Homework #1 – COMP 250, Winter 2017 Mathieu Blanchette

Due date: January 30th 2017 , 23:59.

What to submit?

1. Your TestLargeInteger.java file
2. A PDF file containing your solution to Question 4.

How to submit?

- On MyCourses, under assignment 1 submissions, upload the two files.
- Do not zip the files
- Do not upload any .class files
- Only modify the portions of the code labeled as /*WRITE YOUR CODE HERE */
- You can write new methods if you want, but do not change those we give you.

IMPORTANT: Copying code that is not your own (e.g. found on the internet or elsewhere) is **cheating.** We have tools that will check for that automatically.

Background

Multiplying two integer numbers can be done using a surprisingly large number of different algorithms, some much more efficient than others. In this assignment, you are going to investigate some of these algorithms and you are going to implement them in Java. To make things more interesting, your algorithms will have to be able to multiply arbitrarily large non-negative integers, for example numbers with several thousand digits. For this purpose, we are providing you with a partially implemented Java class called LargeInteger. The code for this class is available at:
http://www.cs.mcgill.ca/~blanchem/250/hw1

The class stores integers in an array of digits, with each element of the array corresponding to one digit of the number. The numbers manipulated can thus have as many digits as will fit in your computer memory (for example, an integer with 1 Million digits (e.g. 10^1000000) would easily be conceivable). The class LargeInteger currently has several methods already implemented: the add method adds two LargeInteger and returns the resulting LargeInteger, the subtract method does subtraction, the equals method tests if two LargeInteger have the same value, etc. We also provide you with a few constructors as well as a toString method. All you need to do (!) is to implement each of the multiplication algorithms described below, test them with small LargeInteger multiplications that you can check by hand, and measure their running time for multiplying LargeIntegers of several hundred digits (see below). Note: Java actually already has a class called BigInteger that is very similar to our LargeInteger class. Please do not use anything from the BigInteger class, except possibly for checking your results if you want.

Before starting writing your own code, take some time to become familiar with the code already provided in the LargeInteger class. This will help you in two ways: (i) it will give

you examples of how these large numbers are stored and manipulated, and (ii) it will give you the tools that you will need to use to implement the multiplication algorithms.

Your code will be evaluated on several criteria

1) Correctness (80%). Your program always returns the correct product.
2) Reuse of available code (10%): You use the code already provide as much as possible.
3) Style and comments (10%): You document the new methods you write. You indent your code properly and use meaningful variable names.
4) Speed (0%), so don't spend time optimizing your program. This is not a speed competition.

Question 1. (20 pts)

The most naive way to compute $a * b$ is to repeatedly add b in a loop executed a times: $8 * 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4$. Implement this algorithm in the method called iterativeAddition of the LargeInteger class. Make sure to use the add and equals methods that are provided! Remember that the counter in your loop will need to be a LargeInteger too!

Question 2. (30 pts)

A better way to multiply two non-negative integers is the method taught to kids in grade school. (Let me know if you're not familiar with this method; different countries use different variants.)

For example:

```
        3758937
    x    370821
  -------------
       3758937
      7517874
     30071496
     0000000
   26312559
+11276811
 ---------------
 1393892777277
```

Algorithm standardMultiplication(a,b)

Input: a = $a_0a_1...a_{k-1}$ and b = $b_0b_1...b_{n-1}$ are non-negative integers of k and n digits respectively. In other words, a = $10^{k-1}\ a_0 + 10^{k-2}\ a_1 + ... + 10^0\ a_{k-1}$, and similarly for b.

Output: a * b

total ← 0

for i ← n-1 **to** 0 **do**

  carry ← 0

  tmpAdd ← Array of k+1 digits

  for j = k-1 **downto** 0 **do**

```
            c ← b_i * a_j + carry          /*This is a single-digit multiplication, so you should
                                             use the normal Java multiplication "*" defined on int. */
            tmpAdd_{j+1} ← c mod 10  // this is the remainder of the integer division
                                     // In java, write c % 10.
            carry ← ⌊c/10⌋           // this is the floor of the division c/10. If c is
                                     // an int, then carry = c/10; will work.
        tmpAdd_0 ← carry
        total ← total + tmpAdd*10^(n-i-1)  // Note: this is not a difficult multiplication:
                                           //simply add n-i-1 zeros at the end of tmpAdd!
    return total
```

Implement this pseudocode in the standardMultiplication method of the LargeInteger class. It is OK to use the Java multiplication operator * defined on int but only to multiply single-digit numbers.

Question 3. (20 pts)

Background: The code given to you contains a method called recursiveMultiplication, which implements the following recursive algorithm. I'm giving you this code as an example, because it will help you implement the "recursiveFastMultiplication" algorithm described further below.

RecursiveMultiplication:
Suppose *a* has k digits and *b* has n digits. Suppose l_a and r_a are the left and right halves of the digits of a, and l_b and r_b are the left and right half of the digits of b. In other words, $a = 10^{\lfloor k/2 \rfloor} l_a + r_a$ and $b = 10^{\lfloor n/2 \rfloor} l_b + r_b$. N.B. The notation $\lfloor k/2 \rfloor$ means the largest integer smaller or equal to k/2 (in java, if k is an int, then simply writing k/2, as an integer division, gives you $\lfloor k/2 \rfloor$).
Then
$a\ b = (10^{\lfloor k/2 \rfloor} l_a + r_a)\ (10^{\lfloor n/2 \rfloor} l_b + r_b) = r_a r_b + 10^{\lfloor n/2 \rfloor} r_a l_b + 10^{\lfloor k/2 \rfloor} l_a r_b + 10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor} l_a l_b$
So, we have reduced the problem of multiplying an n-digit number with a k-digit number to four multiplications of numbers of about n/2 and k/2 digits, three multiplications by powers of ten (which consist of simply adding zeros), and three additions. Here is an example: Assume a= 891 and b=1234, so that k=3, n =4.
Then $l_a = 89$, $r_a = 1$, $l_b = 12$, $r_b = 34$.
term1 = $r_a r_b = 1 * 34 = 34$
term2 = $10^2 * r_a l_b = 100 * 1 * 12 = 1200$
term3 = $10^1 * l_a r_b = 10 * 89 * 34 = 30260$
term4 = $10^{1+2} * l_a l_b = 1000 * 89 * 12 = 1068000$
Finally a b = 34 + 1200 + 30260 + 1068000 = 1099494

Here is the pseudocode:

Algorithm recursiveMultiplication(a,b)
Input: $a = a_0a_1...a_{k-1}$ and $b = b_0b_1...b_{n-1}$ are non-negative integers of k and n digits respectively. In other words, $a = 10^{k-1} a_0 + 10^{k-2} a_1 + ... + 10^0 a_{k-1}$, and similarly for b.
Output: a*b
if (k=1 **and** n=1) **return** $a_0 * b_0$;
term1 ← term2 ← term3 ← term4 ← 0;
$l_a \leftarrow (a_0...a_{k-\lfloor k/2 \rfloor-1})$
$r_a \leftarrow (a_{k-\lfloor k/2 \rfloor}...a_{k-1})$
$l_b \leftarrow (b_0...b_{n-\lfloor n/2 \rfloor-1})$
$r_b \leftarrow (b_{n-\lfloor n/2 \rfloor}...b_{n-1})$
if (n>1 **and** k>1) **then** term1 ← recursiveMultiplication(r_a , r_b)
if (k>1) **then** term2 ← $10^{\lfloor n/2 \rfloor}$ * recursiveMultiplication(r_a , l_b)
if (n>1) **then** term3 ← $10^{\lfloor k/2 \rfloor}$ * recursiveMultiplication(l_a , r_b)
term4 = $10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor}$ * recursiveMultiplication(l_a, l_b)
return term1 + term2 + term3 + term4

RecursiveFastMultiplication: (what you actually need to implement)
In the previous example, to compute the product of two integers, we needed four multiplications of numbers half as long as the original ones. Here, we will show that we can actually do it with only three such multiplications, and the difference will be extremely significant when multiplying very large integers.

Again, suppose $a = 10^{\lfloor k/2 \rfloor} l_a + r_a$ and $b = 10^{\lfloor n/2 \rfloor} l_b + r_b$. Assume that $n \geq k$ (if this is not the case, then simply switch a and b so that the longest number is then b). We have

$$a\,b = (10^{\lfloor k/2 \rfloor} l_a + r_a)\ (10^{\lfloor n/2 \rfloor} l_b + r_b)$$
$$= r_a r_b + 10^{\lfloor n/2 \rfloor} r_a l_b + 10^{\lfloor k/2 \rfloor} l_a r_b + 10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor} l_a l_b$$
$$= r_a r_b + 10^{\lfloor k/2 \rfloor} (10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} r_a l_b + l_a r_b) + 10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor} l_a l_b \quad [1]$$

It turns out that it is possible compute $(10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} r_a l_b + l_a r_b)$ using only one multiplication instead of two. How is this miracle possible? Consider the product
$(l_a + r_a) * (10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_b + r_b) = 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_a l_b + (l_a r_b + 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} r_a l_b) + r_a r_b$

Thus, $(10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} r_a l_b + l_a r_b) = (l_a + r_a)\ (10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_b + r_b) - 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_a l_b - r_a r_b$
But it so happens that $l_a l_b$ and $r_a r_b$ are two of the products needed in [1];
Thus, if we compute
term1 = $r_a * r_b$
term2 = $l_a * l_b$
term3 = $(l_a + r_a) * (10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_b + r_b)$ - $10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor}$ term2 – term1
we get a * b = $10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor}$ term2 + $10^{\lfloor k/2 \rfloor}$ term3 + term1.
Thus, we have computed a*b using just ***three*** multiplications of numbers about half as long as the originals. Notice that the method doesn't work when k=1 (why?), so in that case, we will have to rely on one of the multiplication algorithms defined previously.

Here is an example:
Assume a= 891 and b=1234, so that k=3, n =4.
Then $l_a = 89$, $r_a = 1$, $l_b = 12$, $r_b = 34$.
term1 = $r_a\ r_b = 1 * 34 = 34$
term2 = $l_a\ l_b = 89 * 12 = 1068$
term3 = $(89 + 1) * (10^{2-1*} 12 + 34) - 10^{2-1} * 1068 - 34 = 90 * 154 - 10680 - 34$
$= 3146$
then a b = $10^{1+2}\ 1068 + 10^1\ 3146 + 34 = 1099494$.

So we've worked very hard for saving one single multiplication. Big deal, say you? You'll see that the answer is yes, at least when the numbers multiplied are very long.

So the pseudocode is:

Algorithm recursiveFastMultiplication(a,b)
Input: : a = $a_0a_1...a_{k-1}$ and b = $b_0b_1...b_{n-1}$ are non-negative integers of k and n digits respectively. In other words, a = $10^{k-1}\ a_0 + 10^{k-2}\ a_1 + ... + 10^0\ a_{k-1}$, and similarly for b.
Output: a * b
if (n<k) **then return** recursiveFastMultiplication(b, a);
if (k=1) **then return** standardMultiplication(a, b);
term1←term2 ← term3 ← 0;
$l_a \leftarrow (a_0...a_{k-\lfloor k/2 \rfloor-1})$
$r_a \leftarrow (a_{k-\lfloor k/2 \rfloor}...a_{k-1})$
$l_b \leftarrow (b_0...b_{n-\lfloor n/2 \rfloor-1})$
$r_b \leftarrow (b_{n-\lfloor n/2 \rfloor}...b_{n-1})$
term1 ← recursiveFastMultiplication(r_a , r_b)
term2 ← recursiveFastMultiplication(l_a , l_b)
term3 ← recursiveFastMultiplication($l_a + r_a$, $10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor}\ l_b + r_b$)
$- 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor}$ term2 – term1
return $10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor}$term2 + $10^{\lfloor k/2 \rfloor}$ term3 + term1

Implement this pseudocode in the recursiveFastMultiplication method of the LargeInteger class. This will be a recursive method. It is OK to use the Java multiplication operator * but only to multiply single-digit numbers.

Question 4. (30 pts) (to be turned-in as a PDF file)

So, which algorithm is best? This is for you to discover! For each multiplication algorithm implemented (and for recursiveMultiplication too), try to measure the average running time needed to compute product of two random numbers of

- 2 digits each
- 4 digits each
- 8 digits each
- 16 digits each
- ...
- 1024 digits each

- 2048 digits each
- 4096 digits each

For small numbers, the execution will be too fast to be measured reliably, so you should measure the time to do, say, 1000 such multiplications (each time on a different pair of random numbers), and then divide the total running time by 1000.
You can use

```
long currentTime = System.nanoTime();
```

to get the time (in nanoseconds) before and after the execution of the 1000 repetitions.

Don't worry about the fact that the time measured will include that for generating the random numbers; it is negligible. When running your experiments, make sure that your computer isn't running too many other jobs; this may affect your measurements.

For larger numbers, certain methods will take too long to run, so you should reduce the number of repetitions so that the total running time is not more than a few minutes. Don't spend hours doing this! For some methods, even multiplying two 8-digits numbers will take forever. If a method takes more than one minute to do a single multiplication, simply abort the computation and report that it was too slow.

Question:
a) Report a table for the average running time for a single multiplication using each method for each number of digits above (when possible).

Based on your observations, try to detect the pattern in the increase of running time as the size of the numbers is doubled. The pattern will be clearer for large numbers and should involve small whole numbers. It will help to see the pattern if you understand clearly how each algorithm works.
b) Predict (approximately) what would be the running time of each method for multiplying two 8192-digit numbers.
d) For each method, give a formula that describes (approximately) the running time for multiplying two n-digit numbers? Your formula should only try to be accurate for large values of n, as for small values of n several factors complicate the analysis.
e) For which value of n does the recursiveFastMultiplication algorithm become faster than the other three?

Good luck!