CS 237	Discrete Mathematics I

Bigness

RSA *Exploration*

Goal

The purpose of this exploration is for you to implement and exercise a version of the RSA cryptosystem to understand how it works in a BIG way.

Requirements

Write a C++ program that is *adequate* for solving RSA encryption/decryption exercises 46 and 47 on page 246. (Note: It will *not* be adequate for e-commerce!)

You will need to implement a fast modular exponentiation algorithm, perhaps using the pseudocode Rosen gives on page 226 (Algorithm 5), but you're free to use something else, as long as it's better. This function is called `modPow`.

You will also need two more functions to compute the "modular inverse" as per Theorem 3 on page 234. These functions are called `extendedEuclideanGCD` and `findInverse`.

Your program should accept all required input via command-line arguments, as per the stub `rsa.cpp` file provided. The supplied `main` function ensures that the output of your program makes it clear what it's doing for every step in the encryption/decryption process. The `main` function has a few problems, but don't bother fixing them. For full credit you will have the opportunity of creating — in a separate file — new core functions together with a few helper functions that will use the **ZZ** class instead of `int` types. Name this separate file `rsaZZ.cpp`, and see the **Grading Criteria** for more information.

If conditions are right, you can build, test and submit your code in the Linux Lab via the command:

```
make it just so
```

If your `int`-based code passes all tests, you will see the following four lines of output:

```
encryption of attack succeeded
decryption of attack succeeded
decryption of silver succeeded
encryption of silver succeeded
```

If there is a mismatch of actual versus expected output, you'll see something more like the following:

```
1,2c1,2
< 0120 2001 0311
< 0286 0798 0425
---
> 0019 1900 0210
> 2299 1317 2117
make[1]: *** [test_rsa_1] Error 1
make: *** [just] Error 1
```

The '<' in front of a line identifies actual output, whereas the '>' identifies expected output. As visual inspection shows, they are different.

Now, how would you handle large primes and messages? Suppose you were to find a pair of prime numbers that are large enough to be able to RSA encrypt/decrypt your name (first and last), converted to a base-27 number using only uppercase alphabetic letters and the '@' character (standing in for the space character). You could then use your RSA code (which again, you must first modify to use **ZZ** type parameters and return values) to actually do the encryption and decryption of your name.

For example, my name (`RICK@NEFF`) treated as a base-27 number using

```
'@'  =  0    'I'  =   9    'R'  =  18
'A'  =  1    'J'  =  10    'S'  =  19
'B'  =  2    'K'  =  11    'T'  =  20
'C'  =  3    'L'  =  12    'U'  =  21
'D'  =  4    'M'  =  13    'V'  =  22
'E'  =  5    'N'  =  14    'W'  =  23
'F'  =  6    'O'  =  15    'X'  =  24
'G'  =  7    'P'  =  16    'Y'  =  25
'H'  =  8    'Q'  =  17    'Z'  =  26
```

converts to base-10 (decimal) as

$$18 \cdot 27^8 + 9 \cdot 27^7 + 3 \cdot 27^6 + 11 \cdot 27^5 + 0 \cdot 27^4 + 14 \cdot 27^3 + 5 \cdot 27^2 + 6 \cdot 27^1 + 6 \cdot 27^0 =$$

$$5083731656658+94143178827+1162261467+157837977+0+275562+3645+162+6 = 5179195214304.$$

This 13-digit number only requires two 7-digit primes to RSA encrypt/decrypt, but most of your names will convert to bigger numbers. Encrypt your name-number and then decrypt the result. Convert the decrypted number back to base-27 to see if your name is recovered or not.

If conditions are right, the same magic incantation will serve to build, test, and submit your `ZZ`-based code:

```
make it just so
```

You might have noticed that there is another file (`rsa_theory.pdf`) provided that you might find helpful, as it's completely self-contained.

From the authors' abstract: *This is our proof of the RSA algorithm. There are probably more elegant and succinct ways of doing this. We have tried to explain every step in terms of elementary number theory and avoid the 'clearly it follows...' technique favoured by many text books.*

Where is the **Chinese Remainder Theorem** used in this exposition? (It's not mentioned by name.)

Grading Criteria

The rubric below is meant to guide you in the production of a solution of exceptional quality. (In this case, given the well-defined nature of the task, exceptional quality is not exceptionally difficult to achieve.)

	Exceptional 100%	**Good 90%**	**Acceptable 70%**	**Developing 50%**	**Missing 0%**
Application — using what you've learned 30%	Achieved new knowledge and figured out how it applies. At least ten elements of a good write-up are present.	Achieved new knowledge and figured out how it applies. At least nine elements of a good write-up are present.	Achieved new knowledge and figured out how it applies. At least seven elements of a good write-up are present.	Achieved new knowledge and figured out how it applies. At least five elements of a good write-up are present.	No application of anything learned. Zero elements of a good write-up are present (M.I.A., in other words).
Correctness/ Completeness 40%	**Good**, plus your submitted `rsaZZ.cpp` code passes the required test perfectly, **without modifying** the supplied `rsaZZmain.cpp` file.	**Acceptable**, plus your submitted `ZZ`-based code (`rsaZZ.cpp`) compiles and runs.	**Developing**, but the `int`-based code passes all four of the required tests **without modifying** `main` in any way.	Submitted `int`-based code (`rsa.cpp`) compiles and runs, however, the `main` function was modified.	Code does not even compile.
Elegance 30%	**Good**, plus the `findE` and `findPAndQ` functions are each implemented in at most **four** lines of code using **NO** hardwired primes or other "magic" numbers. Other elegancies together make the total number of lines at most three dozen.	**Acceptable**, plus `modPow` and `findInverse` are properly converted from the `int`-based versions to the `ZZ`-based versions, and the helper functions `fromBase27` and `toBase27` are implemented in at most **five** lines of code each.	Code is correct, and efficient—e.g., `modPow` is **not** a direct implementation of Rosen's pseudocode, and `findInverse` is implemented such that it is clear that proper attention was paid to modularity—i.e., cohesion and coupling.	Code is correct, as first and foremost, an elegant solution is a correct solution.	No elegance whatsoever. The code is not even correct.

NOTE: for line counting purposes, don't count the function header line or any line with just a single curly brace (open or close) on it.

NOTE WELL: while you are free to collaborate with each other, no collaboration is allowed with anyone (including the tutor) who has done this exploration in the past.

Comments

- This was my favorite exploration so far. I have been really interested in cryptography, and being able to go through some of the steps of cryptography was really awesome! I liked having a way to check to see if the program was working. That really helped with making sure that at least I was on the right track. I know that there are many ways to improve the program, but it worked. I also liked going to the lab. All in all, a VERY enjoyable exploration, and I really think that collaboration made it a lot more enjoyable.
- I really enjoyed seeing the encryption/decryption work. Not just because I finally really understand how the whole process works, but to see and understand each part function was extremely satisfying. I also enjoyed having to work through the modular exponentiation. Finding an algorithm that would do this was intriguing. And I finally collaborated on this project! It was a lot more enjoyable seeing more than just my own ideas.
- By collaborating with other people it sure made completing the assignment so much easier. I really enjoyed being able to collaborate with other students on the exploration. I liked the fact that we only had to focus on the encryption/decryption of the program for time constraint reasons. With the time that was given for the assignment I think that it would have been very difficult to complete the entire assignment. I liked how I was finally able to see how the encryption/decryption actually worked. When just doing the math, and completing the exercises it was still a little unclear to me on how the algorithm worked. By having to write code for this algorithm, it really made me understand the fundamentals of how it works.
- I liked being exposed to the idea of the "make it just so" shell script/macro. I also liked being able to find incomplete algorithms and fix them (the spaces thing). In general, I also like assignments that don't take me too long to finish, but still teach me something. The only thing that might have been better is if the lab had been better described in class. However, the instructions in the PDF were sufficient (with a helpful shoulder to look over), so that's not a big deal.
- This exploration was an excellent view into the world of RSA. It made me realize just how complicated and delicate the process is. I came to understand many truths about data encryption. I read an article online that explained how hackers became so sophisticated, they could determine private keys simply by figuring out how long the process took. By knowing how fast a system is and factoring out the number of cycles completed, hackers could drastically the limit the possibilities to just a few values. Who would've thought that in order to be ultimately secure, the system would have to be redesigned to take the same amount of time for all encryption/decryption? Hackers really do think of everything! This assignment was deceivingly straightforward. It seemed only a matter of doing the math right, but bugs prevailed in a number of forms in other areas of the program.
- This assignment was interesting. Though less of a creative feat than other assignments, it was nice to have the feeling of completion (more so than other assignments anyway!) that a test base gives. I also liked doing something that has a lot of real-world practicality.
- I really liked applying one of the concepts we've been learning. It's just been quite enjoyable. I got it functional without much difficulty (though it may not be the best implementation because it's really simple). I enjoyed talking with some of my classmates about it because I felt like I had a grasp on the concepts. Hopefully I was able to help some others although maybe only a little. I was trying to find ideas for making mine better but I mostly ended up trying to help other people get theirs functional.

This was also one of the times I enjoyed working with already somewhat functioning code. I wasn't too lost looking at main and I feel like with a little more work I could have fixed the issues I had dealing with messages that include spaces.

- Of all the programs we have written, this felt the most practical. It was really nice to see our work in action. I also liked how this gave us an opportunity to learn to research more into this subject as well. I feel the urge to go and research more into encryption algorithms. Nothing was really too hard. I've been in the class long enough now to understand the wriggle room Brother Neff gives us, and I have learned to not hang myself with it! So this was actually a very enjoyable program. Just challenging enough to make me learn and pique my interest, but brief enough that it didn't burn me out.
- This was an awesome assignment. I loved it. It was really interesting to learn how this encryption works. I felt like I really got the modular exponentiation down. I like it. It's really intriguing to me how few steps it takes to do such a huge calculation. It seems like it's cheating. Math with that big of numbers shouldn't be so easy. I thought that having all the main already written was really helpful. I have a nephew in high school who is pretty smart and he is about the only one in my family whose eyes don't glaze over when I start talking about computer stuff. I was so excited about this that I bounced the ideas off him to see if I could get him worked up too. I did!
- I enjoyed exploring the different ways of implementing the modular exponentiation function and the modular inverse function. This exploration became fun when I was able to collaborate with other students. I liked how it was short coding yet challenging. I liked the most how Brother Neff, like on all assignments, was willing to sit down with us and inspect our code. It taught me to "ask and ye shall receive."