CPS 222: Computer Science III

Project 1: RSA

Important Dates
Checkpoint 1: 11:59pm, Tuesday, 9/4
Checkpoint 2: 11:59pm, Tuesday, 9/11
Checkpoint 3: 11:59pm, Tuesday, 9/18
Checkpoint 4: 11:59pm, Tuesday, 9/25
Final deadline: 11:59pm, Tuesday, 10/2

At each checkpoint you will receive some high-level written comments intended to help you to improve your solutions (do not expect comments to identify every error/bug!). Each numbered part will also receive a score from the following rubric:
0 points: *No substantive attempt submitted or doesn't compile.*
1 point: *A good start but major functionality is missing or incorrect.*
2 points: *Mostly functional, but has some important errors or does not comply with the specifications.*
3 points: *Nearly complete; only minor issues remain.*
Your accumulated checkpoint scores will comprise 25% of your project grade.

Introduction

Public-key cryptography was a major advance in cryptographic technology and currently underlies the security the world relies upon to function. In older cryptographic systems, the sender and receiver typically share a key that allows messages to be encrypted/decrypted. This raises the tricky problem of how to transmit the *key* securely. You could encrypt the key, but then you'd need to share the key for that encryption, and so on *ad infinitum*.

In public-key cryptography, a person generates a matched pair of keys: the public key and the private key. The public key can be used to encrypt a message, but cannot be used to decrypt the message it encrypts! Only the private key can be used to decrypt messages encrypted using the public key. So, the public key can be transmitted freely, with little danger of compromising security. Anyone who intercepts the public key can only encrypt messages, not decrypt them.

The RSA algorithm (named for its creators, Rivest, Shamir, and Adleman, one of whom is an author on our algorithms textbook) was one of the first public key cryptographic systems and is still used today. In this project you will implement the RSA algorithm. First you will develop programs for encrypting and decrypting messages with provided keys. Next you will develop a program to generate keys. Finally, in the biggest part of the project, you will make your encryption much stronger by allowing your RSA implementation to handle integers with an unlimited number of digits.

Part 1: Encrypting and decrypting using RSA

In RSA, the public key is made of two numbers, e and n. One way to encrypt a message is to turn each character into a number x (say, the ASCII value of that character) and then transform x into the number $y = x^e \pmod{n}$ in the encrypted message. The problem is that e and n are often quite large, so we'll have to be clever about how to perform this operation.

1) Simple modPower
Reading:
Topic 1 – C++ Basics
Topic 2 – Common Control Structures
Topic 3 – Numeric types
(see also: Prata ch. 1, 2, 3, 5, 6, Lynda.com 1.1-6, 1.10-13, 1.15, 2.1-5, 5.1-3)

In *simplemodpower.cpp* write a `main` function and also the function
`unsigned int power(unsigned int base, unsigned int exponent)`,
which should return $base^{exponent}$. Note: there is no exponentiation operator in C++! There is a function called `pow`, but it only operates on floating point numbers, and we are doing integer math. So you'll have to do it yourself: multiply *base* by itself *exponent* times. Don't forget to handle *exponent* = 0!

Now implement the function
`unsigned int modPower(unsigned int base, unsigned int exponent, unsigned int modulus)`,
which takes three non-negative integer values: the *base*, the *exponent*, and the *modulus* value and should return $base^{exponent}$ (mod *modulus*). This version should be very simple: call the `power` function that you just wrote and then apply the modulo operator to the result. You don't need to gracefully handle *modulus* = 0 (which is undefined).

In `main`, write a simple test of your function. Print out the result from your function on the following test cases (print only the results, no other text, each on its own line, in this order):

a. Base: 2, Exponent: 2, Modulus: 3
b. Base: 2, Exponent: 3, Modulus: 3
c. Base: 12, Exponent: 12, Modulus 123
d. Base: 10, Exponent: 19, Modulus: 1019
e. Base: 12345, Exponent: 1234567, Modulus: 123
f. Base: 12345, Exponent: 1234567, Modulus: 123456789
g. Base: 12345, Exponent: 123456789, Modulus: 12345
h. Base: 12345, Exponent: 123456789, Modulus: 1234567891011

You can use the Python interactive shell to check your answers for the first several cases. You'll notice that for big numbers your program takes a little while. Remember, in the GNU/Linux shell you can use Ctrl+C to interrupt a program's execution! You'll also note that your program gets some results wrong. This is the result of **integer overflow** (review Topic 3 to make sure you know what that means!).

Now change your functions so they use `unsigned long long`s instead of `unsigned int`s in order to prevent some overflow. It should fix cases c and d, but not e-h.

2) Recursive exponentiation

If *e* is very large, the program you've written will perform a very large number of multiplications (a fairly expensive arithmetic operator). It is possible to perform exponentiation by performing O(log *e*) multiplications rather than O(*e*) using a recursive algorithm that you may recognize from a lab assignment in CS1. Fast exponentiation is based on a recursive definition of exponentiation:

$$x^e = \begin{cases} x^{\text{floor}\left(\frac{e}{2}\right)} \times x^{\text{floor}\left(\frac{e}{2}\right)} & \text{if } e \text{ is even,} \\ x \times x^{\text{floor}\left(\frac{e}{2}\right)} \times x^{\text{floor}\left(\frac{e}{2}\right)} & \text{if } e \text{ is odd} \end{cases}$$

Because it performs many fewer multiplications, a recursive algorithm based on this formula will be much faster than the iterative version, especially when *e* is large. In *recursivemodpower.cpp* copy your `main` and `modPower` functions from *simplemodpower.cpp* and implement

```
unsigned long long power(unsigned long long base,
                         unsigned long long exponent),
```

which has the same effect as your previous version, but does the exponentiation recursively. Note: to get the efficiency benefit, your recursive function should make *only one* recursive call. That is, you should not recompute $x^{\text{floor}(e/2)}$ every time you want to use it in an expression. Recursively compute it once; store its value; and use the stored value in calculations.

Your new program should produce results nearly instantaneously, though overflow is still a problem.

3) Incremental modulus

You've seen by now that even when we can directly compute x^e we might not even be able to practically store a number that large! Thankfully we can get around this issue by recognizing that (*a* * *b*) mod *c* = ((*a* mod *c* * *b* mod *c*) (mod *c*). That is, rather than performing the modulo operator at the end, it is safe to apply it *to each term in a multiplication* as well as the final result. This ensures that each intermediate value in the computation is relatively small and lowers the risk of integer overflow.

In *modpower.cpp* copy and paste your `main` function from *recursivemodpower.cpp*. Now write a `modPower` function that computes $base^{exponent}$ (mod *modulus*) recursively (no separate `power` function!). The modulo operator should be applied to the result of *every multiplication* (i.e. every time you multiply two numbers together).

Your final program should be able to instantly get every case right except the very last one, which will suffer from overflow. Hey, you can't win 'em all (actually you can; see Part 4). If you still see overflow in some of the cases, make sure you are applying the modulo operator to *every* intermediate term.

4) Encrypting a file

Reading:
Topic 4 – I/O
(see also: Prata ch. 17, Lynda.com 10.4)

Now you will write a program to encrypt a file. Copy your `modPower` function from Part 3 to *encrypt.cpp*. If all this copy/pasting of code makes your skin crawl, good! We will soon learn how to avoid this kind of code duplication, but grin and bear it for now. Your `main` function should:

- Open the file called *public.txt* and read two numbers from it: *e* and *n*.
- Open an input file called *plaintext.txt* and an output file called *encrypted.txt.*
- Read *plaintext.txt* character by character (see Topic 4, slide 15).
- Recall that `char`s are really just numbers (ASCII values)! For each character *x*, use `modPower` to compute the encrypted number $y = x^e$ (mod *n*) and write this number to the output file, followed by a newline.

Create a test message in *plaintext.txt* and encrypt a file using the provided *public.txt* file. After running

your program, *encrypted.txt* should have as many lines as there are characters in *plaintext.txt*, each line containing a single number. The GNU/Linux utility *wc* (short for "word count") may be helpful to check this. In your terminal, type *man wc* to learn more. By the way, the *man* command (short for "manual") brings up a help message for pretty much any command-line utility.

5) Decrypting a file

Decryption using RSA is very much like encryption. The private key, which is used to decrypt a message, is made of two numbers, d and n (the same n as the public key). To decrypt a message, take each encrypted number y and compute $x = y^d \pmod{n}$ to get the original character back.

Copy `modPower` to *decrypt.cpp*. Your `main` function should:

- Open the file called *private.txt* and read two numbers from it: d and n.
- Open an input file called *encrypted.txt* and an output file called *decrypted.txt.*
- Read each number in *encrypted.txt* (see Topic 4, slide 13).
- For each number y, use `modPower` to compute the decrypted number $x = y^d \pmod{n}$, typecast the result to a `char`, and write it to the output file. (Note that if you don't do the typecast, you will write numbers rather than characters to the file).

Decrypt the encrypted message from Part 4 using the provided *private.txt* file. If all is working properly, the resulting *decrypted.txt* should be exactly the same as the original *plaintext.txt*. The GNU/Linux utility *diff* can help you tell for sure (try *man diff* for more information).

6) Improving *encrypt.cpp* and *decrypt.cpp*

Reading:
Topic 5 – Multi-file Programs (up through slide 37)
Topic 6 – Arrays
(see also: Prata ch. 1, 2, 4, Lynda.com 1.9, 3.1, 3.3-4, 3.8)

As of now, your encrypt and decrypt functions only work with fixed filenames (*public.txt*, *private.txt*, *plaintext.txt*, *encrypted.txt*, and *decrypted.txt*). Alter both programs to take 3 command line arguments:

1. The name of a file containing the key (either public or private, as appropriate). The file will simply contain two numbers, separated by a space.
2. The name of the input file.
3. The name of the output file.

Hint: if your program crashes without doing anything, it's probably because you didn't give it the command line arguments it was expecting!

Another issue is that both *encrypt* and *decrypt* use the `modPower` function, and the best we could muster before was to copy it into both files. Duplicating code like that makes projects harder to maintain and more error-prone. If you wanted to make a change to `modPower`, you'd have to do it twice, causing many more opportunities to introduce bugs (or to forget to make modifications everywhere it appears!).

We now know how to fix this:

1. Create a file called *numberTheory.cpp* and add the definition of the `modPower` function.
2. Create a file called *numberTheory.hpp* and add the header for the `modPower` function (don't forget the include guard! See Topic 5, Slide 10).

3. Remove `modPower` from *encrypt.cpp* and *decrypt.cpp*.
4. Instead, have *encrypt.cpp* and *decrypt.cpp* include *numberTheory.hpp*.

Now create a *Makefile* for your project.

1. Create a rule for the object file *numberTheory.o*. Its dependencies should be its header and source files (*numberTheory.hpp* and *numberTheory.cpp*).
2. Also create rules for the executable files *encrypt* and *decrypt*, which depend on *numberTheory.o* and their respective source files (*encrypt.cpp* and *decrypt.cpp*).
3. Finally create a rule called *all*, which invokes the rules for *encrypt* and *decrypt* (check out how this is done in the provided examples!). This should be the first rule in your file.

Now you can use the *make* command to avoid having to constantly type long compilation commands. If you type *make encrypt*, *make* will automatically compile any necessary files that have changed. Handy! As you create more files, continue to add appropriate rules to your *Makefile*.

7) Unit testing

Reading:
Topic 5 – Multi-file Programs (slide 37 onward)

Right now the project is pretty simple, but eventually there is going to be a lot of code with lots of interdependencies. We'd better establish some good testing habits while we can. You'll be writing unit tests for every function as you go so that you can always feel confident that you are building on something solid. We will be using Catch as the unit testing harness and gcov/lcov to measure unit test coverage. For now create *numberTheory_TEST.cpp*. In that file create a `TEST_CASE` with a `SECTION` for your `modPower` function and implement tests for cases a-j from above (remember we know that it will fail on case k). While you are at it, check the coverage of these tests. Did we do a good job? Add tests to ensure 100% coverage. In general, by the final submission, your unit tests will need to have 85% coverage overall for full credit. Add a rule to your *Makefile* for *numberTheory_TEST* as well as a rule called *tests*, which will eventually compile all the unit test programs (like the *all* rule) – for now it need only invoke the rule for *numberTheory_TEST*. Be sure that your make rule compiles your test program to measure coverage.

---You have reached Checkpoint 1---

Files to turn in by 11:59, Tuesday 9/4:

simplemodpower.cpp, recursivemodpower.cpp, modpower.cpp, encrypt.cpp, decrypt.cpp, numberTheory.cpp, numberTheory.hpp, numberTheory_TEST.cpp, catch.hpp, Makefile

and, as always, don't forget to fill out and sign your cover sheet!

Part 2: Generating keys and representing large integers

Now we turn our attention to generating RSA keys. The security of RSA relies on the fact that obtaining the private key from the public key requires solving a computationally hard problem (factoring large numbers). Here is a description of the key generation algorithm:

1. Start with two prime numbers, p and q
2. Let $n = pq$

   Note: for RSA encryption to work, you must have $n > x$ for all numbers x that are being encrypted (so if you are encrypting ASCII values, n must be at least 256). The key must be generated with big enough primes p and q to create a big enough n to satisfy this condition.
3. Let $t = (p - 1)(q - 1)$

4. Pick any number e (usually a small value is selected) such that:
   a) $1 < e < t$
   b) e and t are *relatively prime* (that is, the greatest common denominator of e and t is 1)
5. Let d be the (unique) positive number such that $(ed) \bmod t = 1$
6. The keys are:
   a) Public (encryption): (e, n)
   b) Private (decryption): (d, n)

In order to accomplish all this, you'll need to implement a couple more functions. For each one you should implement a `SECTION` in *numberTheory_TEST.cpp* for thorough unit testing.

8) `isPrime`

In *numberTheory.cpp/hpp* implement the function
`bool isPrime(unsigned long long num)`,
which should take a number and return `true` if it is prime and `false` otherwise. The simplest way to determine if a number is prime is just to check all possible divisors and see if any of them divide the number evenly. Note that 0 and 1 are not considered to be prime, but 2 is prime. *Hint: the smallest divisor you need to check is 2; the largest is the square root of the number. If you* `#include <cmath>` *you will gain access to the* `sqrt` *function.*

9) `extendedEuclid`

Reading:
Topic 7 – Pointers
(see also: Prata ch. 4, Lynda.com 1.7, 2.3)

You may be familiar with the well-known, recursive Euclidean algorithm for computing the GCD of two numbers. Here is the pseudocode for that algorithm:

```
GCD(a, b)
1)  if b == 0
2)      return a
3)  else
4)      return GCD(b, a mod b)
```

You can look online for many lovely visualizations of why this algorithm works. For our purposes, though, we need a little bit more. The *extended Euclidean algorithm* computes the GCD of two numbers, but also computes two other numbers. Specifically, given a and b, the extended Euclidean algorithm additionally computes the integers x and y such that $ax + by = \gcd(a, b)$ (typically at least one of x and y is negative). The pseudocode for the extended Euclid's algorithm is:

```
EXTENDEDEUCLID(a, b)
1)  if b == 0
2)      x = 1
3)      y = 0
4)      return (a, x, y)
5)  else
6)      (d, x, y) = EXTENDEDEUCLID(b, a mod b)
7)      x′ = y
8)      y′ = x – y · FLOOR(a/b)
9)  return (d, x′, y′)
```

In *numberTheory.cpp/hpp* implement the function

```
unsigned long long extendedEuclid(unsigned long long a,
                  unsigned long long b, long long* px, long long* py).
```

An important thing to learn in this class is that **pseudocode is not code.** Pseudocode is a precise, language-independent way to specify an *algorithm* but typically cannot be directly transliterated into a program. There are almost always implementation issues to consider and details to fill in. So, in that spirit, note a couple things about implementing this algorithm in C++.

- First, the floor function is defined as the largest integer less than or equal to the given number. So, C++ *automatically* performs the floor function when it does integer division!
- Second, the algorithm returns three things, but C++ does not allow multiple return values. However, you can pass pointers to functions to act as additional return values. The prototype for `extendedEuclid` takes *four* parameters, two of which are pointers to `long long` values. The pointers are named `px` and `py` to emphasize that they are in fact *pointers* to `x` and `y`. Before the function returns, it should dereference these pointers and set the values of `x` and `y` appropriately (and return the gcd). Of course this means that before you call this function, you must declare two `long long` variables, so you can pass their addresses as parameters. See Topic 7, Slide 42 for a relevant example.

In your testing, verify that the return value is indeed the GCD of a and b, and that $ax + by = \gcd(a, b)$.

10) *keygen.cpp*

Once the functions in *numberTheory.cpp/hpp* are working correctly, you are ready to generate RSA keys. In a file called *keygen.cpp*, write a `main` function that will generate a matched pair of RSA keys using the procedure given above and write them to files. Your program should:

1. Take two prime numbers and two filenames as command line arguments.
   a) If the provided numbers are not prime, you program should print an error that communicates this and return 1 (indicating that the program quit on an error condition).
   b) The public key will be written to the first file. The private key will be written to the second.
2. Compute n and t.
3. Find the *smallest* value of e that satisfies the requirements.
   a) For each value of e, starting at 2, use `extendedEuclid` to compute its GCD with t.
   b) Stop when e is relatively prime to t (the GDC is 1).
4. When you have an e that is relatively prime to t, the x value given by extendedEuclid is d!
   a) To understand why, remember that `extendedEuclid` finds x and y such that $ex + ty = \gcd(e, t)$. But if $\gcd(e, t) = 1$, then $ex + ty = 1$. Since ty is a multiple of t, $(ex + ty) \bmod t = ex \bmod t$. Thus, x is the number such that $ex \bmod t = 1$, which is what d has to be.
   b) *Note*: x may be negative! If it is negative, you should add it to t to get a positive number with the same property.
5. Finally, print the two keys to their respective files. The public key file should contain e and n (separated by a space, followed by a newline). The private key should contain d and n.

Hopefully by now you feel confident that `isPrime` and `extendedEuclid` work correctly. You can test your *keygen* program by working through small examples by hand, and seeing if your output matches. Another good way to test is by encrypting and decrypting messages using your generated keys (make sure you use large enough values for p and q!).

X) Representing large integers

Through some clever algorithmic tricks, you've been able to efficiently perform calculations involving quite large numbers. However, once the key got to be 10 digits or so, even those clever tricks couldn't stop the overflow problem. The issue is that the security of an RSA key depends a lot on how large it is. Remember that part of the public key is the number $n = pq$, where p and q are the original prime numbers from which the keys were generated. So if you have someone's public key, all you have to do to crack the code is figure out how to factor n into two prime numbers. With those prime numbers, you can generate the private key and decrypt the message. If n is small, this is easy to do: just try all possible factorizations until one works. A 10 digit key would be crackable in seconds...on a cell phone...from 2003. The key is just not big enough!

Modern RSA keys need to have hundreds (or thousands!) of digits to be secure. The basic, built-in numerical types will clearly be insufficient for working with numbers that large (a `long long` can only represent numbers with at most 18 digits, let alone be used to do math with them!). As you know, integer overflow is not an issue in Python; you can do math with integers of arbitrary precision! So what is Python doing that C++ isn't? The built-in C++ integer types use *fixed-width* representations. That is, they have a set number of bits, and can therefore only represent as many numbers as there are ways to set the bits. In order to represent integers of arbitrary magnitude, we'll have to have a *variable-width* representation that can be as large as it needs to be to represent a given number.

We will represent a variable-width integer using an array, where each entry of the array holds a digit. If you want to represent a large number, you just need a big array! Representationally, we are only limited by the amount of memory on the machine. Specifically, a "really long int" will be an array of `unsigned int`s, where each entry will represent a single digit. For now you will implement a few basic operations on non-negative numbers. In the next part you will add more features (including negative numbers) and incorporate really long ints into RSA.

Implement these functions in *reallyLongInt.cpp/hpp.* You should create *reallyLongInt_TEST.cpp* with one `TEST_CASE` and a `SECTION` for each function. Add appropriate rules to your *Makefile.*

11) Really long ints from strings
Reading:
Topic 8 – References and const
Topic 9 – Dynamic allocation
(see also: Prata ch. 4, Lynda.com 1.8, 5.6)

We will want to be able to construct really long ints out of strings (for instance, a key we've read from a file). In *reallyLongInt.cpp* implement the function
`unsigned* rlIntMake(const string& numStr, unsigned& numDigits)`,
which should take a `string` of a number and convert it to our really long integer representation. It should **dynamically allocate** an `unsigned int` array of the appropriate size (see Topic 9, Slide 3), and fill it with the digits of the number. Remember that you have to keep track of the size of arrays yourself, so the function additionally takes a reference to an `unsigned int`, called `numDigits`. Before returning, the function should set this variable to the number of digits in the really long int being returned (so this is like a second return value).

The C++ `string` type has many of the same features as Python strings. You can access an individual character using the indexing operator (e.g. `numStr[2]`) and get the length of the string using

`numStr.size()`. Access to `string`s comes from `#include <string>`. You may assume that the string contains only digits. Again, remember that the string contains the characters that *look like* the digits, but numerically are the ASCII values of those characters (for instance, the character `'3'` is the number 51). You can figure out a way to compute the *numerical values* of the digits from the ASCII values (i.e. if the character is `'3'`, put the number 3 in the array, not 51).

Note that you can also call this function with C-style strings (null-terminated `char*`s) since they can be implicitly converted to C++-style `strings`. So, you should, for instance, be able to create a really long int by calling

```
unsigned xSize;
unsigned* x = rlIntMake("325", xSize);
```

For testing purposes, you'll want a way to print out really long ints. In *reallyLongInt.cpp* implement `string rlIntToString(const unsigned* x, unsigned xSize)`, which takes a really long int and generates a string of that number. *Hint: if you* `#include <string>` *you get access to the* `to_string` *function which can convert an* `int` *to a* `string`. *Strings can be concatenated with the* `+` *operator.*

Test your constructor and to-string function! If these functions don't work you won't be able to test anything properly! Always unit test as you go; *do not* leave it to the end!

12) Removing leading zeros

Really long ints should not include leading zeros. The only time `x[0]` should be 0 is when `x` represents the value 0. This will ensure that if two numbers are the same, their arrays will have the same contents. As a side benefit, it also provides a nice, easy way to test if a really long int is 0. To help maintain this property, implement the helper function `void rlIntRemoveLeadingZeros(unsigned* x, unsigned& xSize)`. It should take a really long int (and a reference to its size), and remove leading zeros by shifting the significant digits to the front of the array (adjusting the size as appropriate). This may leave some unused spots at the end of the array; that's okay. If `x` contains only zeros, its size should be set to 1.

Edit `rlIntMake` so it calls this function before returning. Now test!

13) Comparing really long ints

In *reallyLongInt.cpp*, implement the function

```
bool rlIntEqual(const unsigned* x, unsigned xSize,
                const unsigned* y, unsigned ySize).
```

It should take two really long ints and return `true` if they are equal (`false` otherwise). Your algorithm should take O(min(*xSize*, *ySize*)) time. Now in *reallyLongInt.cpp*, implement the function

```
bool rlIntGreater(const unsigned* x, unsigned xSize,
                  const unsigned* y, unsigned ySize).
```

It should take two really long ints, `x` and `y`, and return `true` if `x` > `y`, and `false` otherwise. You might want to work out a few examples on paper to come up with a simple test for determining whether one number is greater than another. Your algorithm should take O(min(*xSize*, *ySize*)) time.

I'm not kidding. Thoroughly test your functions for correctness (I certainly will!).

---You have reached Checkpoint 2---
Files to turn in by 11:59, Tuesday 9/11:
numberTheory.cpp/hpp, keygen.cpp, reallyLongInt.cpp/hpp, numberTheory_TEST.cpp, reallyLongInt_TEST.cpp, catch.hpp, Makefile
and, as always, don't forget to fill out and sign your cover sheet!

Part 3: ReallyLongInt class (part 1)

In the last part you probably noticed how your collection of `rlInt_____` functions started to resemble a class; you had functions to create a data structure and then other functions to work with that data. The main difference, and the main reason the idea of classes ever came about, was that there was no way to enforce a specific interface to the data. Anyone could, in principle, just reach into your arrays and mess with your stuff! Another problem was that a really long int couldn't be distinguished from other `unsigned int` arrays that had nothing to do with really long ints.
In this part you will create a `ReallyLongInt` class which incorporates the work you've already done, and adds some additional features (that's a pun – you'll get it later). Implement the class in *ReallyLongInt.cpp/hpp* (note the capital "R") and create *ReallyLongInt_TEST.cpp* so you can test as you go. Make a single `TEST_CASE` for the class and a `SECTION` for each public method.

14a) Default constructor
Reading:
Topic 10 – OOP
(see also: Prata ch. 10, 11, 12, 13, Lynda.com 4.1-6, 4.8-12)

The class should have three private member variables (please use these names):

- `bool isNeg` – the sign of the number (true if negative)
- `const unsigned* digits` – the digits of the number
- `unsigned numDigits` – the number of digits in the number

Yes, you have to deal with negative numbers now! The `isNeg` member is a `bool`, ensuring that a number is always either negative or positive. Unfortunately this would allow for two distinct values 0 and -0; we'll have to prevent that somehow.

The class will have five constructors. First implement the default constructor
`ReallyLongInt()`,
which should set the member variables appropriately in order to represent the value 0.

In *ReallyLongInt.cpp/hpp* implement this constructor. Note that `digits` is a `const` pointer. This means that you can't use it to change values in the array that it points to. This is a safety measure – we don't want to mess around with the digits of our number after it's created. It does slightly complicate the constructors, since if you simply allocate an array and assign it to `digits`, you won't be able to fill in the values! Instead, allocate the array and assign it to a temporary non-`const` pointer. Then you can fill in the digits as necessary. Finally, you can assign that pointer to `digits`.

14b) String constructor

You should also implement the private method
`void removeLeadingZeros(unsigned* x, unsigned& xSize) const`.
It should do pretty much exactly what the `rlIntRemoveLeadingZeros` function did in the

previous part. It takes an array and a reference to a number of digits, shifts the digits over, and sets the number of digits as appropriate.

Now implement the string constructor.
`ReallyLongInt(const string& numStr)`,
which corresponds to `rlIntMake(const string& numStr, unsigned& numDigits)`. There are a few differences. First, the string may now start with the negative sign '-'. You'll have to take this into account. Second, you have to call `removeLeadingZeros` on the array *before* assigning the array to `digits`, since `digits` is `const`! Finally, if after removing the leading zeros the number is actually 0, make sure that `isNeg` is `false`. This is how we will prevent the value -0.

In C++, constructors that take a single argument are used to perform conversions from one type to another. The existence of these constructors allow you explicitly convert (cast) a numeric type or a string to a `ReallyLongInt`. They also allow for *implicit* conversions. If you call a function or use an operator that is expecting a `ReallyLongInt`, and pass in a number instead, it will implicitly create a temporary `ReallyLongInt`, using the appropriate constructor! This will come in super handy later.

14c) Destructor
Now implement the destructor, which should simply `delete[]` the digits.

14d) toString
Finally, implement
`string toString() const`,
which corresponds to `rlIntToString` (except now it has to support negative numbers too).

Now would be a good time to test constructing and printing `ReallyLongInts`. If this doesn't work, you won't get far.... Make sure you test negative numbers, 0, “-0”, and strings with leading zeros!

15) Creating really long ints from numbers

We will also want the ability to convert built-in integer types to really long ints. Implement
`ReallyLongInt(long long num)`,

which should take an integer value and create a really long int of equivalent value. Note that this function will also work with other integer types, since they can be implicitly converted to `long long`. Some hints:

- If you `#include <cmath>` you will gain access to the `log10` function, which should help you in calculating how many digits the number has. (Warning: what is log(0)??)
- To extract the digits of the number, think about repeatedly applying the `%` and `/` operators.

Test, test test! Note that, if your procedure works properly, you shouldn't have to worry about leading zeros in this function. Also note that in C++, integer literals that start with 0 are octal (in base 8). So the number `051` is actually the decimal (base 10) number 41. So don't pass `051` in expecting to get 51 out!

16a) Making copies

The fourth constructor has a special name: the *copy constructor*. It makes a `ReallyLongInt` out of another `ReallyLongInt`:
`ReallyLongInt(const ReallyLongInt& other)`

The copy constructor is invoked, for instance, when you pass a `ReallyLongInt` as a parameter to a function. In fact, *every* class has a copy constructor by default. The default behavior is to simply copy the value of each member variable. This is called a ***shallow copy***, because if there are pointers, it only copies the pointer values, and does not make copies of whatever the pointers point to! Sometimes this is okay, but usually it's a bad idea: if you have two objects that point to the same array, what happens when one decides to delete their array? *Hint: it's bad.* Defining your own copy constructor allows you to perform a ***deep copy***. In this case, that means making a copy of the array of digits for the new object (as well as copying the values of the other member variables). Note that because this is a method of `ReallyLongInt`, you have direct access to private members of `other`, which is convenient.

Hey, have you been testing all this time? I hope so!

16b) Private constructor

The last constructor is notable because it is private. A private constructor?! Why on earth would that be useful? This is a constructor that only methods of the class can use. When doing arithmetic it will be useful to create a digits array to represent the answer and to fill it in digit by digit. Once the digits are determined, you'll want to put them inside a `ReallyLongInt` object. Implement:
`ReallyLongInt(unsigned* digitsArr, unsigned arrSize, bool isNeg)`.
It should call `removeLeadingZeros` on the given digits array and size, then directly assign the result to `digits` and `numDigits`. It should also set `isNeg` to the given value *unless the number is 0*, in which case `isNeg` should be set to `false`, regardless of the given value.

Note that this constructor essentially takes over responsibility for `digitsArr`. It was created somewhere else in the code, but this constructor alters its values and then points `digits` to it, which means that when the object is deallocated, it will delete the array. In many cases this is a very dangerous thing to do: we don't know where this array has been! What if someone else starts messing with its values? What if someone else deletes it?? The only reason we can comfortably do this is that it can only be invoked by our own code, and hopefully we can trust ourselves to use it correctly.

In order to test this constructor on its own, you can make it temporarily public. Then you can fill in your own digits array to pass in and ensure that the resulting `ReallyLongInt` is as it should be.

17) Comparison operators

Implement the public method
`bool equal(const ReallyLongInt& other) const`.
It should do what `rlIntEqual` did with a few differences:

- Instead of taking two operands, these methods should treat `*this` (that is, the object the method is called on) as the left operand and `other` (the argument) as the right operand. Remember that you can access private members of `other`.
- You must now take sign into account. Numbers with different signs should not be equal.

Now you are ready to tackle the greater-than operator. Implement the private helper method
`bool absGreater(const ReallyLongInt& other) const`,
which should return true if the *absolute value* of `*this` is greater than the *absolute value* of `other`. That is, you should **ignore the sign** of these two numbers. Luckily, you already did that! This should be nearly the same as `rlIntGreater`.

Now, using that as a helper method, implement a public method that takes sign into account:
`bool greater(const ReallyLongInt& other) const`.
(Avoid code duplication! The `greater` method can be small if it calls `absGreater` appropriately).

Have you been testing as you go? If not, make sure you go back and test all of this!

18) Unsigned addition

Do you remember the "column addition" algorithm you (probably) learned in elementary school? That's the one where you line up the numbers and add the digits in each column, keeping track of the carry value. In order to add two really long ints together, you will need to implement this algorithm. Before you start, I **highly** recommend that you work through several examples on paper. It may have been a long time since you used this algorithm (calculators are wonderful devices). Also, you've probably never tried to write the algorithm down in full detail before. As you work through examples, pay careful attention:

- How many digits could/should the result have?
- If the numbers are arrays, what indices are you working with?
- What temporary variables are you implicitly creating?

It would be a good idea to try to write the algorithm down in pseudocode before you start typing.

As with comparison, it will be helpful to be able to perform these operations without taking sign into account. Implement the private helper method
`ReallyLongInt absAdd(const ReallyLongInt& other) const`,
which should create and return a `ReallyLongInt` that is the sum of the absolute values of `this` and `other` – so the result will always be non-negative. You should fill an array with the digits of the result and then use the private constructor at the end to create the object to return. Your algorithm should take O(max(*this.numDigits*, *other.numDigits*) time.

Now implement the public method
`ReallyLongInt add(const ReallyLongInt& other) const`.
For now it should just call `absAdd`, so you can test it. Obviously it will return incorrect results for negative operands. You'll fix that later.

19) Unsigned subtraction

Now it's time to implement subtraction. You remember how to do column subtraction, right? It's a lot like column addition, but instead of carrying over to the next digit when necessary, you *borrow from* the next digit. You should probably do some examples on paper to refresh your memory and to think about how to fully specify the algorithm.

Now implement the private helper method
`ReallyLongInt absSub(const ReallyLongInt& other) const`.
It should create a new `ReallyLongInt` that contains |`*this`| – |`other`|. The result of this might be negative! Also, note that for column subtraction to work, the number with a larger absolute value must be "on top." So if you are asked to compute 34 - 512, you actually compute 512 - 34, and return the negation of the result. The sign of the result should be calculated based on which operand is bigger (positive if `*this` is bigger, negative if `other` is bigger), **ignoring the actual sign of the operands**.

As before, implement the public method
`ReallyLongInt sub(const ReallyLongInt& other) const`,
which should just call `absSub` for now. Test it!

---You have reached Checkpoint 3---
Files to turn in by 11:59, Tuesday 9/18:
ReallyLongInt.cpp/hpp, ReallyLongInt_TEST.cpp, catch.hpp, Makefile
and, as always, don't forget to fill out and sign your cover sheet!

Part 4: ReallyLongInt class (part 2)

In this part you will complete the ReallyLongInt class. Along the way you will learn about operator overloading in C++ (you're already used to this in Python). Finally, you will incorporate it into your implementation of RSA, allowing the encryption to use keys of arbitrary length. To save you some typing, I went ahead and filled out *ReallyLongInt.hpp* for you. **Please do not alter this header file.** If your code is not compatible, you must not have followed all the directions in Part 3 about how things should be named/declared – fix it in the *.cpp* file! As always, make sure you write unit tests as you go!

20a) Operator overloading: output stream

Now it's time to overload your first operator. As in Python, operators in C++ correspond to functions with special names. Unlike Python, they do not necessarily reside within classes. The function
`ostream& operator<<(ostream& out, const ReallyLongInt& x)`
represents the output streaming operator `<<` when an `ostream` is on the left (for instance, `cout` is an ostream, as is an output file stream) and a `ReallyLongInt` is on the right. In our case, it should add a string of the really long int to the given stream and return `out`. Note that this is *not a method of our class!* It is a function defined outside the class.

This operator returns an `ostream&` so you can chain `<<`s together. For example,
`cout << "Result: " << x << endl;`
is really
`((cout << "Result: ") << x) << endl;`,
which is really
`operator<<(operator<<(cout, "Result, "), x);`.
where each call returns `cout`. After implementing this function, you should be able to print `ReallyLongInts` to the terminal or files using the streaming operator.

20b) Operator overloading: assignment

The assignment operator (`=`) can also be overloaded. This needs to be done for the same reason that the copy constructor needs to be defined. Every class has a default assignment operator that performs a shallow copy. To ensure a deep copy, we need to overload the operator. The assignment operator must be a method of a class (it cannot be a stand-alone function):
`ReallyLongInt& operator=(const ReallyLongInt& other)`.
It turns `this` into a copy of `other`, and then returns `*this` (so assignments can be chained).

There are actually some subtle issues associated with implementing the assignment operator. The most straightforward way to do it would be to have `this` `delete[]` its `digits`, and then copy `other`'s `digits`. This seems reasonable until you consider that a variable can be assigned to itself (e.g. `x =`

`x`)! In that case the array would be prematurely deleted and bad things would ensue.

The safest way to implement assignment is called "copy and swap." First implement the private method `void swap(ReallyLongInt other)`.
Note that it takes its argument *by value* not by reference. This function should simply exchange the values of the member variables of `this` and `other`. So after this function is over, `this->digits` should point to the array that `other.digits` used to point to, and vice versa. Similarly, they should swap the values of `numDigits` and `isNeg`.

One you have a `swap` method, `operator=` can be literally two lines:
```
swap(other);
return *this;
```
It's very compact, but there's a lot going on. When `other` is passed by value, a temporary copy is made (via the copy constructor) that is local to the `swap` method. Then the members of `*this` and the temporary object are swapped, making `*this` a copy of the original `other`, and giving the temporary object control over the stuff that used to belong to `*this`. When `swap` returns, the temporary object is deallocated. This invokes its destructor, which neatly deletes all of the stuff that *used to* belong to `this`, but is no longer needed. Pretty clever huh?

You can now safely assign `ReallyLongInt`s using the `=` operator. Furthermore, because we implemented constructors for implicit conversion, you can *also* assign numbers and strings to `ReallyLongInt`s. After this operator is defined, all of the following should work properly:
```
ReallyLongInt x(10);
ReallyLongInt y;
y = x;
y = -58;
ReallyLongInt z("10");
y = string("123456789");
```

20c) Operator overloading: comparison

You can now overload the comparison operators in the same way as the streaming operator. Each of the following functions (*not methods*) represents their respective operator with `x` as the left operand and `y` as the right operand. Each one can be implemented *in one line* that calls either the `greater` or the `equal` method (or both) of either `x` or `y`.
```
bool operator==(const ReallyLongInt& x, const ReallyLongInt& y);
bool operator!=(const ReallyLongInt& x, const ReallyLongInt& y);
bool operator>(const ReallyLongInt& x, const ReallyLongInt& y);
bool operator<(const ReallyLongInt& x, const ReallyLongInt& y);
bool operator>=(const ReallyLongInt& x, const ReallyLongInt& y);
bool operator<=(const ReallyLongInt& x, const ReallyLongInt& y);
```

FYI: In C++, some operators can be defined as either stand-alone functions or as methods of a class. There are pluses and minuses of each approach. In this case, the main advantage of defining these as functions is that they can make use of implicit type conversion.

20) Addition and subtraction

Implement one more private helper method that will come in handy:
`void flipSign()`, a private mutator that does what it says (flips the sign). If the number is 0, then

it should set `isNeg` to false, no matter what it was originally.

While you're at it, you can use `flipSign` to implement the method `ReallyLongInt operator-() const`, which represents the negation operator. It should return a `ReallyLongInt` that is the negation of `*this`.

Now go back and fix the two methods
`ReallyLongInt add(const ReallyLongInt& other) const` and
`ReallyLongInt sub(const ReallyLongInt& other) const`.
They should perform **signed** addition and subtraction by calling either `absAdd` or `absSub` (and possibly `flipSign`). For instance, if you are asked to perform (-13) + 58, that's really 58 - 13. Similarly, (-25) - 37 is the same as -(25 + 37). Carefully consider for each case how to perform the operation.

Make sure to update your unit tests for these methods to *thoroughly* test what you've written. There are lots of cases. Make sure you test them all!

Once you know the methods work, the following operators should simply call them as appropriate:

```
ReallyLongInt operator+(const ReallyLongInt& x,
                                          const ReallyLongInt& y);
ReallyLongInt operator-(const ReallyLongInt& x,
                                          const ReallyLongInt& y);
```

The following operators combine arithmetic and assignment:
`ReallyLongInt& operator+=(const ReallyLongInt& other)`
`ReallyLongInt& operator-=(const ReallyLongInt& other)`
They can be implemented simply using the = operator and the arithmetic operators you've implemented with `*this` and `other` as the operands. Like `operator=`, they should each return `*this`.

Finally, implement the increment and decrement operator methods. The prefix operators (`++x`) are:
`ReallyLongInt& operator++()`
`ReallyLongInt& operator--()`.
They should be precisely equivalent to `x += 1` and `x -= 1`, respectively.

The suffix operators (`x++`) are:
`ReallyLongInt operator++(int dummy)`
`ReallyLongInt operator--(int dummy)`.
The parameter is not given a meaningful value; it's just used to distinguish the two operators. These operators should have the same effect as `x += 1` (or `x -= 1`), *but* they should return the value of `x` *before* incrementing (you'll have to make a copy of `*this` before you increment so you can return it). Make sure you test all of these operations.

22) Multiplication

Did you test your addition function? If not, what are you doing here starting on multiplication?

Do you remember the “long multiplication” algorithm? That's the one where you line up the numbers, multiply the top number by each digit of the bottom number, and then add up the results (each one properly shifted to the left). You can implement that algorithm if you want, but if you learned to use an abacus you may know that there is a more elegant and efficient algorithm. It does essentially the same

thing, but rather than adding all of the shifted products at the end, it adds them into the result as it goes. As an illustration, consider multiplying 56 and 34...

```
  56
x 34
0000
```

```
  56   6*4 = 24. Add to last digit and carry.     56
x 34                                           x 34
0000                                           0024
```

```
  56   5*4 = 20. Add to second-to-last digit and carry.     56
x 34                                                      x 34
0024                                                      0224
```

We have computed 56*4 = 224 in the usual way. Now we'll compute 56*3 and add it to the result, shifted to the left by one digit, as we go (rather than storing it and adding later).

```
  56   6*3 = 18. Add to second-to-last digit and carry.     56
x 34                                                      x 34
0224                                                      0404
```

```
  56   5*3 = 15. Add to third-to-last digit and carry.     56
x 34                                                    x 34
0404                                                    1904
```

I strongly recommend that you do some examples on paper yourself to wrap your head around the algorithm you plan to implement. It will probably help to write it out in pseudocode and then try out that algorithm by hand before you start coding. Once you understand what you are about to write, implement the private helper method
`ReallyLongInt absMult(const ReallyLongInt& other) const`,
which should perform *unsigned* multiplication. The result should always be non-negative. Your multiplication algorithm should take O(*xSize**ySize*) time.

Now you can implement
`ReallyLongInt mult(const ReallyLongInt& other) const`,
which simply calls `absMult` and flips the sign of the result when necessary.

You are now prepared to implement the following operators:

```
ReallyLongInt operator*(const ReallyLongInt& x,
                                          const ReallyLongInt& y);
ReallyLongInt& operator*=(const ReallyLongInt& other)
```

Thoroughly test your multiplication methods!

23) Division

Okay. Are you ready for division? Surely you remember the long division algorithm? The thing is that,

of all the arithmetic algorithms that are commonly taught, long division is typically the least well-specified. Most students are taught to "eyeball" certain steps, which is effective and efficient, but not precise enough to code up! So before you dive in, let's spend some time thinking about division.

To understand the long division algorithm, first consider this simple division algorithm (for positive operands) originally given by Euclid:

UNSIGNEDEUCLIDEANDIVISION(x, y)
1) $r = x$
2) $q = 0$
3) **while** $r \geq y$
4) $\quad r = r - y$
5) $\quad q = q + 1$
6) **return** (q, r)

Essentially, this algorithm answers the elementary school question "How many times does y go into x?" The algorithm divides x by y by repeatedly subtracting y from x until the result would be negative. The number of times y can be subtracted is the *quotient* and the remaining value after all the subtractions is the *remainder*. For instance, if $x = 11$ and $y = 2$, the algorithm would proceed as follows:

$r = 11$ $q = 0$ (lines 1 and 2)
$r = 11 - 2 = 9$ $q = 1$ (lines 4 and 5)
$r = 9 - 2 = 7$ $q = 2$ (lines 4 and 5)
$r = 7 - 2 = 5$ $q = 3$ (lines 4 and 5)
$r = 5 - 2 = 3$ $q = 4$ (lines 4 and 5)
$r = 3 - 2 = 1$ $q = 5$ (lines 4 and 5)

Returning the quotient 5 and remainder 1, which is correct since $5(2) + 1 = 11$.

Division by repeated subtraction is elegant and simple, but inefficient. The loop occurs q times. Each instance of the loop involves a comparison, a subtraction, and an addition, each of which can be performed in time linear in the number of digits in the operands. Since x is the largest number involved, overall, the algorithm runs in $O(q*x.digits)$ time, where *x.digits* is the number of digits in x. If q is large, this algorithm will take a long time.

Long division breaks the division problem up into several small division problems, allowing it to achieve a time complexity of $O(y.digits*x.digits)$, which is much more tolerable. Here individual digits of a number will be referred to using subscripts. So if $x = 342$, then $x_1 = 3$, $x_2 = 4$, and $x_3 = 2$. The following pseudocode assumes that both x and y are non-negative.

UNSIGNEDLONGDIVISION(x, y)
1) $r = 0$
2) **for** $i = 1$ **to** *x.digits*
3) $\quad r = 10r$
4) $\quad r = r + x_i$
5) $\quad d = 0$
6) $\quad$ **while** $r \geq y$
7) $\quad\quad r = r - y$
8) $\quad\quad d = d + 1$
9) $\quad q_i = d$
10) **return** (q, r)

The algorithm fills in the quotient one digit at a time, obtaining each digit by performing Euclidean division on small numbers (the Euclidean division algorithm appears in lines 6-8). Note that after the while loop, r is guaranteed to be less than y. Therefore, after line 3, $r \leq 10(y - 1) = 10y - 10$. Since each digit x_i is in the range 0-9, after line 4, $r < 10y$. Because of this, the loop from lines 6-8 will always produce a $d < 10$, meaning it is suitable as a digit for q, and also that the loop can occur 9 times at most. Also note that since $r < 10y$ at all times, $r.digits \leq y.digits + 1$. Therefore, the arithmetic operations inside the inner loop take $O(y.digits)$ time. Since the outer loop occurs $x.digits$ times, the overall complexity of long division is $O(x.digits*9*y.digits) = O(x.digits*y.digits)$.

In the following example ($x = 123$ and $y = 2$) it is clear that this is the familiar long division algorithm:

$r = 0$		(line 1)
$i = 1$		(line 2)
$r = 1$	$d = 0$	(lines 3-4)
$q = 0$__		(line 9)
i = 2		(line 2)
$r = 12$	$d = 0$	(lines 3-4)
$r = 12 - 2 = 9$	$d = 1$	(lines 7 and 8)
$r = 10 - 2 = 7$	$d = 2$	(lines 7 and 8)
$r = 8 - 2 = 5$	$d = 3$	(lines 7 and 8)
$r = 6 - 2 = 3$	$d = 4$	(lines 7 and 8)
$r = 4 - 2 = 1$	$d = 5$	(lines 7 and 8)
$r = 2 - 2 = 0$	$d = 6$	(lines 7 and 8)
$q = 06$_		(line 9)
i = 3		(line 2)
$r = 3$	$d = 0$	(lines 3-4)
$r = 3 - 2 = 1$	$d = 1$	(lines 7 and 8)
$q = 061$		(line 9)

Because x has 3 digits, the outer loop occurs 3 times (marked by the horizontal lines). The algorithm returns the quotient 61 and the remainder 1, which is correct since $61(2) + 1 = 123$. Note that long division only performs 7 subtractions (Euclidean division would have performed 61!).

It would be a good idea to do some examples yourself, to be sure you understand the algorithm. Also, remember that **pseudocode is not code**. The pseudocode tells you how to perform the algorithm, and helps you understand its flow, but you must still think carefully about how to implement the algorithm in your particular language and to suit your particular needs. Pay special attention to the assumptions made by the pseudocode versus the realities of your implementation.

The algorithm above only handles non-negative operands. Use it to implement the private method
`void absDiv(const ReallyLongInt& other, ReallyLongInt& quotient, ReallyLongInt& remainder) const`.
It should divide `*this` by `other` (**ignoring sign**) and set `quotient` and `remainder` to new `ReallyLongInts` that represent the appropriate results.

Now make the public method
`void div(const ReallyLongInt& other, ReallyLongInt& quotient, ReallyLongInt& remainder) const`
just call `absDiv` for testing purposes. Note that the division algorithm involves several of the operations you've already implemented. If `absDiv` isn't working, but seems logically correct, it might

be a bug in one of those other operations!

Once you are confident in `absDiv`, change `div` so it accounts for the signs of the operands. *A note on division with negative numbers:* The traditional programming convention is that the integer quotient *q* of *x* and *y* is the integer portion of the fraction *x*/*y* (created by just leaving off everything after the decimal) and *x*%*y* is the integer *r* such that *yq* + *r* = *x*. This is what you should do too. The `div` function should call `absDiv`, and then set the signs of `quotient` and `remainder` appropriately. Do some examples by hand to figure out how to set the signs.

Though we sometimes call % the "modulus" operator, strictly speaking, this definition does not coincide with the usual mathematical definition of modulus when it comes to negative operands.[1] However, it is the way C++ does it, because it was how C did it, because it was how Fortran did it, and it does make the algorithm a little bit simpler. This kind of shenanigans is why mathematicians won't sit with computer scientists at lunch. Note, however, that Python handles division in the more mathematically proper way, so if you try to use Python to check your answers (with negative operands) you will get different results! Instead, your results should match what C++ would do.

Once you are confident in your division method, you can overload the appropriate operators:

```
ReallyLongInt operator/(const ReallyLongInt& x,
                                          const ReallyLongInt& y);
ReallyLongInt operator%(const ReallyLongInt& x,
                                          const ReallyLongInt& y);
ReallyLongInt& operator/=(const ReallyLongInt& other)
ReallyLongInt& operator%=(const ReallyLongInt& other)
```

You know what I'm going to say, so I won't even say it. Okay, I will. Test these operations!

24) Converting back to a number

You are nearly ready to incorporate `ReallyLongInt`s into RSA. First, we need to be able to convert a really long int back into a built-in integer type. Implement the public method
`long long toLongLong() const`,
which should convert the `ReallyLongInt` to a `long long`.
That raises the question of what should happen if `this` represents a number that cannot be represented by a `long long`. As you know, if you try to convert a large `long long` to an `int`, no promises are made about what will happen. However, the typical convention is to simply ignore the higher order bits. If *x* is the value of the `ReallyLongInt`, then this is equivalent to returning the `long long` value *x*%(LLONG_MAX + 1). This is what your function should do.

- To gain access to the constant `LLONG_MAX`, you need to `#include <climits>`.
- Note that `LLONG_MAX + 1`, by definition, cannot be represented using a `long long`! You'll have to (carefully!) obtain a really long int that has this value....

Hint: your algorithm will probably involve computing various powers of 10. Remember, however, that the `pow` function works with floating point numbers, so using it will result in a loss of precision! Find an efficient way to get powers of 10 using only integer multiplication instead.

1 For a nice, but somewhat lengthy discussion about this surprisingly thorny issue, Ask Dr. Math: http://mathforum.org/library/drmath/view/52343.html

25) RSA!
Reading:
Topic 11 – Template Functions
(see also: Prata ch. 8, starting on p. 419, Lynda.com 8.1-2)

Wait, what were we doing again? Oh yeah, implementing RSA, the famous encryption algorithm! It's time to edit *numberTheory.cpp/hpp* to use `ReallyLongInt`s. You *could* define all these functions again with `ReallyLongInt`s as their operands, but because of operator overloading, the code would be essentially the same as it was when we were using `long long`s! That's a sign that we should consider turning these into function templates.

In *numberTheory.tpp/hpp* implement the function templates
```
bool isPrime(const X& num)
X modPower(const X& base, const X& exponent, const X& modulus)
X extendedEuclid(const X& a, const X& b, X* x, X* y)
```
[2]
They should be nearly the same as your original implementations from before, but you should make sure that you use only operations that work with *both* `ReallyLongInt`s and built-in numeric types. In `isPrime` you used the `sqrt` function. Obviously that won't work with `ReallyLongInt`. Instead you can stop the loop at `num/2`.

Note that all of these functions now take `const` references for their parameters. This is to avoid unnecessary copying when `ReallyLongInt`s are used (but still prevent the function from altering the values of the operands in any way). The magic of references is that within the function the syntax will be the same as if you were using pass by value.

Make sure you modify *numberTheory_TEST.cpp* to test these functions with `ReallyLongInt`s as well as built-in integer types.

Now edit the following files:
keygen.cpp
- Make it use `ReallyLongInt`s throughout. Be very careful! If you convert to `long long` at any time, you will see integer overflow for large values!
- Remember that command line arguments are given as strings, so rather than converting the prime numbers to integers, convert them directly to `ReallyLongInt`s.
- Our method for primality testing is very slow for large numbers (there are faster methods but that is beyond our scope in this project). Your program should skip the primality test for arguments larger than 100,000. In such cases print a message that informs the user that primality is not being verified.

encrypt.cpp
- Similarly, change it to use `ReallyLongInt`s.
- You should read the two elements of the key in as `string`s, then convert the strings to `ReallyLongInt`s.

decrypt.cpp
- Similarly, change it to use `ReallyLongInt`s.
- Read the two elements of the key in as `string`s, then convert them to `ReallyLongInt`s.
- After decryption you'll have a `ReallyLongInt` whose value is guaranteed to be smaller than 256, but you'll need a `char` to write to the decrypted file (otherwise you'll get a number, not a

2 Hey, remember that last test case in *modpower.cpp* with the giant numbers? It should come out to 519170861217.

character). Use `toLongLong` to convert the result to an integer value.

Makefile

- Make sure it is up-to-date and includes new files and dependencies.

Only your incorporation of `ReallyLongInt`s will be evaluated in this checkpoint. If your original code had bugs, you will not lose points twice. That said, if you have mystery bugs ask for help!

If you'd like to test your code out on a large key, you can find some prime numbers of various sizes here: http://primes.utm.edu/lists/small/small.html. Note that for large-ish keys decryption might be a little bit slow: with a 200 digit key, my program takes a few seconds per character during decryption, so you might want to stick to a short message. There are many ways one could improve the efficiency of the RSA calculations to make the programs more usable. We've only scratched the surface!

RSA doesn't work? Read this, please. So, you've gotten to this point and RSA isn't working properly. Don't panic! It might mean that you've made a small error in converting your RSA programs. Give them another careful look. If that's not the problem, then it probably means you have a bug or bugs somewhere in the large volume of code upon which RSA is now built. Hey, I said not to panic!

Have you been thoroughly testing as you code? If so, and if you felt pretty confident that everything was working, a good strategy at this point would be to try to find the simplest, smallest example on which your programs fail. Now, using GDB or print statements (or both) start tracking the process through the example, verifying intermediate results by hand. Eventually you will encounter an error. Now, dig deeper: what operations were involved in that step? Track the process through those operations, and so on. Eventually you will find the source of the problem.

If, on the other hand, you have not been carefully testing along the way, it is very likely that you have many bugs and errors throughout your project. Probably the best thing to do at this point is to go back and carefully test each component *as if* you were testing as you code. Take your time and be thorough; you'll find those bugs yet! Then write "I will test my code frequently" 100 times on the whiteboard.

---You have reached Checkpoint 4---

Files to turn in by 11:59, Tuesday 9/25:

ReallyLongInt.cpp/hpp, numberTheory.tpp/hpp, keygen.cpp, encrypt.cpp, decrypt.cpp, ReallyLongInt_TEST.cpp, numberTheory_TEST.cpp, catch.hpp, Makefile

and, as always, don't forget to fill out and sign your cover sheet!

Part 5: Clean up

You now have a little bit of time before the final submission deadline. Use it wisely! Tie up any loose ends that were still dangling at a checkpoint. Carefully test your code and fix any bugs you find. You will also be required to turn in a *readme.txt* file. Now would be a good time to write it up. This file should list the files included in the project and, for each program, give brief but clear instructions for compiling and running the program. Please see the example and ask me if you have any questions.

The grade breakdown for the final submission is as follows:
ReallyLongInt.cpp/hpp: 60 pts total as follows...
- Creating/destroying/converting: 10 pts
- Comparison: 10 pts
- Addition: 10 pts
- Subtraction: 10 pts
- Multiplication: 10 pts
- Division: 10 pts

numberTheory.tpp/hpp: 15 pts
keygen.cpp: 10 pts
encrypt.cpp: 10 pts
decrypt.cpp: 10 pts
Unit tests: 5 pts (for full credit, must achieve at least 85% coverage)
Makefile: 5 pts
readme.txt: 5 pts
Total: 120 pts
Your final submission grade will comprise 75% of the project grade.

Note that each component will be evaluated on its own merits, as much as possible. So if you have a bug in subtraction you will not lose points in every component that depends on subtraction if those components are otherwise correct. Furthermore, note that you should turn in what you have, even if it is not all complete. For instance, if you did not get an opportunity to integrate `ReallyLongInt`s with your RSA code, you can still get some credit for what you developed in Parts 1 and 2.

Warning: any file that does not compile on a lab machine will receive 0 credit! This is a real thing that will actually happen. Make sure everything compiles and runs before submitting.

---Final Submission Deadline---
Files to turn in by 11:59, Tuesday 10/2:
ReallyLongInt.cpp/hpp, numberTheory.tpp/hpp, keygen.cpp, encrypt.cpp, decrypt.cpp, ReallyLongInt_TEST.cpp, numberTheory_TEST.cpp, catch.hpp, Makefile, readme.txt
and, as always, don't forget to fill out and sign your cover sheet!

Epilogue: How RSA is really used

Your RSA implementation has a lot going for it, but you should never (really, never) actually use it to protect secrets you care about. It is still insecure in many ways.

The most important way in which it is insecure is that it encrypts each character individually. This basically makes it a fancy substitution cipher (like the Caesar cipher from CS1!). Substitution ciphers can often be cracked with simple methods such as frequency analysis (counting up the frequency of each number and comparing that to the frequencies of letters in English text to infer which numbers correspond to which letters).

It is far more common in RSA to take the entire message and encode it as a *single* (very large) number. Then that one large number can be encrypted using the public key. Of course, this means that the key must also be very large. And this means that encryption and especially decryption will be very slow.

Because of this, RSA is actually not often used to encrypt the message itself. Rather, common practice is to encrypt the message with a more standard key-based encryption method, and then to use RSA to encrypt *the key*. Since the key is fairly short, this is more practical. You can then send the encrypted message and the encrypted key and whoever has the private RSA key can decrypt them both.

Even then, cryptographers use many tricks to improve security. A common idea is to pad the number to be encrypted with random bits. That way the same message will have a different encryption every time, making it even harder for a cryptanalyst to crack the code. If this stuff interests you, you should keep digging. Cryptography is a fascinating and vitally important field!