COMP220 - Project 2

Fall 2015

Abstract

This project comes right from the text: Chapter 14, exercise 13. This document focuses on some design and implementation details and the logistics.

1 Project and Document Overview

The basic requirements of the problem are well documented in the textbook and won't be repeated here. The short version is that you're implementing the core of a *BigInt* class for arbitrarily long unsigned integers and using a simple factorial table program as a testing ground for that class.

We'll discuss algorithms for addition and multiplication in class, they are also well documented on Wikipedia and in Computer Organization and Architecture books (COMP230). For full credit, your multiplication operator should be no worse than $O(n^2)$ for multiplying n digit numbers and $O(n)$ or better for all other operations and methods on n digit numbers.

1.1 Project 2 Lab/Hwk Assignment

Submit your *BigInt* class header (documentation and declaration) as assignment *labp2* via handin. If any class code is declared and documented in a separate header than the one containing the class, then submit that header as well.

2 Logistics

The following are notable dates for this project:

Date	Activity
12/3	Project 2 Lab/Hwk Due by 2pm. Some in-lab work time.
12/10	Project 2 Due by 7am *sharp*

Your grades are based on the following criteria:

Points	Category
15	Neatness and Good Style
25	Library Design Completeness
25	Library Implementation Correctness & Efficiency
15	Main Program Design Completeness
10	Main Program Implementation Correctness

Completeness means that all the necessary code is accounted for by way declared and documented classes and procedures with complete test coverage. It has nothing to do with implementation, with tests passing. It's all about *what* and not all all about *how*. At this point in your studies there is a very high bar for completeness. It's worth 40 of the 100 points of this assignment. If you're clearly missing vital functionality or you're packing too much functionality into a small set of procedures rather than using helpers, then you can expect to receive no better than a C or C+.

Correctness means that your code compiles and that a set of tests with complete coverage would all pass. The main procedure should do what it's supposed to do as well. Correctness is purely a measure of completed implementation. If you submit code that doesn't compile, you can expect to receive no better than a D+. This is an exception to the above rubric.

Efficiency means that multiplication of two n digit numbers takes $O(n^2)$ operations and all other methods/operators require not more than $O(n)$. You should also be prepared to analyze the efficiency of these methods and operations come finals time.

To receive a B- or better your program design should be complete in the sense described above. This means that all the requirements are accounted for by declared and documented procedures and methods with complete tests. At that point your grade is a function of how much of that design is implemented and the quality of that implementation. The key observation here is that you're expected to identify everything that needs doing in a problem and be able to express those things as procedural elements of a program. Once you know what to do, then and only then should you set about determining how to do those things.

3 Base-10, Radix Number Systems, and Recursion

Let's recall why we attribute the the sequence of digits 123 to the quantity one hundred twenty three.

$$\begin{aligned} 123 &= 1 * 100 + 2 * 10 + 3 * 1 \\ &= 1 * 10^2 + 2 * 10^1 + 3 * 10^0 \end{aligned}$$

It is the final form given above, the sum of digits multiplied by successive powers of the base of the number system (10), that clearly highlights the pattern used to represent numbers in a base.

In a base 10 representation, the digits come from the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ or the integer interval $[0, 10)$. Each place then represents the product of a primitive digit multiplied by a power of 10 and the total quantity is the sum of those products. We can write this pattern in a compact form using summation notation. Let a be an n digit number where a_0 is the least significant digit, a_1 the next, and so on to the most significant digit a_{n-1}. Each digit a_i is, of course, from the integer interval $[0, 10)$. Then we can write a as,

$$a = \sum_{i=0}^{n-1} a_i * 10^i$$

Binary numbers fit the same pattern but the base is 2 and the digits are from $[0, 2)$.

$$a = \sum_{i=0}^{n-1} a_i * 2^i$$

In general, for base b, we have digits from $[0, b)$ such that we can represent the number a in the form

$$\sum_{i=0}^{n-1} a_i * b^i$$

It's important to realize that when we use the everyday base 10 numbers we're used to to write these things out, what we're really do is representing a base b number using representations in base 10. This takes some getting used to. The important thing is that we have a standard way of thinking about numbers on a digit-by-digit basis that is uniform for any base b.

The Recursive Structure of Numbers

Representing numbers as sequences of digits lends itself to recursive decomposition, which we can then use to write recursive procedures. The base case is a single digit number. The least significant digit can act as the *first* of the sequence and all the other digits the *rest*. Putting these together we can write a complete

recursive definition for numbers as recursively structured data . For number N, written in base b, and with n digits $d_{n-1} \dots d_0$ each between 0 and b,

$$N = \begin{cases} d_0 * b^0 = d_0 & n = 1 \\ (d_0 * b^0) + \left(\sum_{i=1}^{n-1} d_i b^i \right) & n > 1 \end{cases}$$

In the recursive case, parenthesis were used to separate the singular element, the *first*, from the recursive part, the *rest*. One important observation to make is that if we wish to view the *rest* strictly as an $n-1$ digit number, then we can do so by factoring out b.

$$\left(\sum_{i=1}^{n-1} d_i b^i \right) = \left(\sum_{i=1}^{n-1} d_i b^{i-1} \right) b$$

For our original number, 123, the recursive decomposition separates the 3 from the remaining $120 = 1*10^2 + 2*10 = (1*10+2)10 = 12*10$. Pay extra attention to the different formulations of the rest, they'll be important later when we formulate a recursive implementation of multiplication.

Recognizing and understanding this recursive structure is important because it allows you to lay out a basic recursive template for digit by digit operations on numbers. If you need a refresher, that template is:

```
if( base-case ){
  operate on base element as/if needed
}
else{
  deconstruct into first and rest
  operate on pieces as/if needed
  recombine results of by-parts computation as/if needed
}
```

This pattern should be very, very, very familiar from COMP160. We'll now apply it towards the problem of addition and multiplication.

Iterative Addition

Before delving into n digit addition, let's' consider the base case, where A and B are single digit numbers. Intuitively, you know what happens. We add two digits and we get either another single digit number $(1+2=3)$ or a two digit number $(5+7=12)$. Another way of saying this is that the addition of two single digit numbers always produces a single digit sum and a single digit carry where the carry is the excess of the actual sum equal to or above the base. For $1+2$, the sum is 3 and the carry is 0. For $5+7$ the sum is 2 and the carry is $10 = 1*10^1$.

The carry can be uniformly extracted from the complete sum by strict integer division by the base, 10. When the carry is 0 then the sum must be in $(10, 0]$ and dividing by 10 gives 0. When the carry isn't 0 then the sum is in $(20, 10]$ and dividing by 10 gives the ten's digit, 1. To avoid any confusion, we'll express strict integer division, like you see in C++, mathematically as the *floor function*, i.e. division rounded down to the nearest integer. For integers x and y, the C++ operation x/y is mathematically equivalent to:

$$\left\lfloor \frac{x}{y} \right\rfloor$$

By similar logic, we can extract the single digit sum by using the modulo operator with the base 10. Taking any number from $(100, 0]$ modulo 10 will give you exactly the one's place of that number. In mathematics we just write $\mod$ where you use % in C++. When dealing with strictly positive integers, as we are, the two operations are equivalent. They do differ when dealing with negative values though. So, for positive integers x and y, the C++ operation $x\%y$ is mathematically equivalent to:

$$x \mod y$$

Finally, it's worth noting that when adding only single digit, positive numbers, the total sum will be in $(100, 0]$ as long as we're adding together not more than 11 numbers. (Do you see why?)This means simple division and modulo can be used to extract the single digit sum and carry for the sum.

Now, let's restate the addition base case in mathematical notation. For single digit numbers A and B,

$$\begin{aligned} A + B &= \left\lfloor \frac{A+B}{10} \right\rfloor 10^1 + ((A+B) \mod 10)10^0 \\ &= \left\lfloor \frac{A+B}{10} \right\rfloor * 10 + ((A+B) \mod 10) \end{aligned}$$

where the first term is the carry and the second term is the sum.

The n digit algorithm utilizes the basic logic of a carry-sum adder. It's logic is iterative. While we consider single digit numbers the base case, it's also reasonable to say that a zero digit number is 0 and that $0 + 0$ has a sum and carry of 0. From here we can build up the iterative logic. Assume we've traversed and summed across the first $k < n$ digits of the n digit numbers A and B. We should have accumulated a k digit sum S_k and a single carry digit c_k. The iterative logic then accumulates the $(k+1)^{th}$ digits of A and B by using them to compute the $(k+1)^{th}$ digit of the sum S_{k+1} and the carry c_{k+1}

$$\begin{aligned} S_{k+1} &= \left\lfloor \frac{A_{k+1} + B_{k+1} + c_k}{10} \right\rfloor \\ c_{k+1} &= (A_{k+1} + B_{k+1} + c_k) \mod 10 \end{aligned}$$

If begin with the initial sum and carry of 0 and repeat across the digits in least to greatest order then we should get the final sum as $c_n 10^{n+1} + S_n$ (notice we need can incorporate the final carry). You should recognize this as how you were taught to do addition back in grade school. From here you should be able to adjust for the case when the two numbers are not of equal length but first reducing it to the sum of two equal length numbers then adjusting for any digits left off of that sum.

Recursive Multiplication

When you learned to do multiplication by hand you probably started by learning the multiplication table for single digit numbers. You should now recognize this as memorizing the base case for single digits. The only thing we'll say now about this base case is that for any two single digit numbers, multiplication produces at most a two digit number (Do you see why?). This means we can use the same division and modulo technique for extracting the first and second digits from single digit multiplication.

The algorithm you learned to go with this base case required that you multiply the first number by each digit of the second number and shift the result such that the result started in the same place as the digit from the second number. You then add all these results. In case you need a reminder, here's an example:

$$\begin{array}{cccccc} & & & 1 & 2 & 3 \\ \times & & & & 1 & 5 \\ \hline & & & 6 & 1 & 5 \\ + & & 1 & 2 & 3 & 0 \\ \hline & & 1 & 8 & 4 & 5 \end{array}$$

This process is very cleanly expressed and formalized using our summation-based notation. From there

we can reformulate it around the recursive decomposition of B. For n digit number B and any number A,

$$\begin{aligned} A \times B &= A \times \left(\sum_{i=0}^{n-1} B_i 10^i \right) \\ &= \sum_{i=0}^{n-1} AB_i 10^i \\ &= AB_0 + \sum_{i=1}^{n-1} AB_i 10^i \\ &= AB_0 + \left(\sum_{i=1}^{n-1} AB_i 10^{i-1} \right) 10 \end{aligned}$$

What you were told to do is, in fact, to distribute the first operand, A, across the expanded representation of the second, B, and then carry out the computation implied by the expanded representation of the result. These implied computations were simpler: multiplication of an n digit number by a single digit, multiplication by a power of the base, and basic sums. As the final line of the equation show, this cleanly maps to the recursive structure of B. So, we know how to do the addition, but we need to work out the details of the more constrained variants of multiplication.

The first and simplest variant of multiplication is that of a number A by a power of the base. For power 10^i, this amounts to a shift to the left by i places, padding with 0 as you shift. Mapping this to our our summation-based notation clearly illustrates why. For n digit number A,

$$\begin{aligned} A \times 10^k &= \left(\sum_{i=0}^{n-1} A_i 10^i \right) \times 10^k \\ &= \sum_{i=0}^{n-1} A_i 10^i 10^k \\ &= \sum_{i=0}^{n-1} A_i 10^{i+k} \end{aligned}$$

In the case of your BigInt class, no multiplication need actually take place here, we simply need to extend the length of the number by adding the appropriate number of zeros in the low order places.

The other helper we need carries multiplication of an n digit number by a single digit number. This is really just a special case of the general multiplication process and we can see this by mapping it out across the summation expansion of the n digit number. Let b be a single digit number with A an n digit number. Then,

$$\begin{aligned} A \times b &= \left(\sum_{i=0}^{n-1} A_i 10^i \right) \times b \\ &= \sum_{i=0}^{n-1} A_i b 10^i \\ &= A_0 b + \sum_{i=1}^{n-1} A_i b 10^i \\ &= A_0 b + \left(\sum_{i=1}^{n-1} A_i b 10^{i-1} \right) 10 \end{aligned}$$

For the sake of efficiency, it's important to note that the term $A_0 b$ is at most two digits. If our general purpose addition operation won't stop after dealing with only those digits and their potential carry, then we should work out this operation to do so. What needs to be avoided is a complete traversal of the other $n-2$ or $n-3$ digits of the final product.

We can now put the general multiplication algorithm together. Let's revisit the recursive formulation:

$$A \times B = AB_0 + \left(\sum_{i=1}^{n-1} AB_i 10^{i-1} \right) 10$$

Now reconsider 123×15:

$$\begin{aligned} 123 \times 15 &= 123 \times 5 + (123 \times 1) \times 10 \\ &= 615 + 1230 \\ &= 1845 \end{aligned}$$

Finally, let's look at something with a bit more recursive depth 123×115.

$$\begin{aligned} 123 \times 115 &= 123 \times 5 + (123 \times 11) \times 10 \\ &= 615 + 1353 \times 10 \\ &= 615 + 13530 \\ &= 14145 \end{aligned}$$

If you're not seeing the recursive patterns at play both in the structure of the numbers and in the process of multiplication, you should continue to work examples by hand in the same fashion as the past two examples. Doing the same thing for the helpers is also highly recommended as it lets you peel back the curtain on this top-level perspective.

Mathematical Decomposition

Your *BigInt* class embeds the natural recursive structure of a number in an explicit linked-list structure. You can recursively process the number by recursively processing the list and will do so when implementing addition and multiplication. However, it's important to know how to deconstruct integer values in a digit-by-digit fashion as well.

You've already seen the underlying mechanism by which you can break down a number into its digits: integer division and modulus by the number's base. For an n digit number, the *first* digit is in the 10^{n-1}'s place and by doing integer division by 10^{n-1} we get that digit. Similarly, by doing modulo by 10^{n-1} you get the rest. This is exactly what we did when dealing with sums but the place we cared about was the 10s place. Here's another example.

$$\left\lfloor \frac{1234}{10^3} \right\rfloor = 1$$

$$\left(\frac{1234}{10^3} \right) \mod 10^3 = 234$$

This deconstruction can be repeated iteratively or recursively to get the digits in most to least significant order. The catch here is that you have to know the initial length, in digits, of the number. If you're given the number as a numerical value, then how can we determine the length in digits? The computer always sees an *int* value as 32 bits. To answer this, it's helpful to first consider working digits in the opposite order: least to greatest. In doing so, we'll discover a familiar pattern.

If you'd like to work from *last*, the least significant digit, to *all but the last*, the $n-1$ most significant digits, then a successive division and modulo works for you yet again. The last digit is what you get when you take the modulo by 10.

$$1234 \mod 10 = 4$$

To get all but the last *as a three digit number*, we simply need to do integer division by 10.

$$\left\lfloor \frac{1234}{10} \right\rfloor = 123$$

Repeating this iteratively or recursively gives you the digits in least to greatest order.

Let's now revisit the question of the number of digits in the base 10 representation of a number. Successively dividing by 10 will, eventually, net you a single digit number, the base case. This lets us restate or initial question as, "how many times do we have to divide by 10 in order to get a number between 0 and 9?" The answer to this is exactly one less than the number of digits needed to represent the number.

If the number is an exact power of 10, then this problem takes on a familiar form as the final base case is exactly 1: "how many times do we need to divide by 10 to get 1?" Logarithms come to the rescue. For number $N = 10^k$,

$$\begin{aligned} \frac{N}{10^k} &= 1 \\ N &= 10^k \\ \log_{10} Nk & \end{aligned}$$

So we need $(\log_{10} N) + 1$ digits to represent $N = 10^k$. For example, $10^2 = 100$ requires 3 digits, $1 = 10^0$ requires 1 digit, $10000 = 10^4$ requires 5 digits, and so forth.

If a number isn't an exact power of 10, like 123 then it must fall between two powers of 10, like 100 and 1000, and the number of digits must be the same as the number needed for the power of 10 below the number in question. Mathematically, there's two ways to get this number, one involving rounding down and one involving rounding up, i.e. the ceiling function.

$$\lfloor \log_{10} N \rfloor + 1 = \lceil \log_{10} N \rceil$$

Using the floor gives us something consistent with what happens for exact powers of 10. Using the ceiling, saves us the +1 by recognizing that rounding up is the same as rounding down and then adding one. Either way you view it, we now know how many digits we need to represent the number N is base 10: $\lfloor \log_{10}(N) \rfloor + 1$

Finally, it's important to notice that changing the base to b from 10 simply means replacing all the 10 and powers of 10 logic with b and powers of b. If you do all of this with a base 10 system, then you're just expressing the value of each base b digit as a base 10 number.

4 The Path to the Private Details of a Class

The *BigInt* class represents numbers. We're representing these numbers with general list-like structure. As opposed to previous number classes, this class has a non-obvious representation. The clients of our class need not worry about this nor should they have to change how they think about numbers because of how we implemented these particular numbers. The challenge is then to provide a natural public interface while efficiently handling the private representation.

We want keep operators like + and $*$ outside the class but their implementation clearly needs access to the private structure. The solution is a common design idiom. The non-class operator makes a call to a public class method and the public class method works with the private implementation. In the case of *operator+*, we could call an *add* method. So, the expression $a + b$ makes a call to $a.add(b)$ which then gives us access to the private structure of BigInts a and b. The alternative, which I do not want you to use, is making *operator+* a *friend* procedure like you saw in the textbook's implementation of the *Rational* class. This gives the operator access to the private parts of the BigInt class.

The algorithms for addition and multiplication aren't easily or efficiently expressed in terms of public class methods. This means we need a series of private methods to carry out underlying algorithms. You have two paths towards this end. First, make private methods true class methods. To facilitate this it is often helpful to have private constructor that take in instance of the private data type to initialize the object. Users should almost never directly initialize private data, but you, the class designer and implementer, might have need to in the context of other class implementation. For *BigInt* this would mean a private constructor that takes a pointer to a list node. Such a constructor can turn the *rest* of a number into a *BigInt*. In addition to constructors, you can make private methods like *first* and *rest* that let you easily work with a **BigInt** in a recursive fashion. The trick here is to beware of unintended mutation and when pointers get involved, aliasing.

The second option is to drop all the class stuff and write good old fashioned procedures for the underlying private structure. By making these procedures a part of the class definition, you can call them in the context

of public and private class methods. Let's say you have some class *Fee* with a private integer *anum* and string *astr* and you need to do some involved, perhaps recursive, process involving *anum* and *astr* and resulting in an integer. That process can be defined as a procedure *foo* taking an int and a string and returning an int. It doesn't need to be publicly accessible because it's only meant to facilitate the class implementation. The private part of the class is a perfect place for it. However, it's not a class method, it's just a traditional procedure. To make a private, pure procedure inside a class you declare the procedure as *static* like so:

```
// inside the Fee class ...
   private:
      int anum;
      std::string astr;

      // static, private procedure
      static int foo(int x, std::string str);
```

The static procedure *foo* isn't a class method. It has no *this* and therefore no implicit *anum* and *astr*. To use it you call would instead call *foo* from within some class method like this:

```
//inside a (possibly Public) class method for Fee

 ... Fee::foo(this->anum,this->astr) ...
```

Notice that the class name, *Fee*, is used like a namespace name with respect to the procedure *foo*. It is not a namespace though. You *must* always include the *Fee::* in front of foo. There is no shortcut around this. Now, because this procedure call occurs within a class method, we can use that method's *this* pointer to access and pass the private data stored in *anum* and *astr*, and because *foo* is private to the class it's scope is contained to exactly the place it will be used, the *Fee* implementation.

You are free to use either private class methods or private static procedures. The former has the advantage of being consistent within the class. Everything will be methods or variables. The later forces you to mix pure procedures with class methods within a class and this can require care attention to the task at hand. Using a call to *this* within a static method will cause compiler errors. On the other hand, you have a lot of practice with pure procedures and they aren't complicated by implicit parameters like the *this* pointer. Which route you take is up to you. Just be careful when pointers get involved. It's very easy to to do some inadvertent mutation through a pointer.

Testing Private Methods

By definition, private methods are not accessible outside of the class. This means they are not accessible where we do our testing. One way around this is to not test them directly but instead carefully and deliberately test the public method(s) and procedures that use them. It's OK if you go this route, but when those top-level tests fail and you're not sure why, it's nice to be able to test the private code.

The gTest framework discusses some strategies for testing private class methods in the documentation:

`https://code.google.com/p/googletest/wiki/AdvancedGuide#Testing_Private_Code`.

The simplest route is using the Friend Tests. The problem with this technique is you now have testing code where we don't typically want it, in the declarations and definitions proper. For the purpose of this project, that's OK.

Alternative List Structures [Optional Variations]

The implementation implied by the textbook is to store each digit of the base 10 number in a linked-list structure as an integer. This is an inefficient use of space for a few reasons. First, we could use less storage by using an *unsigned byte*, a strictly positive 8 bit integer. This leaves us with more than enough bits to store a digit and do digit-by-digit addition and multiplication. This just reduces the storage per node though, it's possible to cut down the number of nodes as well by storing the number in something other than base 10.

If we step up the digit to an *unsigned short* (positive 16 bit number) and then think of each digit not as a base ten digit, but a base 256 digit, then we can shrink the representation of the number quite a bit. We choose 256 here because a single 'digit' in this system can be represented with an unsigned byte and an unsigned short gives you enough wiggle room to do digit level operations. We won't have to do anything funny to the under the hood types and the logic of the arithmetic algorithms won't change, it will just adjust to the new base. You'll also need to be sure to adjust things so that the client is unaware of the internal representation, just like you're largely unaware that ints are stored in binary.

Playing with the base of the number system like this is a classic exercise in computer science. If you've taken MATH260 and COMP230, you should try tackling this alternate design. Given the timing of this project, I'm not requiring this, but if you want to step things up and push yourself, this is the route to go for this project.

Another alternative is to create a doubly linked structure. The basic, singly linked structure only provides a link from a digit to the next least significant digit. This forces us to work in terms of most to least significant digits. A doubly linked structure adds a second pointer to the node that will point to the next most significant digit. This allows you to traverse the list in most to least and least to most significant order. This works really well if you maintain a pointer to both the first and last node of the number. The notable advantage of this change is you can now easily convert our recursive algorithms to iterative algorithms that traverse the number from least to most significant digit.