COMPENG 2SI4 LAB 1 & 2 REPORT

Raeed Hassan
hassam41

McMaster University

October 26, 2020

Description of Data Structures and Algorithms

The `HugeInteger` class has two private fields, `int sign` that stores the sign of the integer as an integer and `int[] magnitude` that stores the digits of the integer in an integer array.

The addition operation `public HugeInteger add(HugeInteger h)` returns a `HugeInteger sum` that represents the integer sum of `this` and `h`. It first checks if either `this` or `h` are zero by checking if `sign == 0`. The method returns the non-zero `HugeInteger` if one `HugeInteger` is 0 and returns a `HugeInteger` representing 0 if both are 0. Otherwise, the general algorithm is to create a new `HugeInteger sum` with size of `magnitude` equal to the larger `magnitude.length` between `this` and `h` and iterate through `this.magnitude` and `h.magnitude` to add the values at every index of the arrays multiplied by their respective `sign` to `sum.magnitude`. After this process, we iterate through `sum.magnitude` in reverse order to fix elements that are greater than 9 or less than 0. If it is greater than 9, subtract 10 from current index and add 1 to the previous index. If it is less than 0, increase the current index by 10 and subtract 1 from the previous index. All leading zeros are removed from `sum.magnitude` afterwards by replacing it with a smaller array. If the first index of `sum.magnitude` is greater than 9, a new array of length `sum.magnitude.length+1` is created and the values of `sum.magnitude` are copied over. This new matrix replaces `sum.magnitude`. The method then removes all leading zeros from `sum.magnitude`. The value of `sum.sign` equals the 1 if both integers were positive or if the first element of `sum.magnitude` is positive if the signs of the integers did not match. The value of `sum.sign` equals -1 if both integers were negative or if the first element of `sum.magnitude` is negative if the signs of the integers did not match. The method finally returns `HugeInteger sum`. The entire process is illustrated in Figure 1 with two positive integers.

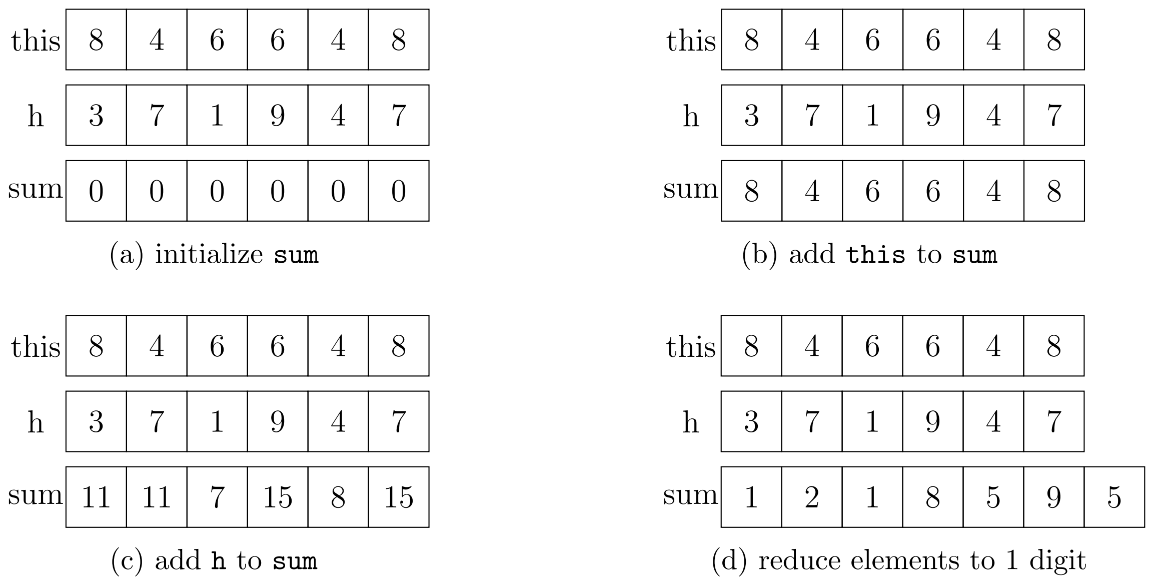

Figure 1: Addition

The subtraction operation `public HugeInteger subtract(HugeInteger h)` returns a `HugeInteger difference` that represents the integer given by of `h` subtracted from `this`.

simply creates a `HugeInteger temp` with the same `magnitude` as `HugeInteger h` but the opposite `sign` and returns the `HugeInteger` that is returned by calling the add method `this.add(temp)`.

The multiplication operation returns a `HugeInteger product` that represents the integer product of `this` and `h`. It first checks if either `this.sign == 0` or `h.sign == 0`, in which case it will return a `HugeInteger` representing 0. Otherwise, the general algorithm is to create a $2n + 1$ digit `HugeInteger product`, then add the products of `this.magnitude` and `h.magnitude` to `product.magnitude`. The operation loops has a nested for loop that iterates through `h.magnitude` in the outer loop, and iterates through `this.magnitude` in the inner loop, adding `this.magnitude*h.magnitude` to `product.magnitude` after each iteration of the inner loop. After each iteration of the outer loop, the starting index for `product.magnitude` is shifted to the left. After the completion of the nested for loop, the method reduces every element of `this.magnitude` to 1 digit, iterating through the array in reverse order and adding `product.magnitude[i]/10` to the previous element and setting `product.magnitude[i]` equal to `product.magnitude[i]%10`. The method then removes all leading zeros from `product.magnitude`. The value of `product.sign` is equal to `this.sign*h.sign`. The method finally returns `HugeInteger product`. The entire process is illustrated in Figure 2 with two positive integers.

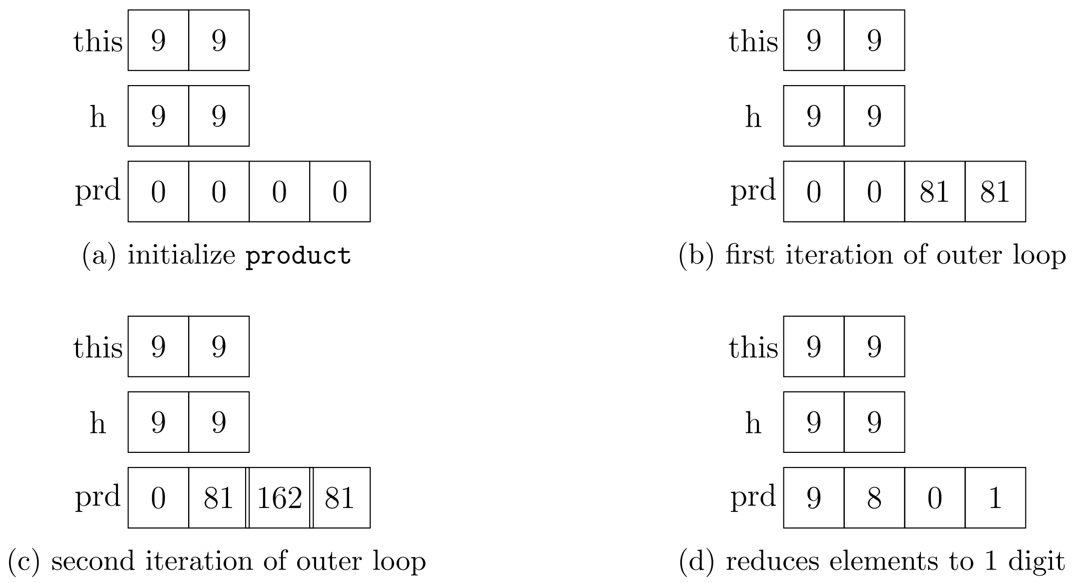

Figure 2: Multiplication

The comparison operation `public int compareTo(HugeInteger h)` returns 1 if `this HugeInteger > h`, 0 if `this HugeInteger == h`, and -1 if `this HugeInteger < h`. The method first compares the values of `this.sign` and `h.sign`. If `this.sign > h.sign`, the method returns 1 as `this HugeInteger` is greater. If `this.sign < h.sign`, the method returns -1 as `HugeInteger h` is greater. If neither condition is satisfied, both integers must have the same value of `sign`. If both integers are positive, it will compare their sizes and return 1 if `this.magnitude.length > h.magnitude.length` and -1 if `this.magnitude.length < h.magnitude.length`. If their lengths are the same, then it will iterate through both arrays and compare the values at each index, returning 1 if `this.magnitude[i] > h.magnitude[i]` at an index or -1 if `this.magnitude[i] < h.magnitude[i]` at an index. If both integers are negative, it will go through the same process, however the value returned will be the negative of what is returned with positive integers in each scenario. If none of these conditions have been meet, then the method will `return 0` as the two integers are equal.

Theoretical Analysis of Running Time and Memory Requirement

The `HugeInteger` class has fields `int sign` and `int[] magnitude`. An integer in `Java` is stored using 4 bytes. Therefore, the memory required to store a `HugeInteger` is $4 + 4n$, 4 bytes to store `sign` and $4n$ bytes to store `magnitude`. A plot of the memory required for a `HugeInteger` of n decimal digits can be seen in Figure 3.

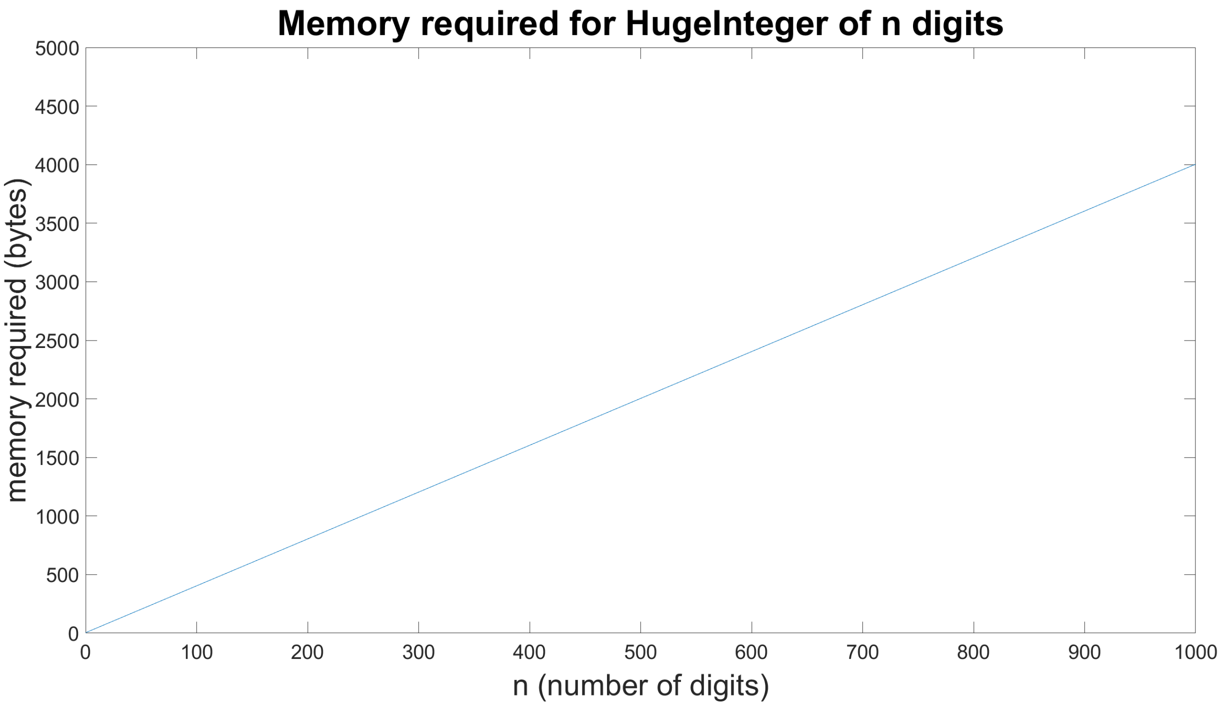

Figure 3: Memory required store a `HugeInteger` of n decimal digits

The running time for the addition operation is $\Theta(n) = n$ for both the average and worst case. The add method consists of operations that take constant time and for loops that take n time. This results in the running time of the operation being $T(n) = c_1 n + c_2 = \Theta(n) = n$ in both the average and worst case, as the method will always encounter a for loop. The addition operation requires up to $2(4 + 4(n + 1))$ extra bytes of memory to store the `HugeInteger` objects that will be returned, an additional $8n - 12n$ bytes of memory to store temporary integer arrays, and up to another 16 bytes of memory to store several integers.

The running time for the subtraction operation is $\Theta(n) = n$ for both the average and worst case. The subtract method creates a new `HugeInteger temp` to represent the negative of `HugeInteger h` and then adds `this HugeInteger` and `temp`. As the method simply creates a new object (which takes constant time) and then calls add, the running time is $c + T(add) = T(n) = c_1n + c_2 = \Theta(n) = n$ in both the average and worst case, as the method will always encounter a for loop. The subtraction operation requires the additional memory required by the addition operation as it calls the add method and also an additional $4 + 4n$ bytes to store `HugeInteger temp`.

The running time for the multiplication operation is $\Theta(n) = n^2$ for both the average and worst case. The multiplication method consists of operations that take constant time, for loops that take n time, and nested for loops that take n^2 time. This results in the running time of the operation being $T(n) = c_1n^2 + c_2n + c_3 = \Theta(n) = n^2$ in both the average and worst case, as the method will always encounter a nested for loop. The multiplication operation requires up to $4 + 4(2n + 1)$ extra bytes of memory to store `HugeInteger` that will be returned, an additional $8n - 12n$ bytes of memory to store temporary integer arrays, and an additional 4 bytes of memory to store an `int` that contains the size of the product.

The running time for the comparison operation is $\Theta(n) = 1$ for the average case and $\Theta(n) = n$ for the worst case. The comparison method consists of operations that take constant time and for loops that take n time. This results in the running time of the operation being $T(n) = c_1 = \Theta(n) = 1$ in the average case where the method does not encounter a for loop and $T(n) = c_1n + c_2 = \Theta(n) = n$ in the worst case where the method encounters a for loop. The comparison operation requires no additional memory.

Test Procedure

To test the functionality of the `HugeInteger` class and its methods, we will create a test class `TestHugeInteger` and perform all possible operations on all possible cases. The constructor and all four operations will be tested.

Possible constructor cases including inputs tested:

- input is zero (”0”)
- input is multiple zeros (”00000”)
- input is negative zero (”-0”)
- input is negative with multiple zeros (”-00000”)
- input is valid string (”1510239”)
- input is valid string with leading zeros (”000000001510239”)
- input is valid negative string (”-690290384”)
- input is negative single digit with leading zeros (”-00000000690290384”)
- input is invalid positive string input with string in middle (”11234-1231)

- input is invalid negative string input with string in middle ("-1863495-105328403924")
- input is $n = 0$ (0)
- input is $n > 0$ (5)
- input is $n < 0$ (-5)

Possible addition cases including inputs tested:

- zero + zero $(0 + 0)$
- zero + positive $(0 + 123)$
- zero + negative $(0 + (-123))$
- two positive integers with different number of digits $(125123 + 173452341)$
- two postive integers with same number of digits with same number of digits in result $(212361 + 501293)$
- two postive integers with same number of digits with different number of digits in result $(512315 + 941238)$
- two negative integers with different number of digits $((-123) + (-6789))$
- two negative integers with same number of digits with same number of digits in result $((-4424) + (-1138))$
- two negative integers with same number of digits with different number of digits in result $((-999) + (-998))$
- positive + negative with positive result $(514 + (-12))$
- positive + negative with leading zeros in positive result $(999 + (-996))$
- positive + negative with negative result $(913 + (-1042))$
- positive + negative with leading zeros in negative result $(555 + (-564))$

Possible subtraction cases including inputs tested:

- zero - zero $(0 - 0)$
- zero - positive $(0 - 123)$
- zero - negative $(0 - (-123))$
- positive - zero $(123 - 0)$
- negative - zero $((-123) - 0)$
- two positive integers with positive result $(1235 - 123)$
- two positive integers with negative result $(125123 - 173452341)$
- two positive integers with leading zeros in result $(555 - 564)$

- two equal positive integers $(123 - 123)$
- two negative integers with positive result $((-123) - (-1234))$
- two negative integers with negative result $((-1234) - (-123))$
- two negative integers with leading zeros in result $((-555) - (-564))$
- two equal negative integers $((-123) - (-123))$
- positive - negative $(514 - (-123))$
- negative - positive $((-514) - 123)$
- positive - negative with negative result $(913 + (-1042))$
- positive - negative with leading zeros in negative result $(555 + (-564))$

Possible multiplication cases including inputs tested:

- zero $\times$ zero (0×0)
- zero $\times$ positive (0×1)
- zero $\times$ negative (0×-1)
- positive $\times$ positive (12×99)
- positive $\times$ negative $(12 \times (-99))$
- negative $\times$ negative $((-12) \times (-99))$

Possible comparison cases including inputs tested:

- zero to positive (0, 123)
- zero to negative (0, -123)
- zero to zero (0, 0)
- positive to negative (123,-123)
- postive to zero (123,0)
- negative to positive (-123, 123)
- negative to zero (-123, 0)
- positive to larger positive (123, 124)
- positive to smaller positive (123, 12)
- equal positives (123, 123)
- negative to larger negative (-123, -12)
- negative to smaller negative (-123, -124)
- equal negatives (-123, -123)

All the test outputs meet specifications as they behaved in the intended manner. Invalid cases produced an exception, and valid cases produced the intended result, whether that be constructing the `HugeInteger` object or returning the correct result of an operation.

There were no difficulties with testing all possible cases or debugging the `TestHugeInteger` class. There were no input conditions that could not be checked.

Experimental Measurement, Comparison and Discussion

The experimental running time for each operation was tested using a modified version of the code provided in the instruction document. The major differences are that `System.nanoTime()` was used instead of `System.currentTimeMillis()` and `startTime` and `endTime` are both doubles, to prevent the need for a cast to a double. Otherwise, the code is the same, with the for loop being run for each value of n specified. The value of `MAXNUMINTS` was set to 110 for all tests, and the value of `MAXRUNS` was set to 100000 for the addition and subtraction operations, 1000000 for the comparison operation, and 10000 for the multiplication operation.

Table 1: Average running time for each operation (seconds)

		n					
		10	100	500	1000	5000	10000
`HugeInteger`	Addition	1.23e-7	2.30e-7	1.00e-6	2.11e-6	9.49e-6	3.00e-5
	Subtraction	1.35-7	2.46e-7	9.26e-7	1.95e-6	9.83e-6	3.07e-5
	Multiplication	1.80e-7	7.08e-6	1.52e-4	6.22e-4		
	Comparison	6.08e-9	3.32e-9	3.71e-9	2.90e-9	2.48e-9	2.27e-9
`BigInteger`	Addition	1.69e-8	3.24e-8	4.38e-8	5.88e-8	2.08e-7	4.52e-7
	Subtraction	2.16e-8	2.78e-8	3.47e-8	6.39e-8	2.13-e7	4.59e-7
	Multiplication	3.23e-8	5.69e-8	1.06e-7	3.56e-7	7.85e-6	3.02e-5
	Comparison	7.57e-9	5.70e-9	2.09e-9	2.14e-9	2.10e-9	2.12e-9
Theoretical	Addition	1.00e-8	1.00e-7	5.00e-7	1.00e-6	5.00e-6	1.00e-5
	Subtraction	1.00e-8	1.00e-7	5.00e-7	1.00e-6	5.00e-6	1.00e-5
	Multiplication	1.00e-9	1.00e-7	2.50e-6	1.00e-5	2.50e-4	1.00e-3
	Comparison	5.00e-9	5.00e-9	5.00e-9	5.00e-9	5.00e-9	5.00e-9

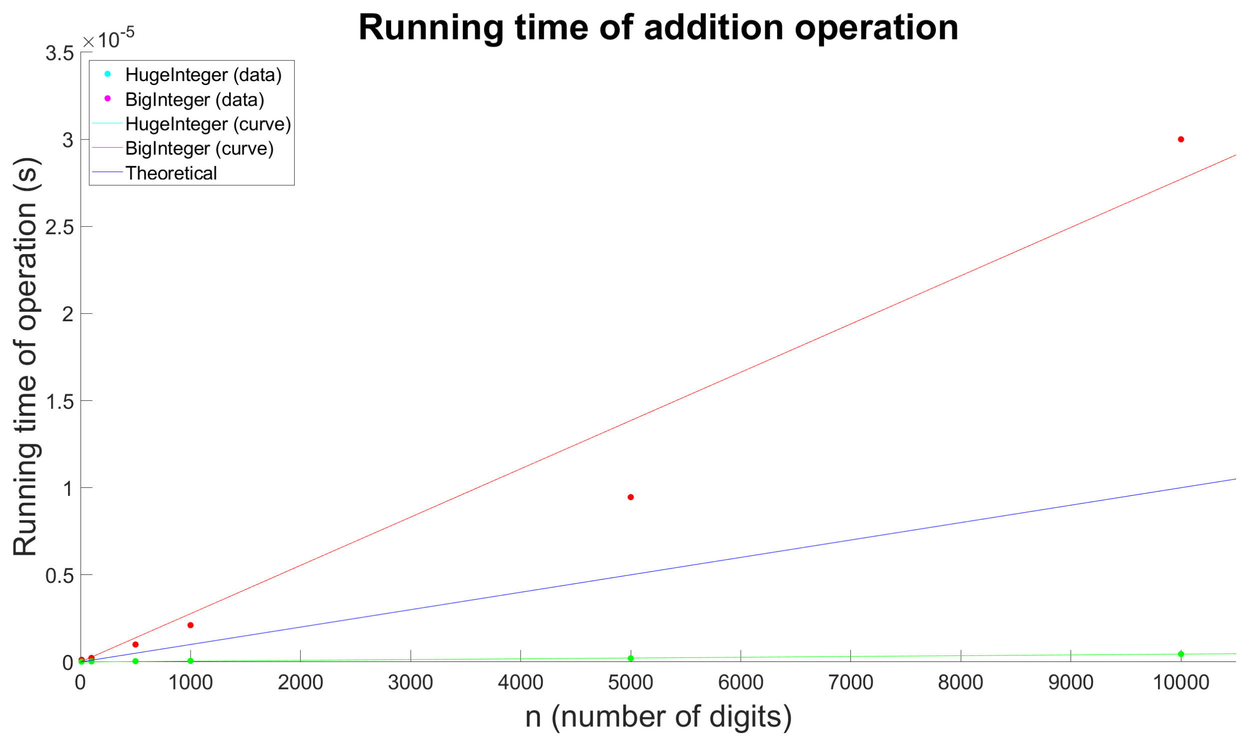

Figure 4: Plot of running time for addition operation

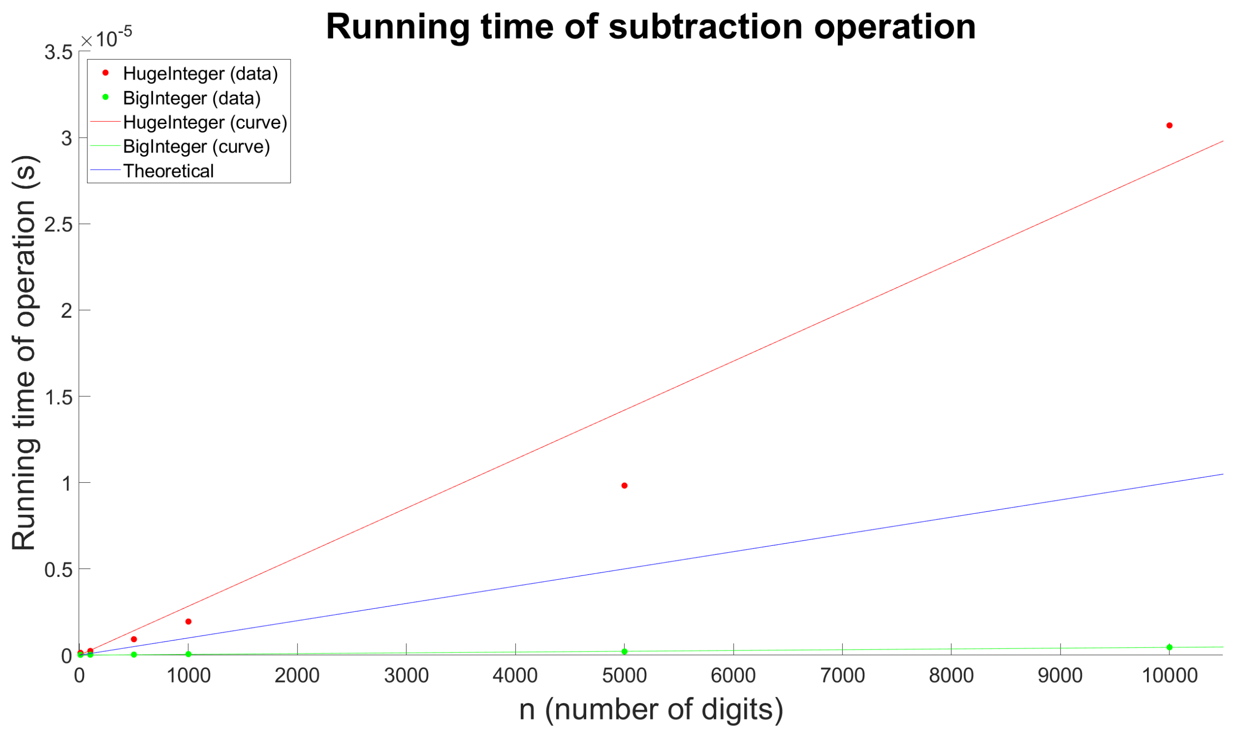

Figure 5: Plot of running time for subtraction operation

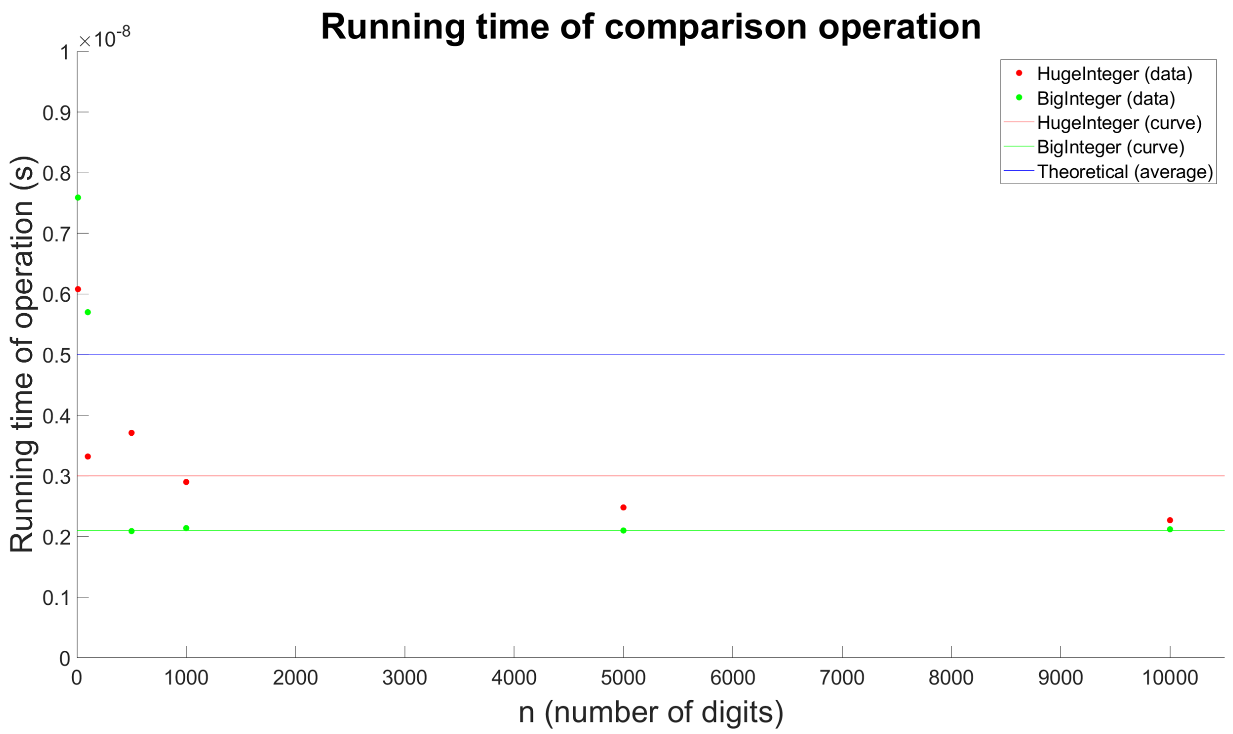

Figure 6: Plot of running time for comparison operation

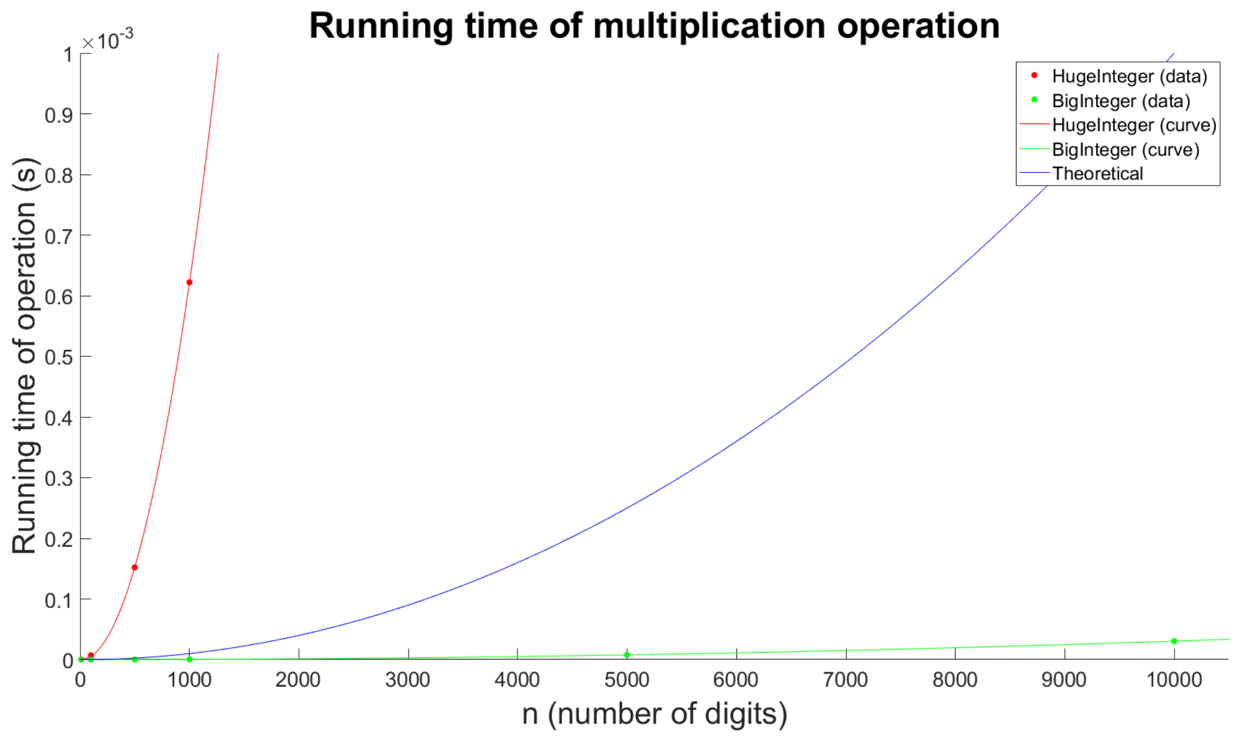

Figure 7: Plot of running time for multiplication operation

The only issue with obtaining experimental data occured with the multiplication operation of the `HugeInteger` class. The running time at $n = 5000$ and $n = 10000$ were ommitted because they would take too long to compute at the chosen parameters, and reducing the value of MAXRUNS to speed up the process would introduce too much variance and noise to the data that it would be likely make the data unreliable.

Discussion of Results and Comparison

The theoretical calculations correspond relatively well to the measured running times. The experimental results generally follow the same growth rate as the theoretical calculations, and can be modelled relatively acurrately with the growth rate found through calculations. The implementation of `HugeInteger` is significantly worse than the implementation `java.math.BigInteger`. While the growth rates match relatively well, the actual running times for `java.math.BigInteger` is multiple orders of magnitude less in all operations except comparison, which still had a notably smaller running time.

An improvement that could be made to the addition operation that would improve the running time and memory complexity of the both addition and subtraction operations would be to increase the size of the initialized array to equal the maximum possible number of digits and then remove leading zeros at the end, which is how the process is implemented in the multiplication operation. The current implementation removes leading zeros, but also needs to create a new `HugeInteger` with an extra digit when the result of the addition requires an extra digit, increasing the memory complexity significantly and also likely increasing the running time as there is an unneeded for loop that the operation must iterate through.

Another potential improvement would be to implement Karatsuba multiplication or an alternative multiplication algorithm that has a lower $\Theta(n)$. A recurvise algorithm like Karatsuba multiplication would increase the memory complexity of the operation, but would significantly increase the running time, as can be seen in the running times for `java.math.BigInteger`, which uses Karatsuba multiplication and Toom-Cook multiplication for multiplication of large integers.

Discussions with Peers and Sources of Inspiration

- The implementation of `java.math.BigInteger` in `Java 13.0.1`