CoE 2SI4
Lab 3/4, section L03
March 8,2016

Kevin Ziemba, ziembak, 001324999

Description of Data Structures and Algorithms

The data structure used for the HugeInteger object is a doubly linked list. The HugeInteger contains a reference to a dummy header and the last node (a useful number, not a dummy tail), an integer size holding the number of nodes linked to the header, and a Boolean displaying if the HugeInteger contains a negative number. Each node contains a reference to the next and previous node, as well as an integer.

By design, the integers held by each node in the linked list should be 4 digits or less. The choice of 4 was made so that the sum or product of these integers can still be held as an int (which holds up to 10 digits). If a number is negative, only the most significant bit will hold the negative sign, and the Boolean instance field isNegative will be marked true.

Method add starts at the least significant node for both this and h. The sum of the two nodes is computed. If the sum is greater than 4 digits (>=10000), the most significant is dropped with modulus, and the remainder sent to the node of the new HugeInteger holding the sum, and the carry on is held for the next sum. This continues for the next least significant node until every node in both numbers have been looked at, and no carry on exists. New nodes are created dynamically to accommodate each new partial sum. Negative numbers are handled by treating all numbers as positive and solving an equivalent operation.

Method subtract examines only the case which this>h. The comparison is completed with the compare method, and if the inequality fails, the subtraction is taken in reverse. Method starts at the least significant node for both this and h. The difference of the two nodes is computed. If the difference <0, the complement of the difference is sent to the node of the new HugeInteger holding the difference, and the carry on of -1 is held for the next difference. This continues for the next least significant node until every node in both numbers have been looked at, and no carry on exists. New nodes are created dynamically to accommodate each new partial sum. Negative numbers are handled by treating all numbers as positive and solving an equivalent operation.

Method multiply starts at the least significant node for both this and h. Nodes are multiplied by each other , the product of which represents a partial sum belonging to 2 different nodes of the answer. Each node is multiplied by nodes of the opposite list of equal or less significance (order of magnitude). The partial sums are stored in a list, each element corresponding to a different node of the answer. The list is processed to handle carry on. A new HugeInteger object is created and holds the values held in the list. This new object represent the product.

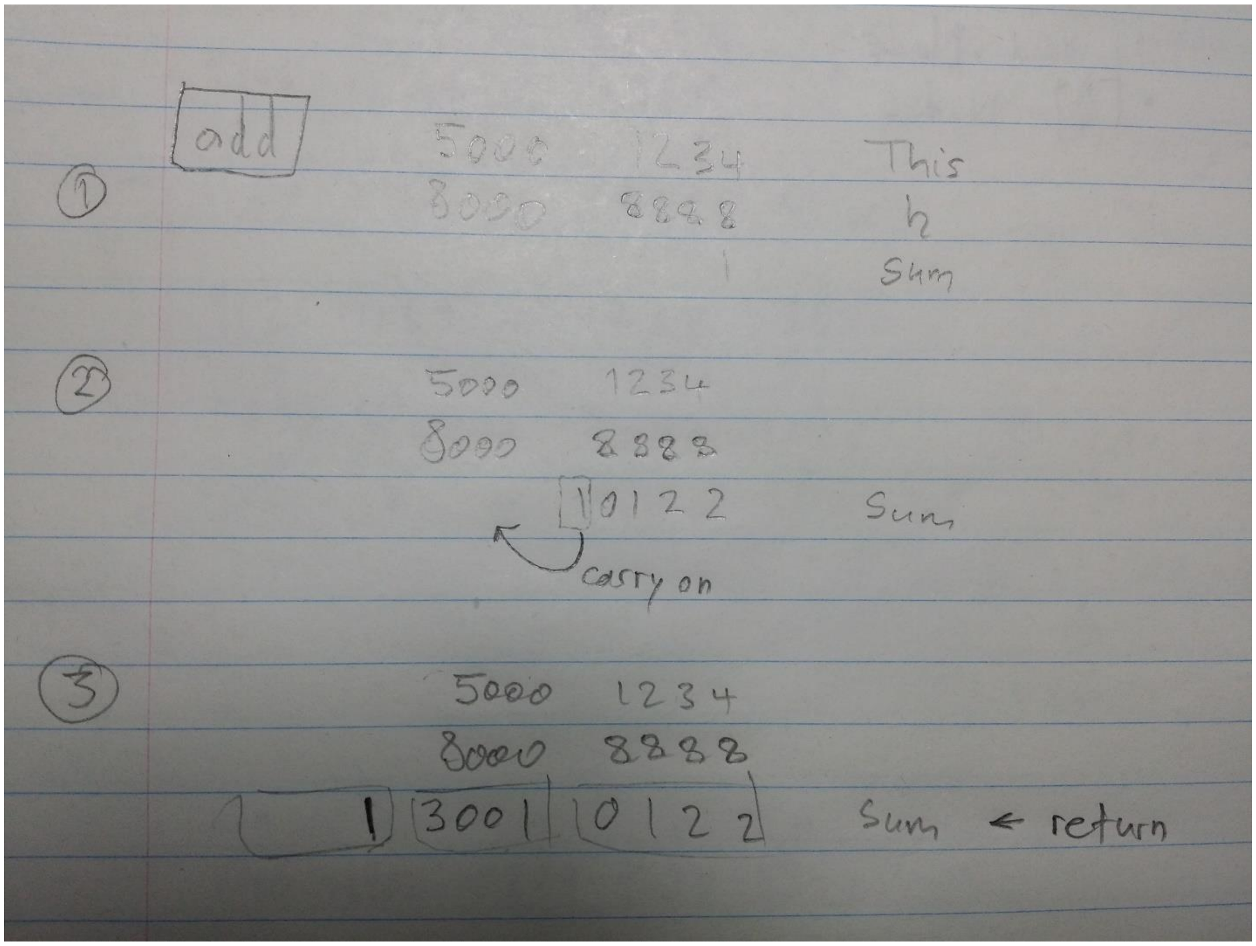
Method compare begins by running checks to see which list or both is negative and which list has the larger magnitude. Under most conditions, these checks make the method run in constant time. In the case both numbers have the same sign and magnitude (number of nodes), a for loop examines each node for an inequality. In the case the numbers are equal, all nodes are examined and a 0 is returned.

Theoretical Analysis of Running Time and Memory Requirement

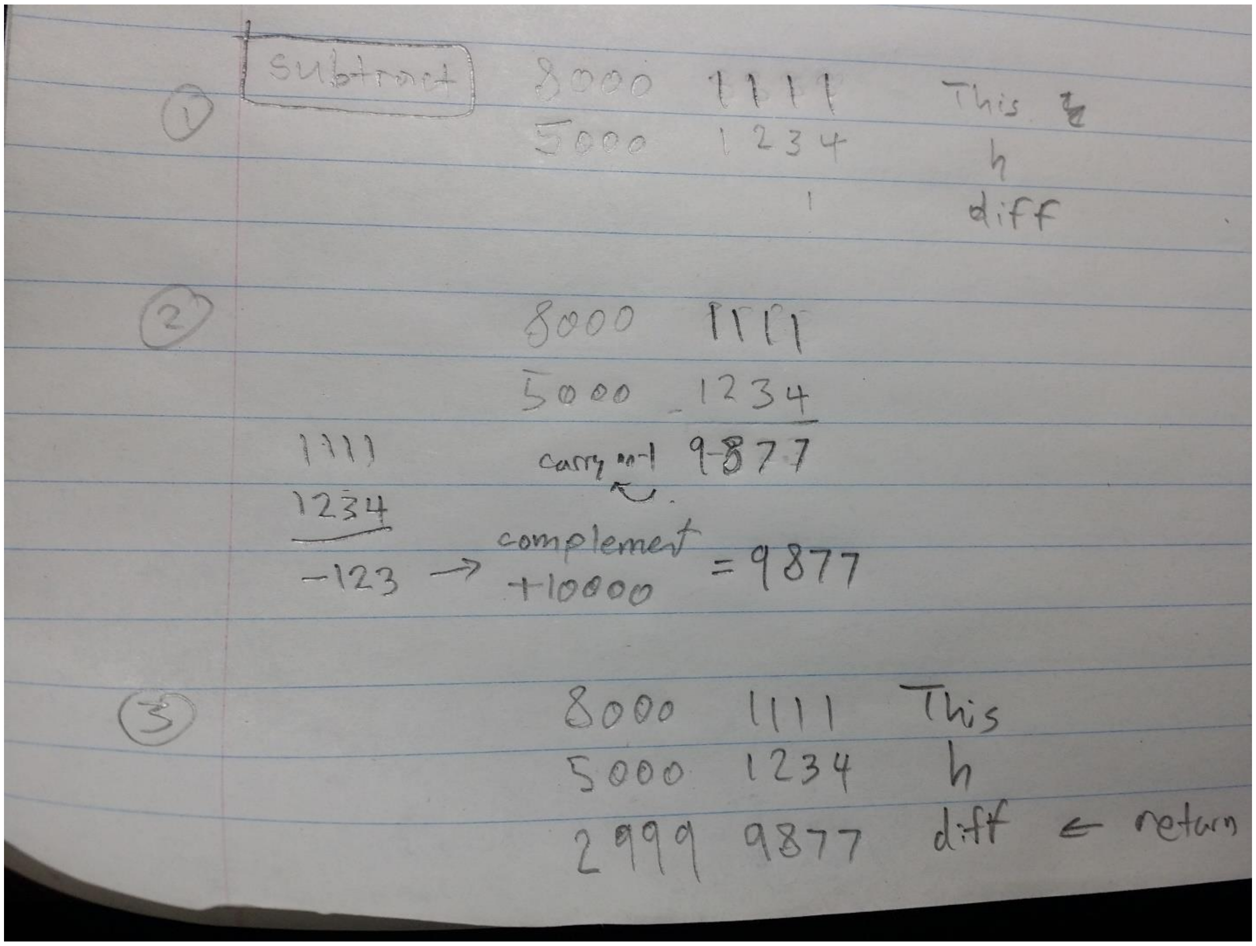
The number of bytes used in a HugeInteger depends on the number of nodes used, plus 4 bytes to hold the integer size, and 12 bytes to hold the head node. For every four digits, a new node is created, and each node holds an integer (4 bytes) and a reference to 2 more nodes (4 bytes each). So for a number of n digits.

The number of bytes used in a HugeInteger is thus 4+[floor(n/4) + 1]*12, where n = the number of digits in the number.

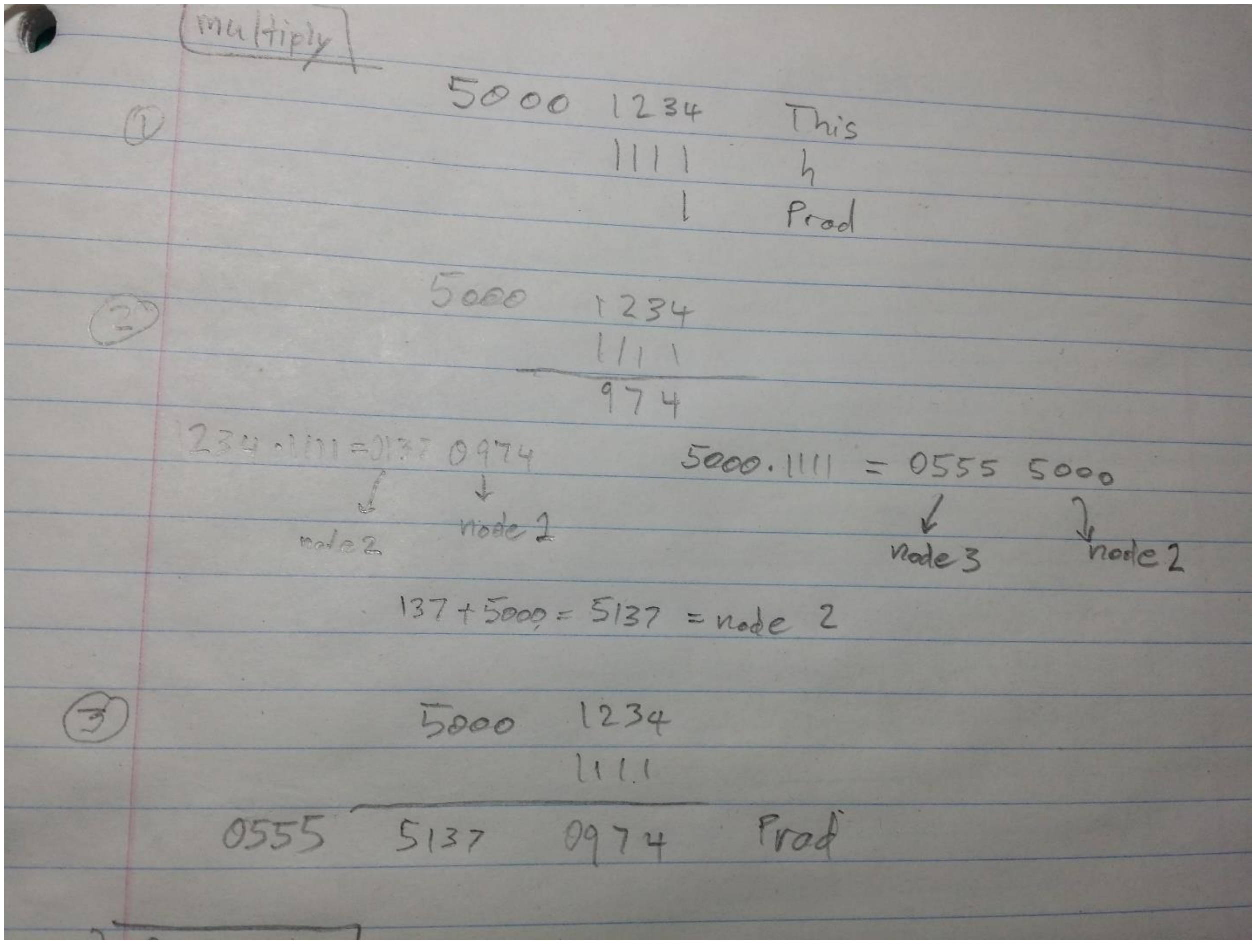
The add method runs in Θ(n) time for both average and worst cases, where n is the size of the larger in magnitude integer. Add completes some initializations and comparisons which run in constant time. The while loop runs through each node in both lists once, and inserts a number into a new node, a step which takes constant time. Memory required is Θ(n) as new nodes may need to be created to hold the sum, and the number of new nodes that have to be made dynamically also depends on the size of the HugeIntegers being added.



The subtract method runs in Θ(n) time for both average and worst cases, and memory required is Θ(n). Reasoning for this is the same as the add method. One difference is subtract calls the compare method. However, compare runs in Θ(n) so does not increase complexity.

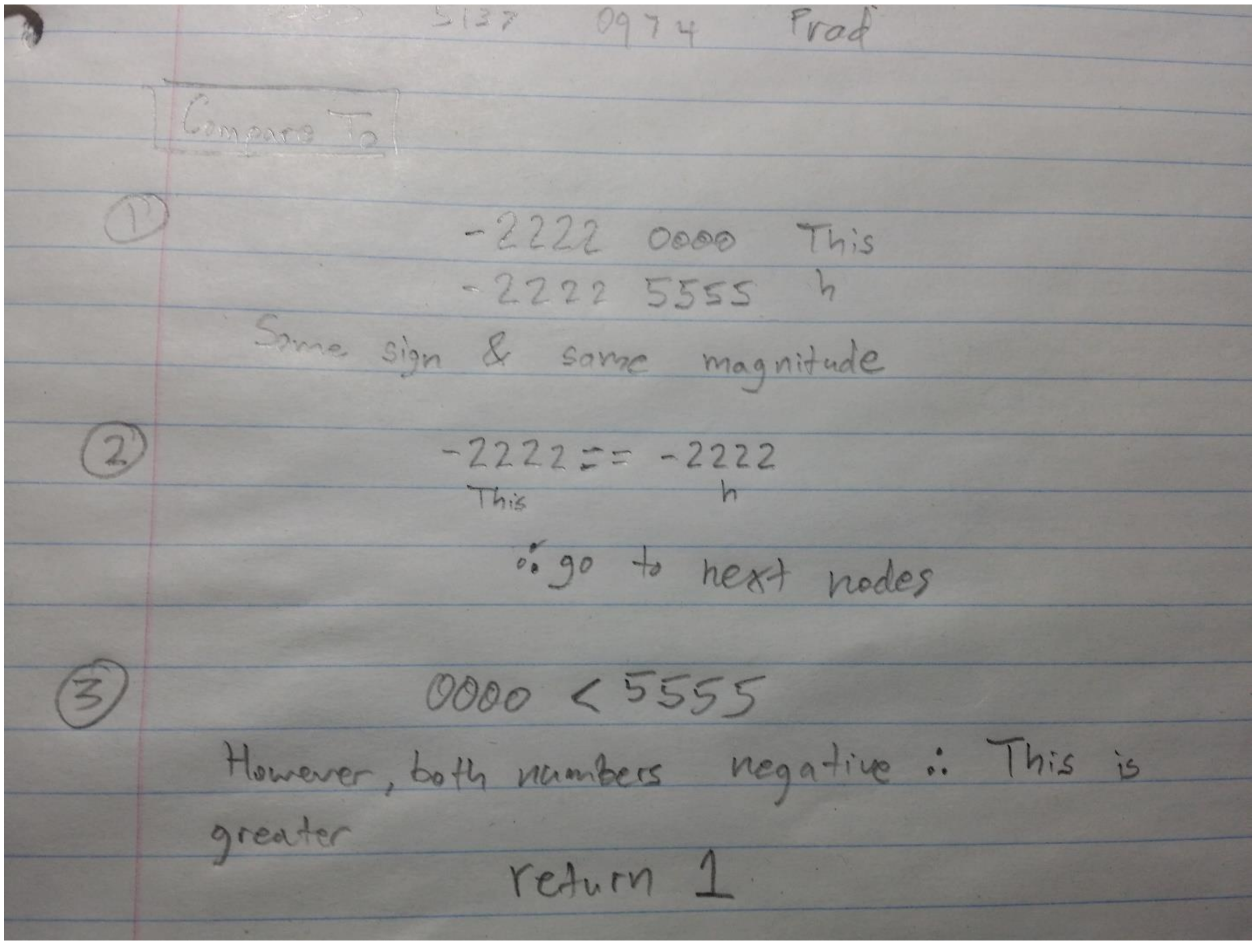


The multiply method runs in Θ(n^2) time for both average and worst cases, where n^2 is the product n1*n2 of the sizes of the integers being multiplied. Multiply performs operations on both objects, multiplying nodes in one object with nodes in the other object of equal or less significance (The most significant node follows the dummy header). Memory required is Θ(n). Multiply creates an int array of size n1+n2, and then must create roughly n1+n2-1 nodes to hold the product.



```
multiply

①            5000  1234    This
                   1111    n
                      1    Prod

②            5000  1234
                   1111
                   ----
                    974

1234·1111 = 0137  0974          5000·1111 = 0555  5000
             ↓     ↓                          ↓     ↓
           node 2  node 1                  node 3  node 2

             137 + 5000 = 5137 = node 2

③            5000  1234
                   1111
             ----------------
     0555    5137  0974    Prod
```

The compare method runs in Θ(n) time for the worst case, and Θ(1) time for the average case. The average case happens as there are several checks that occur in the method that could find the result in constant time. For example, if one number is negative and the other is positive, the positive number must be greater. When all checks fail, the worst case occurs. A loop goes through nodes to find an occurrence where one is greater than the other. If the numbers are the same, every node is checked. Memory required is Θ(1), no extra memory is required to hold values.



5137 0974 Frad

Compare To

① -2222 0000 This

-2222 5555 h

Same sign & same magnitude

② -2222 == -2222

This h

∴ go to next nodes

③ 0000 < 5555

However, both numbers negative ∴ This is greater

return 1

Test Procedure

Testing procedure will analyze combinations of the following inputs:

- Inputs of different sizes, and of the same size
- Inputs of both positive ,both negative, and mixed signs
- Inputs that will need some carry on arithmetic (ex: 5000+6000 = 11000)
- Some basic rules of the operations were satisfied (ex: a+b=b+a , a-b= -(b-a) , a*b=-(a*(-b)))
- Inputs that would cause the output to have leading zeros which would have to be removed (ex: 10003-10001 = 00002 = 2)

The HugeInteger had satisfied all the test cases. There were no test cases that could not be checked.

HugeInteger was directly compared to BigInteger by having both objects undergo equivalent operations, and then printing the results to a string, with strings from both objects side by side.

Difficulties in debugging included testing all cases to see if the resulting numbers and the size instance field matched up with expected results. Particular problems included checking if the carry on was working expectantly, checking if loops ended at their intended position, and checking all the handlers for the negative numbers.

Experimental Measurement, Comparison, Discussion

The running time for each operation was measured using the sample code given in the lab report. The value MAXNUMINTS was held at 100, the number of digits n was varied from 10,100,500,1000,5000,10000, and the value MAXRUN was varied to allow the run time for MAXRUN repetitions of the operation was greater than 500 milliseconds.

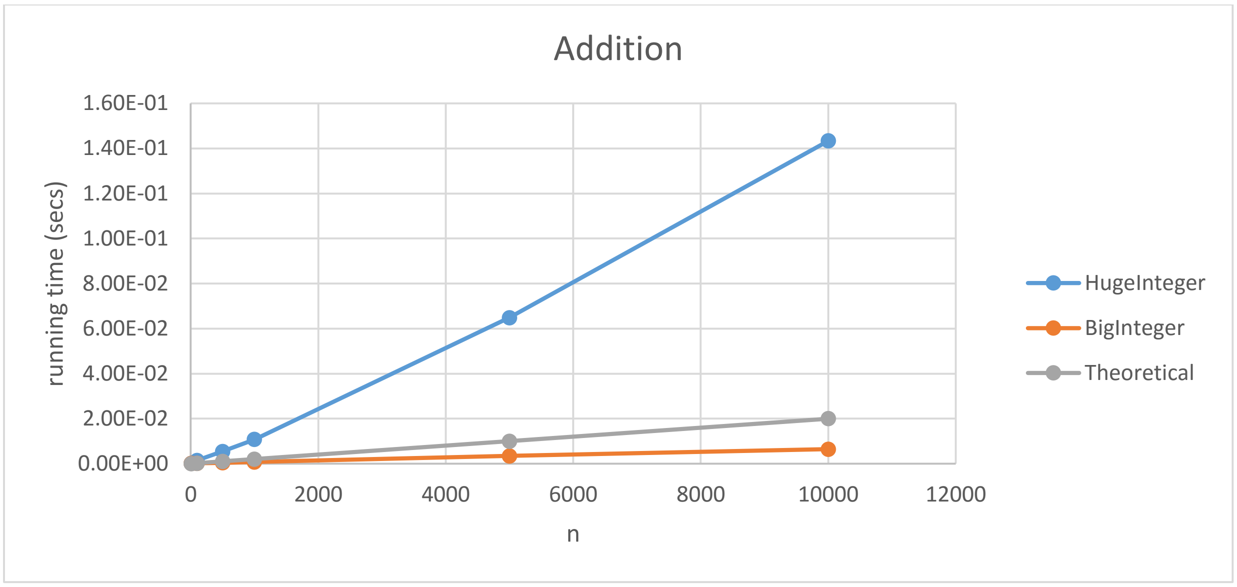
Theoretical values were calculated by plugging in n into the time complexity, and dividing by 500,000. The number 500,000 was chosen since it best matched the data, so it was a good scaling factor.

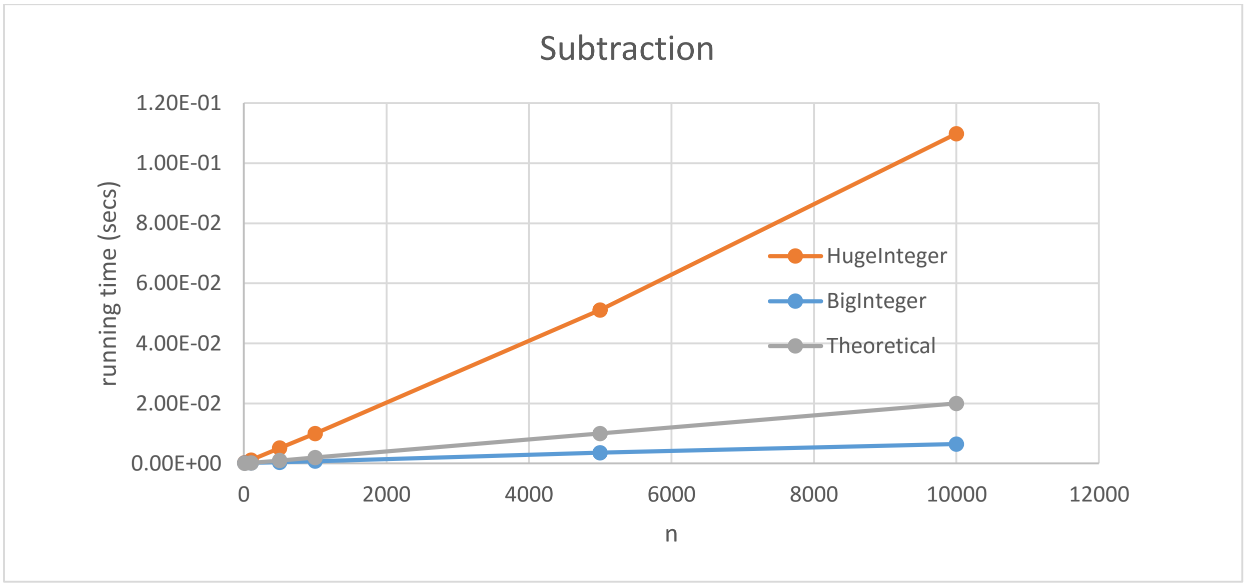
n	HugeInteger				BigInteger			
//	add	subtract	multiply	compare	add	subtract	multiply	compare
10	1.89e-4	2.53e-4	6.16e-4	1.1e-5	1.25e-4	1.25e-4	1.3e-4	1.27e-5
100	1.42e-3	1.1e-3	1.08e-2	8.85-6	1.53e-4	1.59e-4	4.56e-4	1.31e-5
500	5.42e-3	5.15e-3	2.08e-1	8.36e-6	3.66e-4	3.71e-4	5.08e-3	1.35e-5
1000	1.08e-2	9.99e-3	8.59e-1	8.93e-6	7.41e-4	7.47e-4	2.22e-2	1.36e-5
5000	6.45e-2	5.05e-2	20.8	9.07e-6	3.47e-3	3.61e-3	4.31e-1	1.37e-5
10000	1.43e-1	1.09e-1	79.2	9.24e-6	6.45e-3	6.48e-3	1.04	1.41e-5
Θ Time	n	n	n^2	1(average)	n	n	n^2	1 (average)

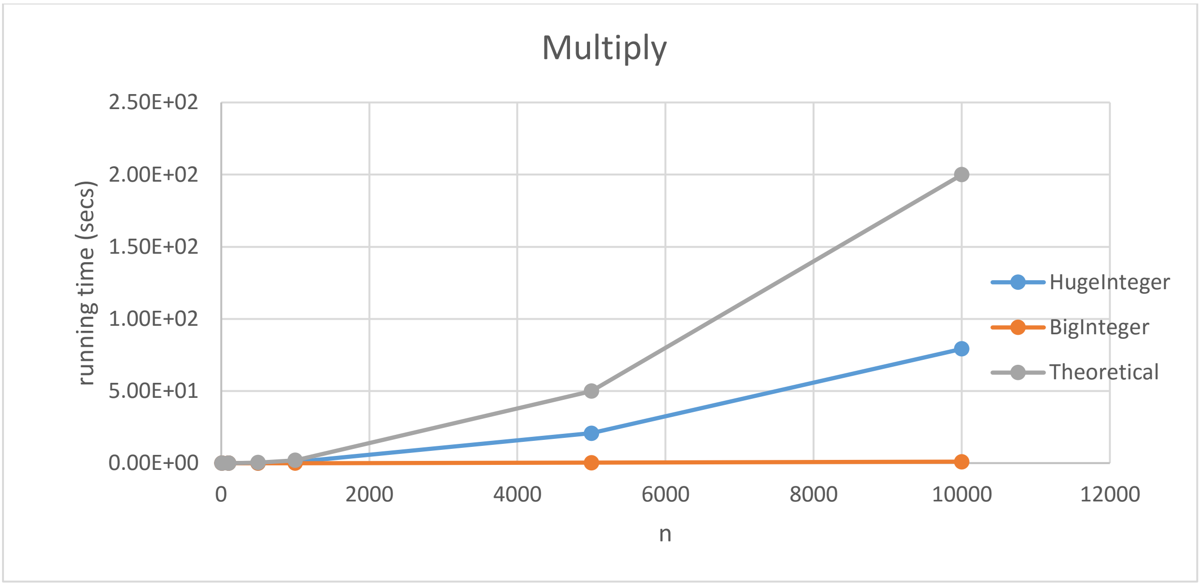
Above: Measured values using given code

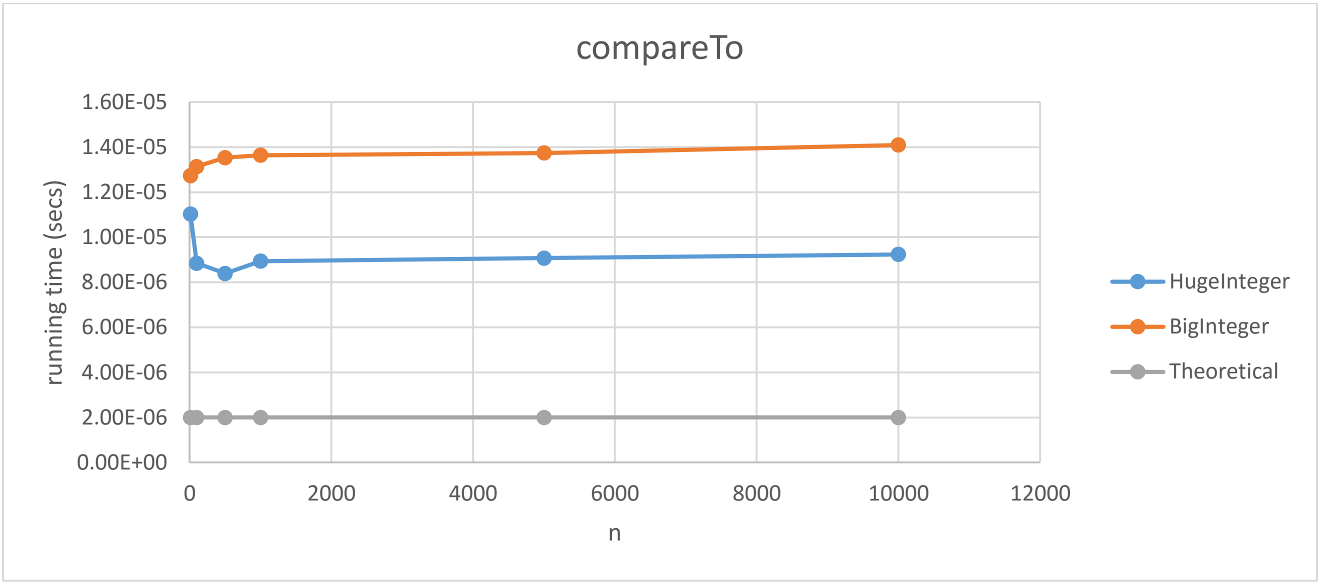
n	Θ Time Scaled			
//	add	subtract	multiply	compare
10	2e-5	2e-5	2e-4	2e-6
100	2e-4	2e-4	2e-2	2e-6
500	1e-3	1e-3	5e-1	2e-6
1000	2e-3	2e-3	2	2e-6
5000	1e-2	1e-2	50	2e-6
10000	2e-2	2e-2	200	2e-6
Θ Time	n	n	n^2	1(average)

Above: Theoretical values based on time complexitiy


Addition
running time (secs)
1.60E-01
1.40E-01
1.20E-01
1.00E-01
8.00E-02
6.00E-02
4.00E-02
2.00E-02
0.00E+00
0
2000
4000
6000
8000
10000
12000
n
HugeInteger
BigInteger
Theoretical


Subtraction
running time (secs)
1.20E-01
1.00E-01
8.00E-02
6.00E-02
4.00E-02
2.00E-02
0.00E+00
0
2000
4000
6000
8000
10000
12000
n
HugeInteger
BigInteger
Theoretical





Discussion of Results and Comparison

There exists resemblance between the theoretical and measured results. The relationship between n and running time is correct (graph has the expected shape), although all values are affected by different scaling factors. Thus the results make sense, the measured results follow the expected relationships (1,n,n^2).

In comparing HugeInteger to BigInteger, BigInteger is often quicker to compute than HugeInteger. Addition and Subtraction are quicker in BigInteger by about a factor of 1.4 . BigInteger appears to have different asymptotic behavior on method Multiply, as BigInteger is vastly more efficient for large n.

An improvement to Multiply can be made by using advanced multiplication algorithms, such as Karatsuba multiplication. This would allow multiply to act at a lower complexity than n^2. Method add and subtract can be improved on by directly handling negative numbers, instead of having to call other methods to complete the task.