SI 335 Spring 2012: Project 4

This project contains only electronic (programming parts). You do not need to hand anything in on paper. The electronic parts (code) must be submitted according to the instructions on this page.

Submissions for this project are due before the beginning of class on **Wednesday, March 21**. Two weeks prior, by **2359 on Wednesday, March 7**, your public key for Part I must be submitted by email. Late submissions will not be accepted, in accordance with the class policy.

Coding Guidelines:

- It should be clear what your code is doing. The best way is to use sensible variable/function/class names, and to clearly state what every non-trivial chunk of your code (function, long loop, long block) does.
- Your program will be tested by an automated script before I even look at your code. This means that
  - Your program **must compile without errors**, on the CS Linux environment. If you use my included makefile, you should be able to run `make` and then `make clean` without any errors. I recommend making back-up copies frequently in case you break something and can't figure out how to fix it. Commenting out half-working code may garner some partial credit.
  - Your program must **behave exactly as I have specified**. Extra messages like "hi prof" are nice and friendly, but they will confuse my auto-testing script and cost you dearly. When arguments are specified on the command line, they must come from the command line, and when input is specified to come from standard in, it must come from standard in. For this project, you do not need to do any file I/O.
  - Your **file and folder names must be exactly as specified** as well. Yes, this includes capitalization. For this project, I'm giving you starter code for everything, so you shouldn't have to change any file names.
- Follow the course honor policy: document any external sources, and don't produce or look at any code in collaboration with any classmate.

Overview

In this project you will implement key generation, encryption, and decryption for the RSA algorithm. This is a big undertaking, but don't worry; it will be step-by-step, and a lot of the code for the multiple-precision integer arithmetic has been provided for you.

At the end, you will have a program called `rsa` that runs like:

```
proj4$ make
g++ -O3 -Wall -Wno-sign-compare -Wno-unused-function -c rsa.cpp
g++ -O3 -Wall -Wno-sign-compare -Wno-unused-function -c posint.cpp
g++ -O3 -Wall -Wno-sign-compare -Wno-unused-function -o rsa rsa.o posint.o
proj4$ rsa -h
Generate a public-private key pair:
      rsa -k
Encrypt a message with public key:
      rsa -e <e> <n>
Decrypt a message with private key:
      rsa -d <d> <n>
proj4$ rsa -k
91980840823 262004802161
235914929287 262004802161
proj4$ rsa -e 91980840823 262004802161
Hide me please!
165880826654 134843922670 84501887192 204706605960
proj4$ rsa -d 235914929287 262004802161
165880826654 134843922670 84501887192 204706605960
Hide me please!
```

Deliverables

By 11:59pm on Wednesday, March 7, you need to respond to my (first) automated email asking for the public key (e, n) that you have generated, according to Part I below.

By the final due date, you need to respond to my (second) automated email asking you to decrypt the message I posted for you and to encrypt another message back to me.

By the final due date, you also need to submit a folder named `proj4` that contains the following files:

- `rsa.cpp`: This is your top-level program (with a `main` method) and will contain all the code that you contribute.
- `Makefile`: This needs to have a target `rsa` that compiles your rsa program. You probably won't need to change this file.
- `posint.hpp`: This is part of the `mp` package and you must not change this file.
- `posint.cpp`: This is part of the `mp` package and you must not change this file either.

If you want to include some code in helper files, you must also submit a `README.txt` file that explains how your code is organized. Otherwise, everything you write should be in the `rsa.cpp` file.

You may include any other files in your submission, but I will ignore them.

Multiple-Precision Arithmetic

Since we're going to be implementing RSA, that means there we will have to deal with some very big integers, ones that can't be stored within a single `int`.

There are many good, free packages available for doing this that have been developed by researchers around the world. The most popular is GNU's GMP package .

However, we're going to use a much smaller, more primitive version that I have developed just for you called `mp`. If you navigate to the directory `~roche/mp` in the Linux environment here, you will see the whole thing, along with an executable program called `calc` that you can use as a bignum calculator.

Since the RSA algorithm only deals with positive numbers, the only part of the `mp` library that you will be using is the `PosInt` class. Unsurprisingly, this is a class to represent (arbitrarily large) positive integers.

You can have a look at the header file posint.hpp for the `PosInt` class to see what the interface is like. What you should notice is that you will use these guys very differently than how you would use a regular `int`.

For example, the "dumb" way of doing exponentiation might be written in pseudocode as:

```
pow(n,k)

1 res := 1
2 for i from 1 to k do
3   res := res * n
4 return res
```

Writing this in C++ using the `PosInt` class would look like:

```
1 void pow (PosInt& res, const PosInt& n, const PosInt& k) {
2   PosInt i(1);
3   PosInt one(1);
4   res.set(1);
5   while (i.compare(k) <= 0) {
6     res.mul(res, n); // Could also write res.mul(n)
7     i.add(i, one);   // Could also write i.add(one)
8   }
9 }
```

Here are some things to notice:

- You can't use the usual C++ operators for arithmetic. So for example to add, you have to use the `add` method in the `PosInt` class, not the `+` operator. This seems annoying, but makes it easier to keep track of exactly what is happening.
- There are no automatic conversions from regular **int**s. This means we can't do something like "i = i + 1" directly, but instead must first create a `PosInt` object (called `one` here) to hold the value 1, and add that to `i`.
- Rather than using the actual return values, most `PosInt` functions take the return value as the first parameter to the function, which is set by the function. The reason for this is to avoid too much unnecessary copying of these (potentially large) integers. So most `PosInt` functions actually just return a reference to the value that gets changed, rather than returning something new.
- The method `set` is used to set the value based on that of another `PosInt` or a regular **int**, and the `convert` method is used to convert a (small) `PosInt` back into a regular **int**.

Of course you can look through any of the files in the `mp` library to see many more examples of how these routines work. You should also know that the way the library has been designed is to throw exceptions of type **const char*** when something goes wrong (such as divide-by-zero). You can catch these exceptions very similarly to how you would in Java:

```
try {
  // Your PosInt-related code here
}
catch (const char* err) {
  // Let's just print the error message and quit.
  cerr << err << endl;
  exit(1);
}
```

Finally, while the normal I/O operators `<<` and `>>` will work just fine for reading and writing `PosInt`s, for debugging you might find the `print_array` method useful. This prints the digits of the actual number in an array just like we did in class, so you can see what the actual representation looks like.

Part I: Key Generation

When your `rsa` program is run with the `-k` flag, that means to generate a public/private key pair, randomly. You should follow the key generation algorithm from Section 5.4 of Unit 3. In particular, you will have to

1) Choose two random PRIME numbers p and q
2) Multiply them together to get n
3) Compute $\varphi(n) = (p-1)(q-1)$
4) Choose a random public exponent e
5) Use the XGCD algorithm to compute d as the multiplicative inverse of e modulo $\varphi(n)$

To do this, you will have to use some of the following functions from the `PosInt` class: `mul`, `sub`, `pow`, `set`, `xgcd`, `mod`, `rand`, and `MillerRabin`. Keep in mind that the `MillerRabin` method only runs the probabilistic primality test *one time*, so you should run it more than once to make sure it is correct before assuming some number is prime.

The output should be two pairs of numbers, corresponding to the public key (e, n) and the private key (d, n), respectively. Each pair should be on a separate line and the two numbers in the pair should be separated by a single space, as in the example from the beginning.

Now as long as you generate valid public/private key pairs, that are randomly generated (should be different every time you run the program), I don't care too much how you might choose to do it. The only requirement is that **your modulus n must be at least** 2^32. This is because we will be encrypting four bytes at a time, so we need at least a 32-bit modulus to make everything work. So you will have to choose your p and q accordingly to make n this large. Of course you can make it much larger, but not any smaller.

Deliverables

1) Modify the `rsa.cpp` program so that it spits out your randomly-chosen public and private keys when run like `./rsa -k`.

2) **By 2359 on Wednesday, March 7** email me (responding to the special email I send out) with your public key (only!). I will post everyone's public key on the website, along with secret messages to each of you.

3) **IMPORTANT**: You may of course keep modifying your `rsa.cpp` program between March 7 and March 21, the final submission deadline. But the public key you submit on the 7th must be "typical" of an output from the final program you submit. For example, if the key you submit is 200 bits, but your RSA program usually spits out 40-bit keys, then there will be a problem.

Part II: Encryption and Decryption

Now that you have some keys, it's time to encrypt and decrypt some messages! There are really four tasks here:

Modular exponentiation

The heart of the RSA cryptosystem is encrypting and decrypting messages with the public and the private key (respectively). If the public key is the pair of integers (e, n), and the message is a single number $M < n$, then we encrypt it by computing

$$C = M^e \bmod n$$

Sometime later, the recipient can decrypt this message by knowing the private key (d, n) and computing

$$M = C^d \bmod n$$

Now of course ultimately the message to be encrypted will consist of a whole series of numbers $(M_1, M_2, \ldots)$ and the encrypted messages will be another list $(C_1, C_2, \ldots)$.

But the basic operation of raising one big integer to another big integer exponent, modulo a third big integer, is really the central operation of the while RSA scheme! So the first thing you need to do is implement the function

```
void powmod (PosInt& result, const PosInt& a, const PosInt& b, const PosInt& n)
```

which will compute $a^b \bmod n$ and store the result in (you guessed it) `result`. The prototype for this function is already there in the starter code; you just have to fill it in.

Now there's a dumb way to do this, and a smart way... Of course I want you to do it the smart way. This means using the square-and-multiply algorithm as discussed in class, and taking the intermediate results mod n at every step along the way. If you don't do it like this, it'll never finish in time once you start plugging in the big numbers for RSA!

From characters to messages

When your program is run like `rsa -e <e> <n>`, it should use the public key (e, n) to encrypt a message. The message will come in via standard in, one ASCII character at a time. Now as each character is read in with `cin.get()`, it is converted to an 8-bit number (between 0 and 255). Your program must take the characters, four at a time, and convert each group of 4 ASCII characters (8-bit numbers) into a single 32-bit number, which will be the message M that is used.

Fortunately, most of this has already been written for you in the starter code. The following diagram should also help illuminate what's going on:

```
Characters read: "RSA FUN"
...
To ASCII:
  R=82     S=83     A=65     " "=32    F=70     U=85     N=78
01010010 01010011 01000001 00100000 01000110 01010101 01001110
...
Group into two 32-bit messages:
01010010010100110100000100100000 01000110010101010100111000000000
M1 = 1381187872 (stores "RSA ")  M2 = 1179995648 (stores "FUN\0")
```

Encrypting messages

After converting the ASCII characters to big numbers like above, the last thing your `rsa -e` program will do is encrypt each of these numbers, one at a time, using the formula $C = M^e \bmod n$ and the `powmod` function you already wrote. The output should be a list of the big numbers for each C, all on a single line and separated by spaces.

See the example at the top for how this should look when you are done.

Decryption

Your decryption program will be run like `rsa -d <d> <n>`, and will use the private key (d, n) to encrypt a message (list of big numbers) that is passed in via standard in. The *first* thing the decrypting part of your program will do is use the formula $M = C^d \bmod n$ to convert each number in the input to another number M.

Each of these numbers M contains (at most) four ASCII characters. You have to extract them, one at a time, and print them out. This will involve dividing and modding by powers of 256 repeatedly.

Some of this has been written for you already, but not as much as for encryption. The good news is that it should be easy to generate examples to test your decryption algorithm if you already have the other parts working! Here's an example that might help:

```
Decrypted message: M = 1381187872
(in binary) M = 01010010010100110100000100100000
...
256^3 = 16777216
...
M / 16777216:
...
   R       S       A      " "
/------\/------\/------\/------\
01010010010100110100000100100000
\------/
       \
         \
           \
             \
               \
                 \
                   \
                     \
                      /------\
00000000000000000000000001010010
                      \------/
                         R
...
M % 16777216:
...
   R       S       A      " "
/------\/------\/------\/------\
01010010010100110100000100100000
        \------/\------/\------/
           |       |       |
           |       |       |
        /------\/------\/------\
00000000010100110100000100100000
        \------/\------/\------/
           S       A       " "
```

Deliverables

This part is the bulk of the project. You should be very happy once you get this far! Now here's what you need to do:

1) Modify the `rsa.cpp` program so that `rsa -e <e> <n>` encrypts ASCII characters using the public key and prints out encrypted numbers, and `rsa -d <d> <n>` decrypts numbers using the private key and prints out the ASCII characters that make up the original message.

2) By the final deadline, go to the website and look up the message I have encrypted for you, as well as my own public key. Then you have to email me (responding to the special email I send out just for this purpose) with the following two things:

   a. The decryption of the message I sent you, and

   b. An encrypted message to me. It should be relatively short, and should have your own name in it somewhere.

   Of course you should use your program to do both of these things!

Part III: Faster Multiplication

Once everything is working, we want it to be as fast as possible of course! Right now the multiplication algorithm is just the standard algorithm that costs $\Theta(n^2)$ to multiply to n-digit numbers.

To speed this up, I want you to implement Karatsuba's algorithm for integer multiplication. There are two extra functions that I have defined in the `PosInt` class for doing "faster" multiplication, and some skeleton of their implementation is near the bottom of the starter code for `rsa.cpp`.

The first function is

```
PosInt& PosInt::fasterMul (const PosInt& x, const PosInt& y)
```

which is the top-level function to do the multiplication of x times y and store the result in the `PosInt` object that it was called on. You should use it just like you would use the usual `mul` function. Mostly what this function has to do is set things up, do some memory allocation and zero-padding, and make the first recursive call to the recursive `fastMulArray` function below. It has pretty much been written for you already...

Which means you will really spend your time implementing the second function:

```
void PosInt::fastMulArray (int* dest, const int* x, const int* y, int len)
```

This is a *static* function in the `PosInt` class, meaning that it doesn't get called on any particular object. Really, it's just working on arrays of **int**s instead of `PosInt` objects. This (should) make it easier to set up and call your recursive calls, since they will just be on sub-arrays of the original ones. The basic skeleton of this function is there, but you will definitely have to look back at the karatsubaMul algorithm from Unit 4 and implement it very carefully, step by step.

Once you get this working, you should incorporate it into everything you did in parts II and III for the RSA algorithm. Like I said, this should really just involve replacing some calls to `something.mul(...)` with `something.fasterMul(...)`.

Finally, **add a new option flag to the `rsa` program**. If I run something like `rsa -m <a> <b>`, it should use your new code to multiply a and b and print the resulting number to standard out (no input required).

For example:

```
proj4$ rsa -m 4278 87124
372716472
proj4$ rsa -m 41274981789257 90876523976523
3750926872201963955201613411
```

And feel free to play around with it! The ultimate goal is *fast code*, which means you might want to tweak with the algorithm one way or another to really make it as fast as possible. You have permission to do pretty much anything you like, as long as it actually makes an improvement and you carefully document what you are doing.

Deliverables

1) Modify the `rsa.cpp` program to complete the implementation of the `fasterMul` and `fastMulArray` functions. Then add a new `-m` option to the program that reads two numbers from the command line and prints out their product.

Part IV: EXTRA CREDIT

The extra credit is about having the fastest, secure RSA implementation. Note that there is one big prize but also **multiple smaller prizes** and also **opportunities for sabotage**.

The basic idea is to get your encryption and decryption to be really fast, without sacrificing security. It's based on the following general principles:

- **The bad guys have more computing power than you.** I will run your encryption and decryption algorithms with a time limit of just a few seconds, but there is no time or memory limit on what your classmates can do to try and crack your key.
- **Security is more important than speed**. I am going to encrypt a secret message to you using your public key and post it on the website. If anyone else sends me that secret message, you are not eligible to receive any extra credit points.

- **But a slow system is useless.** Like I said, I will test your code for encryption and decryption using a timeout of just a few seconds. If you choose a key size so large that encryption takes a long time, or don't write good code, then no one will use your cryptosystem even if it's really secure.

So the extra credit competition is to get the fastest encryption and decryption of random messages that I choose, **using random keys generated by your own program**. But in case you are tempted to just make the keys you generate really short, remember that if anyone cracks the message (for example by factoring your public modulus n), then you are not eligible for any extra credit.

Deliverables

1) Nothing extra to get the extra credit, just submit your solutions to the other parts as usual.

2) If you crack someone's secret message, send me a (regular) email. I will update the website as the messages get cracked. Also since no points are directly associated with cracking other people's keys, **you can work together to crack other people's messages**. However, this doesn't mean you can violate the honor policy for the rest of the project, for example by looking at any classmate's code that will be submitted.