Computer Science 52

Assignment 8

Due Friday, November 17, 2017, at 5:00 PM

Reading This assignment is about RSA encryption. Refer to the course document *RSA Encryption* and the original RSA paper for details on the encryption method; there are links to both on the course Resources page. In particular, be sure to read Sections I, II, IV, and V of *RSA Encryption* before you start to work on the assignment.

Preparation The template file contains some type declarations to handle extremely large integers and some functions that will be useful.

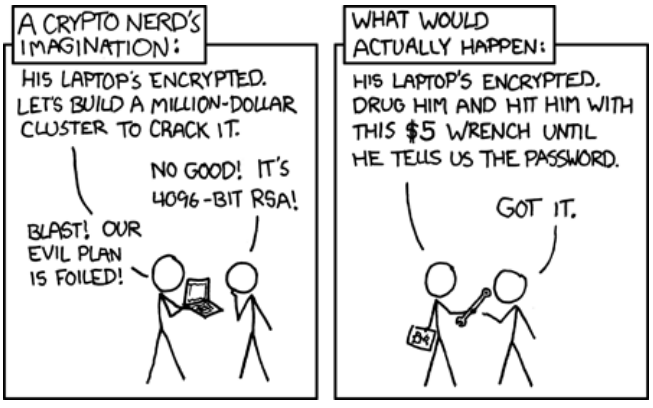
For encryption, we must use integers that are much larger than the ordinary SML `int` whose largest value is $2^{30} - 1$, or $1,073,741,823$. On current 64-bit processors, the largest integer is $2^{63} - 1$, a considerably larger number that is nearly 10^{19}, but is still too small for our purpose. Fortunately, SML (like most other programming languages) provides "infinite precision" integer facilities through a structure named `IntInf`. Its integer type is `IntInf.int`.

One of the features of the starter code in the template file is that it converts the ordinary arithmetic operations to use the type `IntInf.int` instead of the ordinary `int`. With those declarations, you can create a function like this

```
fun squareMod (k,n) = k * k mod n;
```

and obtain the type signature of

```
val squareMod = fn : IntInf.int * IntInf.int -> IntInf.int
```



https://xkcd.com/538

As you write your functions, be sure that all the type signatures contain `IntInf.int` and not something like `?.intinf`.

Most of the features of the starter file are described in comments. Be sure to read them and understand how they are used. A few functions require more explanation than we want to put in comments and are explained in the appendix to this assignment.

Submission Problem 8 asks you to create a special file named `asgt08.rsa`. For the rest of the assignment, work as usual: obtain the template and check files Rename the template file to `asgt08.sml`, and put your solutions in it. When you are ready, run the check file. Submit *both* `asgt08.sml` and `asgt08.rsa` in the usual way.

Assembling the Tools

1. [1 point] Declare a function `powerMod` that computes b^e mod n using the recursive exponential function below.

$$b^e = \begin{cases} 1 & \text{if } e = 0, \\ (b^{e/2})^2 & \text{if } 0 < e \text{ and } e \text{ is even, and} \\ (b^{e/2})^2 \cdot b & \text{otherwise.} \end{cases}$$

The expression $e/2$ in the recursive step signifies integer division, with truncation.

```
powerMod : IntInf.int * IntInf.int * IntInf.int -> IntInf.int
```

The order of the arguments is `b`, `e`, and then `n`. You may assume that `b` and `n` are positive, and `e` is non-negative. The function `squareMod`, used as an example earlier, will be helpful. Be sure to take the remainder after *every* arithmetic operation to keep the size of the intermediate results under control.

A comment on the algorithm: You will use `powerMod` at least twice later in the assignment. It is critically important that you follow the formula above. The more obvious strategy—of multiplying b by itself e times—could take centuries. The reason that the "repeated squaring" strategy is better is that the recursive step has the exponent $e/2$ instead of $e - 1$. The exponent e gets one bit shorter with each recursive call, so the number of recursive calls is the number of bits required to express e in binary. For example, if e is one million, about 2^{20}, the "repeated multiplying" strategy will do about a million multiplications, compared to at most 40 for the "repeated squaring" strategy. In our work, the exponents will be much larger, and the contrast will be even more dramatic.

2. [2 points] To be able to translate arbitrarily long messages we need to be able to break a number is larger than n into a collection of numbers that are all smaller than n (and then the reverse operation), all *in a way that is efficient in space usage.*

Declare a pair of functions to convert between integers and their base-`n` list representations. The functions `block n` and `unblock n` should be inverses of one another.

```
block : IntInf.int * IntInf.int -> IntInf.int list
unblock : IntInf.int * IntInf.int list -> IntInf.int
```

The expression `block(n,m)` creates a list `[x0,x1,...,xk]` each of whose entries is a non-negative integer less than `n` and which satisfies

$$m = x0 + x1 \cdot n + x2 \cdot n^2 + \ldots + xk \cdot n^k.$$

Effectively, the result of `block n m` is the base-`n` representation of the integer `m`. The expression `unblock(n,[x0,x1,...,xk])` recovers the value of `m` from the list.

3. [2 points] RSA encrypts numbers. To encrypt a string, we need to be able to convert from any string into some corresponding number. One way to do this is to treat each character in a string as a digit. Characters correspond to integers between 0 and 255. The function `ord` maps a character to the corresponding integer. Its inverse is the function `chr`. A string may be represented, via `explode` and `ord`, as a list of values between 0 and 255. That list, in turn, may be considered to be an `IntInf.int` represented in base 256.

Type `man ascii` in a terminal window to see the correspondence between integers and characters.

Declare a pair of functions that translate between a string and the corresponding `IntInf.int`.

```
messageToIntInf : string -> IntInf.int
intInfToMessage : intInf.int -> string
```

These functions allow us to translate between a string and its numerical representation.

```
- val abcInt = messageToIntInf "abc";
val abcInt = 6513249 : IntInf.int

- intInfToMessage (abcInt);
val it = "abc" : string
```

The value 6513249 is `(ord #"a")` + `(ord #"b")`256 + `(ord #"c")`256^2.

Suggestion: The functions `block` and `unblock` may be useful.

Important convention: For this assignment, let us agree that strings are encoded so that the first character corresponds to the low-order part of the integer. This is the same convention that we used in earlier assignments.

Carrying Out the Encryption

4. [2 points] Write a function `rsaEncode` that takes a key (e, n) and an integer less than n and returns the encryption of the integer. The result will also be an integer less than n.

```
rsaEncode : IntInf.int * IntInf.int -> IntInf.int -> IntInf.int
```

The result of `rsaEncode (e,n) m` is $\texttt{m}^{\texttt{e}}$ mod `n`. Do not forget that you wrote `powerMod` in Problem 1.

We will be generating keys in a minute, but if you would like to use a "valid" key for debugging, you can use $(7, 111)$.

5. [2 points] With a set of keys, we now have all of the pieces to support encryption and decryption. Remember, to encrypt a string, we

- turn the string into a number,
- break this number into n-sized blocks,
- encrypt each of the blocks using the public key,
- and finally convert the blocks back into a single number.

To decrypt, reverse the steps and use the private key.

a. Write a function `encodeString` that takes a key (e, n) and a string, and produces a single `IntInf.int` value that encrypts the message contained in the string. This function is an easy combination of functions that you have already written.

```
encodeString : IntInf.int * IntInf.int -> string -> IntInf.int
```

b. Suppose that you have the public key $(7, 111)$. Use the function from part a to encode the phrase below. The result will be a single `IntInf.int`; include an expression that computes that value, named `sampleEncryption`.

> Don't panic and always carry your towel.

```
sampleEncryption : IntInf.int
```

c. Write an analogous function `decodeString`, then use the private key $(31, 111)$ to decode the result from part b. Again, include an expression that will produce a string named `sampleDecryption` to show that your function works properly.

```
decodeString : IntInf.int * IntInf.int -> IntInf.int -> string
sampleDecryption : string
```

Generating Keys

6. [3 points] You now have encoding and decoding functions that depend on having appropriate keys. The next step is to generate

public and private RSA keys.

a. Declare a (pseudo-) random number generator. See the notes at the end of this assignment. Name it `generator`.

```
generator : Random.rand
```

b. Create a function `industrialStrengthPrime` that takes a random number generator and an integral number of bits. It creates a "random" prime with at most the specified number of bits. We declare a number p to be prime if $a^p \bmod p = a$ for twenty random a's that are less than p. See the description at the end of this assignment for instructions on how to generate random `IntInf.int`'s.

There is additional material and class discussion about why this test produces numbers that are "very likely" to be prime.

```
industrialStrengthPrime : int -> Random.rand -> IntInf.int
```

Important: There must be only *one* random number generator that generates all the random values. It must be declared outside of the function.

Use the function `randomRangeIntInf`, provided in the template file and described in the appendix to this assignment, to produce random numbers in the desired range.

7. [2 points] Using the result of Problem 6, create a function that will take an integer k anda random number generator, generate two k-bit primes, and then create a pair of RSA keys:

0. Find the two different industrial strength k-bit primes p and q.
1. Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
2. Generate a random number d less than n and apply the function `inverseMod` to it and $\varphi(n)$.
3. If the result from `inverseMod` is `SOME` e and $d \not\equiv e \pmod{\varphi(n)}$, then you have correct values for d and e. If the result is `NONE`, try again with a different random number d. Repeat from step 2 until you have all three values.

The function `inverseMod` is provided in the template file and is described in the appendix to this assignment.

```
newRSAKeys : int -> Random.rand ->
                  (IntInf.int * IntInf.int) * (IntInf.int * IntInf.int)
```

See the end of Section V of the course document *RSA Encryption* for an explanation of why we must avoid the situation in which $d \equiv e \pmod{\varphi(n)}$. Interestingly, if you generate primes with three or fewer bits, there are *no* satisfactory values of d and e; the algorithm above will never terminate. There are only two 4-bit primes; in that case n is always 143.

It makes sense to test your code with small values of k—just be sure that k is at least 5.

Encrypting and Decrypting

8. [2 points] Use the function from Problem 7 to generate keys from a pair of 30-bit prime numbers. Choose a secret message, one or two

sentences in good taste, and encrypt it with your *public* key (as if someone were sending the message to you). Copy your *public* key and the encrypted message into a file named `asgt08.rsa`. The file should contain two `val` declarations, as shown in the following example.

```
val myPublicKey = (7:IntInf.int, 111:IntInf.int);
val mySecret = 917305171560109201789422813828 7:IntInf.int;
```

Notice the type specifications. They are crucial, and you will have to add them by hand. Without them, we will not be able to read your data into the SML system. See the file http://www.cs.pomona.edu/classes/cs052/rsa/rbull.rsa for another example.

The file asgt08.rsa is the *only* material that you should submit for Problem 8. Keep a record of your private key and your secret message, but do not reveal them to anyone, and do not include your computations in the file `asgt08.sml`. After everyone has completed the assignment, we will post the submissions for Problem 8 and attempt to break one anothers' codes.

9. [2 points] If someone knows my public key (e, n) *and* knows the factors of n, it is easy to recreate the steps in generating the key and learn my private key. The security of the RSA scheme lies in the presumption that factoring is a time-consuming process.

a. If my RSA public key is $(22821, 52753)$, what is my private key? The value of n is small enough to permit brute-force factoring. Include an executable expression that evaluates the private key and puts that result in a variable called `crackedPrivateKey`. Additionally, place the actual value of the key in a comment.

```
crackedPrivateKey : IntInf.int * IntInf.int
```

Restriction: Please write your own brute-force factoring function. It will be only a few lines of SML. Do not resort to an external program or one of the "factoring services," like WolframAlpha, on the internet.

b. The numbers in the key of part a can be represented with 16 bits. Suppose that the private key can be found in t seconds. Estimate, as a multiple of t, the time it would take to find the private key if the public key numbers required 160 bits to represent. Give a brief answer in a comment.

Appendix: Additional `IntInf.int` Functions

Arithmetic on ordinary integers The file `asgt08-template.sml` contains several declarations to make your work easier. Among them are re-declarations of the usual arithmetic operators so that they operate on values of type `IntInf.int`. When you write something like

`x + y` the SML system assumes that `x` and `y` are of type `IntInf.int`, and the result will be of the same type. One minor consequence of making the change is that the ordinary operations on type `int` are "hidden," and it is a little cumbersome to use them—one must write `Int.+(x,y)`. Fortunately, there are only a few places where you will need to use ordinary `int` values.

Random number generation In SML, the random number generator is of type `Random.rand`. It is created with a *seed*, an ordered pair of integers. If you create two generators with the same seed, they will give you the same sequence of numbers. The file `asgt08-template.sml` provides a function

```
randomRangeIntInf : IntInf.int * IntInf.int -> Random.rand -> IntInf.int
```

that generates numbers in a given range. Here is an example of its use.

```
- val seed = (47,42);
- val generator = Random.rand seed;
- val randomFirst = randomRangeIntInf (20,100) generator;
val randomFirst = 40 : IntInf.int
- val randomSecond = randomRangeIntInf (20,100) generator
val randomSecond = 67 : IntInf.int
```

Each subsequent call `randomRangeIntInf (20,100) generator` gives a different result. The values are uniformly distributed between the two bounds, 20 and 100 in this case, including the endpoints. It is important to create *one* random number generator and use it throughout your program.

If you want to generate a random number with `k` bits, as a candidate for an industrial strength prime, simply make the call

```
randomRangeIntInf(pow2 (Int.-(k,1)), pow2 k - one)
```

Here, the phrase "b-bit prime" means a prime number whose binary representation has exactly k bits. The numbers which can be represented in exactly k bits are those numbers c which satisfy $2^{k-1} \le c \le 2^k - 1$.

The `inverseMod` function The function `inverseMod` is an SML implementation of the extension to Euclid's algorithm described in Section III of *RSA Encryption.*

```
inverseMod : IntInf.int * IntInf.int -> IntInf.int option
```

Given the pair of integers `(u,m)`, `inverseMod` attempts to find the modulo-`m` inverse of `u`. If `u` and `m` are relatively prime to one another,

the function returns `SOME a`, where `a` is the unique positive integer satisfying

```
u*a div m = 1   and   a < m.
```

If `u` and `m` are not relatively prime to one another, then there is no modulo-`m` inverse of `u`, and the function returns `NONE`.