Darren Chang
1406398
changd

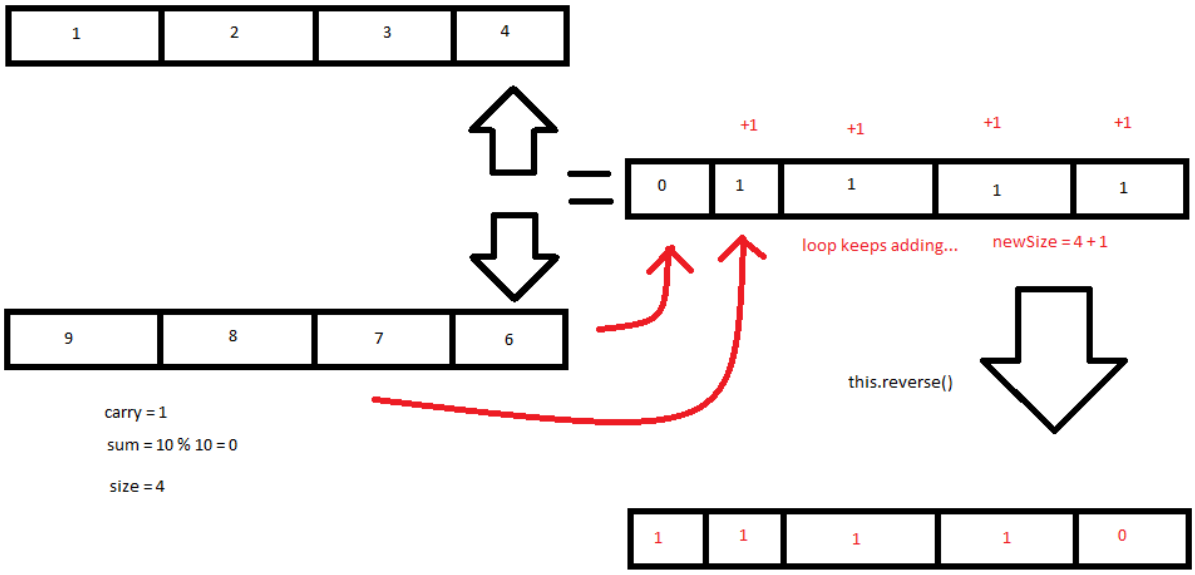
2SI4 Lab 3 & 4 Report

Description of functions:

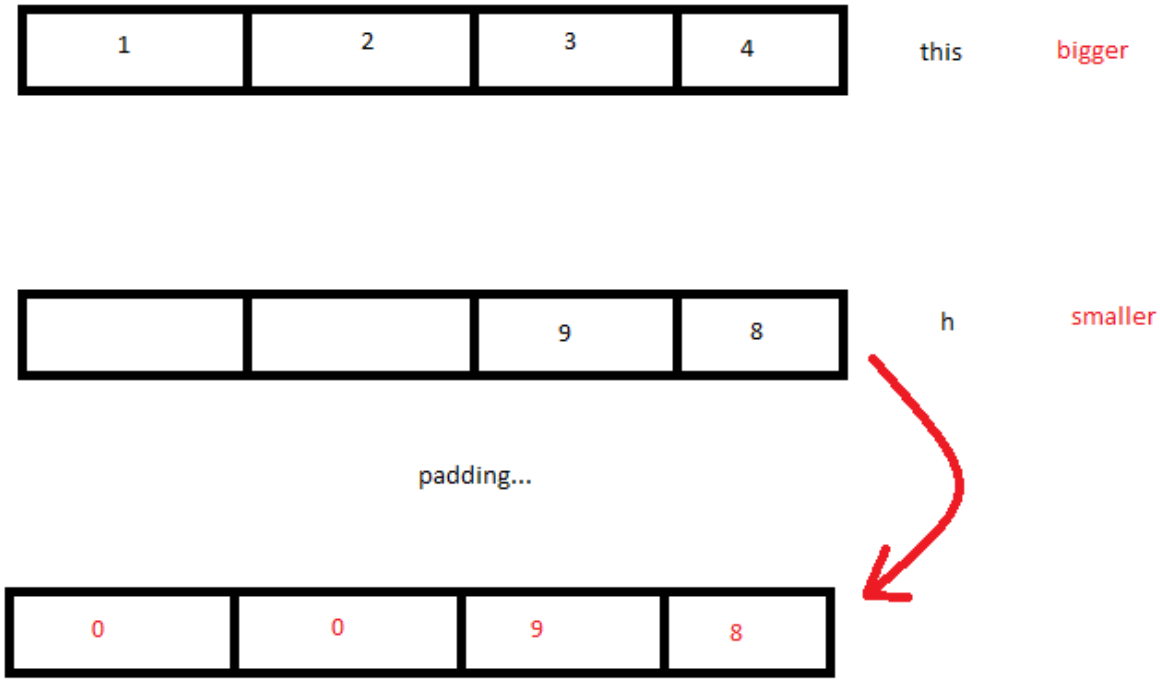
public HugeInteger (String val) -> This constructor checks for valid string inputs and creates an array with the same size as the string. A loop is used to insert the individual characters into the array. There are two private variables, one to store the array and one to keep track of the size.

public HugeInteger(int n) throws Illegal ArgumentException -> This constructor checks to make sure that n is greater than 1. If not, it will throw an exception. If n is a valid number, then a new array will be made with the size equal to n elements. A loop is created to populate the array with random numbers ranging from 0-9 using the Math.random() function.

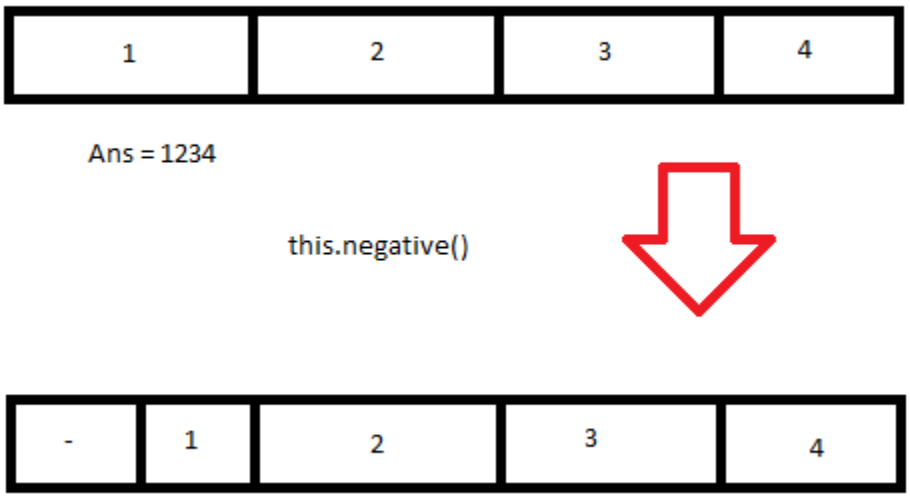
public HugeInteger add (HugeInteger h) -> This function first checks three cases, if both numbers are negative, if this number is negative or if h is negative. If none of these checks are satisfied, then both numbers are assumed to be positive. If both numbers are positive, then the function will proceed to check whether this is bigger or if h is bigger in terms of digits. If they are the same, then, a loop is used to add each element starting from the last element[size-1].The resulting additions are stored in a new array. If the resulting sums of the individual additions are greater than 9, then a carry is stored in a temporary variable and added to the next addition. The sum is mod by 10 to store the sum but not the carry. This is done until the loop ends. The result is then reversed using a method called reverse. Then that result is returned. If the numbers this and h are of different sizes, then the smaller number will be padded with zeros in front and the same algorithm is used.



Darren Chang
1406398
changd

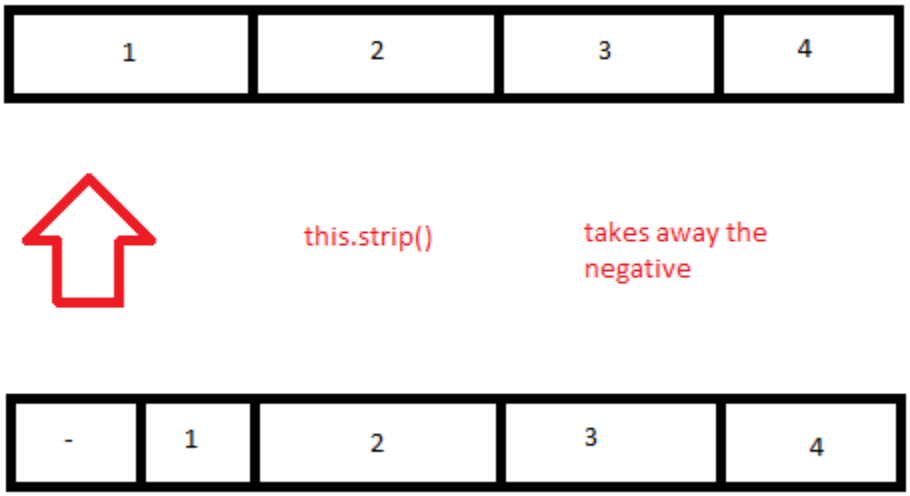


When the numbers are both negative, the same addition algorithm with be used, but the result will add a negative sign in front using the function negative().

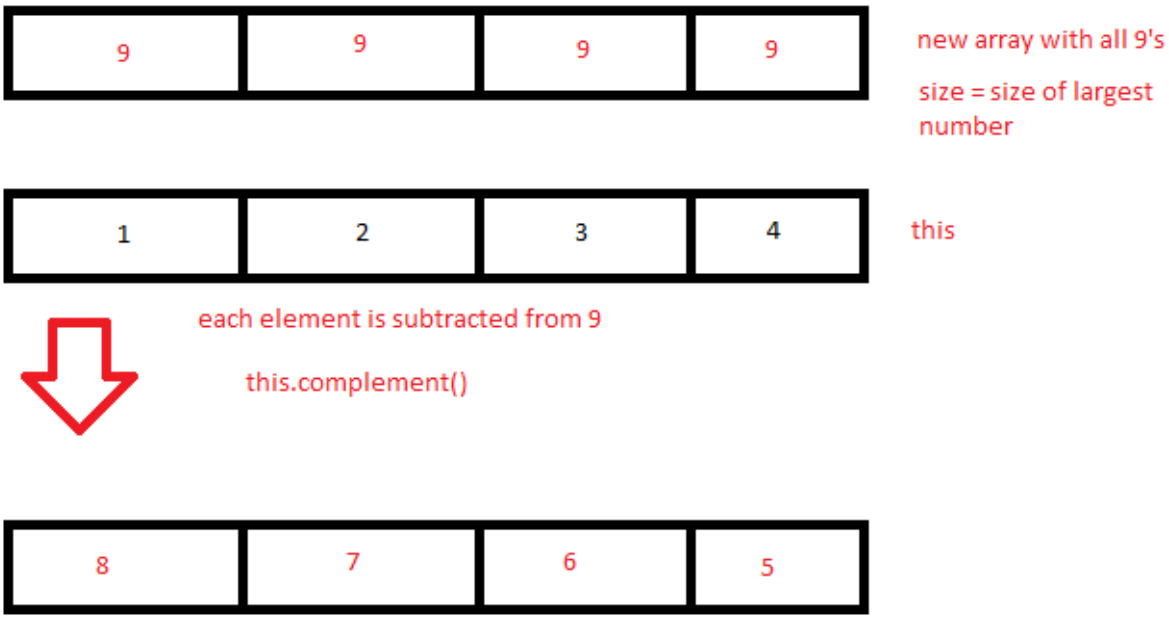


If one of the numbers is negative, then the add function will first strip the negative by using the strip() function then check to see which number is bigger using the compareTo() function.

Darren Chang
1406398
changd



The smaller number will be complemented using the complement() function and then added to the bigger number. The size of the 9's array will be determined by the size of the larger number.



Once completed, the same addition algorithm is used to add the larger number with the complemented smaller number. If the larger number had the negative sign, then the result will get tagged with a negative sign using the function negative(). If the larger number is positive, then the result will be added without a negative sign.

Darren Chang
1406398
changd

public HugeInteger subtraction (HugeInteger h) -> This function will call on the add function to the subtraction. This function checks three cases, if both numbers are negative, if one of the numbers is negative, or if both numbers are positive. When both numbers are positive, a negative number will be tagged on to the smaller number via the negative() function. Then, both numbers are added via the add() function, this way, a subtraction will be completed. If both numbers are negative, then the second number is stripped of the negative using the function strip(). Both numbers are both added via the add() method. The add() method will essentially add the correct negatives. If the first number is positive and the second is negative, then the negative sign will be stripped using the function strip() and the add function will be called. If the first number is negative and the second is positive, then the negative sign will be added to the second number using the function negative() and the add function will be called. All cases are then covered.

public HugeInteger multiply (HugeInteger h) -> This function uses classic grade school multiplication. First a check to see which numbers were negative. If both were positive or both negative a flag would stay false. If either one were negative then the flag would be true. This indicates whether the final answer should be a negative or not. Then a check to see which array was bigger, as that will determine which to loop through first. After that is determined, loop through the bottom Huge Integer, then have an inner loop looping through the top Huge Integer. Then multiply the bottom by each element of the top. 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 of length. Then store the integer division of the value into one variable, while storing the modulus in another, adding them together at the end. Finally, once the looping is done, add the final carry out back into the final running total. Then create a new object and then return it.

public int compareTo(HugeInteger h) -> This function would first check for the trivial cases where if this number were negative and the h one was positive, it would return the -1 for example. If they were both negative or both positive, then it would then check for the array size to see which of the arrays were bigger. It would return 1 if this.size were bigger than h.size for example. Then it comes to the case where both numbers are positive and negative with the arrays being the same size. In this case, a loop will be formed comparing each individual element starting from the 0th to nth element. If one element were bigger, then the function would immediately stop and return. If all the elements are the same, then both the numbers are the same and the function will return 0.

public String toString() -> Checks if there is a valid input, if so loop through each index of the array and create string by appending each index together. Finally, once done, return it.

Running Time and Memory Requirement Analysis:

Using an array implementation, each array element takes up the amount of bits of an integer. It takes 2 times the amount of bits for the sign and size when implementing Huge Integer. So for n decimal digits the memory would be (n+2) * bits of and integer (32 bits) or (n + 2) * (4 bytes).

Darren Chang
1406398
changd

Addition:
For the addition function, the amount of new memory is the 10 new local integer variables, plus the length of the largest number + 1. For addition the average case and the worst case would be the same. Therefore, it would be, (10+ n + 1) * (4 bytes) = (11 + n) * (4 bytes).
The run-time of addition is big theta of n, because regardless of which one is longer, you always completely loop though the bigger one completely, hence looping through the smaller one completely as well.

Subtraction:
For the subtraction function, there is no new local variables meaning that the memory would be the same as the addition function because this function calls on the addition function. The run-time for subtraction is big theta of n, because regardless of which one is longer, you always completely loop through the bigger one completely, hence looping through the smaller once completely as well. Basically the same as addition.

Multiplication:
For the multiplication function, a new Huge Integer object is created and 7 local integer type variables. Therefore, (n + 7) * (4 bytes) is the amount of memory used. The running time for multiplication is big theta n^2. This is because it's looped through each element of the lower array and multiply it with a single element of the top array. Since both of the loops are independent of each other, you can multiply the runtime of the lower array by the top array. Therefore, n * n = n^2.

Compare:
For the compare function, 3 local variables are created, so the memory is just a constant.
For the runtime, it's loop through the most significant vales to the lowest significant vales. Hence, the worst case if they are both equal is big theta of n. However, for the average case, it would be n/2 times assuming that half the time it is bigger than h and half the time it is less than h.

Test Procedure:

When designing the test procedure, all test cases and sub-cases for each of the functions. This included test cases for both positive and positive, negative and positive, and negative and negative arrays. Then for each of those cases you also had to consider when THIS object is greater, less, or equal length, this was tested using the compareTo () functions. Furthermore, you also have to test the zero case, in which the add, subtract, and multiply all have inputs of zero. The biggest difficulty in debugging the code was that the functions would work for majority of the larger number test cases, but failed when using smaller numbers like 99, 1 and 0. This is because my logic for using the carries wasn't implemented correctly. Possible cases could be error cases where a number isn't given or where a first value that is not 0 or 1 is given. To make sure that these cases work exceptions need to be thrown if any of these cases are inputted. To make sure that the proper inputs are working the test procedure above will need to be used. The outputs for my code did not meet the specifications. Only about half of the designated outputs for the four functions worked while the other half gave wrong answers. To test the HugeInteger class all the 48 possibilities need to be tested, where both numbers are negative, both are positive, the first is positive and the second is negative and lastly the first being negative and the second being positive. For each of these cases, three more cases need to be

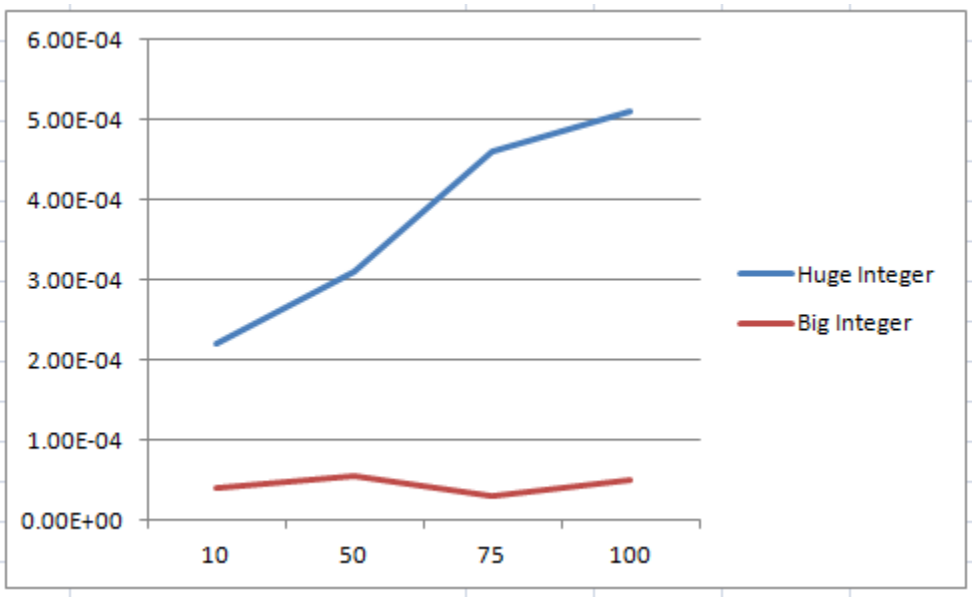
Darren Chang
1406398
changd

checked, where they are the same size, where the first integer is bigger and where the second integer is bigger this gives a total number of cases equal to:
4 (number of functions) x 4 (number of positive/negative combinations) x 3 (number of bigger smaller integer combinations = 48 cases. Possible cases could be error cases where a number isn't given or where a first value that is not 0 or 1 is given. To make sure that these cases work exceptions need to be thrown if any of these cases are inputted. To make sure that the proper inputs are working the test procedure above will need to be used. The outputs for my code did not meet the specifications. Only about half of the designated outputs for the four functions worked while the other half gave wrong answers.

Graphed Results:

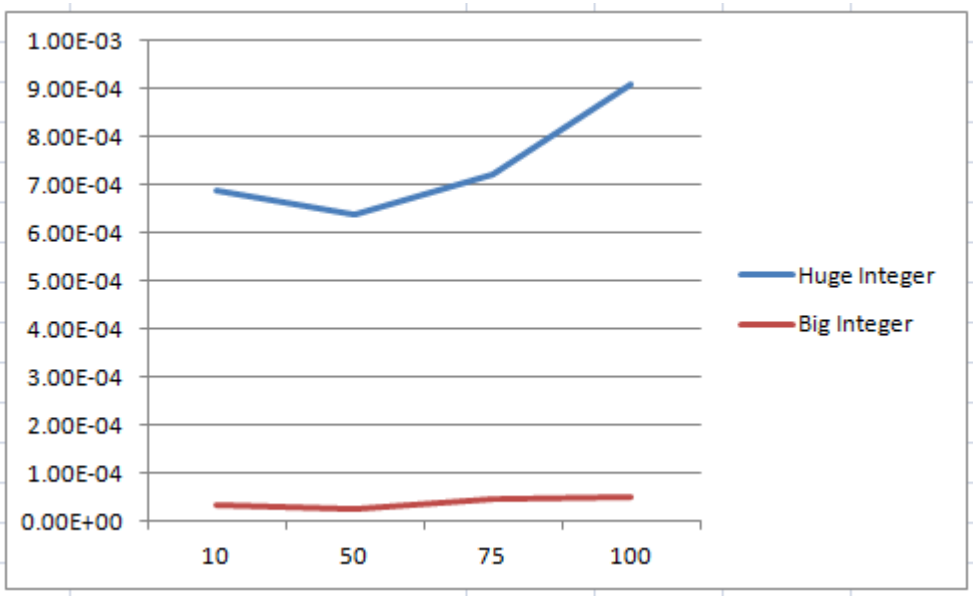
MAXRUN = 10, MAXNUMINTS = 100

Addition

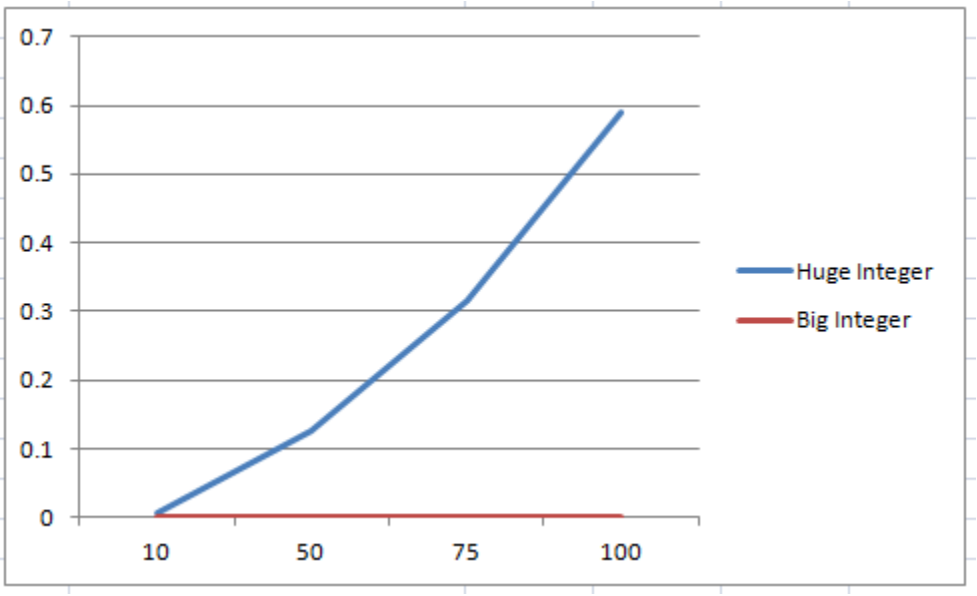


Darren Chang
1406398
changd

Subtraction

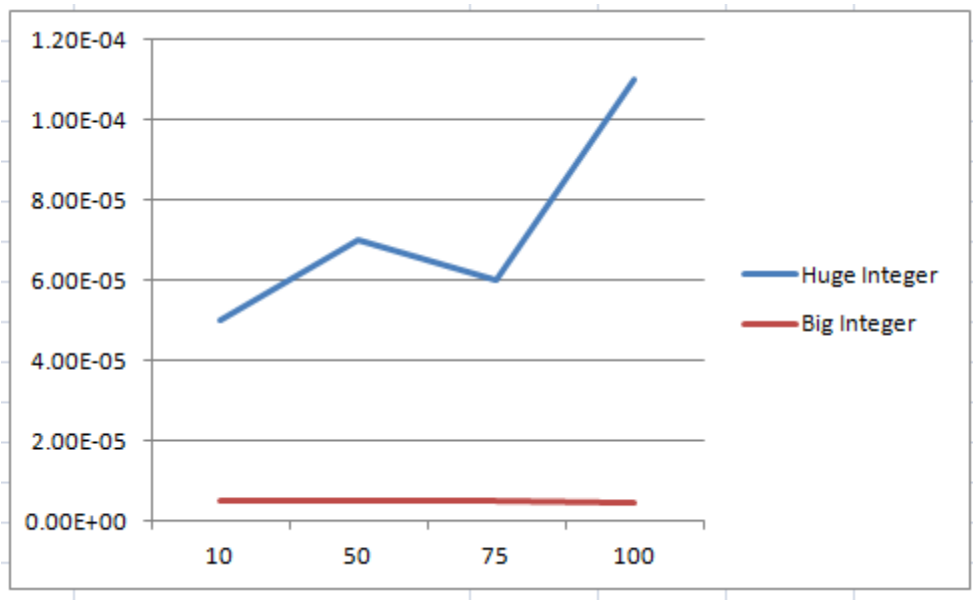


Multiplication



Darren Chang
1406398
changd

Compare



Raw Values:

n = 10
add: 2.2000000000000006E-4
sub: 6.900000000000004E-4
mult: 0.006869999999999903
comp: 5.0E-5

n = 50
add: 3.100000000000001E-4
sub: 6.400000000000004E-4
mult: 0.1258100000000002
comp: 7.000000000000001E-5

n = 75
add: 4.6000000000000023E-4
sub: 7.200000000000004E-4
mult: 0.31601000000000135
comp: 6.0E-5

n = 100
add: 5.100000000000003E-4
sub: 9.100000000000005E-4
mult: 0.5913599999999982
comp: 1.0999999999999999E-4

Darren Chang
1406398
changd

Second Attempt:

Addition

Integer Size	Huge Integer	Big Integer
10	0.033000000000000015	4.039999999999998E-5
50	0.03633000000000002	5.648079999999989E-5
75	0.03436330000000001	2.9712961600000025E-5
100	0.03134363300000002	5.08594259231999E-5
250	0.055313436329999986	3.53017188518464E-5
400	0.0775531343632999	6.247060343770354E-5
500	0.07777553134363294	4.652494120687534E-5
1000	0.27577775531343596	6.889304988241355E-5

Subtraction

Integer Size	Huge Integer	Big Integer
10	0.009757777553134358	3.213778609976485E-5
50	0.011097577775531345	2.6864275572199555E-5
75	0.010110975777755313	4.7253728551144336E-5
100	0.012101109757777555	5.009450745710221E-5
250	0.02812101109757779	3.570018901491421E-5
400	0.05028121011097577	4.5671400378029765E-5
500	0.07050281210110967	4.769134280075599E-5
1000	0.21170502812101083	7.049538268560132E-5

Multiply

Integer Size	Huge Integer	Big Integer
10	0.0790000020201019	3.400721923847055E-5
100	0.6417900000202019	6.6496992346143935E-5
500	1.4294179000002023	1.3413299398469197E-4
1000	2.5702941790000016	4.754925365319772E-4

Compare-To

Integer Size	Huge Integer	Big Integer
10	6.178838667889E-5	4.9351920834358145E-6
50	9.142596467486969E-5	5.1154681690423275E-6
75	3.1318300050789107E-5	4.9850224174503895E-6
100	7.102025370757862E-5	4.723204949625405E-6
250	1.5222535405121725E-4	5.013163896902684E-6
400	2.7424952626879494E-4	5.044119615209829E-6
500	1.9465966687783957E-4	4.900132594981822E-6
1000	4.2619102081017213E-4	4.922780869003992E-6

Darren Chang
1406398
changd

Discussion:

The results obtained from my Huge Integer and my theoretical runtimes make sense. As you can see the graphs for addition and subtraction are both increasing linearly, which makes sense as my theoretical analysis resulted in a big theta of n running time. As expected, my multiply function is increasing in a quadratic function which I predicted when I got a theoretical result of big theta of n^2. It doesn't look like it but the compare-To numbers are so small that it is basically a flat line, which also works for my theoretical analysis of big theta of constant time. The implementation of my code compared to Big Integer is extremely inefficient. Java is probably using a different method of computing really big numbers, instead of using arrays; like I did. However, it does run it in the same big theta subspace as my Huge Integer methods. For example, it seems that the Big Integer run time for addition and subtraction also increases linearly as you increase the size of the array. Thereby big theta of n. The theoretical calculations corresponded to my measurements very well except for the few outliers but these cases are attributed to the fact that the algorithms were not implemented properly so some of the values obtained did not correlate to the algorithm itself. Java.lang.biginteger was better in almost every case of the runs times whether it was addition, subtraction, multiplication or the compareTo. This is obviously a given because the implementation of java.lang.biginteger was done by professionals who were able to extract the most efficient way of handling the problem. If I had extra time I would have implemented the Karatsuba multiplication method which does multiplication in Θ(nlog(n)) time. The theoretical calculations corresponded to my measurements very well except for the few outliers but these cases are attributed to the fact that the algorithms were not implemented properly so some of the values obtained did not correlate to the algorithm itself. Some problems that I had in obtaining my experimental data, was that my multiply function took a long time to run and get the results. To combat this, I made it so that I would only test my multiply function for four smaller (but still pretty big) data points (i.e. 10, 100, 500, 1000). This would still allow me to get a good approximation of the graph for the run time of multiply, without waiting forever for it to finish.