Computer Engineering 2SI4 Lab #3 and #4 Report

Start Date: February 27th, 2016
Mid-point Date: March 10th, 2016
End Date: March 12th, 2016
Acknowledgements: Christine Horner, for discussing how to handle the case of addition in which the sum has more digits than the two arguments (has a carry out).

Description of Data Structure and Algorithms

Data Structure: My HugeInteger class was implemented with an array of integers, 'array', with each element of the array corresponding to a digit. A boolean instance field 'negative' identifies a number that is positive or negative – it is true if negative, false if positive. An integer instance field 'size' stored the size of the array. My implementation in Lab 3 did not have the 'negative' field but instead identified a negative value by having array[0] = 0. The implementation was changed to simplify the following algorithms, as the current one separates the sign from the digits.

For example, the number –16039 is implemented as the following:

negative = true; size = 5;
array =

1	**6**	**0**	**3**	**9**

Addition: The basis of this algorithm is a series of cases. There are 8 accounted cases: 4 for this.size > h.size and 4 for h.size > this.size. In each, the 4 cases are: a positive and positive input, a negative and negative input, and two cases for a negative and positive, but in opposite order. Most of these call either add() or subtract() after temporary field changes – so, I will focus on describing the algorithm for a simple case in which this and h are both positive.

To start, a string val_sum is initialized to "0" - the sum will be added to it, and then a new HugeInteger sum will be created through the use of the first constructor. There are two integer values, i and j, which loop through each HugeInteger array, starting at the end. In the first part, the code will add array[i] and h.array[j] through a loop that runs h.size times. In the second part, the code will simply add array[I] to val sum, as there are no more digits from h (this case is when this.size > h.size). A carry out is accounted for in the code, as if the intermediate sum is larger than 10, the modulus is added to val_sum and the next digit is incremented by 1. It is important to note that the way val_sum is being added to is val_sum = add + val_sum, as it must add onto the string at the front. Sum = new HugeInteger(val_sum) is returned at the end.

For example, the addition between 1780 and 503 is performed as the following:

Beginning Of Execution:

this.array =

1	7	8	0

h.array =

5	0	3

I = 3, j = 2;

Val_sum = "0";

Middle of Execution:

I = 2, j = 1;

Val_sum = "83";

End of Execution:

I = 0, j = 0;

Val_sum = "2283"

Sum.negative = false;

Sum.size = 4;

Sum.array =

2	2	8	3

Subtraction: The basis of this algorithm is a series of cases – please refer to the first paragraph of the addition method as the cases are the same. Again, I will focus on just describing the algorithm for the case in which this and h are both positive.

To start, a string val_diff is initialized to "0" - the difference will be added to it, and then it will be passed on to the first constructor to create a new object that is the difference between the two inputs. Two pointers, i and j, loop through the two inputs. If the difference between the corresponding digits is greater or equal to 0, the variable sub is assigned the difference. If it is less than 0, the carry over process is executed using a while loop. The while loop finds the next digit that is greater than 0 to take away 1 from, and then it will proceed to loop back to the current element and increase it by a factor of 10. Then, after the take away process, sub is assigned the value of the difference. For the remaining digits of array[i] where h.array[j] has no digits, sub = array[I]. Val_diff = sub + val_diff each time the loop runs, and then finally difference = new HugeInteger(val_diff) is returned.

For example, the subtraction between 6040 and 720 is performed as the following:

Beginning Of Execution:

this.array =

6	0	4	0

h.array =

7	2	0

i = 3, j = 2;

Val_diff = "0";

Middle of Execution:
I = 2, j = 1;
Sub = 2;
Val_diff = "20";

End of Execution:
I = 0, j = 0;
Val_diff = "5320";
difference.negative = false;
difference.size = 4;
difference.array =

5	3	2	0

Multiplication: This algorithm does not account for a large number of cases, as the sign can be accounted for separate of the digits, and the sizes can be accounted for by simply swapping the pointers a and b – a being a pointer to the larger array. To start, an array of HugeInteger objects is created called sum_array[] - this is for when the code produces a series of sums that need to be added together to result in the product. The sum is 'to_add' and it is a string that will be added on with intermediate sums.

j is a pointer to the smaller digit, and i is a pointer to the larger digit. For each digit of the b array, there will be a different to_add string produced. First, a variable m is used to see if 'to_add' has any trailing zeros – this occurs in long multiplication (ie. The second sum has 1 trailing zero). Then, a loop runs through all a array elements – intermediate_add = a[i]*b[j] + carry. If intermediate_add >= 10, the modulus is added to 'to_add' and the carry variable is incremented. When one to_add is finished, it is copied over to the sum_array[] and then reset for the next to_add string. The process is repeated b.length - 1 times. Then, add() is invoked on sum_array[] to add up all the elements and return the product. The sign of product is assigned by an if/else statement – if both are + or -, it is + but if it is both + and -, it is -.

For example, the multiplication between 8410 and -22 is performed as the following:

Beginning Of Execution:
this.array = a =

8	4	1	0

h.array = b =

2	2

i = 3, j = 1;
To_add= "";

Middle of Execution:

I = 0, j = 1;
To_add= "16820";
Sum_array =

16820	

End of Execution:
I = 0, j = 0;
To_add = "168200";
Sign = true;
Sum_array =

16820	168200

Product.negative = true;
product.size = 6;
product.array =

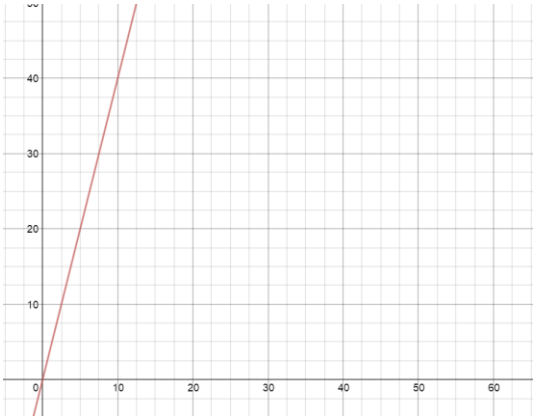
1	8	5	0	2	0

Comparison: The basis of this algorithm is a series of if/else statements that lead to the correct output. First, if the size of this is larger than h, or if this is positive and h is negative, the program returns 1 as this is larger than h, and vice versa. If the size and the sign of both this and h are the same, a loop through the arrays is executed and will exit once the digits no longer match (ie. When condition of array[i] == h.array[i] is broken). The element at which the loop stops at is used in another comparison, and the output will depend on if the signs of the HugeInteger objects are positive or negative (as negative inputs will result in the opposite output).

An example of a case would be comparing this = -456 and h = -472. Both are negative and of the same size, and so the while loop will execute. When I = 1, the elements do not match – array[i] < h.array[I]. However, since both are negative, this indicates that this input is larger than h input – thus 1 is returned.

Theoretical Analysis of Running Time and Memory Requirement

The amount of memory required to store n decimal digits using my class is 4*n. This is because it is implemented with an array of integers. Each digit is represented by 1 integer, and 1 integer needs 4 bytes of memory. On the graph below, the horizontal axis is the number of digits (n) and the vertical axis is amount of bytes.



Running Time and Memory Requirement of Addition: The amount of extra memory required will be constant. This is because a constant amount is required for variables such as I, j, val_sum, and add. In the worst case scenario, the running time is θ(n). This is because the algorithm loops through the numbers together – so it is directly proportional to the number of digits that the larger number has (n). As the inputs are random, the average running time is θ(n).

Running Time and Memory Requirement of Subtraction: The amount of extra memory require will be constant. This is because a constant amount is required for variables like I, j, k, m, val_diff and sub. The worst case scenario has a running time of θ(n). This is because the algorithm loops through the numbers together – so it is directly proportional to the number of digits that the larger number has (n). As the inputs are random, the average running time is θ(n).

Running Time and Memory Requirement of Comparison: There is a constant amount of extra memory required as only one variable in allocated, i. In the worst case scenario where both numbers are of the same sign, size and have the same digits up until the last digit, the running time is θ(n). This is because the program loops through both digits simultaneously in the while loop. The average running time is θ(n) as the majority of inputs will be of varying signs and sizes, and will result in a constant number of comparisons.

Running Time and Memory Requirement of Multiplication: The amount of extra memory required is n – the amount of digits of the smaller number. This is because an array called sum_array[] is allocated to store all the intermediate sums while performing the loop multiplication for each digit of the smaller number. The running time of the method is $\theta(n^2)$. This is because the algorithm loops through the first digit n times and loops through the second digit m times. The worst case arises when m = n. The average running time is also quadratic, as no matter what input is given, the algorithm loops through both arrays using nested loops.

Test Procedure

The test class created encompasses all the cases that were predicted of the inputs. Please refer to the dropbox submission for the full test class. The cases tested are described below:

First Constructor: A positive input of 15835 and a negative input of –499 are tested and give the expected output. The test class does not include a test of an incorrect input. However, a test was run where the input was a string "hello" and the expected exception was thrown.

Second Constructor: An input of 5 is tested and gives the correct output of a random number with 5 digits. The test class does not include a test of an incorrect input. However, a test was run where the input was -10 and the expected exception was thrown.

Addition: There were 8 cases tested – 4 where the first number is larger, and 4 where the second number is larger (equal is also accounted for). The four subcases are 1) Both inputs are positive 2) Both inputs are negative 3) The first input is positive and second is negative 4) The first input is negative and the second is positive. These eight cases account for all possible inputs of addition as each case must be treated differently. For instance if this = -100 and h = 56900, subtract() is actually invoked, along with a manipulation of a negative field. The corresponding output is 56800, which is the correct output. Due to the volume of tests, please refer to the test class for the full cases.

Subtraction: There were 8 cases tested – 4 where the first number is larger, and 4 where the second number is larger (equal is also accounted for). The four subcases are 1) Both inputs are positive 2) Both inputs are negative 3) The first input is positive and second is negative 4) The first input is negative and the second is positive. These eight cases account for all possible inputs of subtraction as each case must be treated differently. For instance if this = -250 and h = 60, add() is actually invoked, along with a manipulation of a negative field. The output is –310, which is the correct output. Due to the volume of tests, please refer to the test class for the full cases.

Comparison: Three cases are tested – 1) The first input is larger with this = 15385 and h = -499 which gives the correct output of 1. 2) The second input is larger with this = -127 and h = 127 which gives the correct output of –1. 3) The two inputs are the same with this = h = 59 which gives the correct output of 0.

Multiply: Three cases are tested – 1) The first input is bigger 2) The second input is bigger 3) Both inputs are negative. These account for all cases of multiplication and do not require a large amount of cases because multiplication is commutative, while addition/subtraction are not. Also, the signs are accounted for separate of the digits. An example of an input is from 3) where the tested inputs are –210 and –29. The resulting output is 6090 which is correct. Please refer to the test class for the full cases.

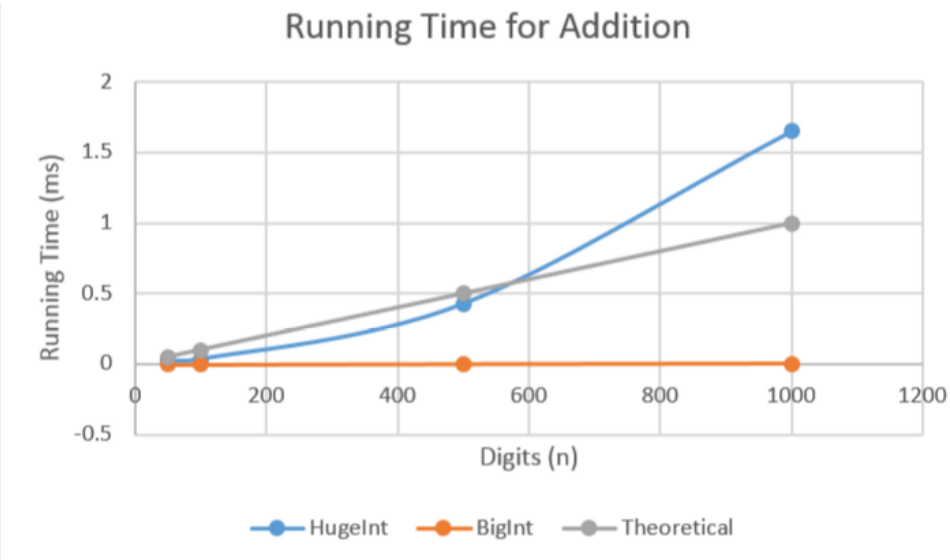
As seen above, the outputs meet the specifications, as the test cases show that each method performs properly. There were no inputs that were not accounted for, as this test class is very detailed. However, a couple of generic test cases were not tested – for instance, the multiply method did not include a test of having two negative inputs with the second input having more digits. This is because the size of each input did not matter.

There were no issues in debugging the code. However, a special case arose while testing the addition method in which the sum of the inputs resulted in a carry out at the very end. This was accounted for by modifying the addition method and making it so that once the loop finishes execution through the larger number, it will check for an end carry and if it exists, it will add onto val_sum at the end. After testing this case with inputs 700 and 315, the special case was accounted for as the output was correct - 1015.

Experimental Measurement, Comparison and Discussion

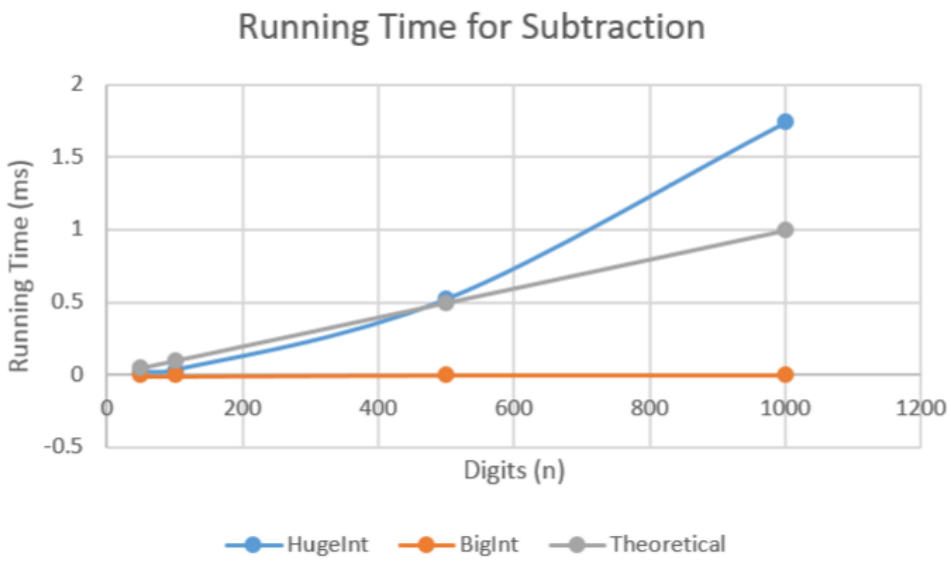
Experimental Measurements for Addition:

Digits (n)	HugeInteger Running Time (ms)	BigInteger Running Time (ms)	Theoretical Running Time (ms)
50	0.02369	0.00001	0.05000
100	0.03004	0.00001	0.01000
500	0.42385	0.00002	0.50000
1000	1.6515	0.00003	1.00000



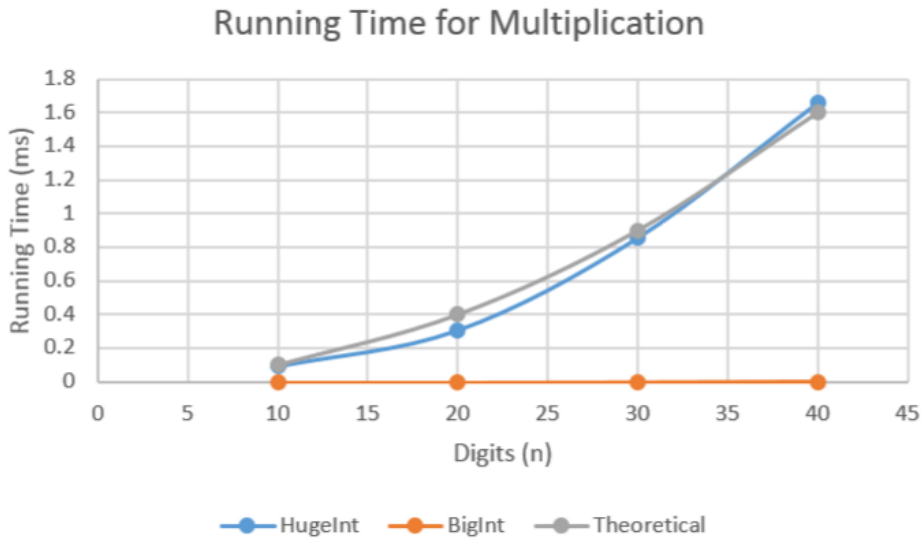
Experimental Measurements for Subtraction:

Digits (n)	HugeInteger Running Time (ms)	BigInteger Running Time (ms)	Theoretical Running Time (ms)
50	0.02420	0.00002	0.05000
100	0.03200	0.00002	0.10000
500	0.52515	0.00004	0.50000
1000	1.7425	0.00004	1.00000



Experimental Measurements for Multiply:

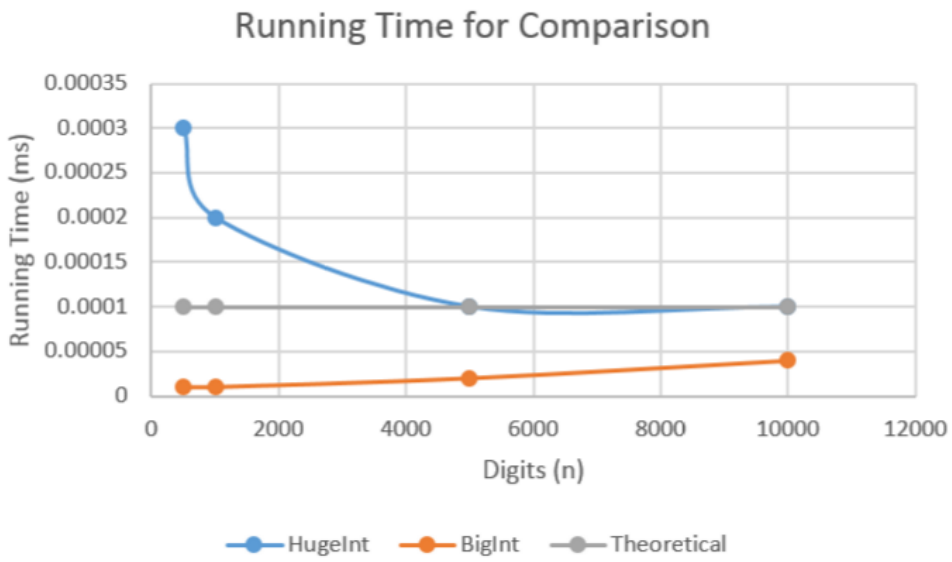
Digits (n)	HugeInteger Running Time (ms)	BigInteger Running Time (ms)	Theoretical Running Time (ms)
10	0.09250	0.00001	0.10000
20	0.30655	0.00001	0.40000
30	0.85545	0.00002	0.90000
40	1.65615	0.00002	1.60000



Experimental Measurements for Comparison:

Digits (n)	HugeInteger Running	BigInteger Running	Theoretical Running

	Time (ms)	Time (ms)	Time (ms)
500	0.00030	0.00000	0.00001
1000	0.00020	0.00000	0.00001
5000	0.00010	0.00000	0.00001
10000	0.00010	0.00000	0.00001



To calculate the running time of each operation, the code from the lab document was slightly modified. The code given will test the running time of one value of n (number of digits). To create an efficient class that calculates the running time of multiple n, an array of integers was created – array_of_n = {10, 50, 100, 500}. This array is used in a for loop that runs for the length of the array and will calculate the running time of each n. The same code is ran a total of eight times – four times for testing the HugeInteger class and four times to test the BigInteger class. The code is slightly modified by changing the objects created, and changing the methods called. The parameters used are the following: MAXNUMINTS = 100 and MAXRUN = 500 but for compareTo, a value of 1000 was used to obtain more accurate results. This is because the method has a constant run time and would require further accuracy.

There were no issues in retrieving the experimental data. For comparison, values of 0.00000 were printed in the output and so it was difficult to retrieve more accurate data. However, the values do agree with the theory of a very small running time, and so it was still sufficient to prove the theory.

Discussion of Results and Comparison

As seen from the graphs above, the theoretical calculations show a similar trend to the HugeInteger testing. For addition and subtraction, the experimental values agree with the linear trend that the theoretical suggests, with $\theta(n)$. For multiplication, the graph shows that the theoretical and experimental results are very similar, which confirms that the multiply method has a quadratic running time. For comparison, the experimental values show some fluctuation, but stabilizes to a constant

running time that is independent of the number of digits. This agrees with the theoretical calculation where compareTo has a running time of $\theta(1)$.

In general, the running time of each method is vastly faster for BigInteger compared to the running time using the HugeInteger class. As seen in the graphs, the BigInteger running time is much smaller in relation to HugeInteger, and so it appears as though the running time is 0 for each method. This speaks to the fact that the implementation of the HugeInteger class is inefficient compared to the BigInteger class.

If given the time to improve the code, there are two things I'd change. First, to reduce the memory used for the implementation, I would store three digits per array element. This would complicate the implementation of each method but would reduce the amount of memory used. In order to reduce the running time of the program, I would modify the multiply method to be implemented through Karatsuba multiplication, which has a running time that is faster than $\theta(n^2)$.