Report

Labs 1, 2

Macid: alia78

Student #: 400075937

Lab section: L02

Date: April 2, 2018

Description of Data Structures and Algorithms

I chose to **represent my HugeIntegers as integer arrays**, where the index of the array corresponded to the power of 10 that the digit corresponded to. For example,

Construction of number:

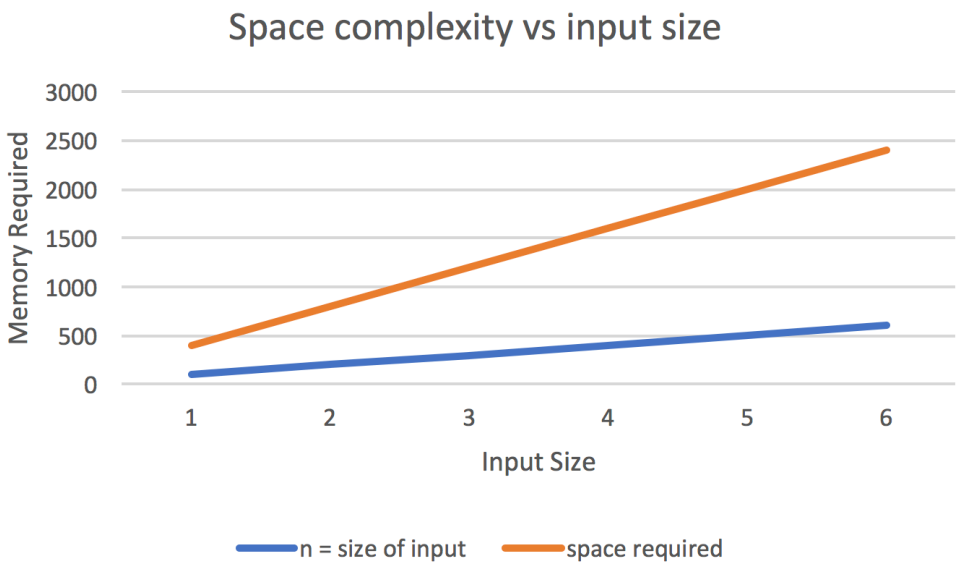
1 5 3 9

would be represented as array = {9, 3, 5, 1} i.e. the 0^{th} element (9) corresponds to 10^0 i.e. 1*9, then 10^1 * 3 = 10* 3 = 30, then 5*10^2 =500, and so on and so forth.

This would allow me to represent very large integers with each digit corresponding to a different placement in the array. Right off the bat, it is clearly self-evident that my method requires much more memory than other representations might. This is because I am allocating an entire integer of space, when I only need enough bits to represent the digits 0-9 i.e. four bits, when an integer gives me 32 bits of space. This means the memory complexity has quite a large constant.

In addition to this, instead of allocating an datatype or instance field to hold the size of my array, since an array already has a length method, I just started everything one digit/decimal place to the left. What I mean by this is I stored the sign (whether positive or negative) in the first index i.e. if an 11 was stored in the first index it was a positive number, and if a 10 was stored in the first spot it was a negative number. This decreased the space complexity of my program.

Here is a graphical representation of the space complexity of my algorithm



I shall now **describe the algorithms** I used to achieve the methods that I have in my source code. Firstly, for the comparison method, I checked both sign and number of digits primarily, as that was where I felt I could save the most time. I.e. if one HugeInteger has more digits and it's also negative, it MUST necessarily be smaller than the other HugeInteger because there is no way for it to be bigger. If the other number was also negative and had less digits, it would be smaller, or if the other number was positive then obviously it would be smaller. I then checked the digits to look for the first digit (i.e. the digit with the highest power) that was different from the other integer. If it was greater than the other integer, then it meant that the integer was greater. I only did this after ensuring that everything else was the same, so that I could save time. To implement my addition method, I utilized a number of the same principles. I would check to see the signs of the numbers so that if possible, I could pass along the work to the subtraction method. I.e. if I wanted to add a negative number to a positive number, then I could just subtracted that number instead. This would allow me to save time overall, instead of repeating operations needlessly. For the addition method, I split my method into a number of different cases, looking for which integer had more digits. Then I went through and added each digit individually, until the integer with fewer digits had been exhausted. I then added an end carry to the other digit and kept adding. This same algorithm but in reverse was what I used for subtraction. I recorded the number of borrows instead of recording the number of carries. Using this method, I was able to subtract each digit one by one. Finally, my multiplication method was by far the most difficult to come up with an algorithm for. I at first considered using a regular multiplication method in which I would mimic how people multiply their numbers by hand. Thereby, I would multiply each number by the first digits place, then do it for the second digits place, and so on and so forth until I had reached the end of the number. This would result in me achieving a quadratic time algorithm, which is what I ended up choosing in the interest of time. I did not choose to use the karat-suba algorithm, which would have allowed me to achieve a sub quadratic multiplication algorithm. In my multiplication algorithm, I first checked the signs of the two integers to be multiplied. Then I accordingly set the sign of the outputted integer, i.e. negative times negative makes a positive. Then, I found which number would be smaller. Multiplication only needs to run for whichever number is smaller. Then, I iterated through that number and multiplied it's least significant bit by each digit in the larger number. Then I stored that number and did the same thing for the next one, keeping in mind to multiply it by ten since it was one larger. This process would continue (along with any carries needed) and eventually the correct answer would be returned.

Theoretical Analysis of Running Time and Memory Requirement

Amount of memory (in bytes) of storing my HugeIntegers is quite simply theoretically calculated. If we assume that the array that we have calculated is exactly the right size for the number of digits that have to be stored i.e. there is no excess empty spots in the array, then each array position holds an integer which requires four (4) bytes. Thus, the amount of space required by the integer of n digits would be n*4 bytes. I.e. a HugeInteger with 100 digits would require 400 bytes, which is quite a bit of memory.

The **theoretical analysis** of run time is easily achieved. I shall do this with respect to each method separately, as per the requirements.

- Beginning with the constructor, this has a space complexity of Big theta n. As the size of the integer increases, so too does the memory required of the constructor, in a linear fashion. This makes sense because there is only one digit represented per element of the array. The time complexity is constant.
- The addition method has a constant space complexity as you only need to hold a few extra integers i.e. the carries. The time complexity is linear, as it will grow by 2*n, with n being the number of digits in each HugeInteger. This is because I am adding each element one by one.
- The subtraction method has a constant space complexity, for the same reasons as the addition method. There is also a linear time complexity of big theta n, as I am again subtracting each integer from each integer in that array element. This means that it will grow linearly with the number of digits in the array.
- The comparison method is worst case linear as well. There are certain times when the comparison can be made very quickly i.e. when one number is negative and the other is positive, or when the number of digits differs between numbers. But there are also certain times when this comparison requires that we check every digit i.e. when the two numbers only differ by one. This means that the time required to compare their sizes will grow linearly with the number of digits. Space complexity is constant.
- My multiplication method is not a sub-quadratic multiplication. What this means is that my multiplication method takes more than quadratic time complexity, when considered in the asymptotic case (worst-case i.e. big theta). This is further discussed above where I fully explain how I implemented the multiplication algorithm. The space complexity is constant because I only have to hold the carry, which is not dependent on the size of the inputs.

Test Procedure

In order to test these theoretical times, I used a test class that utilized the Java start and stop methods to measure time in milliseconds and then subtracted their difference. This worked for every single one of my methods; all I had to do was specific start and end times in my test classes. I tested a number of inputs of different ranges in order to test my software. Namely, I tried most the general cases for the addition, subtraction, comparison and multiplication cases that I could anticipate. I then also not only used different cases in terms of signs (positive and negative signs), but I also used different numbers of integers. I started with trivial cases such as when n (number of digits) was below around 10 digits. These were numbers in the billions and thus should not have taken long to multiply, compare, add or subtract (relatively speaking).

There are a number of **possible cases** that can arise. For example, For the subtraction method, we can try to subtract two positive numbers, or we can subtract a positive from a negative, or we can subtract a negative from a negative, or we can subtract a negative number from a

positive number. In each of these cases, there are simplifications that can be made and achieved if we are economical with our method calls. There are also a number of possible cases that can arise when we test the multiplication method. I.e. when we're multiplying two values, they can either be positive or negative, so this means that we may be multiplying two negatives, one positive one negative, or two positives. In these cases, though the algorithm for multiplication would be the same, we would have to adjust the final sign of the multiplied number (i.e., the result) in order to reflect the initial conditions of the multiplication. These cases can also arrive when comparing numbers as well. As outlined in my description for my comparison algorithm, there may be times in which you can drastically reduce the number of comparisons and calculations you have to make if you first consider all the base cases that can possibly be achieved by your program. This is also useful later on when deciding to test the robustness of your program at different endpoints.

Experimental Measurement, Comparison and Discussion

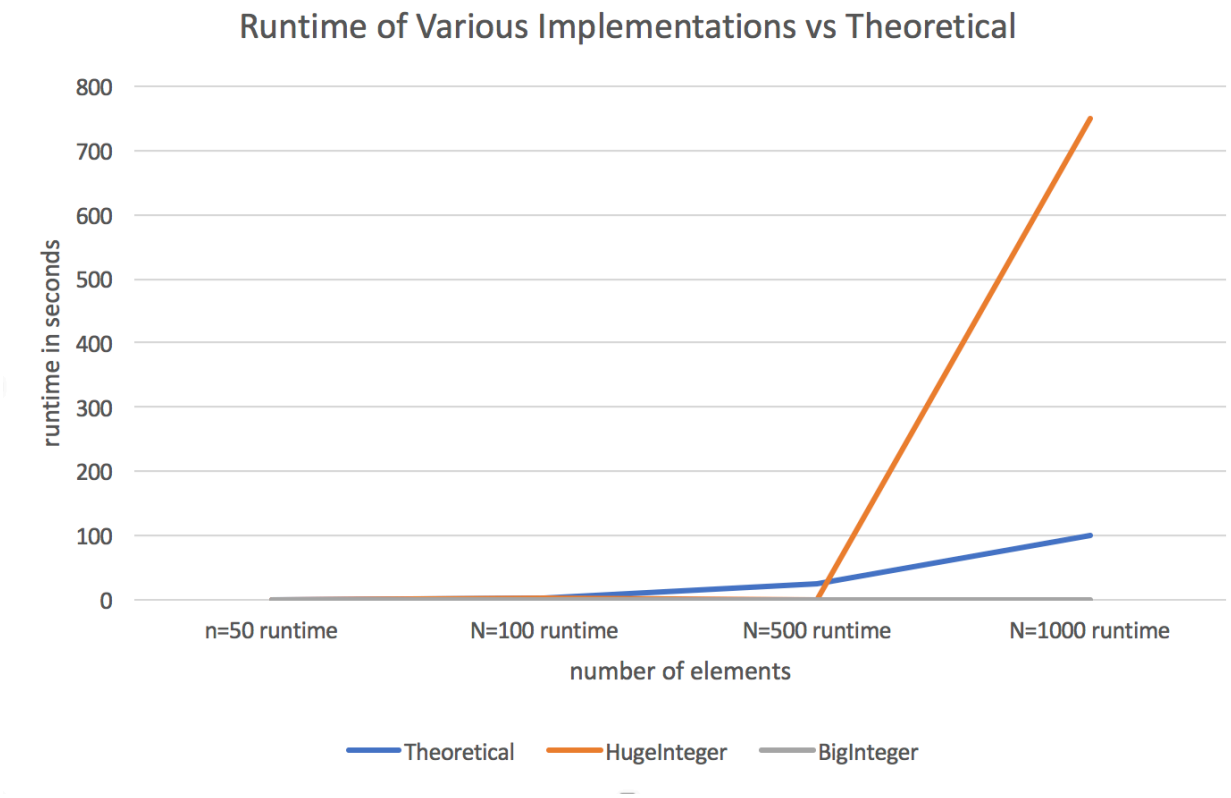
After outlining my test procedure, as shown above, I decided to **actually measure** these methods and their runtime. These are my results. As you can see, I used different parameters at different points of time in my program in order to obtain these results, and then I took system times and subtracted them from the beginning to the end of each of these methods. I then averaged them over the number of iterations that I did to arrive at the numbers that I have below.

My tables of different runtimes of different implementations are shown below.

Plots of my different runtimes can be found below. I plotted these using Microsoft excel. I took the start and end times and subtracted them to get the time that the method had ran for, then averaged these over the number of times I ran the method. You can see the Figures below.

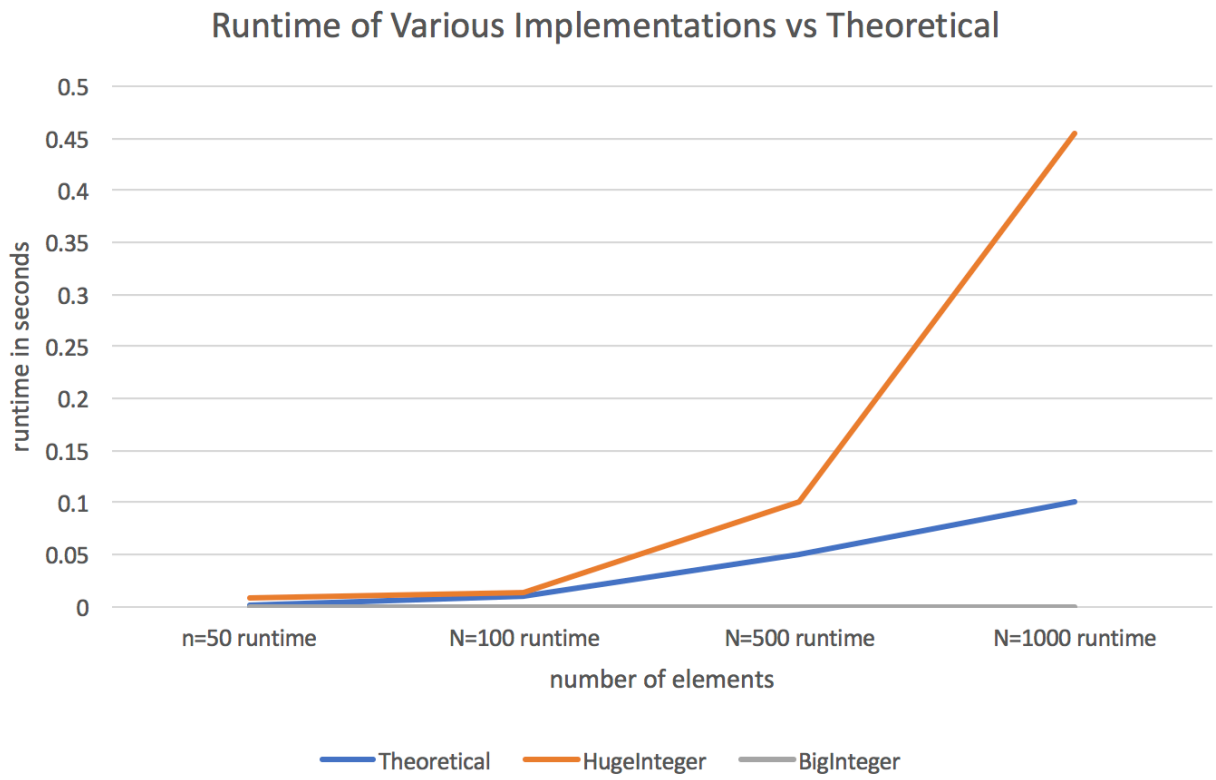
Multiply: Big theta n^2

Class:	n=50 runtime	N=100 runtime	N=500 runtime	N=1000 runtime
Theoretical	1.00E-01	1.00E+00	2.50E+01	1.00E+02
HugeInteger	8.73E-02	3.50E+00	10.01E+0.1	7.50E+02
BigInteger	3.96E-05	7.00E-05	2.37E-04	5.20E-04



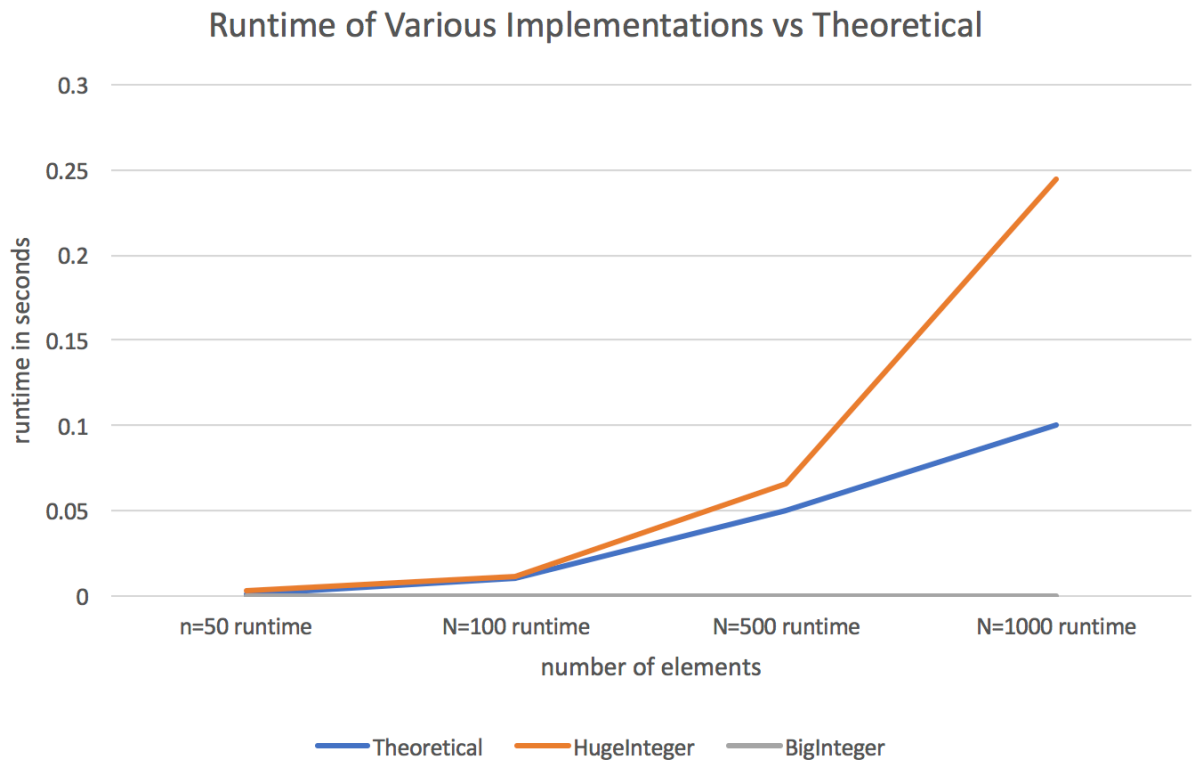
add: big theta n

Class:	n=50 runtime	N=100 runtime	N=500 runtime	N=1000 runtime
Theoretical	1.00E-03	1.00E-02	5.00E-02	1.00E-01
HugeInteger	8.73E-03	1.35E-02	10.01E-02	4.55E-01
BigInteger	3.96E-05	4.19E-05	4.90E-05	6.7E-05



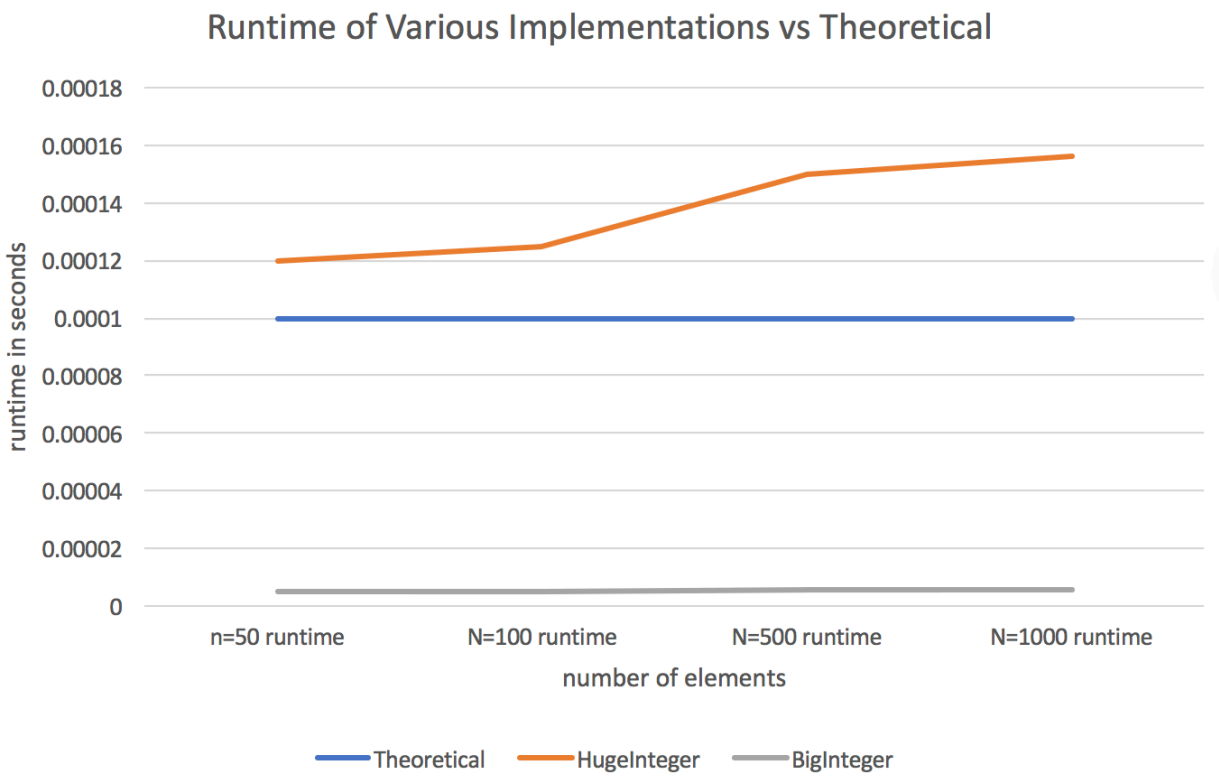
Subtract: big theta n

Class:	n=50 runtime	N=100 runtime	N=500 runtime	N=1000 runtime
Theoretical	1.00E-03	1.00E-02	5.00E-02	1.00E-01
HugeInteger	2.60E-03	1.10E-02	6.54E-02	2.45E-01
BigInteger	3.16E-05	4.00E-05	5.45E-05	6.7E-05

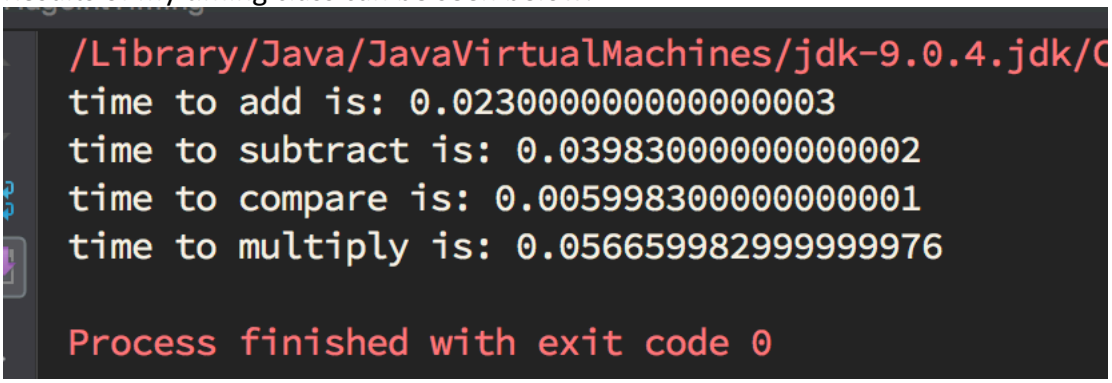


compareto: big theta n worst case, average case constant time

Class:	n=50 runtime	N=100 runtime	N=500 runtime	N=1000 runtime
Theoretical	1.00E-04	1.00E-04	1.00E-04	1.00E-04
HugeInteger	1.20E-04	1.25E-04	1.50E-04	1.56E-04
BigInteger	4.97E-06	5.11E-06	5.24E-06	5.50E-06



Results of my timing class can be seen below:



Discussion of Results and Comparison

My theoretical calculations agreed with my experimental and measured results, as documented above through my adherence to a specific testing protocol (also documented above). However, my values were not exactly the same as the theoretical predicted them to be, likely due to differences in the runtime environment of the computer or specific test cases that would increase or decrease the average running time of a program. The theoretical is only supposed to be an asymptotic analysis, and thus I am not overtly concerned with the exact value of the runtime, rather I am concerned with the shape of the graphs. My experimental

results thus make sense, as though they are not exactly what I expect, they follow the general asymptotic trend.

My implementation of the **HugeInteger class is much slower than the BigInteger** class used by Java. Specfically speaking, my runtime for the addition method took 1.35E-02 seconds for the addition method, and then when I used the exact same test case with the Big Integer class, it took 4.19e-05 seconds only. Considering my subtraction method, I saw a similar case where it took 1.10E-02 seconds and the big integer class took 4.00E-05 seconds. Finally, with my multiplication method, my method too 3.50E+00 seconds and the big integer class took 7.00E-05 seconds. As you can see, my method is much slower than BigInteger's implementation. This might be due to a number of factors and improvements, discussed below.

In order to **improve upon my program**, I could perhaps use my memory more sparingly. There are many instances where I allocate multiple temporary variables, which is unnecessary as I could reuse variables that I have already allocated memory for. This would in addition help with time complexity, as although those constants do not increase the run time asymptotically, they are relatively expensive operations (i.e. allocating a whole new temporary HugeInteger using my constructor in my subtraction method). Additionally, I could have perhaps taken a closer look at the exact karat-suba multiplication theory and decided upon a method of splitting the integer such that my recursive calls would have been more efficient in certain cases i.e. when the integer is of even digits, etc. Another improvement that I could have made to my program is to make an array holding short integers instead of full integers. This would have drastically reduced the amount of memory that my program takes, but would have made no difference asymptotically speaking as the memory complexity would still grow linearly with time.

Also, I could have tried to implement a sub quadratic multiplication algorithm. Since mine contains two nested for loops, it is quite inefficient, and is quadratic on the size of the input integers passed to it. A subquadratic algorithm would have greatly increased my programs speed for large inputs to the multiplication method.