CSE 341, Autumn 2010
Assignment #2 - ML: RSA Encryption (100 points)
Due: Thursday, October 14, 2010, 11:30 PM

This assignment uses the basic building blocks of ML, along with patterns and the `let` construct. You are allowed to use any constructs from Chapters 1-3 of the Ullman text. Place all of your functions and variables into a file called `rsa.sml` and submit it from the class web page. Those doing the bonus should also submit a second file called `rsa-big.sml`.

Background about RSA Encryption:

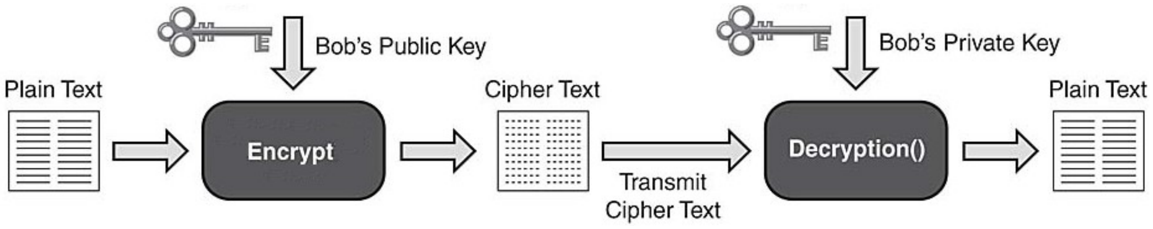
In this assignment we will explore an algorithm known as *RSA encryption*, named for the three mathematicians who first described it, Rivest, Shamir, and Adleman. **You don't have to understand all of the details of the algorithm** to do the assignment, but you might want to read about it online first:

- http://en.wikipedia.org/wiki/RSA
- http://mathworld.wolfram.com/RSAEncryption.html

Encryption is the process of transforming *plaintext* information using an algorithm into an encoded form called *ciphertext* so that it can be read only by its intended recipients. One quality of a good encryption algorithm is that even if an attacker intercepts the ciphertext and knows what algorithm was used to encrypt it, he/she still cannot decode the message. The intended recipient knows a secret value called a *key* that is fed into the algorithm to decode the message.

Keys: RSA uses a pair of keys: a *public key* to encrypt messages, known to both the sender and recipient (and potentially also known to the attacker); and another *private key* to decrypt messages, known only to the intended recipient(s). Suppose Alice wants to send a message to Bob. They choose their keys in the following way:

- Choose two random unequal large prime numbers; call them p and q. Compute their product, $n = p \cdot q$.
- Choose another random large integer e (for "encryption") such that $1 < e < (p - 1)(q - 1)$, where e and $(p - 1)(q - 1)$ are relatively prime (share no factors with each other). Alice's public key for Bob will be the two numbers **(n, e)**.
- Find another integer d (for "decryption") such that $d \cdot e \equiv 1 \bmod (p - 1)(q - 1)$. In other words, find some integer d such that $e \cdot d - 1$ is divisible by $(p - 1)(q - 1)$. Bob's private key will be the number **(d)**.
- Once the keys are chosen, Alice tells her public key (n, e) to Bob. Private key d is kept secret.



Encrypt: Alice wants to send a message M to Bob. M is a long string. The message is encrypted and sent as follows:

- M is converted into a large integer m such that $0 < m < n$, generally by combining M's ASCII character values.
- The plaintext integer m is encrypted into a ciphertext integer c by applying the following formula (recall that n and e are the components of Alice's known public key for Bob):

  $c = m^e \bmod n$
- Ciphertext integer c is sent to Bob.

Decrypt: Bob receives ciphertext integer message c and wants to decrypt and read it.

- Bob uses his secret private key value d to convert c back into its original integer form m using the formula:
  - $m = c^d \bmod n$
- Now that Bob has m, it can be converted back into a string by applying the reverse conversion process that was used to turn M into m (by breaking apart the integer into its ASCII values and interpreting these as characters).

It is believed that the encryption can't be broken unless the attacker can efficiently factor n. For this reason, very large values of n are chosen so that it would take an impractically long time to do the factoring. In our assignment, it wouldn't be very difficult to break the encryption because we will use relatively small values of n.

Our Implementation of the RSA Algorithm (Functions to Implement):

The following sections describe our implementation of the RSA algorithm for this assignment. You are not yet allowed to use higher order functions (functions that take other functions as arguments, like the `map` function), and you should not use the built-in list function called `rev` even though it is briefly mentioned in Chapter 3. You can use the built-in `length` function for lists, but do not use other list operations not described in Ullman Ch. 1-3.

You are expected to use good programming style, such as naming, indentation, and avoiding redundancy when possible. You should also place a descriptive comment on each function, including any preconditions assumed by the function.

As always, if the solution to a previous problem is useful in helping you solve a later problem, you should call the earlier function from the later function. You are once again allowed to define **"helper" functions** to solve these problems and you can include any testing code you develop (although it should be labeled as such). All helper functions should be defined with a `let` construct so that they are not part of the top-level bindings. Favor the use of patterns rather than `if-then-else` statements to distinguish between base and recursive cases.

(The `pack`, `unpack`, and `modPow` functions are worth the majority of the points for this assignment.)

Part A: String ↔ Integer Conversion

RSA encryption involves computing with very large integers. This requires special code to perform the math more efficiently. We need functions to convert between strings and sequences of integers representing the ASCII ("ordinal") values of each character in the string. We will turn each character into its ordinal using the built-in `ord` function, and we will convert it back to a character using the built-in `chr` function. We will use the built-in `implode` and `explode` functions to convert between string values and character lists. To keep the math simpler, we'll restrict ourselves to ASCII characters that have a range of 0 to 127 and we'll even assume that we have no characters with ordinal value 0.

1. `stringToInts`

```
val stringToInts = fn : string -> int list
```

Define a function called `stringToInts` that takes a string *s* as an argument and that produces a list of integers that represent the ordinal values (ASCII codes) of the sequence of characters in *s*. For example, the call of `stringToInts("hello")` produces `[104,101,108,108,111]`.

2. `intsToString`

```
val intsToString = fn : int list -> string
```

Define a function called `intsToString` that is the inverse of `stringToInts`, taking a list of ordinal values as an argument and returning the corresponding string.
For example, the call of `intsToString([104, 101, 108, 108, 111])` produces `"hello"`.

(continued on next page)

Part B: Packing and Unpacking

Being able to convert a string into a list of ASCII integers is a good start, but we have to treat the entire message *M* as a single very large integer *m*. Because of limitations on the range of values that can be stored by ML's `int` type, we can't actually pack all of the integers together into a single value. But we can pack small groups of integers together into larger integers and then encrypt/decrypt each integer individually using the same public/private key values.

We need the ability to combine small groups of characters together, where each group joins to form a larger integer that we'll encrypt and later decrypt. We will refer to this task as *packing* and *unpacking* in the rest of this document.

For example, suppose you were dealing with numbers between 1 and 99 and you want to "pack" several of them together. If you want to pack together the numbers `[38, 3, 97]`, you could turn this into the integer 380397 by computing:

$$(38 \cdot 100^2) + (3 \cdot 100^1) + (97 \cdot 100^0)$$
$$380000 + 0300 + 97$$

Given the integer 380397 and knowing that we used 100 for packing, we could undo this to get the original sequence.

You are going to write this in a general way as a function called `pack` that takes a list of integers to pack along with the number of integers to pack together and the base to use.

3. **`pack`**

```
val pack = fn : int list * int * int -> int list
```

Define a function called `pack` that takes three arguments: a list of integers *lst*, a number *k* for the maximum number of integers to pack, and a base *b* to use for packing. The function produces the list obtained by combining integers in the given base. In particular, consecutive sequences of *k* values from *lst* ($a_1, a_2, \ldots, a_m$) are replaced with the single integer:

$$(a_1 \cdot b^{k-1}) + (a_2 \cdot b^{k-2}) + (a_3 \cdot b^{k-3}) + \ldots + (a_k \cdot b^0)$$

If the list ends with a sequence of fewer than *k* values, those values should be combined using the same formula above as if *m* were the length of the list instead. Your function should produce an empty list if passed an empty list.

For example, suppose that you have defined the following list:

```
val lst = [18, 3, 95, 48, 22, 39, 47, 12, 73, 15];
```

Below are examples of calls on `pack`, most using base 100 but packing a different number of numbers on each call:

- `pack(lst, 2,  100)` produces `[1803,9548,2239,4712,7315]`
- `pack(lst, 3,  100)` produces `[180395,482239,471273,15]`
- `pack(lst, 4,  100)` produces `[18039548,22394712,7315]`
- `pack(lst, 3, 1000)` produces `[18003095,48022039,47012073,15]`

Notice that there might be extra values at the end of the list that are only partially packed. For example, the last call above has two stray values at the end (`[73, 15]`) that are converted into `7315` rather than `73150000`. By packing the lists this way and guaranteeing that 0 does not appear in the list to be packed, we can unpack the list using just the original base (100 in the examples above) without knowing how many values were packed into each integer.

You may assume that *k* and *b* are both greater than 0 and that the numbers in the list are all greater than 0 and less than *b*. In testing your program, be careful about which cases you test because you might get integer overflow.

4. **`unpack`**

```
val unpack = fn : int list * int -> int list
```

Define a function called `unpack` that is the inverse of the `pack` function. It takes two arguments: a list and a base. It produces the list of integers obtained by expanding each integer using the given base. For example, the call of `unpack([180395, 482239, 471273, 15], 100)` returns the same elements as in the `lst` shown previously.

Keep in mind that the original list did not contain any 0s, so your function should pull apart each integer as many times as it can until it reaches 0. Use the `div` and `mod` operators to pull apart each integer into its constituent parts. Assume that the list/base combination to unpack was generated by a legal call on `pack` with the same base.

Part C: RSA Encryption/Decryption Functions

Now we'll write a few functions for the actual encryption/decryption of our packed lists of integers, following the RSA algorithm described previously. We have already gone ahead and decided the various integers and public/private keys to start with, such as p, q, n, e, and d. Include the following variable bindings at the beginning of your file:

```
val p = 131;      (* first chosen prime for RSA encryption *)
val q = 239;      (* second prime chosen prime for RSA encryption *)
val n = p * q;    (* used for modulus *)
val e = 1363;     (* public key *)
val d = 227;      (* private key *)
```

5. `modPow`

```
val modPow = fn : int * int * int -> int
```

Define a function called `modPow` that takes integers x, y and n as arguments and that efficiently computes $(x^y \bmod n)$. You may assume that y is greater than or equal to 0. Use your `pow` function from Homework 1 as a starting point.

This might seem like an easy function to implement. But you cannot just raise x to the y power and then mod it by n. The reason is because x^y might be very large and therefore might exceed the maximum integer value allowed by ML. But remember that $(x^y \bmod n)$ must be between 0 and n-1, so the overall result is not too large to be represented as an ML integer. It is possible to compute $(x^y \bmod n)$ without having to actually fully compute and store x^y in our code. To do this, you will need to make two modifications to your code:

- Include a special case for even powers that takes advantage of the fact that for even values of y,

  $$x^y = (x^{y/2})^2 = (x^2)^{y/2}$$

  For example, $3^8 = (3^4)^2 = (3^2)^4$, which also equals $((3^2)^2)^2 = 6561$.

  If you correctly add this optimization, your function makes at most roughly $\log_2 n$ calls where n is the exponent.

- Make sure that your function computes the mod often. It would be highly inefficient to compute x^y in its entirety first and then to mod it by n only at the end of the computation. Instead, we want to break up the computation into smaller computations that we mod by n. The principle we will apply is that, for any two integers x and y,

  $$(x \cdot y \bmod n) = ((x \bmod n) \cdot (y \bmod n)) \bmod n$$

So in writing your `modPow` function, make sure that every time you multiply two values together that you mod the result by n before using the result in any other multiplications. If we assume x is less than n, this will ensure that you never compute a value larger than $(n - 1)^2$, which will improve the efficiency and bound the range of the computation.

6. `encrypt`

```
val encrypt = fn : string * int * int * int * int -> int list
```

Write a function called `encrypt` that performs the overall RSA encryption algorithm we have described. The function produces a list of encrypted integers based on five arguments: a string s, an integer key e, a modulus n, an integer *per* (how many ordinals to pack into each encrypted integer), and an integer *base*. The function converts the string into a list of ordinal values, packs the ordinal values with the given base and given number of ordinals to pack, and then encrypts each integer x in the resulting list by replacing it with $(x^e \bmod n)$.

7. `decrypt`

```
val decrypt = fn : int list * int * int * int -> string
```

Write a function called `decrypt` that performs the overall RSA decryption algorithm we have described. The function produces a decrypted string based on four arguments: a list of encrypted integers, an integer key d, an integer modulus n, and an integer base. Decrypt each int x of the list by replacing it with $(x^d \bmod n)$, then unpack the resulting list using the given base, and convert the list of ordinal values into a string.

Testing Your Code:

Once you have completed the previous functions, include the following variable bindings at the end of your file. If your functions work, `message` and `decoded` should store the same value.

```
val message = "Twas brillig and the slithy toves did gyre and gimble";
val code = encrypt(message, e, n, 2, 128);
val decoded = decrypt(code, d, n, 128);
```

NOTE: Be advised that the *per* value of 3 does not work for packing and unpacking. Your code will produce strange results even if it works for other values. You do not need to worry about this case or write any code to account for it.

Extra Credit:

(1 point) **RSA on Big Integers**

Once you have verified that you have a working solution to the preceding problems, save a new copy of the file as `rsa-big.sml`. In this second version we will use the `IntInf` structure that can handle large integers. Replace the variable bindings at the beginning of the file with the following (including the command to open the `IntInf` structure):

```
open IntInf;

val p = toLarge(30000000000000000041);              (* first prime *)
val q = toLarge(40000000000000000019);              (* second prime *)
val n = p * q;                                      (* modulus *)
val e = toLarge(1255949);                           (* public key *)
val d = toLarge(668816966294013531996920257112350 9); (* private key *)
```

Before you can proceed, you will have to modify your definitions for `stringToInts` and `intsToString`. The `ord` and `chr` functions deal with ordinary `int` values (what will be listed as `?.int` after you open the `IntInf` structure). These ordinary `ints` can be converted back and forth with big `ints` by calling the functions `fromInt` and `toInt`. The `fromInt` function converts an ordinary `int` into a big `int`. The `toInt` function converts a big `int` into an ordinary `int`. Once you have made the appropriate modifications, the function prototypes should look like this:

```
val stringToInts = fn : string -> int list
val intsToString = fn : int list -> string
```

Then replace the final two variable bindings with the following:

```
val code = encrypt(message, e, n, 16, 256);
val decoded = decrypt(code, d, n, 256);
```

The rest of the code should work without modification and should quickly encrypt the message, this time packing 16 characters into each encoded integer and allowing characters to have ordinal values as high as 255. The call on `decrypt` should correctly decrypt the message.