COMP ENG 2SI4: Data Structures, Algorithms, and Discrete Mathematics

Lab 3/4 – Operations with Huge Integer

Instructor:

Dr. Dumitrescu

Moshiur Howlader – HOWLAM – 1316948

L08

04/07/2016

Description of Data Structures and Algorithms

The data structure used to implement my HugeInteger class was arrays containing digits to all of the HugeInteger, and there were 4 instance fields, which were op, size, myArray[], and sign. The field op was used to keep track of whether the HugeInteger object underwent an arithmetic operation so that we can keep track of what was happening to There were 4 main operations for HugeInteger class to be implemented, which were addition, subtraction, multiplication, and compareTo.

For my addition method, several book keeping variables were used to carry out the long hand addition that many have learned in elementary school. First, the method checked to see if this.size or h.size object was greater. The greater size was set to be the size of my temp HugeInteger which stored the values of the addition between the integers of this.hugearray and h.hugearray. Worst case we extend the size of temp by one extra digit if the addition requires us to. As addition of all the digits from both array is required, we loop through both of the array. We add the array bit by bit, carry over the carry when necessary. However, if the addition involves a negative number we call the subtract method. Regardless, the output array is removed with leading zeros, as necessary at the end since one of the array will contain leading zeros.

For my subtract method, we check if op is 0 or 1. If op is 0, that means arithmetic operations has not occurred on the HugeInteger object, therefore, we simply subtract h object to this object. To subtract we again loop through from highest index to the lowest index, carrying the borrow as required. If we are subtracting two numbers when op is 1, that means we call the add functions by reversing the h object sign (recall opposite of subtraction is addition).

For my multiplication method variable x stored the size of the larger size between this.size and h.size. Variable y stored the lesser size of the array or integer. Two variables tempsize1 and tempsize2 were used to store this.size and h.size. A for loop is then used to loop through the array backwards and setting the respective index values. As well, setting the values of h.myArray into origDig, if larger or equal use h or setting the values of this.myArray into origDig if tempsize2>tempsize1. Similar algorithm was also used for fDig, which was used to store the value of myArray[i-1] or h.myArray[i-1] depending on tempsize1 and tempsize2. Also depending on the carry value (0 or 1), I had my output variable equal to (origDig*hDig+carry)%10, and my product variable = product+finalsum. Also within the loop, there was an if statement as follows: if(k-1==0). This if statement was used to check whether we have approached the front of the array to add the final carry to the product. After all of this, since the output is reverse of what we want, we reverse the product using the reverse() from the stringBuilder class (an Input/Output class). After we call the add method to sum up all the results of the multiplication and store in my final output string.

For my compareTo method, the code was very simple; there were mainly 3 cases to consider. First was whether this or h object has positive or negative sign. Second was whether this or h

object had more digits. After that (knowing the number of digits and sign for both was the same) we brute force and check to see which object had the largest number checking from MSB to LSB bit by bit until we found which number was larger.

Theoretical Analysis of Running Time and Memory Requirement

After inspecting the code, the add method can be concluded to run in $\Theta(n)$ because at most the code only uses single layer loop. Intuitively makes sense as well because we are implementing the long-hand addition we learned from elementary school. This requires loops through both of the HugeInteger object arrays, and addition bit by bit, doing my constant time operation such as carrying the bits over. Subtract method uses similar techniques as the add method in which we loop through the arrays bit by bit and subtracting, doing my constant time operations such as carrying the borrows over. Subtract method therefore has running time of $\Theta(n)$. When we multiply two numbers by long hand form, we are effectively looping through each digits of the operand with each digits of the operator, and summing everything up. This suggests that we require 2 nested for loops, which we have in our multiply method. Therefore, our multiply method is $\Theta(n^2)$. The compareTo method had several if statements all running in $\Theta(1)$ as well as for loop used to compare individual digits bit by bit. This meant that overall run time was $\Theta(n)$ in the worst case and $\Theta(1)$ in the best case.

Test Procedure

To test that my code works as expected, I wrote a test class consisting of 16 test cases. All of my methods seem to have complied with the expected output. And the TA confirmed in lab that my outputs were correct for all the test cases. Test 1 to test 3 just verified the constructors work as intended from lab 3. Test 4 to 6 tests whether addition works. Tests 7 to 9 tests for subtraction. Tests 10 to 12 checks for whether multiply works. And last 4 test cases checks whether compareTo works. Please check the TestClass.java to verify that all the operations are consistent with what the results should be.

Below is the log of the outputs:

```
Test 1 : constructor 1 (positive case)
123113
Test 2 : constructor 1 (negative case)
-43553
Test 3 : constructor 2 (normal case)
-6335567694
Test 4 : add (positive numbers)
180578
Test 5 : add (negative numbers)
```

```
-180578
Test 6 : add (positive and negative numbers)
3
Test 7 : subtract (positive numbers)
10733000
Test 8 : subtract (negative numbers)
1
Test 9 : subtract (positive and negative numbers)
-13780578
Test 10 : multiply (positive numbers)
30625
Test 11 : multiply (negative numbers)
69495489
Test 12 : multiply (positive and negative numbers)
-69495489
Test 13 : compareTo (positive numbers)
-1
output should be -1
Test 14 : compareTo (negative numbers)
1
output should be 1
Test 15 : compareTo (positive and negative numbers)
-1
output should be -1
Test 16 : compareTo (same numbers)
0
output should be 0
```

Experimental Measurement, Comparison, and Discussion

We measured the running time of each operation by using professor's running time code and checking the running time for same sized inputs for both HugeInteger class' method and BigInteger's method all the add, subtract, multiply, and compareTo methods.

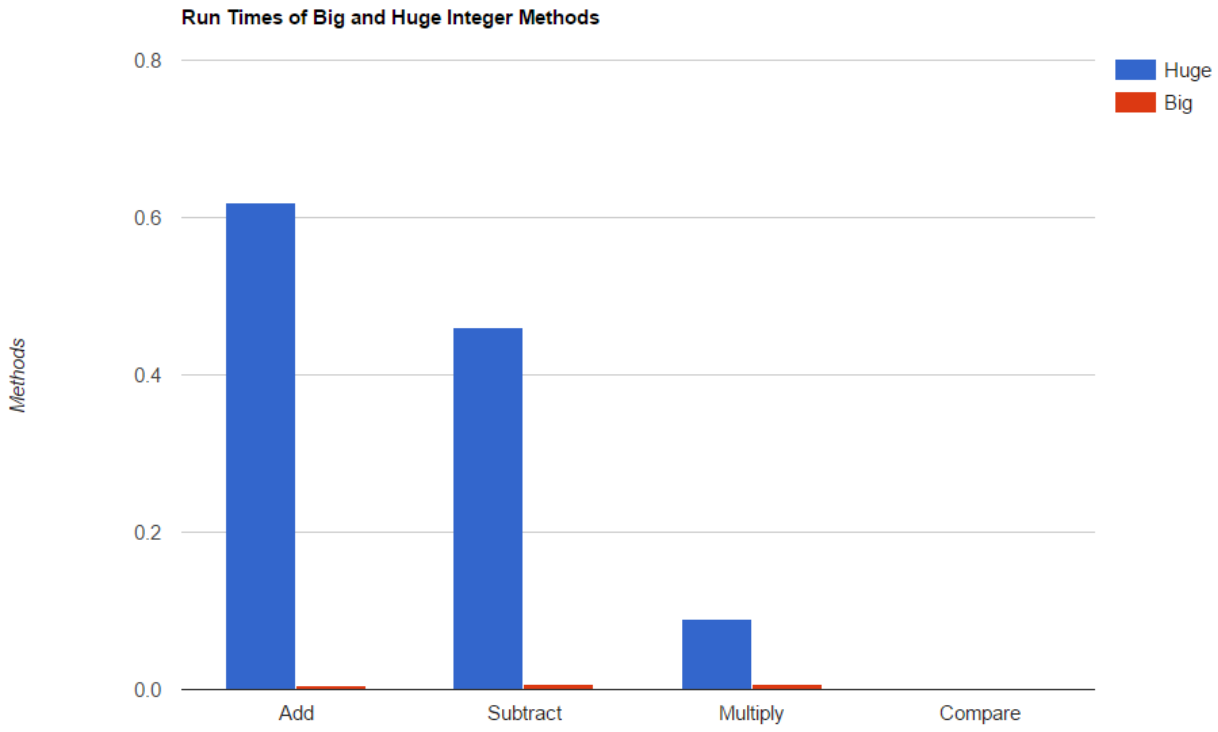
The following table is the log of the experimental results of the HugeInteger and BigInteger class below. Note that I wrote getRandomNumber(int digCount) and its associated helper method getRandomNumber(int digCount, Random rnd) to help me create an array of random integers to input into the BigInteger's constructor class (as there were no constructors that created a random BigInteger of n digits). We tried different size digits, and different number of loops and found the results to indicate similar results.

```
Run time for add with my HugeInteger Class:
The endTime-startTime is:
309.49000000000007
The run time (ms) for add is:
0.6189800000000001
****************************************
Run time for add with BigInteger Class:
The endTime-startTime is:
```

```
2.4000000000000012
The run time (ms) for add is:
0.004800000000000002
*****************************************
Run time for subtract with my HugeInteger Class:
The endTime-startTime is:
322.08000000000004
The run time (ms) for subtract is:
0.46011428571428575
*****************************************
Run time for subtract with BigInteger Class:
The endTime-startTime is:
3.1700000000000017
The run time (ms) for subtract is:
0.006340000000000004
*****************************************
Run time for multiply with my HugeInteger Class:
The endTime-startTime is:
0.09
The run time (ms) for multiply is:
0.09
*****************************************
Run time for multiply with BigInteger Class:
The endTime-startTime is:
326.11999999999966
The run time (ms) for multiply is:
0.006522399999
*****************************************
Run time for compareTo with my HugeInteger Class:
The endTime-startTime is:
256.64
The run time for compareTo is:
5.132799999999999E-6
*****************************************
Run time for compareTo with BigInteger Class:
The endTime-startTime is:
0.02
The run time (ms) for compareTo is:
4.0E-9
*****************************************
```

Discussion of Results and Comparison

As expected, the Java's implementation if much more efficient than our implementation for all equivalent methods. Above the reader can see the data all measured in milliseconds. We estimate that the add method is $\Theta(n)$, subtract method to be $\Theta(n)$, and multiply method to be $\Theta(n^2)$, and compareTo method to be $\Theta(1)$. These big theta running time is proportional respective to each class's results. Given more time, I might have tried to implement the Karasuba's algorithm for the multiplication method. Below is the graph of the results. Graphically, our theoretical results align, for example the compareTo method is constant time. However the multiply method seems to run faster than we expect. This is strange considering our multiply method should be $\Theta(n^2)$.


Run Times of Big and Huge Integer Methods
0.8
0.6
0.4
0.2
0.0
Methods
Huge
Big
Add
Subtract
Multiply
Compare