COMP ENG 2SI4

Lab 1 & Lab 2

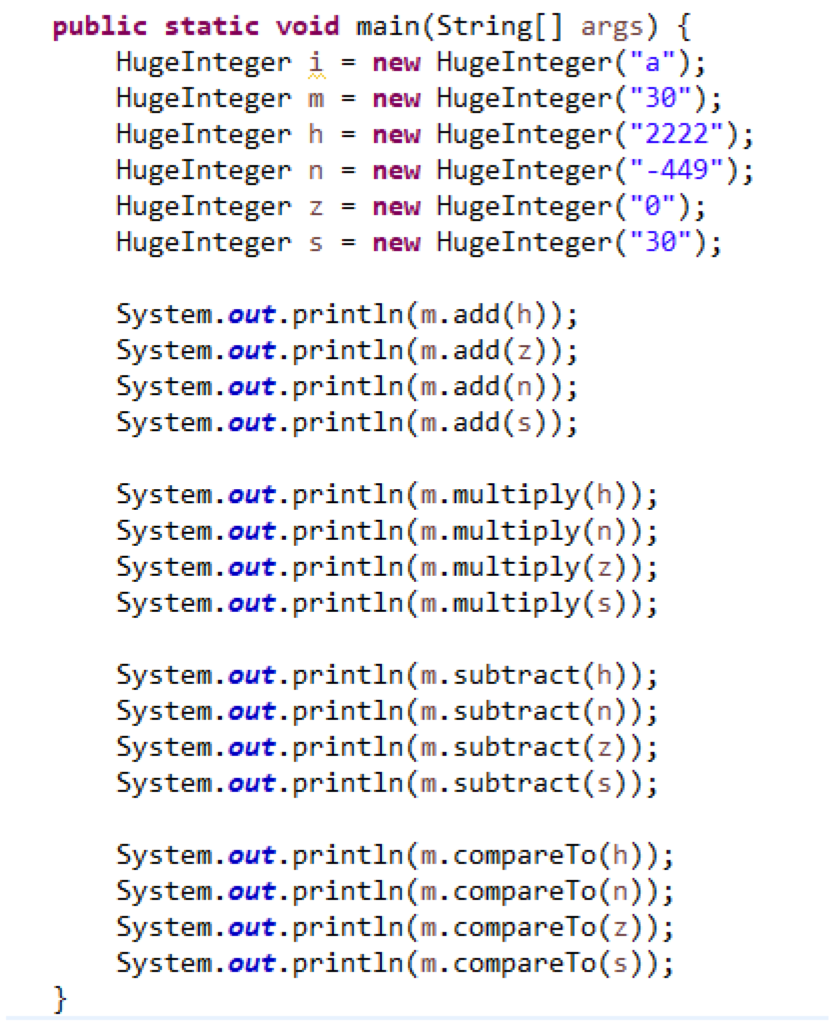
YIMING CHEN
cheny466@mcmaster.ca

1. Data Structure: In this lab, for the main class HugeInteger I used a single int array for its filed. The function of this int array is to store the huge integer number by splitting the whole number into several digits. For index 0 of each int array, I used a specific number 1/-1 to determine the sign of the HugeInteger and this sign, for sure, will be correctly printed as +/- in the output. So in my codes, you can find when countering loops in calculation, the start index is usually 1 instead of 0.

   In this lab, the requirement is to solve for 4 main methods. I united the addition and subtraction together so in the code the subtraction part is simple because it has been covered in the addition part.
   For addition part, to make it clear, I divided it into 4 circumstances: +/+, +/-, -/+, -/-, and under each circumstance it has 2 cases: this has more digits or h has more digits. For the output, I first chose to create an empty array. The length of the array is determined by the longer input +1 to store the carry.
   For multiplication, the basic idea is just the same as the previous two methods, but there is one point I want to mention: because each time we need to shift the calculation one digit left so I used double for loop and for index I chose to +1 to adjust it to the proper digit.
   And for the last method, compareTo, it is easy to implement. The codes which I analyze all the circumstances one by one, from the same two digits. Then for the obvious differences which occur in different signs. And then to compare the length and the most complex circumstance is that the two number has the same length and most of the digits are the same. For this case I used a for loop to compare each digit. And after finishing the codes, I found that this method can be easily done by subtracting both to get the result, but I think it will be less complicated to finish like what I wrote now.

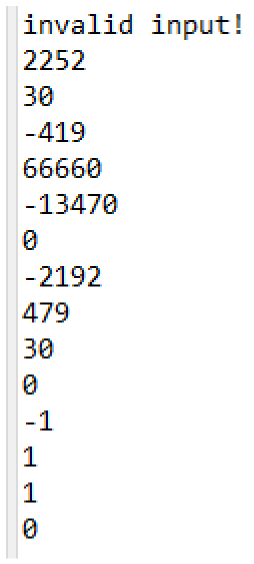
2. Theoretical Analysis of Running Time and Memory Requirement
   We need to allocate 4 bytes * array.length to store a huge integer.
   Run-time and memory using the knowledge learned from class.
   Addition: $T(n)= \Theta(n)+c$ and $S(n)=S(n)$
   Subtraction: $T(n)= \Theta(n)+c$ and $S(n)=S(n)$
   Multiplication: $T(n)= \Theta(n^2)+c$ and $S(n)=S(n)$
   Comparison: $T(n)= \Theta(\log n)+c$ and $S(n)=S(n)$

3. Test Procedure
   Test Cases:



```java
public static void main(String[] args) {
    HugeInteger i = new HugeInteger("a");
    HugeInteger m = new HugeInteger("30");
    HugeInteger h = new HugeInteger("2222");
    HugeInteger n = new HugeInteger("-449");
    HugeInteger z = new HugeInteger("0");
    HugeInteger s = new HugeInteger("30");

    System.out.println(m.add(h));
    System.out.println(m.add(z));
    System.out.println(m.add(n));
    System.out.println(m.add(s));

    System.out.println(m.multiply(h));
    System.out.println(m.multiply(n));
    System.out.println(m.multiply(z));
    System.out.println(m.multiply(s));

    System.out.println(m.subtract(h));
    System.out.println(m.subtract(n));
    System.out.println(m.subtract(z));
    System.out.println(m.subtract(s));

    System.out.println(m.compareTo(h));
    System.out.println(m.compareTo(n));
    System.out.println(m.compareTo(z));
    System.out.println(m.compareTo(s));
}
```

Outputs:



```
invalid input!
2252
30
-419
66660
-13470
0
-2192
479
30
0
-1
1
1
0
```

For constructors: it is important to judge whether the content of the input String is a number or not. So I decided to create a HugeInteger("a") firstly to see if the constructor could distinguish it and it succeeded. And the other circumstances can be easily checked when implementing the methods.

For the rest of the methods, I chose a 2-digits positive number as a basic sample. And for each method I chose a positive number with more digits, a negative number, zero and the same number to do the test. It has covered almost all the possibilities and succeeded.

The reason to choose these cases is that. For a positive number with more digits it can verify whether the carry system works well and when taking consideration on the results with more digits, whether it could successfully deal with it. As for the negative number it is as well critical to test it not only for the positive and negative addition but also when dealing with the subtraction including negative number, the negative number should be turned into positive and apply addition and it succeeded with the codes. It is also important to take 0 into consideration, because under many circumstances 0 can be considered as a special case. And finally, I chose the same number. For compareTo method, it can be thought as a requirement because it should return 0 if the two numbers are equal. And for other cases, especially for the subtraction case, the same number should focus on the output: to see if it could correctly print the number 0. And all the test cases are passed with my codes.

Some Discussion: because in this experiment I chose to use +1/-1 and the index 0 of each array, so I went into some problem. When dealing with the subtraction, it is hard to control the correct output because if it has less digits, the array will be like [-1,0,0,0,0,1] and the toString() method should return -1 but at first it returns –11. It took me a lot of time to fix this bug. At last I chose to analyze more cases in the toString() method to get more precise results.

4. Experimental Measurement, Comparison and Discussion:
   Sample Code:

```
int MAXNUMINTS = 500;
int MAXRUN = 100;
```

the runtime of the methods will be described as the following format using my own code.
(addition; subtraction; multiplication; compareTo)

n = 10: (3.800000000000001E-4; 3.800000000000001E-4; 5.400000000000001E-4; 1.7999999999999998E-4)

n = 100: (0.0014800000000000008; 0.0014800000000000008; 0.007299999999999935; 2.6000000000000003E-4)

n = 500: (0.004539999999999991; 0.004539999999999991; 0.1422000000000003; 5.200000000000002E-4)

n=1000: (0.007159999999999936; 0.007159999999999936; 0.5571400000000044; 0.0010600000000000004)

n=5000: (0.032059999999999686; 0.032059999999999686; 2.155700000000003; 0.003180000000000002)

n=10000: (0.06138; 0.06138;*; 0.0050999999999999795)

Note: * means the value is too large and takes too long to run

5. Discussion

   The measured running time is basically corresponding to the theoretical calculation, which can be easily found from the data shown above.

   From the theoretical calculation, the runtime should be:
   compareTo() < add() = subtract() < multiply()
   and the result is the same.

   When calculation sample 10000 for multiply because the calculation sample is too big for my computer to calculate. It took so long time so I used * to represent the failure for such a huge integer.

   Compared to the java.lang.BigInteger I think the compareTo part and addition as well as subtraction are basically meet the speed of the ones in JDK. However, because of the implementation of nested for loop in the multiplication part, it is relatively slower. But when dealing the number digits less than 10000, it performs well I think.