CPS 222: Computer Science III
Project 1: Our own data-type - RSA

Important Dates
Checkpoint 1: 5:00pm, Thursday, 9/19
Checkpoint 2: 5:00pm, Thursday, 9/26
Checkpoint 3: ~~5:00pm, Thursday, 10/3~~ 5:00pm, Thursday, 10/10
Final deadline: ~~11:59pm, Thursday, 10/10~~ 11:59pm, Thursday 10/7

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; Unit Test coverage < 50%*
2 points: *Mostly functional, but has some important errors or does not comply with the specifications; Unit Test with at least 50% coverage.*
3 points: *Nearly complete; only minor issues remain; Unit Test with at least 85% coverage.*
Your accumulated checkpoint scores will comprise 20% of your project grade.

Computer
All submissions are checked on lab systems. If they do not compile/run on lab systems they do not work! If you are using your own system, you must check to make sure your project/checkpoints work on the lab systems! AGAIN --- IF IT DOES NOT COMPILE/RUN ON LAB SYSTEMS, YOU GET A ZERO.

Submissions
Your submission is whatever is on your master ("main") branch of your repo the instructor pulls at the due date. This means that if you just added something and it breaks everything, you just don't get points. Therefore, you must use branches for development and merge into the master before the due date. Below is a Git-Flow that should help you with this.

Checkpoint Git-Flow
1. Clone the repo
2. Make a readme file in the master branch with your name
3. Branch – dev_checkpoint1
4. Work on checkpoint 1 in dev_checkpoint1
5. When you are done with checkpoint 1, merge (Give yourself at least 1 hour to merge to make sure things are ok)
    a. Merge master into dev_checkpoint1
    b. Fix any merge errors
    c. Merge dev_checkpoint1 into master
6. Now don't touch master again until the next checkpoint (I would also make a Fork or download a zip at this point as a backup!)
7. Make a new Branch – dev_checkpoint2
8. Repeat steps 4-6 for the new checkpoint.

Makefile

You will need to start your Makefile right away. All submissions should have a proper Makefile. You may lose points if your Makefile does not compile your project correctly. You should have rules in your Makefile for coverage in addition to debug and release.

Unit Testing

Our goal is to write unit tests for every function we can for our classes and have at least 85% coverage for all defined classes by the end. We will be using Catch2, which was introduced in class for our unit testing harness, and gcov/lcov to find our unit test coverage. Please refer back to canvas, the Vector class test, and your class notes for this information. We will build one C++ (.cpp) file for each class we will test. For a given class X, the unit test will be named X_TEST.cpp. Each class will be a TEST_CASE, can each permutation of a function we test is a SECTION. There are many other ways we *could* outline our unit tests, but this is the one we will use. You will include the Catch.hpp in every submission for this project, so the unit tests can be compiled easily. If the unit tests do not compile, you get a zero. You will include a rule in your Makefile called tests (similar to the rule all), which will make all the unit tests. Before you start stubbing out any class, make sure you stub out SECTION tests (at least one) for each function you stub for the class. This will also help you so make sure all your code will compile against test cases.

Introduction

You have been using built in simple data-types, such as Integers and Floats, since CS1. Many of the most common ones are built into your programming languages. However, the time will come when built in simple data-types are just not enough. There are many many examples. One such is the need for quad-precision for certain numerical modeling or finding the 3000^{th} digit of Pi. In this project, we will need a very large unsigned long (i.e., a very large positive integer) of a size not supported by C++. For example, C++ supports up to 2^64. However, we are going to be consider the RSA algorithm for public-key cryptography, and even 2^64 can be considered small.

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.

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.

Because of the exponent part of the formula (i.e., x^e), number can get large fast, and we need to be able to represent those number using our own data type. In order to complete this, we will build our own type using C++ classes to represent and unsigned binary number.

Introduction to Unsigned Binary Numbers

Every number is represented relative to some *base*. We normally use base 10 to represent numbers 0,1,2,3,4,5,6,7,8,9 before we increment the base and start counting again. In binary, the base is 2 (See Appendix A in your C++ textbook). We use number 0 and 1 before starting over.
Here are some examples of numbers in binary:

- 0 = 0
- 1 = 1
- 2 = 10
- 3 = 11
- 4 = 100
- etc

It is often the case in computers to work with a fixed size data type (You will learn more about why in Computer Organization). For example, we might want to work in 4-bits. This just means that we always have 4 slots and we zero out the ones we do not use.
For example:

- 0 = 0000
- 1 = 0001
- 2 = 0010
- 3 = 0011
- 4 = 0100
- etc.

Project Steps
Checkpoint 1 Steps

Step 1. Builing a ReallyLongInt Class

In this part, you will construct the initial structure for your ReallyLongInt class in the file ReallyLongInt.cpp. Make sure you are unit testing along the way in the ReallyLongInt_TEST.cpp file. The class could have the following three private member variables (please use these names):

- `vector<bool> *digits` – the binary digits (see: textbook page 627) – you will use this as an array – not as a vector. This means you will not use things like push_back or length.
- `unsigned int size` – the number of digits, (i.e., the fixed size).
- `bool isNeg` – the sign of the number (true if negative)

a.) Constructors

You will have to implement 4 constructors.

1. `ReallyLongInt()` – represents the value 0.

2. `ReallyLongInt(long long num)` – sets based on num.
3. `ReallyLontInt(const string& numStr)` – takes in a string representation and sets the private member variables. You may assume to start that the string has a number that can be represented as long long.
4. `ReallyLongInt(const ReallyLongInt& other)` – this is needed to make copies.

b.) Destructor
You must make an appropriate destructor.

c.) toString
You will have to make a toString class method (i.e., `string toString()` const) that returns a string representation of the number stored in base 10. You may assume to start that the number can be represented as long long.

Also implement a toStringBinary class method (i.e., `string toStringBinary()` const) that returns a string representation of the number stored in base 2.

d.) Comparison operators

Implement the public method
`bool equal(const ReallyLongInt& other) const`.
It should do the following:

- 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 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.

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` appropriate

e.) Makefile
You should start with making a makefile for this. More information about makefile is given in class.

---You have reached Checkpoint 1---
Your github master branch should contain:
ReallyLongInt.cpp/.hpp, ReallyLongInt_TEST.cpp, catch.hpp, Makefile, and a coversheet in pdf format.

Checkpoint 2 Steps

This part of the project is where you roll up your sleeves and get down with binary arithmetic. You will have to remember a lot of stuff from elementary school, so it might take a while. I am sure you never thought in fifth grade that you would have to be programming long division in binary in C++ to make your own data types. Look how far you got, and without calculus ☺ So which ones are we going to need. Well, you are going to need the major ones that make them into a semi-ring (i.e., a big math term for a number of operational properties the number system needs.) For you, this means they need addition, subtraction, multiplication, division, and modulus.

f.) Removing leading zeros

It can be very easy to get extra "0" bits that we don't need. This can be confusing when making comparisons. So, we would like a way to get rid of them. For example, `6 = 00110` has 5 bits, but really only needs 3 bits to represent the number. In the example, our class function should change our internal representation of `00110` to `110`. This means that it will change both size and `digits`. Note that we want to treat digits as an array not a vector!

Let the class function have the signature `void removeLeadingZeros(void)`;

Right now, this should not be an issue for you if you created your constructor correctly, but you will need it later.

g.) Operator overload: 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");
```

h.) 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 public 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.

After, implement the public method
`ReallyLongIn add(const ReallyLongInt& other) const`

i.) 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 public 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!

j.) 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);
```

k.) 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);
```

Thoroughly test your multiplication methods!

l.) 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) $r = r - y$
5) $q = q + 1$

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) $r = 10r$
4) $r = r + x_i$
5) $d = 0$
6) **while** $r \geq y$
7) $r = r - y$
8) $d = d + 1$
9) $q_i = d$
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);
```

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

---You have reached Checkpoint 2---

You should have the following in your github master branch:
ReallyLongInt.cpp/.hpp, ReallyLongInt_TEST.cpp, catch.hpp, Makefile, and a coversheet in pdf format.

Checkpoint 3 Steps

This checkpoint is about now using our new data type to start doing RSA, sorry for the long wait! So lets review the RSA algorithm:

Generating Keys:

1. The user gives two prime number *p* and *q*.
2. You calculate *n = p x q*
3. You calculate *t = (p-1) x (q-1)*
4. You pick a number *e*, such that *1 < e < t*, and *e* and *t* are relatively prime (i.e., the greatest common divisor of e and t is 1)
5. Find a *d* such that *(ed) % t = 1*
6. The keys are
    a. Public (encryption): *(e,n)*
    b. Private (decryption): *(d,n)*

Encrypt:

1. Loop over characters in message
2. For each character '*c*' get ASCII value *x*
3. Let *y = (x^e) % n*, from the Public key set above
4. Covert the number *y* back into a character '*c*' and place into the new message

Decrypt:

1. Loop over charaters in the encrypted message
2. For each character '*c*' get ASCII value *y*

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

3. Let *x = (y^d) % n*, from the Private key set above
4. Convert the number *x* into a character '*c*' and place into a new message

So why does the above work!?! Well it is based on factoring very large numbers, i.e., factor *n* and *t*. This is difficult as there is no polynomial time algorithm exists for it. So therefore it is hard for anyone to break.

So what parts do we need to get this working? We are going to break it into three pieces. The first, generating the keys (keygen.cpp). The second, encrypt.cpp. The third decrypt.cpp.

m.) Exponents
Note that we need to keep raising things to powers (i.e., exponents). The dirty and fast way to do this is with recursion, such as

```
def exp(base, e):
      if e == 0:
     return 1
       return base*exp(base,e-1)
```

However, this means that as n grows, so does the number of recursions linearly. So for huge n, this would be super slow (why?, is explained in Computer Organization). So, you need to be a little bit faster. We can observe the following:

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:

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

This reduction from O(e) to O(log e) is huge and makes it possible.
Therefore, you will add a exp method to your ReallyLongInt class using this method:
ReallyLongInt exp(const ReallyLongInt e).
Note that you might need a helper method in the class to tell if it is even or odd. When you make that helper function, do not use modulus operator!!!! Division is slow!!! Think about how you can tell if a number is even or odd quickly if the number is in unsigned binary. Also you might need a private recursive helper call for the exp, since e is a const).

n.) isPrime
You are going to continue to add onto your ReallyLongInt class and make a method that determines if a number is Prime. It should return the bool true if prime and bool false if not prime.

o.) extendedEuclid
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 % 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) $y' = x - y \cdot \text{FLOOR}(a/b)$
9) **return** (d, x', y')

In *numberTheory.cpp/hpp* implement the function with the type being ReallyLongInt.

p.) Keygen

Now you should have all the pieces you need for your keygen.cpp based on the algorithm provided before.

q.) Encyrpt

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).

r.) Encrypt and decrypt

Make two programs, encrypt.cpp and decrypt.cpp. Each should 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!

Have these programs follow the algorithms above.

---You have reached Checkpoint 3---

You should have the following in your github master branch:
ReallyLongInt.cpp/.hpp, ReallyLongInt_TEST.cpp,
numberTheory.cpp/.hpp, numberTheory_TEST.cpp,
keygen.cpp,
encrypt.cpp, decrypt.cpp
catch.hpp
Makefile, and a coversheet in pdf format.

Final Submission

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: 70 pts total as follows...
- Creating/destroying/converting: 10 pts
- Comparison: 10 pts
- Addition: 10 pts
- Subtraction: 10 pts
- Multiplication: 10 pts
- Division: 10 pts
- Exp: 10 pts

numberTheory.cpp/hpp: 10 pts
keygen.cpp: 10 pts
encrypt.cpp: 10 pts
decrypt.cpp: 10 pts
Unit tests: 10 pts (for full credit, must achieve at least 85% coverage)
Makefile: 10 pts
readme.txt: 10 pts
Total: 140 pts
Your final submission grade will comprise 80% of the project grade.