# 2SI4 Lab Report

Abdelmoniem Hassan 400248003 - L04

McMaster University

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is our own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario.

## Data Structures and Algorithms

This implementation of the HugeInteger class stored each integer in an integer array, with each index representing a unit place of the HugeInteger. The only other class variable is the HugeIntegerSign that stores the sign of the integer as a char data type.

Comparison Method

The compareTo method uses a series of if statements to determine which HugeInteger is larger. The program first checks if the arrays containing both integers are equivalent and if their respective signs are equal and then returns zero to indicate that the HugeIntegers are equal, if that condition is not met the method then performs a series of checks the check if the signs of each HugeInteger are different and then returns one if the local HugeInteger is larger or negative one if h is larger. If none of those conditions are met then the signs of each HugeInteger are the same, in that case the method iterates through each HugeIntegers array from the most significant unit to least significant to determine which HugeInteger has a larger magnitude and then returns one or negative one depending on the case.

Addition Method

The addition method begins with a series of if statements to simplify the addition operation to simply adding a smaller positive integer to a larger positive integer. This is achieved by routing some operations through the subtraction method and the absoluteValue method which returns a new HugeInteger which is the absolute value of the HugeInteger passed in. By doing this cases of positive and negative numbers added together are simplified to a subtraction and cases of two negative numbers being added together are simplified to an addition of positive integers with the result being converted to a negative number. The rest of the method operates based on the addition of two positive integers.

Six variables are initialized to make the addition process easier

- Int[] solution - This integer array which initialized to be slightly larger than the one storing the longer integer stores the solution as the calculation takes place.
- Int shorterLen, longerLen - Both of these store the length of each HugeInteger.
- Int[] longerInt, shorterInt - Both of these arrays are references to each HugeInteger Array
- String strSolution - An empty string that will be used to convert the solution stored in an array to a HugeInteger.

The first for loop in the addition method handles the addition of the overlapping unit places. The for loop iterates from zero to the length of the shorter HugeInteger. It takes the sum of both numbers in that specific unit place, takes the mod of that sum by 10 and adds it in the equivalent unit place in the solution array. The sum is then divided by ten and is stored in a more significant unit place, this gives us the carry of the operation.

The next for loop handles the non overlapping units from the larger number, it simply adds the number, and the already existing value mod 10 to the equivalent unit place and the sum divided by ten to the next most significant unit place.

The solution array is then converted to a string, by converting each number to a string and concatenating it to the strSolution. A new HugeInteger is then initialized using the strSolution and that is the object that is returned.

Subtraction Method

Similar to the addition method the subtraction method begins with a series of if statements to simplify the subtraction process to simply subtracting a smaller positive integer from a larger positive integer. This is again achieved by using the absoluteValue method and routing some operations to addition.

Six variables are initialized to make the subtraction process easier

- Int[] solution - This integer array which initialized to be slightly larger than the one storing the longer integer stores the solution as the calculation takes place.
- Int shorterLen, longerLen - Both of these store the length of each HugeInteger.
- Int[] longerInt, shorterInt - Both of these arrays are references to each HugeInteger Array
- String strSolution - An empty string that will be used to convert the solution stored in an array to a HugeInteger.

The first for loop handles both the overlapping and nonoverlapping subtraction cases. The loop iterates from zero to the length of the longer integer. While i is strictly smaller than the shorterLen variable the loop is handling the overlapping case. There are two cases within the overlapping case when there needs to be a carry for the subtraction and when the subtraction can just take place. During the no carry case the value from the smaller integer is subtracted from the longer integer and placed in the equivalent unit place in the solution array. During the carry case the next significant unit place in the longerInteger is reduced by one and our current unit place is increased by 10, then the value from the smaller integer is subtracted from the larger integer and stored in the equivalent unit place in the solution array. During the non overlapping case the value in the solution array is simply increased by the value in the longer integer array.

The next for loop handles an edge case where the a carry is taken from a zero value, this is done by reducing the next significant unit in the solution array by one and increasing the current unit by 10.

The solution array is then converted to a string, by converting each number to a string and concatenating it to the strSolution. A new HugeInteger is then initialized using the strSolution and that is the object that is returned.

Multiplication Method

The multiplication method begins by checking if any of the HugeIntegers are zero. The multiplication arithmetic by designating the integer with more units as the longerInt and the other as shorterInt. The sign of the result is handled after the arithmetic.

Six variables are then initialized to make the multiplication process easier

- Int[] solution - This integer array which initialized to be slightly larger than the one storing the longer integer stores the solution as the calculation takes place.
- Int shorterLen, longerLen - Both of these store the length of each HugeInteger.
- Int[] longerInt, shorterInt - Both of these arrays are references to each HugeInteger Array
- String strSolution - An empty string that will be used to convert the solution stored in an array to a HugeInteger.

This multiplication method takes the grade school approach of multiplication where you iterate through each unit in the smaller number and multiply it by each unit in the larger integer. This is achieved by a nested for loop where the outer loop iterates from zero to the length of the shorter integer and the inner loop iterates from zero to the length of the longer integer. The value from the shorterInt is then multiplied by the value from the longerInt modded by ten and stored in the appropriate units placed in the solution array. The carry is the result of the multiplication divided by 10, and is stored in the next significant unit.

## Theoretical Analysis of Running Time and Memory Requirement

Comparison Method

The comparison method had an average running time of Θ(1) as 14 out of the 16 cases are resolved using conditional statements. The other two cases require iteration through each of the elements in each array which gives us a worst case running time of Θ(n). This method has a space complexity of Θ(1).

Addition Method

The addition method has an average and worst case running time of Θ(n). The addition method only contains independent loops which at most run the length of the longer integer. It also calls absoluteValue() and the constructor which both run in linear time. This method has a space complexity of Θ(n).

Subtraction Method

Similar to the addition method, the subtraction method has an average and worst case running time of Θ(n).This method has a space complexity of Θ(n).

Multiplication Method

The multiplication has an average and worst case running time of $\Theta(n^2)$. The multiplication method has two nested for loops which each run n times. This method has a space complexity of $\Theta(n)$.

Memory Analysis

The HugeInteger values are stored in an integer array meaning for each n we require 4 bytes we then require another 2 bytes to store the sign of the HugeInteger as a char. This means our equation for memory requirement to store the HugeInteger is S(n) = 4n + 2.


**Test Procedure**

There are 6 main cases (and their combinations) that the code must pass to demonstrate complete functionality.

- Different Sizes
- Same size
- Different signs
- Same sign
- With Carry/Borrow
- Without Carry/Borrow

By testing each method for each combination of these cases we can say that it is fully functional and meets the specification. The values can be compared to an already existing library such as bigInteger to confirm the validity of the values.
The outputs from this implementation of HugeInteger all passed the required test cases and are up to specification and all input conditions were checked.

While debugging edge cases can be difficult at times, stepping through the code using a debugger allows you to identify issues and resolve them quickly.

## Experimental Measurement, Comparison and Discussion

Process

The program runs for MAX_RUN times for each n, times the entire process and divides the result by MAX_RUN to get the average.

Parameters

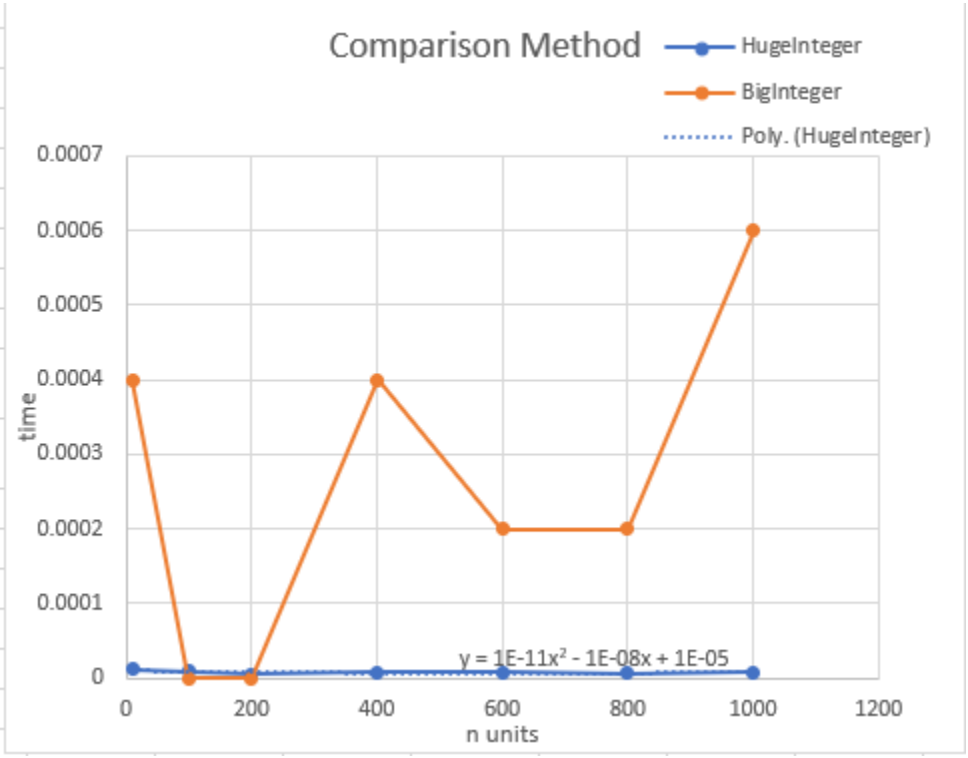
```
int MAXNUMINTS = 100;
        int MAX_RUN = 50;
        int[] n = {10,100,200,400,600,800,1000};
```

Data

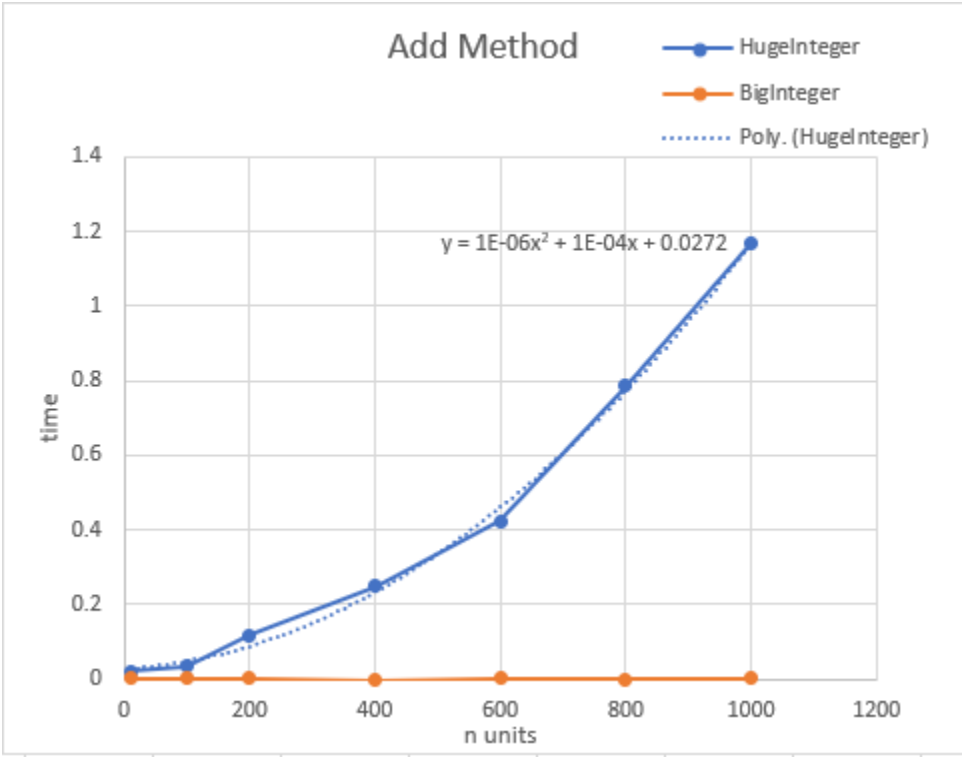
| HUGEINTEGER | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| add | time | | subtract | time | | multiply | time | | compareTo | time |
| 10 | 0.0216 | | 10 | 0.0192 | | 10 | 0.0272 | | 10 | 0.000011 |
| 100 | 0.033 | | 100 | 0.0594 | | 100 | 0.1168 | | 100 | 0.000009 |
| 200 | 0.1164 | | 200 | 0.052 | | 200 | 0.3322 | | 200 | 0.000005 |
| 400 | 0.2484 | | 400 | 0.0888 | | 400 | 1.2466 | | 400 | 0.000008 |
| 600 | 0.423 | | 600 | 0.4936 | | 600 | 2.4406 | | 600 | 0.000007 |
| 800 | 0.7866 | | 800 | 0.2338 | | 800 | 4.3082 | | 800 | 0.000006 |
| 1000 | 1.169 | | 1000 | 0.344 | | 1000 | 6.644 | | 1000 | 0.000008 |
| | | | | | | | | | | |
| BIGINTEGER | | | | | | | | | | |
| add | time | | subtract | time | | multiply | time | | compareto | time |
| 10 | 0.0008 | | 10 | 0.0016 | | 10 | 0.0014 | | 10 | 0.0004 |
| 100 | 0.0004 | | 100 | 0.0006 | | 100 | 0.0022 | | 100 | 0 |
| 200 | 0.0006 | | 200 | 0.0004 | | 200 | 0.002 | | 200 | 0 |
| 400 | 0 | | 400 | 0.0016 | | 400 | 0.0044 | | 400 | 0.0004 |
| 600 | 0.0006 | | 600 | 0.0002 | | 600 | 0.0022 | | 600 | 0.0002 |
| 800 | 0.0002 | | 800 | 0.0006 | | 800 | 0.0238 | | 800 | 0.0002 |
| 1000 | 0.0006 | | 1000 | 0.0004 | | 1000 | 0.0312 | | 1000 | 0.0006 |

## Operations Graphed in comparison to BigInteger

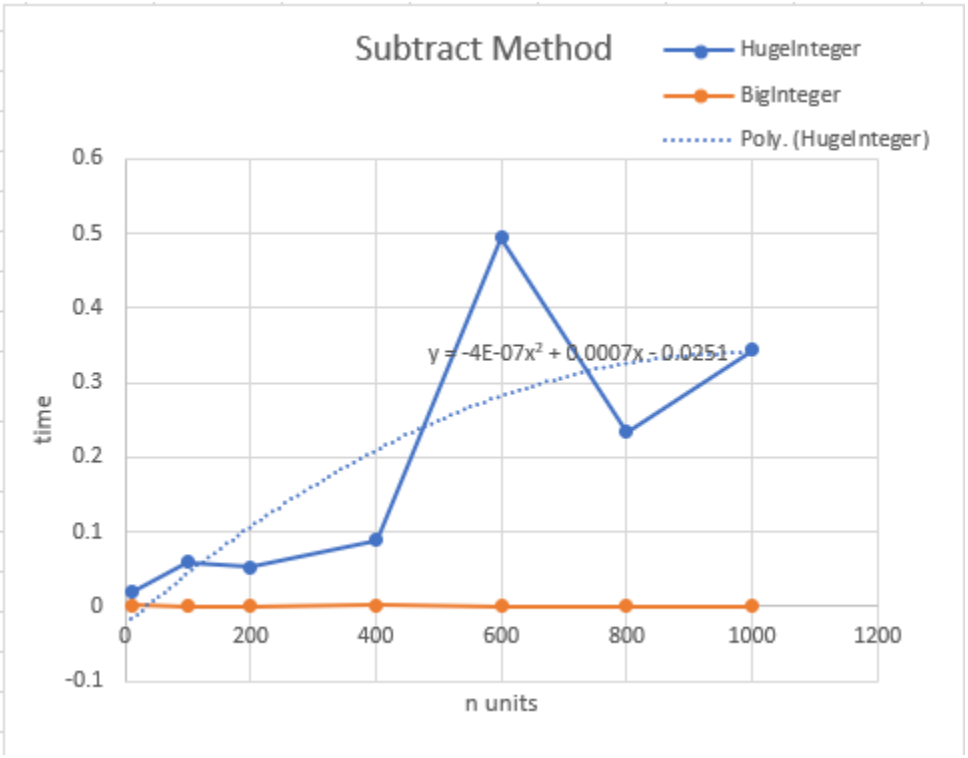
### Comparison Method



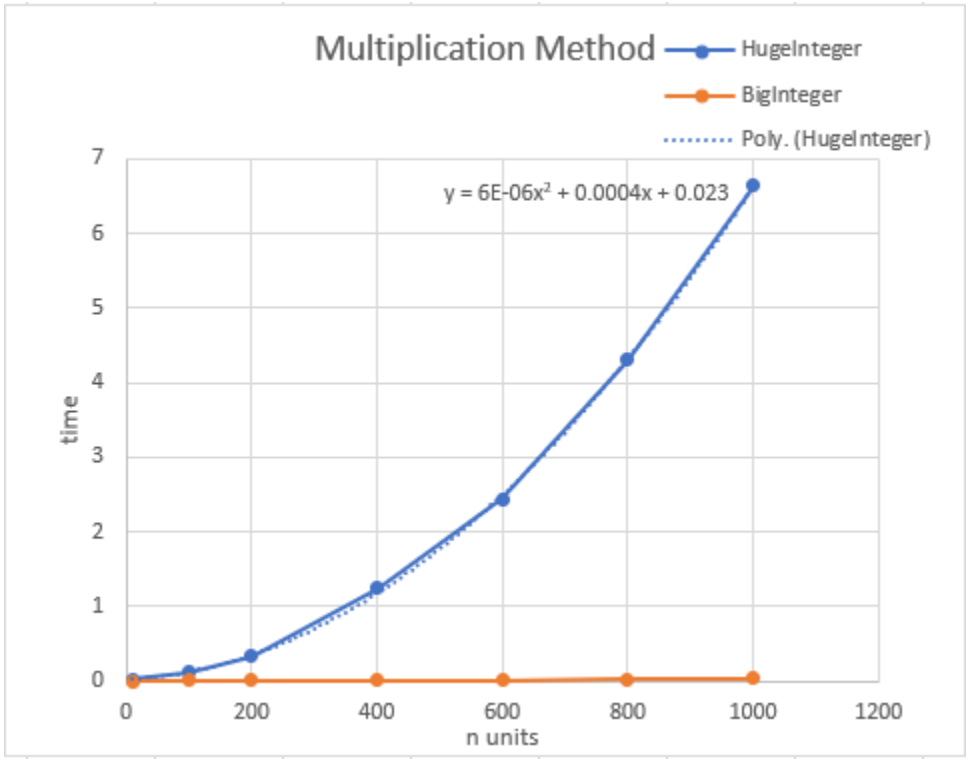

### Addition Method



Subtraction Method

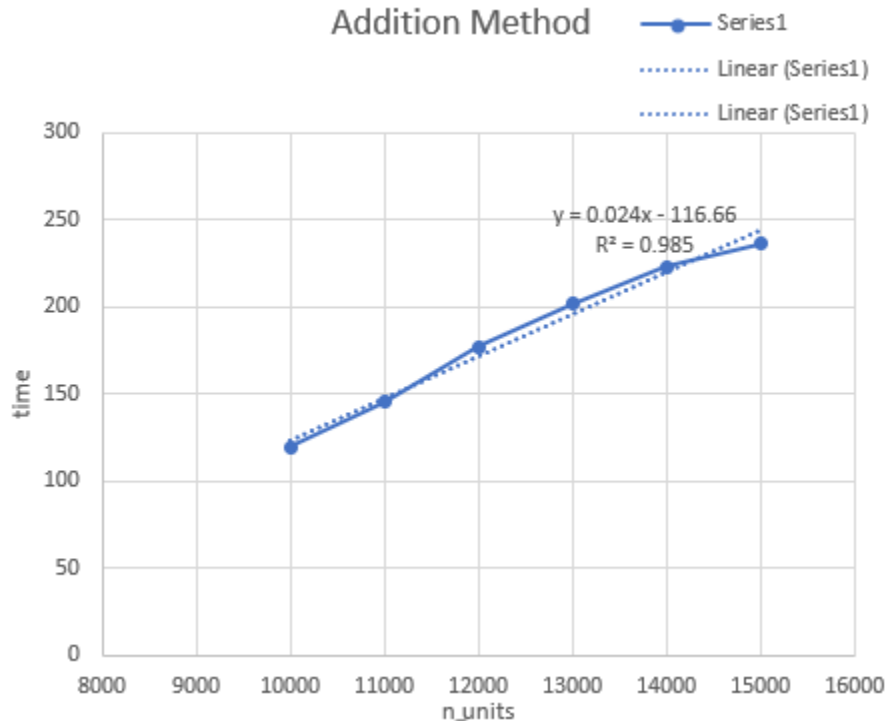



Multiplication method



**Discussion of Results and Comparison**

The measured results for compareTo and multiplication line up very accurately with the theoretical results. The subtraction and addition methods seem to be on the order of $n^2$ this could be due to the relatively low n values used and the large factor of the n value if you were to calculate T(n), However as the n value actually increases to infinity the graph would be expected to approach the theoretical result of $\Theta(n)$.




(results for large n, that would have taken too long to run for all the methods)

```
int MAXNUMINTS = 100;
int MAX_RUN = 10;
int[] n = {10000, 11000, 12000, 13000, 14000, 15000};
```

This implementation of HugeInteger is significantly slower than BigInteger for arithmetic operations, however compareTo() performance is very similar to its BigInteger counterpart. Improvements to the Addition and Subtraction methods would require a bitwise approach similar to that of BigInteger. The multiplication would also require a shift to a bitwise approach and a more complex algorithm to reduce the complexity from $\Theta(n^2)$ to something smaller. Memory wise the HugeInteger array could be converted to a byte array to reduce the memory for storing n units by 4 times.

## Appendix

Source code of runtime analysis.

```
import java.math.BigInteger;
public class RunTime {
    public static BigInteger random(int n){
        int[] rand = new int[n];
        String Rand= "";
        for (int i = 0; i<n ;i++){
            if (i == 0){
                rand[i] = Math.abs((int)(Math.random()*(9)) + 1);
            } else {
                rand[i] = Math.abs((int)(Math.random()*(10)));
            }
        }
        for (int i = 0;i<rand.length;i++){
            Rand += Integer.toString(rand[i]);
        }
        BigInteger x = new BigInteger(Rand);
        return x;
    }
    public static void main(String args[]){
        int MAXNUMINTS = 100;
        int MAX_RUN = 10;
        int[] n = {10000, 11000, 12000, 13000, 14000, 15000};
        int x = 0;

        //System.out.println(ANSI_PURPLE + "FINAL TOTAL MARK IS " + totalMark + ANSI_RESET);
        long startTime, endTime;
        double runTime=0.0;

        for (int i= 0;i<n.length;i++){
            runTime=0.0;
            for (int numInts=0; numInts < MAXNUMINTS; numInts++){
            HugeInteger number = new HugeInteger(n[i]);
            HugeInteger number2 = new HugeInteger(n[i]);
            HugeInteger number3;
            //BigInteger number = random(n[i]);
            //BigInteger number2 = random(n[i]);
            //System.out.println(number.toString());
            //System.out.println(number2.toString());
            startTime = System.currentTimeMillis();

            for(int numRun=0; numRun < MAX_RUN; numRun++){
                number3 = number2.add(number);
            }
            endTime = System.currentTimeMillis();
            runTime +=(double) (endTime - startTime)/((double) MAX_RUN);
            }

            runTime = runTime/((double)MAXNUMINTS);
            System.out.printf("%f\n", runTime);
        }
    }
}
```