MCS 320 Project One : a prototype for a password manager

The goal is to use Sage to prototype a password manager with RSA public key cryptography. The data stored on file by the password manager is in comma separated values (csv) format. Each line contains three items: (1) URL to a login form, (2) login name, (3) password. For example: one line for your webmail could be

```
https://webmail.uic.edu/,mynetid,mypassword
```

where the strings mynetid and mypassword would be an actual netid and password.

Storing this file unencrypted is very unsecure. Each line will be encrypted as one large natural number with the RSA public key cryptography system.

1. strings as numbers

To compute with strings, we have to convert them into large numbers. This is done by the given functions `to_number` and `to_string`. We start by loading a library of Sage functions.

```
mydir = "/Users/jan/Courses/MCS320/Spring16/Project_One/"
mylib = mydir + "pubkeyfun.sage"
load(mylib)
```

You will have to adjust `mydir` in the definition of `mylib` to the location where you saved the file `pubkeyfun.sage`. Before we encrypt, we must encode strings as numbers. Let us encode and decode a message as a large number.

```
message = 'symbolic computation is cool'
encoded_message = to_number(message)
print encoded_message
decoded_message = to_string(encoded_message)
print decoded_message
```

and what is then printed is

```
192513021512090300031513162120012009151400091900031515 12
symbolic computation is cool
```

Assignment One. The given functions `to_string` and `to_number` cannot handle an URL like `http://www.uic.edu`. Extend the functions so they can handle URLs, special punctuation symbols (we will need a comma for example) and also upper case letters.

2. public key cryptography

We use the RSA public key cryptosystem to encrypt our data. The security of this system is based on the hardness of factoring large numbers into primes.

```
encryption_key = 27
modulus_factor = 55
```

To encrypt a message, we use modular exponentiation, defined in the function `modexp` of the module `pubkeyfun.sage`.

```
encrypted_message = modexp(encoded_message, encryption_key, modulus_factor)
print encrypted_message
print to_string(encrypted_message)
```

We see that the decoding of the encrypted message is meaningless:

```
s?????al???????n???????g????
```

To decrypt, we use a decryption key and again modular exponentiation.

```
decryption_key = 3
decrypted_message = modexp(encrypted_message, decryption_key, modulus_factor)
print to_string(decrypted_message)
```

and we get the original message back:

```
192513021512090300031513162120012009151400091900031515l2
symbolic computation is cool
```

The encryption key and the modulus factor are public information, but the decryption key is private. The modulus factor (the number we use in the modular computations) is the product of two prime numbers:

```
factor(modulus_factor)
```

and we see that the factors of 55 are the primes 5 and 11. While the modulus factor is public, the two factors are private. Extracting the factors goes as follows:

```
(p, q) = (f[0][0], f[1][0])
print p, q
```

To make our own encryption scheme, we first choose two large enough primes `p` and `q`. Then we choose the encryption key so that it is relative prime with respect to `(p-1)*(q-1)`.

```
gcd(encryption_key, (p-1)*(q-1))
```

We find the decryption key via the extended gcd algorithm, as one of the cofactors of the gcd:

```
(d, a, b) = xgcd(encryption_key, (p-1)*(q-1))
print d, a, b
print a*encryption_key + b*(p-1)*(q-1)
```

So we find `a = 3` as the decryption key.

```
print (decryption_key*encryption_key) % ((p-1)*(q-1))
```

The problem with a small modulus factor is that the system is easy to crack, as shown above.

Assignment Two. Experiment with increasing sizes of p and q to make the modulus factor that is very hard for Sage to factor. In your experiment, take primes of increasing number of digits. Use successive applications of the `next()` method on a `Primes` object. To obtain primes with k digits, take 10^k as argument of `next()`. Measure how long you must wait for the result of factor. Extrapolate from these timings a save size for the values of `p` and `q`. Justify your choice of size, referring to the algorithm to factor a number `n`: try all divisors from 2 to the square root of `n`.

3. a password manager

The messages we will encrypt will be strings like

```
s1 = "urla logina passa"
s2 = "urlb loginb passb"
```

The module `cryptfun.sage` contains the definition of `encrypt`. The function `encrypt` takes on input a string, the encryption key, modulus factor, and the name of a file. It returns the number (the encryption of the string) that was appended to the file.

```
mycode = mydir + "cryptfun.sage"
load(mycode)
datafile = mydir + "mypasswords.txt"
```

After loading the code for `encrypt` and defining the `datafile`, we can do

```
n1 = encrypt(s1, encryption_key, modulus_factor, datafile); print n1
n2 = encrypt(s2, encryption_key, modulus_factor, datafile); print n2
```

and we see the numbers

```
212043940688668233953445980695952 6
212044094033464872375461405224852 3
```

that are written to the `datafile`. The function `decrypt` takes on input a number of a line on file, the decryption key, modulus factor, and the name of a file. It returns the decryption of the given line on file.

```
d1 = decrypt(1, decryption_key, modulus_factor, datafile); print d1
d2 = decrypt(2, decryption_key, modulus_factor, datafile); print d2
```

and what is printed is

```
urla logina passa
urlb loginb passb
```

Assignment Three. Write code for the encrypt and decrypt functions. The example above is with strings that do not contain commas or special symbols and with small keys to show that you could complete this assignment independently from the other two assignments. In your answer you have to illustrate that your prototype works on plausible data (I do not need to know your actual account information, just invent some account urls, login names and passwords) and with sufficiently large numbers to guarantee the security of the encrypted data on file.

4. the deadline is Monday 22 February 2016, at 11AM

Bring to class the printout of a Sage notebook or SageMath worksheet that contains

1. listings of the Sage functions you have written; and
2. output of the experiments.

Structure your document along the answers to the assignments. Write appropriate comments.

This project must be solved individually, collaborations are not allowed.

Your worksheet should run like a program, via the menu `Action` → `Evaluate All`.

In addition, also email your Sage notebook or SageMath worksheet and the code for the functions to `janv@uic.edu`.

If you have questions, concerns, or technical difficulties, feel free to come to my office for help.