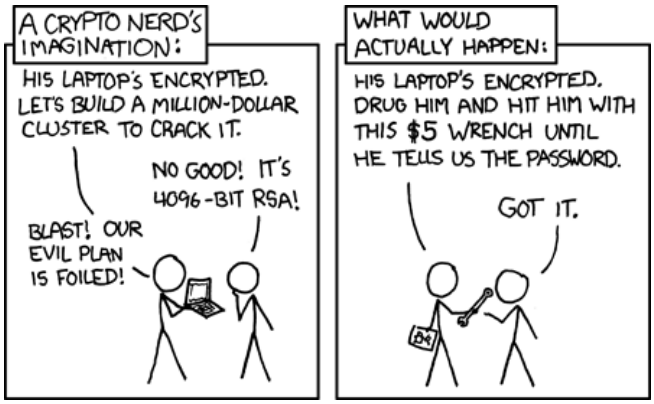
Computer Science 54

Assignment 6

Sunday, October 16 @ 11:59pm



https://xkcd.com/538/

Partners On this assignment you may (and I'd encourage you to) work with a partner. If you choose to, you must both be there anytime you're working on the assignment.

Submission When you're all done, run the check file. If all is well, submit your assignment via Gradescope. If you worked with a partner, make sure that both names are at the top of the file. Only one person needs to submit, but make sure you add the other person when you submit. **For this assignment you need to submit two files: your assign6.pdf file as well as your assign6.sml file.**

The starter file

Take a few minutes to read through the starter file. In addition to sections for your functions, for this assignment, we also provide a number of important additional functionality that you *will need to use* for writing your functions.

The main thing that the starter provides is functionality to handle extremely large integers.

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 rem n;
```

For efficiency reasons, use `quot` and `rem` instead of `div` and `mod`.

and obtain the type signature of

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

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 an appendix to this assignment.

Latex

For most of the assignment going forward, you will use Latex to generate your assignments. Latex is a very powerful software tool for generating documents that is particularly good when doing math (among other things!).

0. **[1 point]** Your first Latex document
To make sure you're ready for the next assignment, you need to be able to at least setup a basic Latex project and generate a document. On the course webpage is a file called `assign6.tex`. You must download this file and use Latex to create a version of this document where you edit the "author" tag to change it from "Me" to your name and then generate a pdf file. There are many Latex editors out there, but we recommend using overleaf, which allows you to access and generate your pdfs via a web interface.

Here is one path towards doing this:

- Follow the instructions at:
  `https://www.overleaf.com/learn/how-to/Creating_a_document_in_Overleaf`

  for creating an overleaf account and starting a new project. I'd recommend calling the project "Assignment 6" (or something like that.
- Download the "assign6.tex" file from the course webpage.
- Get the contents of "assign6.text" into your overleaf project. The easiest is to upload this file, though you could also make a blank .tex file and copy the contents.
- Get it compiling in overleaf and change the author tag to your name.
- Download the .pdf file from overleaf!

Assembling the Tools

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

Whenever you want to use a constant (like, 0, 1, or 2), make sure to use the `IntInf.int` constants (`zero`, `one`, `two`) since this will help avoid any ambiguity for SML.

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

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 that 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.

These should feel *very* similar to some of the functions we've written previously.

```
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

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

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.

For example:

In practice, we'll use n that is *much* larger. Notice that the lower order "digits" are at the beginning (left) of the list.

```
- block (2,26);
 val it = [0,1,0,1,1] : IntInf.int list
- block (111,12345678910);
 val it = [58,110,5,36,81] : IntInf.int list
- unblock (2, block (2,26));
 val it = 26 : IntInf.int
```

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. Once we have a list of numbers (representing the "digits") we can turn it into an `IntInf.int` represented in base 256.

You can see the correspondence between integers and characters at `https://www.asciitable.com/`.

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

Hint: `block` and `unblock` will be helpful! They deal with `IntInf.Ints`. The `fromInt` and `toInt` functions convert between `int`s and `IntInf.int`s.

```
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.

$97 + 98 * 256 + 99 * 256^2 = 6513249$

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. **[1 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
```

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

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

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

- We turn the string into a number,
- then break this number into n sized chunks.
- then encrypt each of these chunks using the public key.
- And finally, put it all back together into a single number.

Decryption is just these steps in reverse using 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 should be a straightforward combination of functions that you have already written.

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

b. Write an analogous function `decodeString`, then use the private key (31, 111) to decode the result from part b.

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

Assuming everything works, you should be able to encode and then decode a message with a pair of keys (e.g., (7, 11) and (31, 111):

```
-  val encrypted = encodeString (7, 111) "CS54 is my favorite class!";
val encrypted =
  39720533575853127514224941186385866282342882609003703184535861 6 : IntInf.int
-  decodeString (31, 111) encrypted;
val it = "CS54 is my favorite class!" : 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.

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 : Random.rand -> int -> 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 (like in Assignment 4).

7. **[0 points]** We now have all the pieces that we need to generate a pair of public/private keys. The general algorithm to generate a key is:

0. Find the two industrial strength primes p and q.
1. Compute their product $n = pq$ 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, 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.

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

We have provided you with a solution. Take a few minutes and look through it and make sure you understand it. Try generating a few keys. Note that the function does require `industrialStrengthPrime` to work correctly.

8. **[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`.

To get it to be of type `IntInf.int` you may need to mulitply by `one`.

```
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.

Extra credit

This assignment is due at the normal time on Sunday. However, to encourage you to finish early and take a break from CS54 over fall break, you will

receive **[0.5 points]** of extra credit if you submit the assignment by 11:59pm on Friday.

Appendix: Additional `IntInf.int` Functions

The template file 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.

For efficiency reasons, we use `quot` and `rem` in place of `div` and `mod`.

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 `asgt07-template.sml` provides a function

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

that generates numbers with the given number of bits. Here is an example of its use.

```
- val seed = (47,42);
- val generator = Random.rand seed;
- val randomFirst = randomIntInf generator 100;
val randomFirst = 116702885620608507845473577974 : IntInf.int
- val randomSecond = randomIntInf generator 100;
val randomSecond = 560072882339451739794345051343 : IntInf.int
```

Each subsequent call `randomIntInf generator 100` gives a different result. It is important to create *one* random number generator and use it throughout your program.

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 rem 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`.