based on the Diffie-Hellman key-agreement protocol. It is used in PGP[9] and other common cryptography applications.

[8] See: Taher ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," in *IEEE Transactions on Information Theory*, vol. **IT-31**, no. 4, pp. 469–472, 1985.

[9] PGP was invented by Phil Zimmerman. It is best described in Simson Garfinkel, **PGP: Pretty Good Privacy**, O'Reilly & Associates.

A public-key cryptosystem does not use the same key for both encrypting and decrypting messages. There are two keys: a *public* key used for encrypting, and a secret *private* key for decrypting. The public key can be known by anybody; in fact, it's often put in a public directory, so that Alyssa can send secret messages to Ben simply by looking up his public key in the directory. His private key d must be known only to Ben, so that nobody else can decrypt messages sent to him. Your MIT browser certificate contains a public-key/private-key pair.

Here is how the ElGamal system works: Each receiver chooses a secret private key, S. The receiver publishes a public key, which gives a, p, and $P = a^S \pmod{p}$. The receiver also supplies a procedure that a sender can call with the ciphertext. Given a message, m, the sender chooses his own secret, T, and sends a ciphertext that has two components, (x, y), where $x = a^T \pmod{p}$ and $y = mP^T \pmod{p}$. The Diffie-Hellman shared secret is $x^S \pmod{p} = a^{ST} \pmod{p} = P^T \pmod{p}$. So the receiver decrypts the message by computing $m = y(x^S)^{-1} \pmod{p}$.

In the file `p0utils.scm` you will find code for making a receiver:

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

This program is intended to work with a procedure that you will write that sends messages to the receiver:

```
(define (eg-send-message message receiver)
  YOUR CODE HERE)
```

An example of the way this pair of procedures can be used is:

```
(define dh-system (public-dh-system 100))

(define Alyssa (eg-receiver dh-system))

(eg-send-message "Hi there." Alyssa)
;Value: "Hi there."
```

In the file `p0utils.scm` you will also find definitions of all of the data abstractions that you will need for this exercise, including the code that translates strings to integers and integers to strings. For example:

```
(string->integer "hello") ; -> 1578072040808
(integer->string 1578072040808) ; -> "hello"

(string->integer "") ; -> 1
(integer->string 1) ; -> ""
```

In order to use the provided code in your own file without having to copy and paste it, put the provided file `p0utils.scm` in the same folder as your own file and add the expression `(load "p0utils.scm")` at the top of your file. The `load` procedure evaluates all the definitions and expressions in `p0utils.scm`, so that you can use `string->integer` and `integer->string` in your own code.

We represent the message we want to send as an integer.[10]

Write a procedure `eg-send-message` that takes a message string and a receiver and calls the receiver's decryption procedure, encrypted with the receiver's public key. The receiver we provided will decrypt the original message and return the string, if your sender is correctly written.

Demonstrate your `eg-send-message` with a few short strings sent from Ben to Alyssa.

What is the longest string you can send that will be correctly decrypted with a 100 digit system? You will find that it is not too long!

Problem 8: "Man In The Middle" Attack

One way we can spy on a cryptosystem is by a "man in the middle" attack. For example, Eve can shadow Alyssa and observe all of the messages sent to her using the following code.

```
(define (Eve receiver)
  (let ((receiver-public-key
         (eg-receiver-public-key receiver))
        (receiver-decryption-procedure
         (eg-receiver-decryption-procedure receiver)))
    (let ((my-spying-procedure
           (lambda (ciphertext)
             (write ciphertext)
             (newline)
             (receiver-decryption-procedure ciphertext))))
      (eg-make-receiver receiver-public-key
                        my-spying-procedure))))
```

[10] There are several ways to do this. One way is to encrypt one byte of the message at a time, using its numeric value as the integer representation. Another way is to treat the bytes of the message as digits in a base-256 number, so that the entire message becomes the integer. We'll use the latter approach in this project. Practical cryptosystems do something in between, converting chunks of the message as integers and encrypting one chunk at a time.

We provide you with this Eve code. You can try it as follows. After defining Alyssa, execute:

```
(define Alyssa (Eve Alyssa))
```

Thus Eve will catch and print every message that Ben tries to send to Alyssa. However, Eve will not be able to interpret the message, because it is encrypted. However, Eve can still make trouble for the Ben-Alyssa relationship using this trick. How?

Modify the `Eve` program to make it possible for Eve to make trouble in the relationship. Explain and demonstrate your nasty trick.