Description of Data Structures and Algorithms

public HugeInteger()

This is the constructor that creates an empty object with null fields, to be declared later

public HugeInteger(int n)

If n is lesser than 1, it throws an index out of bounds exception. Otherwise, it initializes the hugeint[] field to an array of size n. Following this it loops through the array and sets each value to a random number 0-9 inclusive. Lastly negative and positive are set using another random choice between 0-1 with 0 being positive and 1 being negative for the negative field.

public HugeInteger(String val)

First it loops through all characters except the first one to verify that they are numbers (0-9) using their ASCII values. Secondly it checks if the first character is a negative sign: if true, negative is set to 1 and the size field is set to the array length minus 1 since it's not a number; if false it checks if the first character is indeed a number, and if so the size is set to array length and if not it throws an exception as mentioned before. Then it checks for the first non-zero place (if not negative) and copies the characters from the array backwards so that the lowest position in the hugeint[] array will be occupied by the lowest decimal place of the number.

public HugeInteger add()

First it checks for the longer HugeInteger and creates a new HugeInteger to store the sum with an array length equal to the longer one plus 1 (in case of end carry). This function only considered cases where both numbers had the same magnitude (pos/neg) and otherwise it would be calculated using the subtract() method since it is implicit subtraction. While adding, the carry would be the sum of the two numbers at a certain index divided by 10 while the value in the sum array would be this value mod 10 (this separates the two digits). Following the end of the smaller HugeInteger, the remainder of the larger HugeInteger's digits would simply be copied into the sum array, but the carry was still considered in case it remained from the summation. Lastly the sign of the sum would be determined on whether it was a sum of 2 negatives or 2 positives and set accordingly. It also made sure to check if the one extra digit (most significant digit) allocated for a possible end carry was filled, and if not the size would be adjusted accordingly. In the case that subtraction was used, the sign would be set to the value with the largest absolute value, which was checked by temporarily setting both this and h to positive and using the compareTo() method.

public HugeInteger subtract()

First the method goes through potential magnitude scenarios: if its + subtract – or – subtract + it is essentially addition and in this case this is routed to add(). Now the two remaining cases are – subtract – and + subtract +: for the former case it is easier to temporarily convert the numbers to positive and send through the subtract() positive case, because it leaves only one case, so this is what is done. Now the subtrahend and minuend have to be determined, and using the compareTo() method, the larger and smaller one are known (if they are equal a zero HugeInteger is returned). Now the subtraction portion is very simple: the larger number subtracts the smaller number minus the borrow, digit by digit, and if this is less than 0 then the borrow is set to 1 and 10 is added to the difference digit. Borrow will be set back to zero if the difference is greater than zero (including the subtraction of the borrow), and this continues until the size limit of the shorter HugeInteger has been reached. After this, if there is a size difference between the two HugeIntegers, the values of the minuend are copied to the result HugeInteger with the borrow value considered as well. Lastly leading zeroes are removed through a for loop and the magnitude is set based on whether this HugeInteger is larger than h HugeInteger: if true its positive if not its negative.

public HugeInteger multiply()

Firstly I created a new HugeInteger object to store the result with the size of both HugeIntegers summed together. Then I determined which HugeInteger was longer and this would be at the top and the other one would be at the bottom; if they were the same length it didn't matter and they were arbitrarily assigned. Then for the multiplication I loop through the bottom HugeInteger, then have an inner loop looping through the top HugeInteger. This way I can multiply the bottom by each element of the top. I would do the multiplication of each element of the top by the ith element of the bottom, then store that in the position of the new allocated array position ith position of the bottom array + ith position of the top array (since they were copied backwards in the constructor). Then I would store the integer division of the value into the next greatest position and the modulus in the current position. 4. For the sign of the new Hugeinteger, if both negative or positive, new sign is positive and if different then the sign is negative

public int compareTo(HugeInteger h)

This method considered multiple cases to check which HugeInteger was larger. First it would check if they were different signs, and if so it was simple to determine the larger one. If they had the same sign, first their lengths were considered: the larger one would be the greater value for positive signs and the lesser value for negatives and vice versa. If they were the same sign, it would loop through each of the digits and compare them one by one and as soon as one was greater/lesser than the same digit in the other HugeInteger, the larger/smaller value was determined. Lastly if it passed all these tests it would return 0 to signify equality.

<u>Theoretical Analysis of Running Time and Memory Requirement</u>

To store the HugeInteger requires first memory for the array of integers, which is number of bits per integer times the length of the HugeInteger. Additionally there is 2 more integers stored, which define the size and sign of this number so for n decimal digits the memory would be (n+2)*32 bits (for integers). In a plot this figure would be a straight line with a positive slope.

For add(), the amount of new memory is the 5 new local integer variables and the 1 temporary local HugeInteger value with a size of the larger HugeInteger (to determine which number is on top of the addition) and the sum HugeInteger with a size of the largest number + 1. I assumed for all cases to add 1 to the length and all variables are also declared so the memory is same for worst and average. So (n+2)+(n+2+1)+(5)*32 bits (integer). Run-time is n, because of regardless of which one is larger in size, you will always loop through the length of the smaller one, then the remaining of the longer one.

For subtract() 3 new local integer variables and 3 HugeInteger objects were declared so the additional memory usage is (3)+(3*(n+2))*32 bits. The run time is always n because you have to loop through the larger array elements always.

For multiply(), 3 new HugeInteger objects and 7 local integer variables are declared. So additional memory usage is (7)+(3*(n+2))*32 bits. The runtime is n^2 because of the double for-loop used to multiply each element in both HugeIntegers by each other.

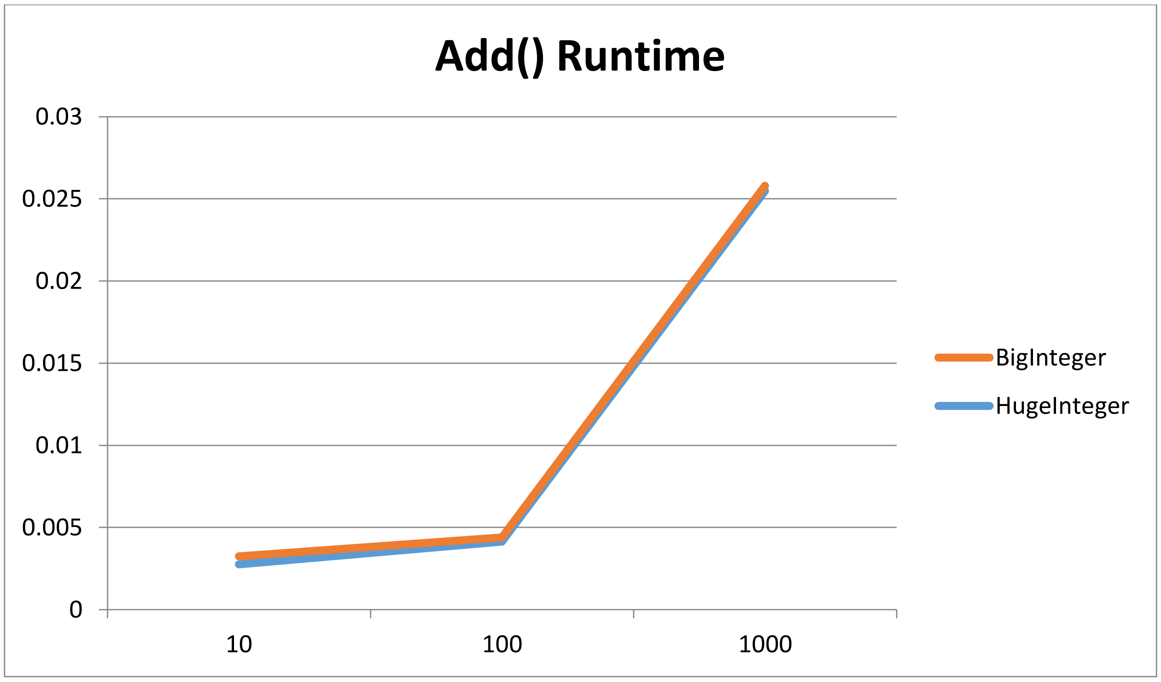
The compareTo() method does not declare any variables or create any objects so the additional memory is just 0. For the runtime, I loop through the most significant values to the lowest significant values. The worst case is if they are both equal and I would run to the very end n. The average would be n/2 which is n as well, assuming that half the time this HugeInteger is smaller than h HugeInteger and vice versa.

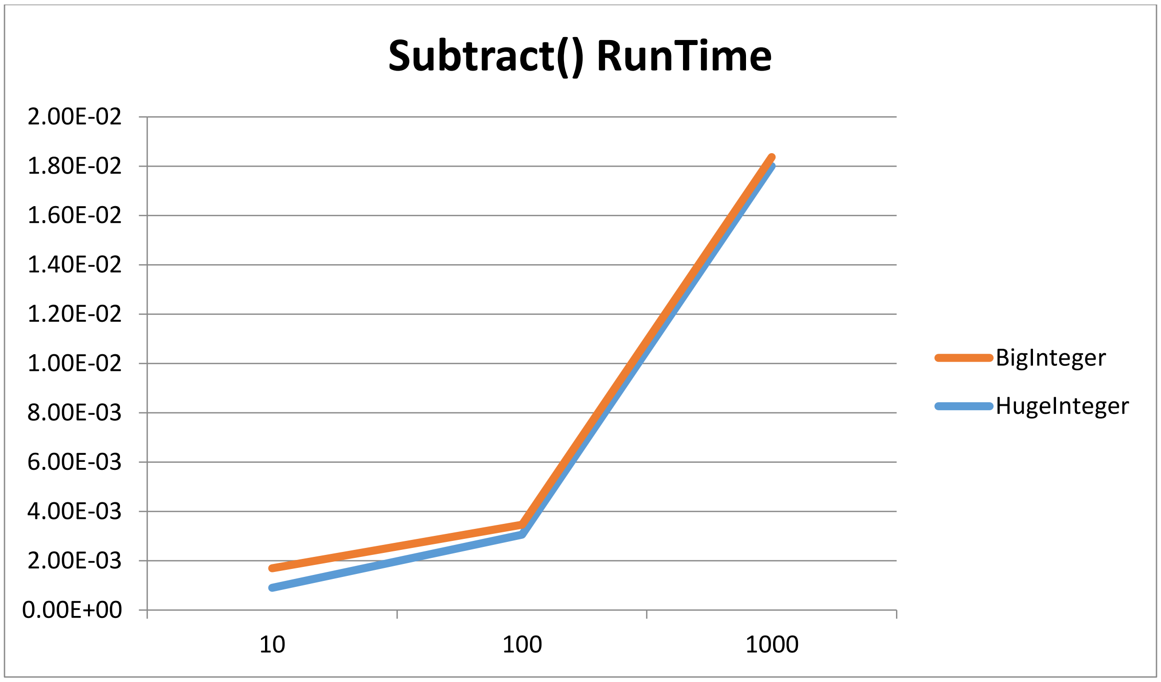
Test Procedure

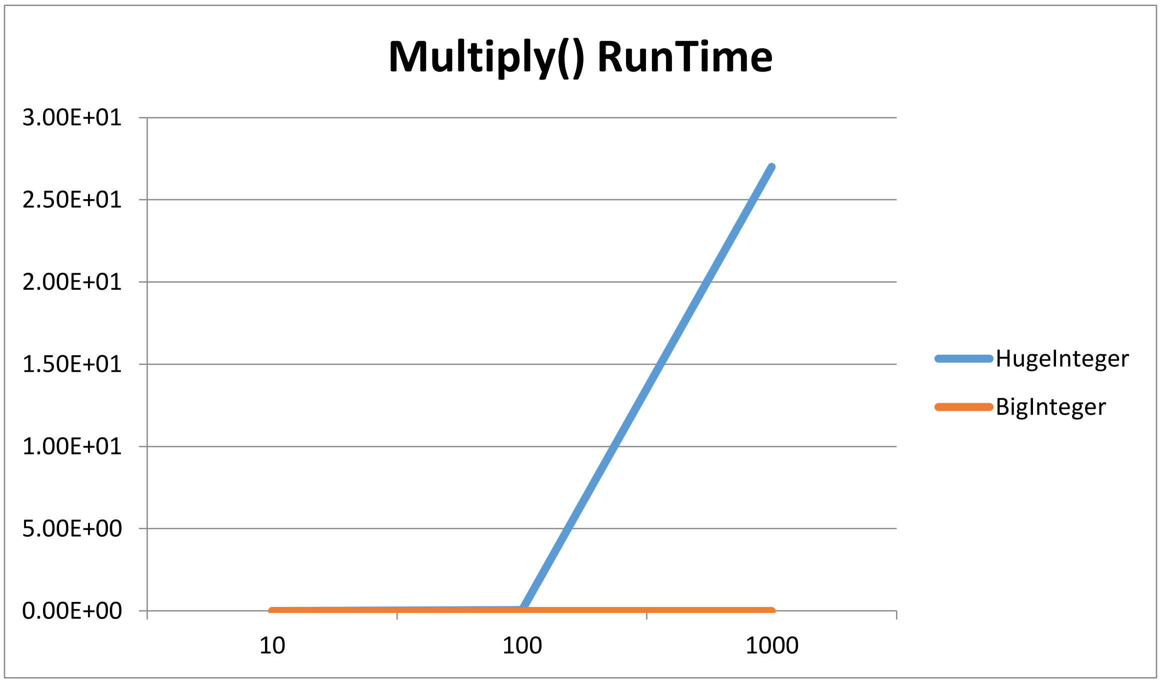
For each of the methods, there are many cases to consider: all combinations of signs (4 in total), the varying lengths, and even zero cases. This means all the methods have to test for all combinations of these variations. For example in the subtract() method one would have to test all 4 sign combinations to see if they were directed to the correct method (as 2/4 went to add()), and then varying lengths to see if the carry worked correctly, and then and equality case where they would equal zero, and finally a case where a smaller number subtracts a larger number since this is counter-intuitive to hand-written subtraction. For constructors all manners of exceptions would have to be tested as well: strings full of non-numbers, strings with a non-number in the middle, strings starting with a non-number and a non-negative, and even a legible string of all zeroes. When writing the code I had to consider all these cases in my head and adjust my code necessarily, therefore all outputs met specifications and all test cases were tested. There were no difficulties debugging code, and input that could not be thoroughly checked were operations performed on extremely large numbers (cannot be plugged into a calculator or Google) but since it worked on all cases on a smaller scale, I can reasonably confirm it would work on extremely large numbers.

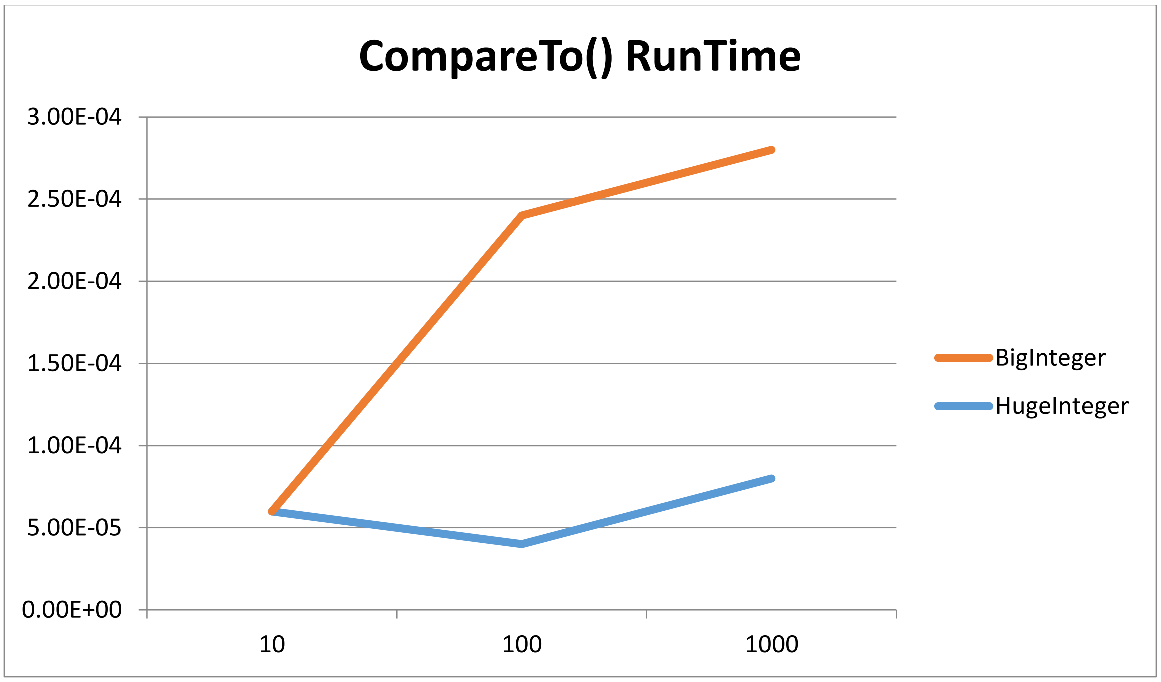
Experimental Measurement, Comparison and Discussion

Method Tested	Value of n	HugeInteger runtime	BigInteger runtime
Add()	10	0.0027600000000000007 ms	4.8E-4 ms
Add()	100	0.004140000000000003 ms	2.600000000000001E-4 ms
Add()	1000	0.025479999999999996 ms	3.2000000000000013E-4 ms
Subtract()	10	9.000000000000005E-4 ms	8.0E-4 ms
Subtract()	100	0.003060000000000002 ms	4.000000000000001E-4 ms
Subtract()	1000	0.018000000000000013 ms	3.600000000000002E-4 ms
Multiply()	10	8.600000000000005E-4 ms	0.001 ms
Multiply()	100	0.044339999999999956 ms	8.0E-4 ms
Multiply()	1000	27.334930000000564 ms	0.001 ms
CompareTo()	10	6.0E-5 ms	0.0 ms
CompareTo()	100	4.0E-5 ms	2.0E-4 ms
CompareTo()	1000	8.0E-5 ms	2.0E-4 ms




Subtract() RunTime
2.00E-02
1.80E-02
1.60E-02
1.40E-02
1.20E-02
1.00E-02
8.00E-03
6.00E-03
4.00E-03
2.00E-03
0.00E+00
10
100
1000
BigInteger
HugeInteger


Multiply() RunTime
3.00E+01
2.50E+01
2.00E+01
1.50E+01
1.00E+01
5.00E+00
0.00E+00
10
100
1000
HugeInteger
BigInteger



The runtime was measured using the code given in the Lab documentation, but modified for each of the methods. MAXRUN was 500 and MAXNUMINTS was 100 and n was changed from 10-100-1000 for each method and checked for both HugeInteger and BigInteger.

Discussion of Results and Comparison

These results match with the theoretical data because as observed on the graphs, all except multiply are linear. In the small scale they do not appear linear, but with enough data points it will look linear. Multiply is quadratic because of the large gain between n values in terms of time. It goes from 8.6E-04 to 27 ms which is characteristic of a quadratic function (large gains). Multiply is much slower than BigIntegers implementation to a degree that the line for BigIntegers runtime is not visible. Other than this all of my other methods are comparable in runtime to BigInteger with all appearing slightly faster than BigInteger, which may suggest my code is more efficient for values of n between 10 and 1000. The only problem I see is with multiply in terms of runtime because it is extremely inefficient compared to BigInteger. This may be due to a fault in the algorithm that is limited by my knowledge in mathematics. Also I would try and find a way to not declare temporary HugeInteger Objects (for top/bottom, minuend/subtrahend) and instead find a way to use the given HugeIntegers which would save additional memory usage and make my program more efficient.