MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2007

Project 1

Release date: 16th February, 2007
Due date: 2 March, 2007, at 6pm

Purpose

The purpose of this project is for you to gain experience with writing and testing simple procedures and data abstractions. You should create a file with your project solutions for uploading to the submission site on the 6.001 on-line tutor. 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.

Additional guidelines for project submission are available under the "How to write up a project" link off of the course web page.

Read the entire project description below before you start working. **Every sentence in boldface** describes something that you should include in your final handin — e.g., procedures to write, test cases to run, or questions to answer.

This project includes some provided code in the file `rsa.scm`, which can be obtained from the Projects link on the course web page.

Scenario

Alyssa P. Hacker and Ben Bitdiddle have founded a startup company, SCrypto, that will provide cryptography services for Scheme programmers, allowing them to encode and decode secret messages using a code that is very hard to break. The first product they decide to create is an implementation of an encryption algorithm called *RSA*.

Named after its inventors, Rivest, Shamir, and Adelman (Prof. Rivest is still here at MIT), RSA is a *public-key* encryption algorithm. This means that instead of using the same key for both encrypting and decrypting messages, RSA has two keys: a *public* key e used for encrypting, and a *private* key d 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 e in the directory. His private key d must be known only to Ben, so that nobody else can decrypt messages sent to him. RSA is widely used for web site security today; your MIT browser certificate contains a public-key/private-key pair.

The basic idea of RSA is wonderfully simple. First, we represent the message we want to send as an integer.[1] Then, to encrypt a message m (represented as a number), we use the public key e as an exponent:

$$c = m^e \pmod{n}$$

The resulting number c is the encrypted message. (The notation "(mod n)" means take the remainder after dividing by a certain number n, which will be explained later.)

To decrypt c, we raise it to the power of the private key d:

$$c^d \pmod{n}$$

which recovers the original message m.

RSA is simple, but the devil is in the details. The public key e and private key d must be chosen so that for all integers m, it is the case that

$$(m^e)^d = m \pmod{n}$$

In other words, encrypting and then decrypting must always give back the original message. Furthermore, since we want the encryption to be hard to break, we need to make sure that the private key d is very hard to discover, even if you know the public key e. The beauty of RSA lies in these details.

Your job in this project is to implement RSA. By the end of the project, you will have written procedures that generate a public-key/private-key pair, and procedures that encrypt and decrypt messages. Along the way, you will have to create procedural and data abstractions for solving the subproblems involved in implementing RSA, 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

[1] 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 RSA does something in between, treating chunks of the message as integers and encrypting one chunk at a time.

Problem 1: Modular Arithmetic

The first thing we need for implementing RSA are 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 actually 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 $7 + 5 \pmod{8}$.

```
(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) ; -> ?
```

Note about preparing your submission

As you are working on your project, you may want to plan ahead for the document that you will submit online as your work. We assume that you are writing your procedures in the *definitions* window and then using the *Run* button to place those definitions into the Scheme environment so that you can try some examples in the *interactions* window. When you have completed your set of test cases, one easy way to proceed is to make a copy of these interactions.

To do this, select them by pressing the left mouse button with the cursor positioned at one end of the sequence, then drag the mouse to the other end and release the button. This will highlight the text, which you can then copy with Ctrl-C. If you then click the mouse on the *definitions* window, you can insert the copied text into that window by placing the cursor and typing Ctrl-V to paste it. Now highlight the text you just inserted, click on the Scheme menu, and click on *Comment out with semicolons*. This converts the text into a comment, so that the Scheme interpreter will ignore it when you press Run.

You should also insert your own comments on lines preceded by semicolons. Your answers to the **questions in boldface** that appear throughout this project should be included in your file in this way. Your procedures should *not* be commented out, of course, so that your TA can run your file to try out your code.

When you are finally ready to submit this file, you can save a version in **text mode**, using some appropriate file name, and upload it on the tutor site.

Problem 2: Raising a Number to a Power

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

Recall that the basic operation in RSA encryption and decryption is raising a number to a power (modulo n). For example, to encrypt a message m using public key e, we need to compute $m^e \pmod{n}$.

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
  (lambda (a b n)
    (if (= b 0)
        1
        (*mod a (slow-exptmod a (- b 1) n) n))))
```

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 it faster using the trick of *repeated squaring.*[2] 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
```

Write a procedure `exptmod` that computes $a^b \pmod{n}$ using repeated squaring. You should use your modular arithmetic operations, particularly `*mod`, in your solution. Do not use `expt` or `slow-exptmod` in your solution.

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

Test your code for at least the following cases:

[2] This technique was also shown in lecture for computing `expt`, and is also discussed in section 1.2.4 of the textbook.

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

In order to generate RSA keys at random, we will need a source of random numbers. 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 DrScheme is not sufficient for our purposes, because its parameter n can be no larger than $2^{31} - 1$, which is only a couple billion. We're going to want random numbers at least as large as 2^{128}, if not larger.

So our goal for this problem is 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. The basic idea 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, you may find the `let` special form useful. `Let` evaluates one or more subexpressions and assigns names to each one, then evaluates its final expression (using those names) and returns that as its result. For example, the code below computes the square of a random number, using `let` to name the number:

```
(let ((n (random 100)))
  (* n n))
```

Note that `(* (random 100) (random 100))` does not do the same thing!

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 the textbook.

In order for RSA encryption and decryption to work correctly, the modulus n must be the product of two prime numbers, p and q. In order for it to be secure against easy cracking, these prime numbers must be large. So we need a way to find large prime numbers.

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 test-factors
  (lambda (n k)
    (cond ((>= k n) #t)
          ((= (remainder n k) 0) #f)
          (else (test-factors n (+ k 1))))))

(define slow-prime?
  (lambda (n)
    (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.[3]

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?**

Problem 5: Random Primes

Now we're ready to find the large prime numbers p and q that we need for RSA. Fortunately, prime numbers are fairly common[4], 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-prime` that returns a random prime number p. It should take a parameter n that limits the size of the prime returned, so that $p < n$.

[3] 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.

[4] 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`.

```
(define random-prime
  (lambda (n)
    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-prime` on at least the test cases below.

```
(random-prime 3) ; -> 2
(random-prime 3) ; -> 2  (must be always 2)
(random-prime 100) ; -> ?
(random-prime 100) ; -> ?  (is it different?)
(random-prime 100000) ; -> ?
```

Problem 6: Multiplicative Inverses

Recall that RSA needs two keys e and d such that $(m^e)^d = m \pmod{n}$ for every possible message m. It turns out that this works if e and d are chosen so that $ed = 1 \pmod{(p-1)(q-1)}$.

So we 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. Otherwise, if $r \neq 1$, recursively 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 `inverse-mod` 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 inverse-mod
  (lambda (e n)
    YOUR-CODE-HERE))
```

Test `inverse-mod` on at least the test cases below.

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

Problem 7: RSA

We're now ready to implement RSA. First, we need to represent public and private keys. We'll use the same abstraction for both kinds of keys. A key will consist of an exponent (e or d) and a modulus (n).

Write a data abstraction for keys, including a constructor `make-key` and two selectors `get-modulus` and `get-exponent`.

Next, we need to be able to generate public keys and private keys. First choose two random primes p and q, and let the modulus $n = pq$. The value e can be any number such that $\gcd(e, (p-1)(q-1)) = 1$. A random number less than n that meets this criterion is fine (although it may take a few tries to find it). Finally, the value d should be the multiplicative inverse of $e \pmod{(p-1)(q-1)}$.

Write a procedure `random-keypair` that generates a random public key and its corresponding private key (represented using your key abstraction) and returns them as a list. Your procedure should take a single parameter m, and produce a key pair capable of encoding any message in the range $[0, m-1]$. That means, in particular, that the modulus n of the generated key pair must be at least as large as m.

```
(define random-keypair
  (lambda (m)
    YOUR-CODE-HERE))
```

Write a procedure `rsa` that takes a key and a message integer and returns the encrypted or decrypted form of the message. Recall that in RSA, both encryption and decryption do the same thing, raising the message integer to a power. Use your key abstraction to implement this procedure; don't violate the abstraction boundary.

```
(define rsa
  (lambda (key message)
    YOUR-CODE-HERE))
```

Test your `rsa` procedure on several generated key pairs and several message integers, to ensure that it encrypts and decrypts properly. What happens when you try to encrypt and decrypt a message integer which is too large for the key – i.e., larger than the modulus n?

Finally, let's tie it all together by writing procedures that can encrypt and decrypt message *strings*, not just integers. The provided code for this project includes two procedures, `string->integer` and `integer->string`, that convert a string into an integer and vice versa. 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 rsa.scm in the same folder as your own file and add the expression `(load "rsa.scm")` at the top of your file. The `load` procedure evaluates all the definitions and expressions in rsa.scm, so that you can use `string->integer` and `integer->string` in your own code.

Write a procedure `encrypt` that takes a message string and a public key and returns the message as an integer, encrypted using the public key. Also write its counterpart procedure, `decrypt`, that takes an encrypted message integer and a private key and decodes it to produce the original string. Note that although the original message is a string, the encrypted message should be represented as an integer.

```
(define encrypt
  (lambda (public-key string)
    YOUR-CODE-HERE))

(define decrypt
  (lambda (private-key encrypted-message)
    YOUR-CODE-HERE))
```

Demonstrate that your program can encrypt and decrypt the following messages. Use a randomly generated key pair that is big enough to encrypt all the messages below.

```
"hello, world!"
""
"meet me in 6.001 lab"
```