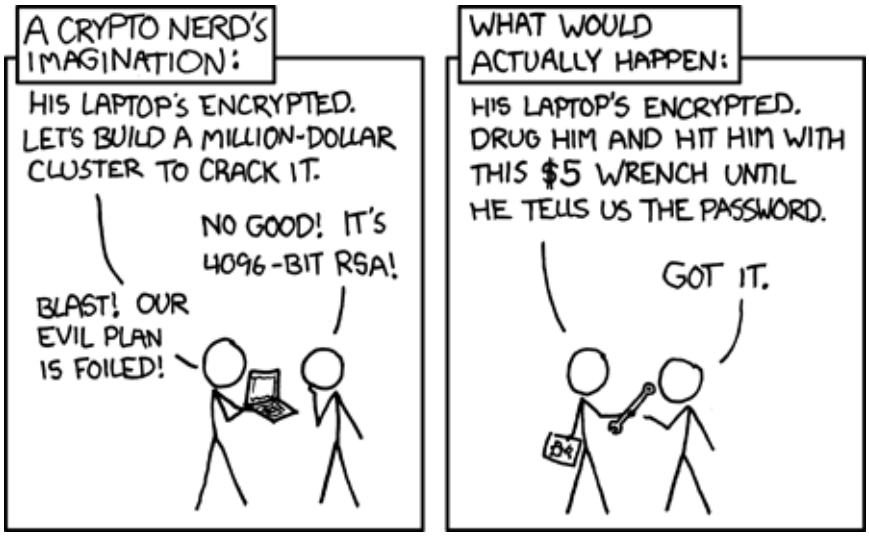
CS52 - Assignment 7

Due Friday 4/8 at 6:00pm



`https://xkcd.com/538/`

For this assignment we will be writing code that allows us to encrypt and decrypt strings using RSA encryption. Put your solution in a file called `assign7.sml` and submit in the usual way.

Starter

To ensure that everyone has working code to get started, we have provided a special version of SML that already contains a `cs52int` datatype and functions.

Running the special version of SML

The version we are providing only works on mac OS or Linux, so if you are working on a Windows computer of your own, you will have to do this assignment in the lab.

- **If you are working in the lab**

  To start SML, run the following command from Terminal:

  ```
  /common/cs/cs052/bin/sml-cs052
  ```

- **If you are working on your laptop/desktop and it's a Mac or Linux computer**

  1. Create the `assign7` directory where you will be doing all of your work.
  2. Go to:
     `http://www.cs.pomona.edu/~dkauchak/classes/cs52/assignments/assign7/`
     and download the zip file into the `assign7` directory.
  3. Unzip the contents of the file. You should see three files now.
  4. To run the special version of SML, make sure that you are in your `assign7` directory and then type:
     ```
     ./sml-cs052
     ```
  5. To check that this is working correctly, just call one of the functions listed below.

Available functions

All of the functions, `sum`, `diff`, `compare`, `toInt`, `fromString`, etc., are available that you implemented in assignment 3 as well as the constants `zero`, `one`, and `two`. You may use any of the following functions on this assignment:

```
zero    negate    isOdd       compare      fromInt       gcd
one     sum       abs         greater      toInt         inversemod
two     diff      min         greatereq    fromString    randomCs52Int
        prod      max         less         toString
        quo       sign        lesseq
        rem       sameSign    unequal
        square
```

You may not, however, use any of the list-based functions, nor are you allowed to use the constructor `CS52Int`. (We would like to have hidden that constructor, but SML makes it difficult. There is no need to use it because any values you need may be constructed from the integer- and string-conversion functions.)

Note: instead of `divide` the function that divides two numbers is called `quo`.

Additional functions

If you look closely at the list above you'll notice three additional functions that we didn't have before. I have included the code for these functions at the end of this assignment in an appendix. The comments of these functions explain their usage. *You will need to use all three of these functions for this assignment so make sure that you understand what each of the three functions do.*

Hints/Advice

I've put a number of suggestions throughout this assignment, so please do pay attention to those. Two other general pieces of advice:

- As always, make sure to test as you go. None of the functions below are particularly complicated, but many of the latter functions utilized the earlier ones, so it's important that they are correct.
- `map` will be your friend on this assignment. You should be doing very little list recursion.

Implementing RSA

1. **[2 point]** Warmup

   I know it's been a while since we've programmed in SML. As a warmup (and to create a useful function for later on) write a function `powermod` that computes $b^e \bmod m$. Use the method for exponentiation from Problem 5b on Assignment 6, and be sure to take the remainder after *every* arithmetic operation.

   ```
   powermod : cs52Int -> cs52Int -> cs52Int -> cs52Int
   ```

   The order of the arguments is `b`, `e`, and then `m`. You may assume that `b` and `m` are positive, and `e` is non-negative. You will use `powermod` at least twice later in the assignment.

2. **[2 points]** Handling arbitrary length messages

   As we discussed in class, RSA only works for numbers less than n. 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 number that are all smaller than n (and then the reverse operation), all *in a way that is efficient in space usage.*

   Write a pair 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 : cs52Int -> cs52Int -> cs52Int list
   unblock : cs52Int -> cs52Int list -> cs52Int
   ```

   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

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

   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. We have done this many, many times now, so this should feel very familiar.

3. **[2 points]** From strings to numbers

   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. We've already seen this with hexadecimal numbers, but that only used the letters 0-9 and a-f.

We will be using the ascii representation of characters, which has 256 possible characters represented by integers between 0 and 255.[1] The function `ord` takes as input a character and returns the number corresponding to that character. Its inverse is the function `chr`.

To convert a string to a number we just think of a string as a number represented in base 256. *For this assignment, we will assume that strings are encoded so that the first character corresponds to the low-order part of the integer.*

Declare a pair of functions that translate between a string and the corresponding `cs52Int`.

```
stringToCs52Int : string -> cs52Int
cs52IntToString : cs52Int -> string
```

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

```
val abcInt = stringToCs52Int "abc";
toString abcInt;
val it = "6513249" : string

cs52IntToString (abcInt);
val it = "abc" : string
```

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

Suggestion: The functions `block` and `unblock` that you wrote above and the built-in functions `explode` and `implode` may be useful.

4. [**1 point**] Almost there

   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 : cs52Int * cs52Int -> cs52Int -> cs52Int
   ```

   The result of `rsaEncode (e,n) m` is $\texttt{m}^{\texttt{e}} \bmod \texttt{n}$. 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**] Encryption at last

   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.
   - and finally we encrypt each of these chunks using the public key.

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

In addition, to make life simple for passing it around, for this assignment we will also then package each of these encrypted chunks back into a single number. Decryption is just these steps in reverse using the private key.

The two functions below should do this and should be a straightforward application of the previous functions we have written so far for this assignment.

(a) Write a function `encryptString` that takes a key (e, n) and a string, and produces a single `cs52int` value that encrypts the message contained in the string.

```
encryptString : cs52int * cs52int -> string -> cs52int
```

Remember that `rsaEncode` can only encrypt numbers that are less than n.

(b) Write an analogous function `decryptString` that takes a private key (d, n) and a single `cs52int` representing an encrypted string and decrypts the string.

```
decryptString : cs52int * cs52int -> cs52int -> string
```

Remember that decrypting a number is the same process as encrypting a number, except using (d, n) instead of (e, n), that is computing $\texttt{m}^{\texttt{d}}$ mod `n`.

6. **[1 point]** Does it work?

Test to make sure that it works. Given the public key $(7, 111)$ and the private key $(31, 111)$ write a few lines of code (i.e. not any functions) that encrypt and then decrypt the following string:

```
Don't panic and always carry your towel.
```

7. **[3 points]** That's so random

The last thing we now need is a way for generating RSA keys. A critical part of this processing is finding a pair of different k-bit primes. Review the material on industrial strength primes from the reading: *Some Facts from Number Theory* and from the class notes. For our case, a number p is prime if $a^p \bmod p = a$ for twenty random a's that are less than p. Any p where this is not true is not prime.

Write a function that takes an integer `k` and produces a `k`-bit industrial strength prime.

```
industrialStrengthPrime : int -> cs52Int
```

Note that `k`-bit means that the number uses at most `k` bits.

See the appendix for how to generate random k-bit `cs52Int`'s. *Important:* There must be only *one* random number generator that generates all the random values. It must be declared outside of the function.

8. **[2 points]** The key to success is ...

Using the result of Problem 7, create a function that takes an integer k and produces a pair of RSA keys based on two k-bit primes.

```
newRSAKeys : int -> (cs52Int * cs52Int) * (cs52Int * cs52Int)
```

(Note that it doesn't actually matter which you consider public/private.)

Given two numbers u and m the `inversemod` function (see appendix) attempts to find a v such that $0 < v < m$ such that $uv \bmod m = 1$. If it finds one, i.e. returns `SOME` v, then you know that $gcd(u, m) = 1$ and if it doesn't find one, i.e. returns `NONE`, then $gcd(u, m) \neq 1$.

We can use this fact to help us generate a pair of keys (e, n) and (d, n). To find a d and e such that $de \bmod \varphi(n) = 1$ repeat the following until a valid pair d and e are found:

- Generate a random d, where $0 < d < n$.
- Use `inversmod` to try and find an e.

9. **[1 point]** Just for fun :)

   Use the function from Problem 8 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 `assign7.rsa`. The file should contain two `val` declarations, as shown in the following example.

   ```
   val myPublicKey = ("7","111");
   val mySecret = "917305171560109201789422813828 7";
   ```

   Important: The three integers should be represented by *strings of digits,* which can be created with the function `toString`. Your strings will be much longer than the ones in the example; each one must appear on a single line and cannot be broken across lines. See the file `/common/cs/cs052/rsa/example.rsa` for another example.

   The file assign7.rsa is the *only* material that you should submit for Problem 9. No material will go into your `assign7.sml` file for this problem. Keep a record of your private key and your secret message, but do not reveal them to anyone. After everyone has completed the assignment, we will post the submissions for Problem 9 and attempt to break one anothers' codes.

   Programming detail: For this problem it will be necessary to see a long string in its entirety, without truncation. The best way to do this is to tell SML to display more data when it echoes the values. Do do this, you will have to set a parameter using the statement

   ```
   Control.Print.stringDepth := 2000;
   ```

   in SML. The `stringDepth` value is the maximum number of elements in a string that will be written; the default value is 70.

   In some cases, this will not work. If it doesn't, the other option is to explicitly `print` the value, which does display it in its entirety. The `print` function will print out the value of a variable. For example, if you have your encrypted message in a variable called "mySecret", you can print out the entire string as:

```
print mySecret;
```

Do note that "val it = () : unit", the return value, is also at the end of the string, so be careful not to copy this part as well.

10. **[3 points]** Cracking the code

    If someone knows my public key (e, n) *and* knows p and q, 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 n into p and q is a time-consuming process.

    (a) **[1.5 points]** Write a function called `bruteForceCracker` that takes as input a public RSA key and returns the corresponding private key.

    ```
    bruteForceCracker : cs52int * cs52int -> cs52int * cs52int
    ```

    You may assume that the input is a valid key.

    (b) **[0.5 points]** If my RSA public key is $(22821, 52753)$, what is my private key? Place your answer in a comment in the file.

    (c) **[1 point]** The numbers in the key of part b can be represented with 16 bits. Let x be the time it takes to crack a 16-bit code. Estimate the time it would take to find the private key if the public key numbers required 160 bits to represent. Your answer should be in terms of x. Give a brief explanation. Again, place your answer in a comment in the file.

When you're done

Double check the following things:

- Make sure your code runs without any errors (i.e. `use "assign7.sml"` does not execute an error). If you didn't finish a function and it gives an error, just comment it out and leave a note.
- Make sure that your functions match the specifications *exactly*, i.e. the names should be exactly as written (including casing) and make sure your function takes the appropriate number of parameters and is curried/uncurried appropriately.
- Make sure you have used proper style and formatting. See the course readings for more information on this. Be informative and consistent with your formatting!
- Make sure you've properly commented your code. You should include:
  - A comment header at the top of the file with your name, the date, the assignment number, etc.
  - Each problem should be delimited by comment stating the problem number.
  - Each function should have a comment above it explaining what the function does.

– Complicating or unusual lines in functions should also be commented.

Don't go overboard with commenting, but do be conscientious about it.

When you're ready to submit, upload your assignment via the online submission mechanism. You may submit as many times as you'd like up until the deadline. We will only grade the most recent submission.

Don't forget that you are submitting two files: `assign7.sml` and `assign7.rsa`.

Grading

functions	19
comments/style	3
Total	22

Appendix: Additional `cs52int` Functions

The following three functions have been added to the ones specified in Assignment 3.

```
(*
 * gcd : cs52int -> cs52int -> cs52int
 *
 * gcd x y returns the greatest common divisor of x and y. It uses
 * Euclid's algorithm.
 *)
local
    fun euclid x y =
        if x = zero
           then if y = zero
                   then raise Overflow
                   else y
           else if y = zero
                   then x
                   else euclid (rem y x) x;
in
    fun gcd x y = euclid (abs x) (abs y);
end;


(*
 * inversemod : cs52int -> cs52int -> cs52int option
 *
 * inversemod u m returns (SOME v) when 0 < v < |m| and uv = 1
 * (mod m). The value of v, if it exists, is unique. inversemod u m
 * returns NONE if there is no such v.
 *
 *)
fun inversemod u m =
    let
        (*
         * Invariant: There are constants b and d
         * satisfying
         *     x = au+bm  and  y = cu+dm.
         * If ever x=1, then 1 = au+bm, and a
         * is the inverse of u, mod m.
         *)
        fun extendedEuclid x y a c =
            if x = zero
               then NONE
            else if x = one
               then SOME (if less a zero
                             then sum a m
                             else a)
               else extendedEuclid (rem y x)
                                   x
                                   (diff c (prod (quo y x) a))
```

```
                                    a;
    in
        extendedEuclid u m one zero
    end;

(*
 * randomCs52Int : int -> Random.rand -> cs52Int
 *
 * Usage example: To generate two 100-bit random cs52Int's, one
 * can execute the following declarations. It is important to use
 * only *one* generator for all the random values. Notice that
 * random number generation violates the functional paradigm, in
 * that different calls to the same expression yield different
 * values.
 *
 *    val seed = (47,42);
 *    val generator = Random.rand seed;
 *    val randomFirst  = randomCs52Int 100 generator;
 *    val randomSecond = randomCs52Int 100 generator;
 *
 *)
fun randomCs52Int len rg =
        if len <= 0
           then zero
        else if 8 <= len
           then sum (fromInt (Random.randRange (0,255) rg))
                    (prod (fromInt 256) (randomCs52Int (len-8) rg))
           else sum (fromInt (Random.randRange (0,1) rg))
                    (prod two (randomCs52Int (len-1) rg));
```