Hamdan Basharat
basham1
400124515
COMPENG 2SI4

Lab 1 & 2 Report

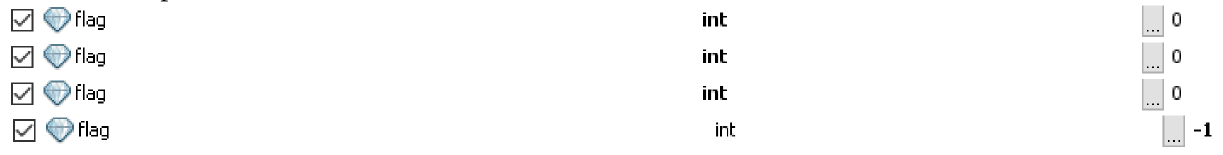
Description of Data Structures and Algorithms

The motive behind creating the HugeInteger class was to create a class that could do integer arithmetic with numbers that are larger than the maximum allowed size of java integers. In order to do this, I created HugeInteger objects that hold string variables to represent the integers. With this, the objects can store integers that have up to 2^31-1 digits. If the integer representation is directly inputted, it is taken as a string where each character is checked to make sure it represents valid digits. Also, if there was a negative sign at the start of the integer, it is removed and the variable "neg" of the HugeInteger object is set to "1" to represent it. Another constructor which takes integer input is also available if only the number of digits is specified. Upon valid input, it randomly generates the specified number of characters representing digits from 0 to 9 and stores it into a character array to be turned into a string. This constructor also makes sure that the first digit generated is never zero, and that the integer could be positive or negative.

Each of the arithmetic operations has their own method that is called by referencing the current HugeInteger object and placing another HugeInteger object as an argument.

Comparison is done through the compareTo() method where a flag is set to return either 1, -1, or 0 if the current integer is greater than, smaller, or equal to the passed integer. Initially, through if statements, the signs of the integers are compared immediately to determine which of the integers is larger. If both signs are the same the number of digits are compared. Finally, if each of the integers have the same number of digits, the leading digits are checked one by one until one of the digits are bigger or smaller than the other (depending on whether the signs were negative or positive). Depending on the scenario the flag is altered and is returned after the cases have been checked. Another method called magnitude() was also created to compare the magnitude of two integers, which uses the same underlying code as the compareTo() method, except for the checks for the sign.

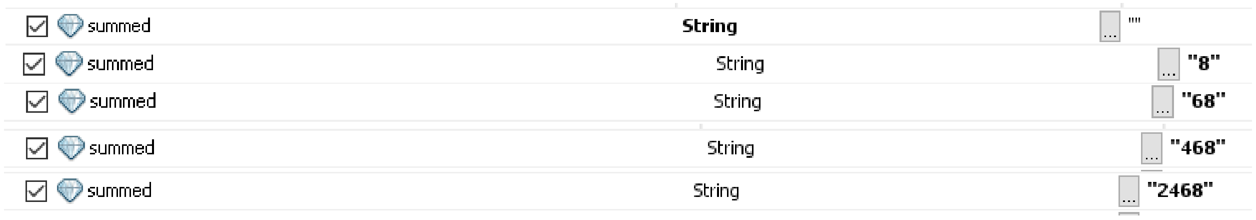
Ex. 123 compared to 126



```
flag    int    0
flag    int    0
flag    int    0
flag    int    -1
```

Addition and subtraction were implemented through the add() and subtract() methods. These methods are intertwined because depending on the scenario, one of the methods calls the other to compute the result. At the beginning of both methods, if the sign of the integers differs ,the other method is called and a variable of the HugeInteger called "check" is set to true so that it isn't called back to the same method. The integer with more digits is set to a string called "big" and the smaller to "small". The smaller integer with less zeros is then padded with leading zeros until both integers have the same number of digits. At this point the methods begin to differ as they run the algorithms. In addition, a loop is run for the length of the “big” string, and for each index of the two integers starting from the end, the sum of the characters is found and converted to digits through their ASCII values. This sum is then added to the front of a string called “summed” that is made into a HugeInteger object. If the sum of the digits at any given index is greater than 9, the value is subtracted by 10 and a “carry” variable adds one to the next index to symbolize the tens position. Finally, if both “neg” variables of the objects symbolized negatives, the resulting sum is also set to negative. The new sum of the integers is then returned.

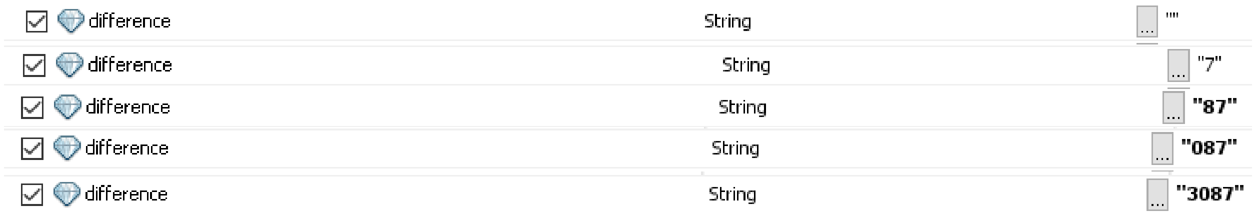
Ex. 1234 + 1234



```
summed    String    ""
summed    String    "8"
summed    String    "68"
summed    String    "468"
summed    String    "2468"
```

In subtraction, copies of the current and passed HugeInteger are made in order to not alter the original integers. Like addition, a loop is run for the length of the “big” starting from the ends of each of the integers. The character from each index of “small” is subtracted from the index of “big” and the value is converted to digits from 0-9 using the ASCII values and stored in a variable called “num”. If the value of “small” at an index is greater than the value of “big” at that index, the subtraction is completed but 10 is added to the value. A variable called “carry” is then set to -1 and is added to the next index to symbolize taking ten away. The variable “num” is added to a string called “difference” which is used to make a new HugeInteger, “diff”. Then through a series of if statements the signs of the HugeIntegers are used to determine and set the sign of “diff” which is then returned.

Ex. 4321 – 1234

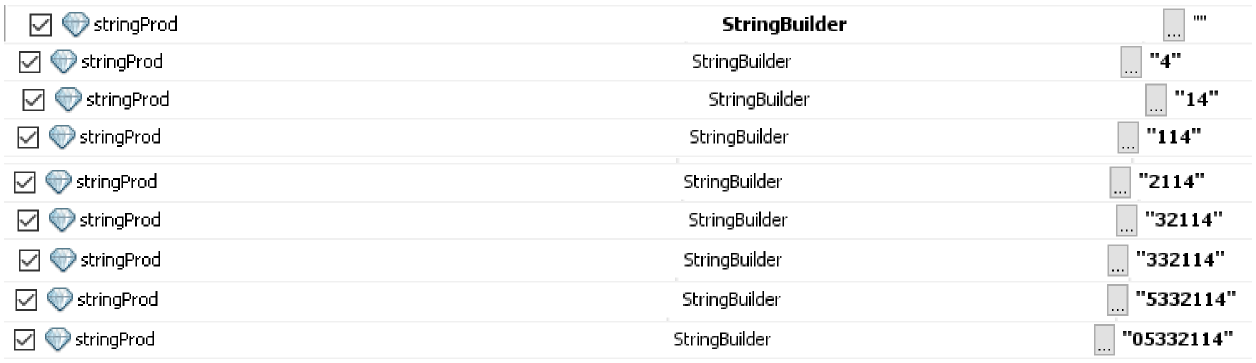


```
difference    String    ""
difference    String    "7"
difference    String    "87"
difference    String    "087"
difference    String    "3087"
```

The method multiply() is used to perform multiplication in the HugeInteger class. Initially, through the use of the built in StringBuilder class, the HugeInteger strings are stored as reversed strings “int1” and “int2”. Another StringBuilder string “stringProd” is also made to represent the product. If either of the HugeIntegers is “0”, the method immediately returns a HugeInteger with string “0” symbolizing multiplication by 0. For the actual multiplication of the integers, a nested for loop is used to multiply the character of each index of int2 by the current index of int1. The value is stored onto the current index of an integer array “prod”. Then another for loop is run to take the value of each index of prod and append the value in the one’s column onto stringProd. If the index carried a value greater than 9, the value in the ten’s column is added over to the next index. The string is then used to create a new HugeInteger object, “returnProd”.

If the signs of the inputted integers differed, the variable “neg” for the returnProd is set to “1” and returnProd is returned.

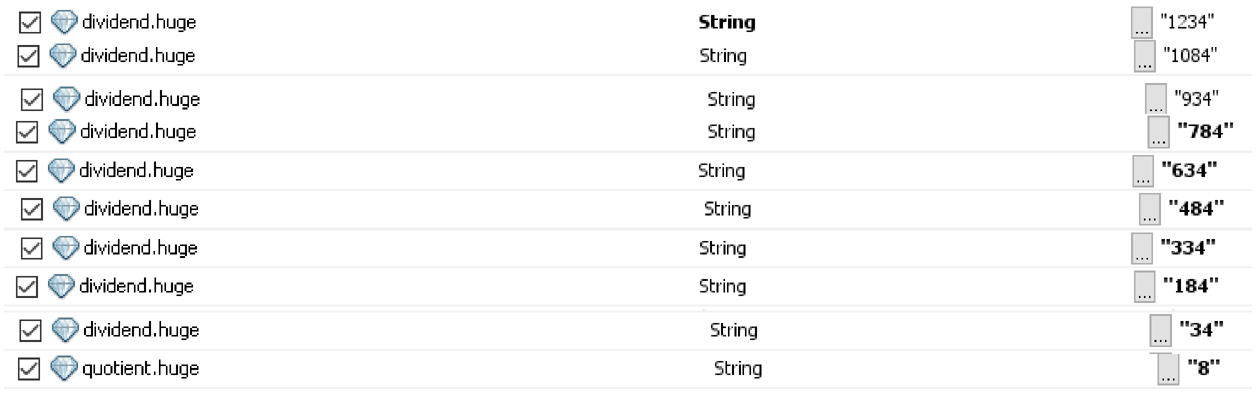
Ex. 4321 x 1234



✓ stringProd	**StringBuilder**	""
✓ stringProd	StringBuilder	**"4"**
✓ stringProd	StringBuilder	**"14"**
✓ stringProd	StringBuilder	**"114"**
✓ stringProd	StringBuilder	**"2114"**
✓ stringProd	StringBuilder	**"32114"**
✓ stringProd	StringBuilder	**"332114"**
✓ stringProd	StringBuilder	**"5332114"**
✓ stringProd	StringBuilder	**"05332114"**

Lastly, division is implemented using the method, division(). This method first creates copies of the inputted HugeIntegers called “dividend” and “divisor” so that the original values are not altered. If the dividend has a value of ”0”, a HugeInteger of value “0” is immediately returned. However, if the divisor has a value of “0”, an exception is thrown because division by zero can’t be completed. Also, if the magnitude of the numbers is equal, a HugeInteger of value ‘1” is returned. For the algorithm, another HugeInteger called “quotient” is created to be used as a counter and return the quotient of the integer division. A loop is run which subtracts the divisor from the dividend and adds one to the quotient every time it does. The loop terminates when the dividend is no longer greater than the divisor. If the signs of the inputted integers differed, the variable “neg” for the quotient is set to “1” and quotient is returned.

Ex. 1234 / 150



✓ dividend.huge	**String**	"1234"
✓ dividend.huge	String	"1084"
✓ dividend.huge	String	"934"
✓ dividend.huge	String	**"784"**
✓ dividend.huge	String	**"634"**
✓ dividend.huge	String	**"484"**
✓ dividend.huge	String	**"334"**
✓ dividend.huge	String	**"184"**
✓ dividend.huge	String	**"34"**
✓ quotient.huge	String	**"8"**

Additional to the arithmetic methods created, the methods removeZeros() and to String() were used. The first removed the leading zeros from a HugeInteger string and was used through the class. The second was used to print the HugeInteger to the screen and was called in the main class.

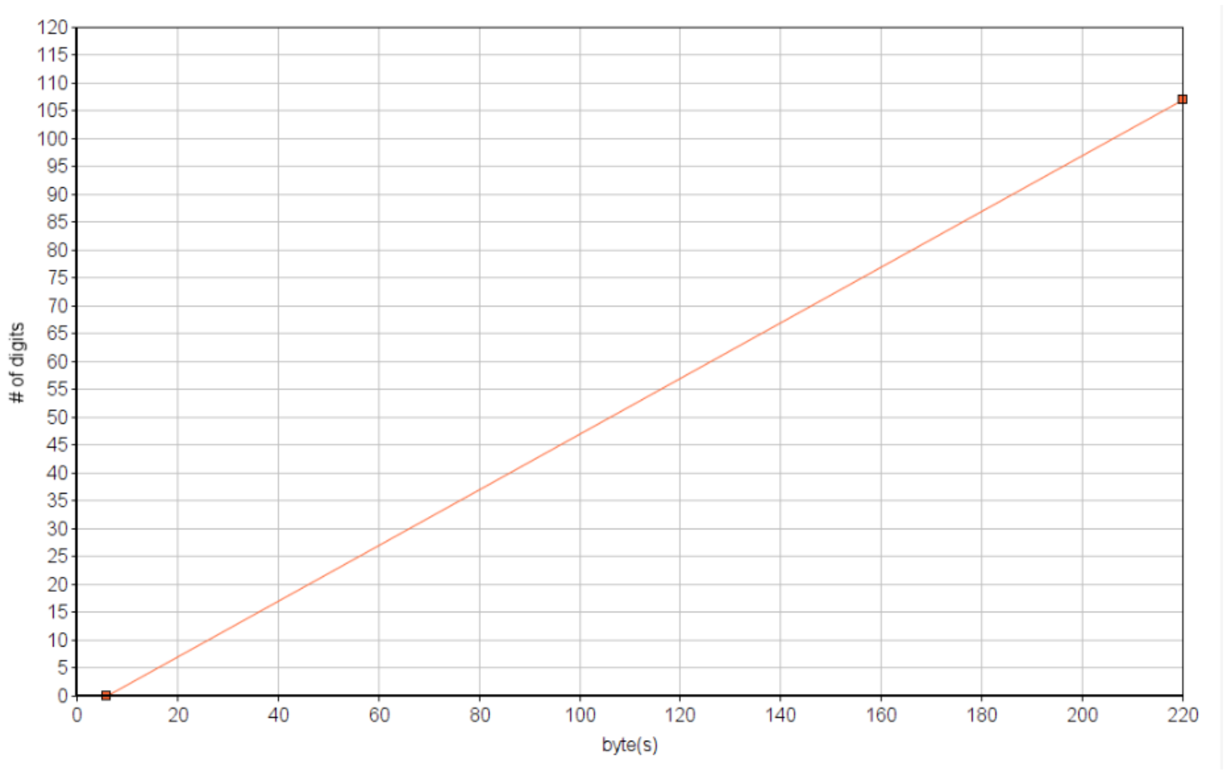
Theoretical Analysis of Running Time and Memory Requirement

To figure out how much memory is required to input an integer or n decimal digits using my HugeInteger class, first I must see how much memory different data types in the program take to allocate,

Memory Required	char	int	Boolean
byte(s)	2	4	1

Storing a single HugeInteger in my class requires the initializing of a String to store the integer, which uses n characters. Three other variables are also initialized; an integer “neg” determines if there was a negative sign, a Boolean “larger” determines if the integer is larger than the other, and a Boolean “check” to serve as a flag when calling a method from another. Adding all of these gives a function for memory required to store a HugeInteger;

$$2*n + 4 + 1 + 1 = 2n + 6 \text{ (bytes)}$$



Addition:

Big Theta Time Complexity: Θ(n) = n + c

Memory Requirement:2*n + 6 + 2*2*n + 4 + 4 + 2 *n + 2*n + 2*n + 6 = 10*n + 20 (bytes)

Subtraction:

Big Theta Time Complexity: Θ(n) = n + c

Memory Requirement: 2*n + 6 + 2*n + 6 + 2*n + 6 + 4 + 4 + 2*n + 2*n + 6 +2*n + 6 + 2*n + 6 = 14*n + 44 (bytes)

Comparison:

Big Theta Time Complexity: Θ(n) = n + c

Memory Requirement: 4 + 4 + 2*n + 4 + 2*n = 4*n + 12 (bytes)

Multiplication:

Big Theta Time Complexity: Θ(n) = n^2 + c

Memory Requirement: 2*n + 2*n + 2*2*n + 2*n*4 + 4 + 4 + 2*2*n + 2*n + 6
= 22*n + 14 (bytes)

Division:

Big Theta Time Complexity: Θ(n) = 10^n + c

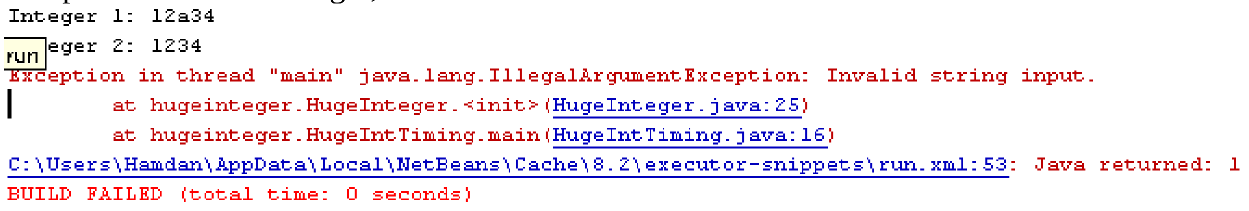
Memory Requirement: 2*n + 6 +2*n + 6 + 2*n + 6 + 2*n + 6 + 2*n + 6 + n*(14*n +44)
= 14*n^2 + 54*n + 30 (bytes)

In order to calculate Θ(n) I analyzed each of the methods and the routines they used. Initializing or altering a variable or doing a non-loop logic statement accounted for 1 complexity unit. Therefore, by determining how many times loops ran and any additional complexity units, one can find the Θ(n) for the method. Since we are only looking for the most significant part of the function, loops typically running n times were the primary functions, plus an arbitrary number of additional processes. In cases like multiplication where a nested for loop was used, the complexity was n^2 to represent the number times the loop is run. In division, the worst-case scenario is 10^n where the dividend is n bases of 10 larger than the divisor. To find the memory requirement of the methods, I used the debugger and went through the data types that were initialized in the memory for each method and added them.

Test Procedure

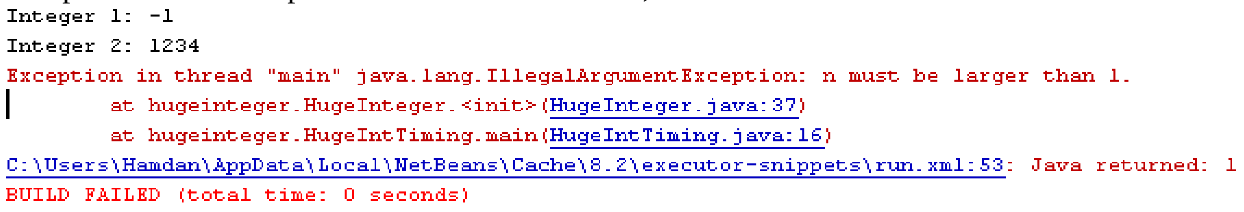
In order to completely test the functionality of this HugeInteger class, it would have to be tested against every possible input by the user. The constructors must be able to take both valid and invalid inputs and respond to them accordingly, and the methods need to output the correct mathematical value to the arithmetic operation they are subjected to. The cases for the constructors that must be accounted for include inputs with valid and invalid integers taken as strings, potential negative signs, and requests to randomly generate numbers. The methods have more scenarios that they must account for, first of which is arithmetic with one, two, or no negative numbers. These cases must be valid for any arbitrarily sized integers, where the integers may be the same or different in number of digits. Finally, all the methods must be able to handle inputs where one or both integers have value “0”. The following are 17 test cases that cover all the possible test cases for the HugeInteger class;

1. Input of an invalid integer,



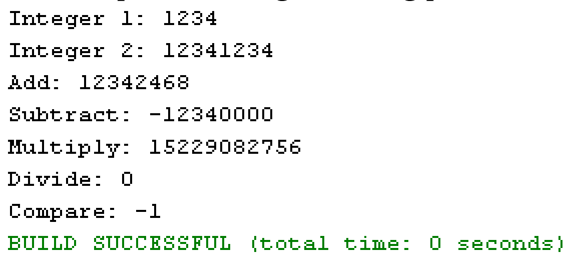
```
Integer 1: 12a34
run eger 2: 1234
Exception in thread "main" java.lang.IllegalArgumentException: Invalid string input.
        at hugeinteger.HugeInteger.<init>(HugeInteger.java:25)
        at hugeinteger.HugeIntTiming.main(HugeIntTiming.java:16)
C:\Users\Hamdan\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

2. Input for random input with number less than 0,



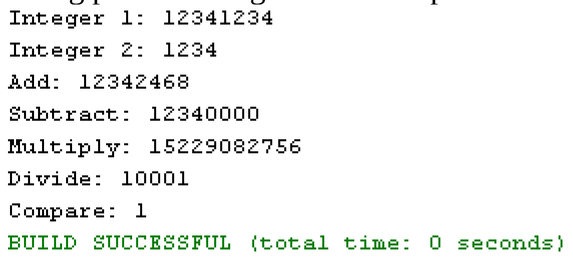
```
Integer 1: -1
Integer 2: 1234
Exception in thread "main" java.lang.IllegalArgumentException: n must be larger than 1.
	at hugeinteger.HugeInteger.<init>(HugeInteger.java:37)
	at hugeinteger.HugeIntTiming.main(HugeIntTiming.java:16)
C:\Users\Hamdan\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

3. Small positive integer and big positive integer,



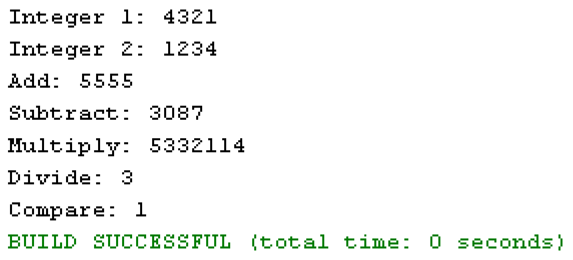
```
Integer 1: 1234
Integer 2: 12341234
Add: 12342468
Subtract: -12340000
Multiply: 15229082756
Divide: 0
Compare: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Big positive integer and small positive integer,



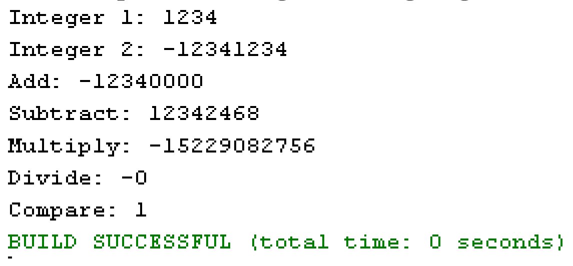
```
Integer 1: 12341234
Integer 2: 1234
Add: 12342468
Subtract: 12340000
Multiply: 15229082756
Divide: 10001
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

5. Two positive integers with same number of digits,



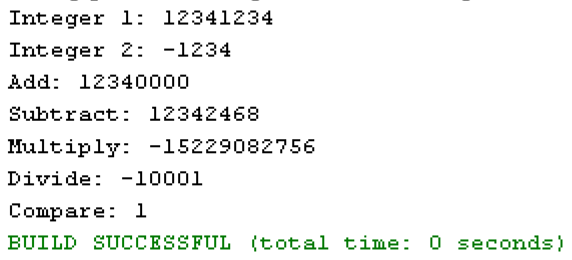
```
Integer 1: 4321
Integer 2: 1234
Add: 5555
Subtract: 3087
Multiply: 5332114
Divide: 3
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

6. Small positive integer and big negative integer,



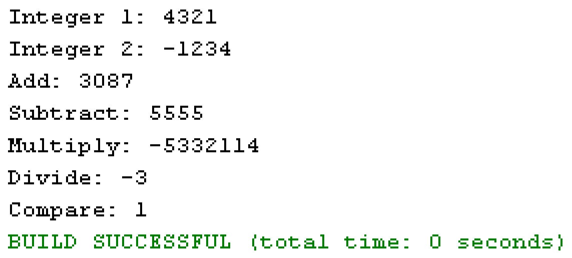
```
Integer 1: 1234
Integer 2: -12341234
Add: -12340000
Subtract: 12342468
Multiply: -15229082756
Divide: -0
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

7. Big positive integer and small negative integer,



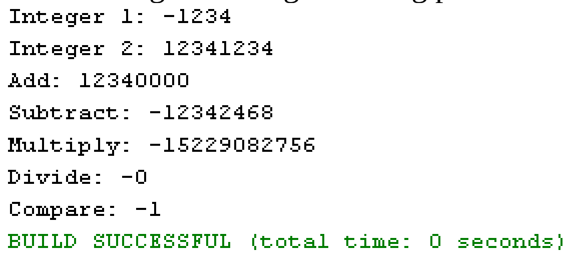
```
Integer 1: 12341234
Integer 2: -1234
Add: 12340000
Subtract: 12342468
Multiply: -15229082756
Divide: -10001
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

8. Positive and negative integer with the same number of digits,



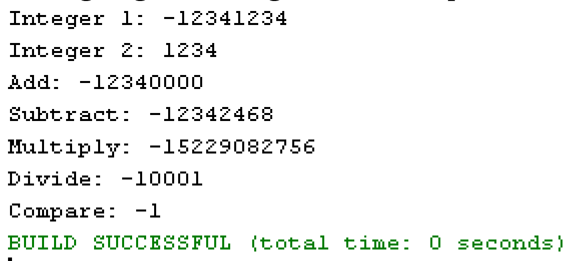
```
Integer 1: 4321
Integer 2: -1234
Add: 3087
Subtract: 5555
Multiply: -5332114
Divide: -3
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

9. Small negative integer and big positive integer,



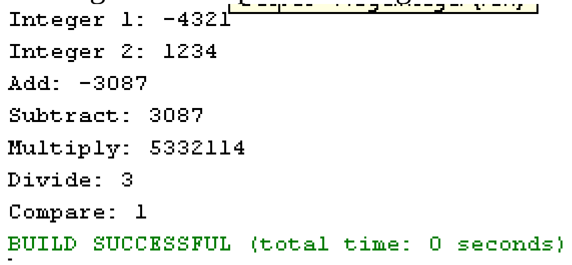
```
Integer 1: -1234
Integer 2: 12341234
Add: 12340000
Subtract: -12342468
Multiply: -15229082756
Divide: -0
Compare: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

10. Big negative integer and small positive integer,



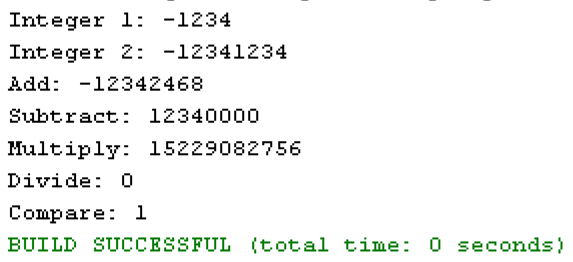
```
Integer 1: -12341234
Integer 2: 1234
Add: -12340000
Subtract: -12342468
Multiply: -15229082756
Divide: -10001
Compare: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

11. Negative and positive integer with the same number of digits,



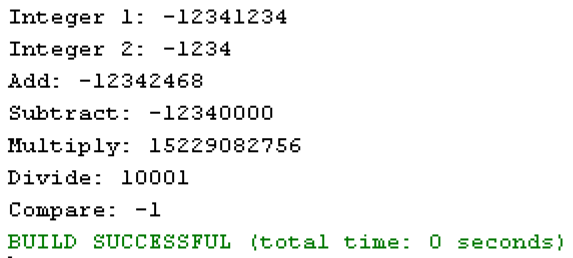
```
Integer 1: -4321
Integer 2: 1234
Add: -3087
Subtract: 3087
Multiply: 5332114
Divide: 3
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

12. Small negative integer and big negative integer,



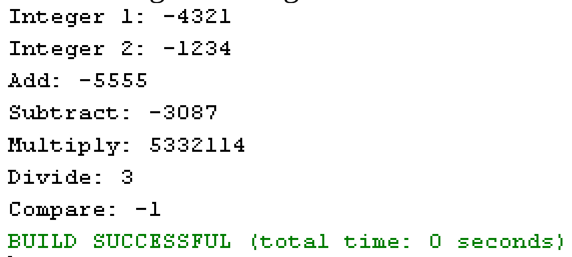
```
Integer 1: -1234
Integer 2: -12341234
Add: -12342468
Subtract: 12340000
Multiply: 15229082756
Divide: 0
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

13. Big negative integer and small negative integer,



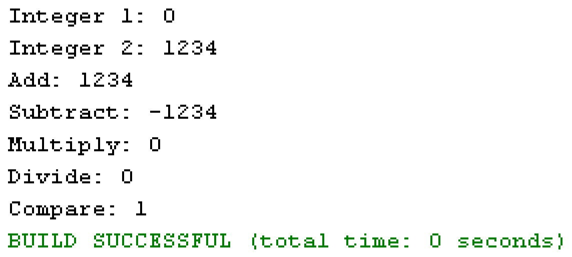
```
Integer 1: -12341234
Integer 2: -1234
Add: -12342468
Subtract: -12340000
Multiply: 15229082756
Divide: 10001
Compare: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

14. Two negative integers with the same number of digits,



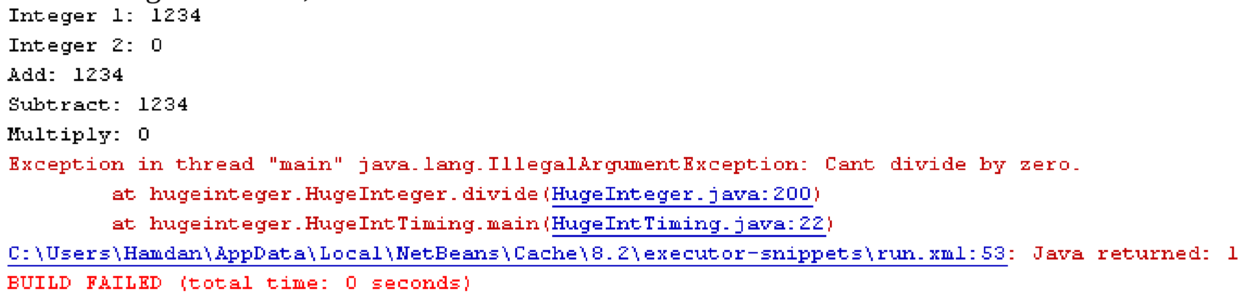
```
Integer 1: -4321
Integer 2: -1234
Add: -5555
Subtract: -3087
Multiply: 5332114
Divide: 3
Compare: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

15. Zero and an integer,



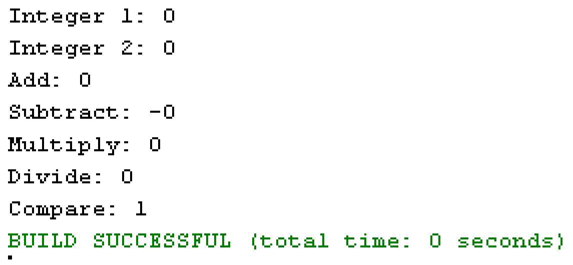
```
Integer 1: 0
Integer 2: 1234
Add: 1234
Subtract: -1234
Multiply: 0
Divide: 0
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

16. An integer and zero,



```
Integer 1: 1234
Integer 2: 0
Add: 1234
Subtract: 1234
Multiply: 0
Exception in thread "main" java.lang.IllegalArgumentException: Cant divide by zero.
        at hugeinteger.HugeInteger.divide(HugeInteger.java:200)
        at hugeinteger.HugeIntTiming.main(HugeIntTiming.java:22)
C:\Users\Hamdan\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

17. Zero and zero,



```
Integer 1: 0
Integer 2: 0
Add: 0
Subtract: -0
Multiply: 0
Divide: 0
Compare: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

As shown above, my HugeInteger class outputs the correct result for any input given, meeting all the required specifications. When I was writing the code for the add and subtract methods, I had a fair bit of difficulty debugging the overlapping cases. The way I decided to approach addition and subtraction was to have them work in conjunction with either other depending on the scenario presented. With this, if a problem is called in add or subtract and could be more easily solved in the other, it would call the other. Initially, I had used many logic statements in an attempt to capture all these possible scenarios, but I would often run into problems where the program would be stuck in an infinite loop calling methods back and forth. To fix this I used a flag that made sure that if a method called another method, it couldn't be sent back. I then reduced the number of logic statements used and the code ran as intended. As well, when I initially wrote the methods for lab 1 I didn't account for cases where the integers had the same number of digits. I realized this and updated my code, so now it accounts for any input condition that is presented.

Experimental Measurement, Comparison and Discussion

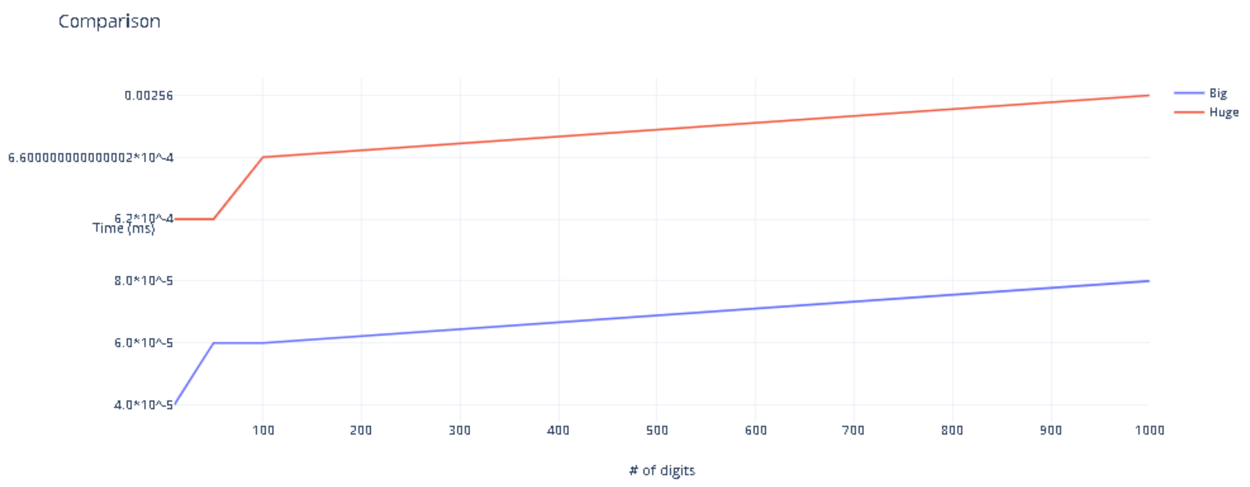
To measure the running time for each operation in the main class I had a section of code that would initialize a timer variable to zero milliseconds, run the specified method, and stop the timer once the method completes. This is within a loop that runs the same method a certain number of times and averages out the run time to get a more accurate result.
For these cases MAXNUMINTS = 100 and MAXRUN = 500;

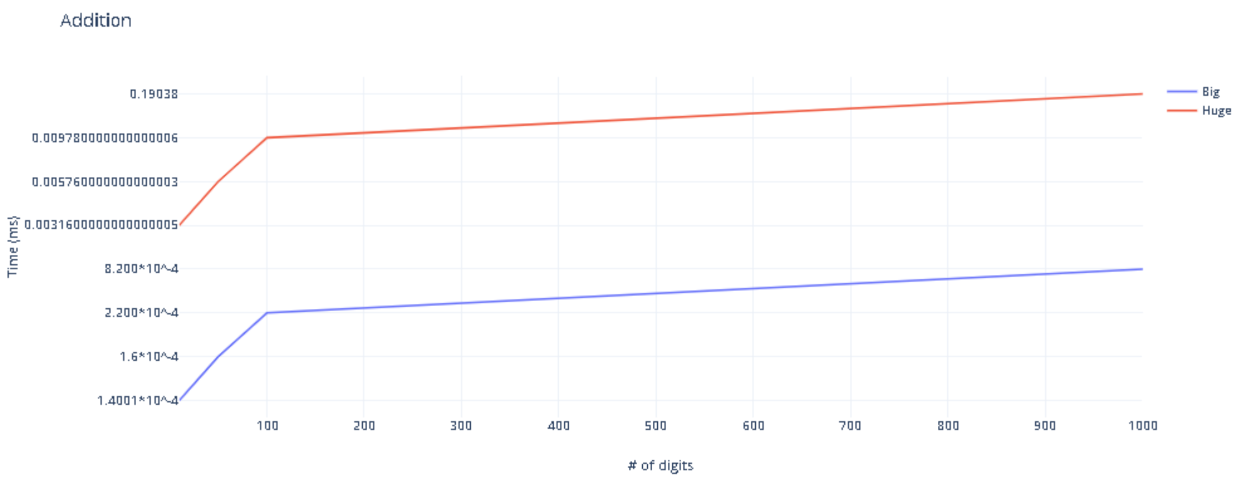
n = 10	Comparison	Addition	Subtraction	Multiplication
BigInteger	4.0*10^-5	1.4001*10^-4	1.2*10^-4	1.2*10^-4
HugeInteger	6.2*10^-4	0.0031600000000000005	0.0020800000000000003	0.003620000000000001

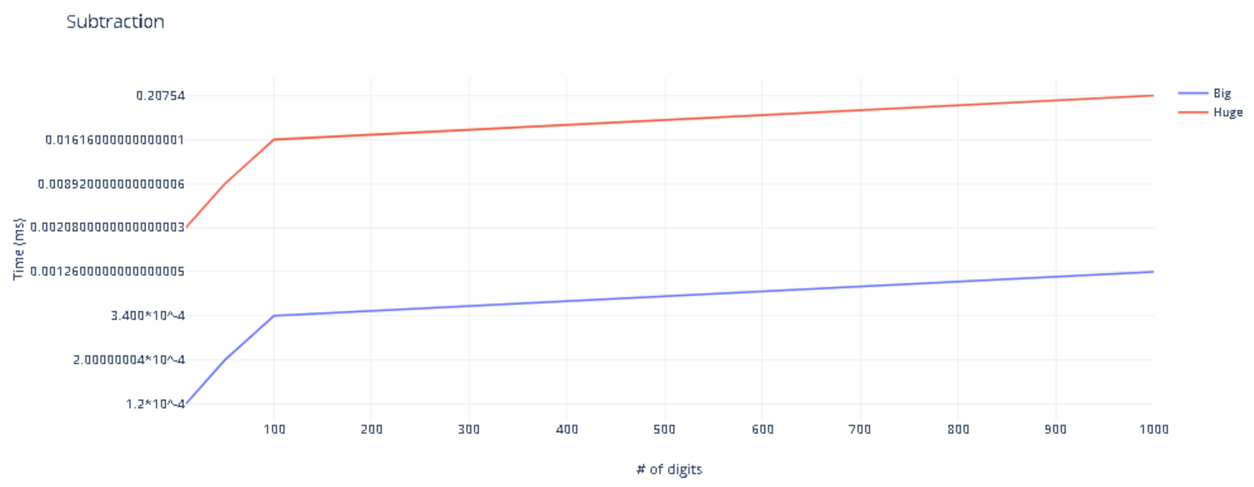
n = 50	Comparison	Addition	Subtraction	Multiplication
BigInteger	6.0*10^-5	1.6*10^-4	2.00000004*10^-4	2.8000001*10^-4
HugeInteger	6.2*10^-4	0.005760000000000003	0.008920000000000006	0.010340000000000002

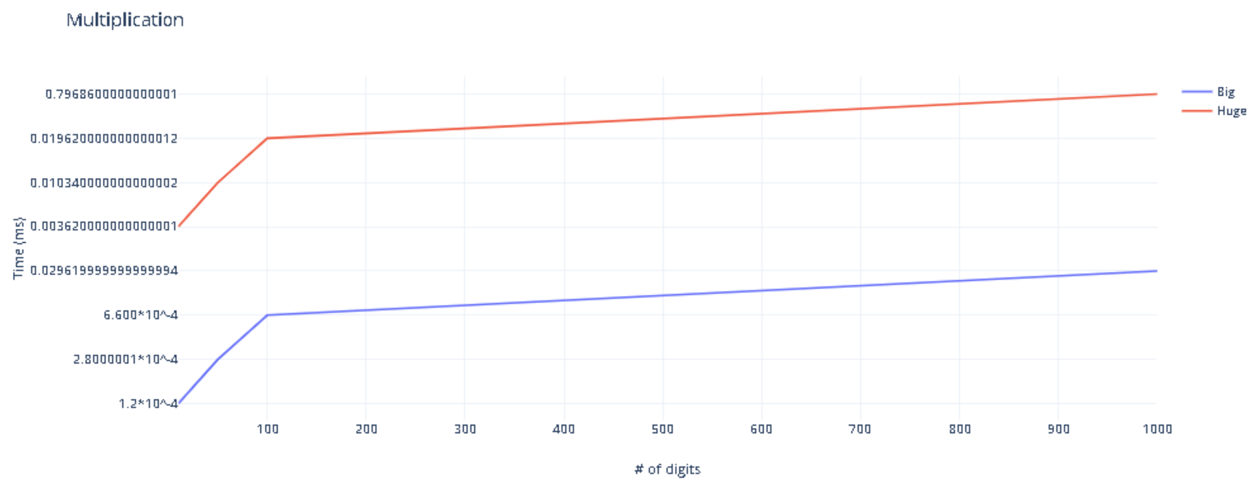
n = 100	Comparison	Addition	Subtraction	Multiplication
BigInteger	6.0*10^-5	2.200*10^-4	3.400*10^-4	6.600*10^-4
HugeInteger	6.600000000000002*10^-4	0.009780000000000006	0.01616000000000001	0.019620000000000012

n = 1000	Comparison	Addition	Subtraction	Multiplication
BigInteger	8.0*10^-5	8.200*10^-4	0.0012600000000000005	0.029619999999999994
HugeInteger	0.00256	0.19038	0.20754	0.7968600000000001









Discussion of Results and Comparison

My implementation of the HugeInteger class is considerably slower than the built-in java BigInteger class when looking at run time. In every method, the BigInteger class takes less time to run than my own class, even though some of the values are very close. This is more likely due to inefficiencies in my own code, and the extra processes that are ran cause the run time to be slower. Given extra time, I would try to implement faster ways of performing some of my methods in order to close the gap between my own run time and that of the BigInteger class and to use less memory. My current multiply function uses long grade school multiplication and a good alternative could be a sub quadratic algorithm like Karatsuba multiplication or Fast Fourier transform. As well, my current divide method is very inefficient as it relies on continuously calling my subtract method, and if the gap between the dividend and divisor becomes too large, the run time and memory usage get exponentially larger. A possible improvement that I could make to this is to implement long division, which would be much efficient.

Citations

https://www.geeksforgeeks.org/sum-two-large-numbers/
https://www.youtube.com/watch?v=D6xkbGLQesk