Java Data Structure Homework 2 – Coding Part

Instructions: ***Please make sure that you push your code to your repository after complete your assignments*!**

Please be aware of the deadline. You can still submit your codes after the deadline, but late homework **will NOT be graded**.

You are freely consult any Internet resource included but not limited to Google, Stack Overflow, etc. But please do cite them when you use part of the code or are inspired by their solution. I take plagiarism really seriously.

Note that you are **encouraged to work together**. You are also welcome to work individually. As a note, in my teaching experience, working as a group can significantly improve your understanding in the class material and enhance your ability to present your ideas, so please spend some time to say hello to your classmates.

While you can discuss your thoughts with other students, for this assignment you need to submit your own solution, i.e., **you needs to submit solution that is written by yourself**.

In general, I will use the following guidelines when reviewing your solution:

- **Coding style**. You will get 5 points deduction for bad or unrecognisable variable names, comments, etc.
- **Robustness**. ANY buggy code will receive **zero** on that specific problem, so make sure you test your program before you submit.
- **Instructions**. Please follow the instruction on each problem, do not overkill any problem.

The best way to check your program is actually run the program by yourself. As long as the output is correct, it is fine.

Problem 1. BigInt.java

In Java, the int data type can only represent integers from -2,147,483,648 to 2,147,483,647. This isn't big enough to store very large numbers—for example, the population of Earth (approximately 7.53 billion as of 2017)—or to compute n! for large values of n.

Representing integers using an array of digits

Each BigInt object will use an arrays of integers, where each element of the array represents a single digit of the integer.

We will design the class to handle non-negative integers with up to 20 digits, and thus the array will have a length of 20.

For example, the integer 57431 would be represented using this array:

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 7, 4, 3, 1}
```

Note:

- Each element of the array is a single digit (i.e., an integer from 0-9).
- The last element of the array represents the least significant digit of the integer (i.e., the rightmost digit).
- The first non-zero element (if there is one) represents the most significant digit of the integer. (The one exception is when we are representing 0 itself. Its most significant digit is the same as its least significant digit, and it is a zero!).
- The remaining elements are leading zeros that are not significant digits.

As another example, the integer for 7.53 billion would be represented using the array

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 5, 3, 0, 0, 0, 0, 0, 0, 0}
```

BigInt.java template is provided to you, read reach field carefully with the comments on each top of the filed.

Important Note:

- You may not add any additional fields to the BigInt class.
- You may not use any variables of type long, any of Java's built-in classes for numbers (Long, BigInteger, etc.), or any of Java's built-in collection classes (e.g., ArrayList).

(a) Add your first additional constructor

Add an additional constructor with the following header:

```java
public BigInt(int[] arr)
```

It should use the contents of the array that is passed in as the basis of the new BigInt object. For example, we should be able to create a BigInt for 57431 as follows:

```
int[] arr = {5, 7, 4, 3, 1};
BigInt val = new BigInt(arr);
```

Hint:

- Validate the array that is passed in, which can have any length between 0 and SIZE. If the parameter is null, if the length of the array is greater than SIZE, or if any of the elements of the array don't make sense as digits, you should throw an *IllegalArgumentException*.

  Note: You may want to consider writing a private helper method that helps you to validate the input array, although doing so is not required. More generally, we encourage you to use a private helper method whenever doing so would help to simplify your code and/or your logic.

- Create a new digits array as we do in the original constructor, and use the input array arr to initialize the elements of the digits array and to determine the correct value for the numSigDigits field. Make sure that you don't include non-significant leading zeroes in the value that you compute for numSigDigits.

Important: Make sure that the digits field ends up referring to a new array of length SIZE. You should not make it refer to the array that is passed in.

(b) Add two methods useful for testing
You should now add the following methods:

- an accessor method called getNumSigDigits() that returns the value of the numSigDigits field. For example, running this test code:

  ```
  int[] arr = {0, 0, 0, 5, 7, 4, 3, 1};
  BigInt val = new BigInt(arr);
  System.out.println(val.getNumSigDigits());
  ```

  should print a value of 5.

- a toString() method that overrides the default toString() method (the one inherited from the Object class). It should return a string that can be used to print a BigInt object in the way that we would ordinarily write the corresponding integer—with no leading zeroes. For example, running this test code:

  ```
  int[] arr = {0, 0, 0, 5, 7, 4, 3, 1};
  BigInt val = new BigInt(arr);
  System.out.println(val);
  ```

  should print: 57431

(c) Add the remaining methods
You are now ready to implement the remaining methods of the class.

- public BigInt(int n)

  A constructor that creates a BigInt object representing the integer n. If n is negative, throw an IllegalArgumentException. (Note: You don't need to worry about n being too big, because all non-negative int values can be represented using fewer than 20 digits!)

  For example, the following statements:

  ```
  BigInt val = new BigInt(1234567);
  System.out.println(val);
  System.out.println(val.getNumSigDigits());
  ```

  should produce this output:

  ```
  1234567
  7
  ```

  You will need to figure out how to determine the individual digits of the int that is passed in, and we recommend that you use concrete cases to help you. For example, let's say that you wanted to determine the individual digits of 123:

  – What computation would allow you to determine just the units digit of 123?

  – Once you have determined the units digit, how could you compute a new integer that includes everything but the units digit? (In the case of 123, how could you go from 123 to 12?). Doing so will allow you to continue determining the remaining digits!

- public int compareTo(BigInt other)

  This method should compare the called BigInt object to the parameter other and return:

  – **-1** if integer represented by the called object is less than the integer represented by other

  – **0** if integer represented by the called object is equal to the integer represented by other

  – **1** if integer represented by the called object is greater than the integer represented by other

  The method should not modify the original objects. If the parameter is null, throw an IllegalArgumentException.

- public BigInt add(BigInt other)

  This method should create and return a new BigInt object for the sum of the integers represented by the called object and other. For example:

  ```
  BigInt val1 = new BigInt(1111111);
  BigInt val2 = new BigInt(2222);
  BigInt sum = val1.add(val2);
  System.out.println(sum);
  ```

should print: 1113333

The method should not modify the original objects.

Special Case Hint:

- what if parameter is null?*IllegalArgumentException.*
- what if result overflows? *ArithmeticExceptions*

- public BigInt mul(BigInt other)

  This method should create and return a new BigInt object for the product of the integers represented by the called object and other. For example:

  ```
  BigInt val1 = new BigInt(11111);
  BigInt val2 = new BigInt(23);
  BigInt product = val1.mul(val2);
  System.out.println(product);
  ```

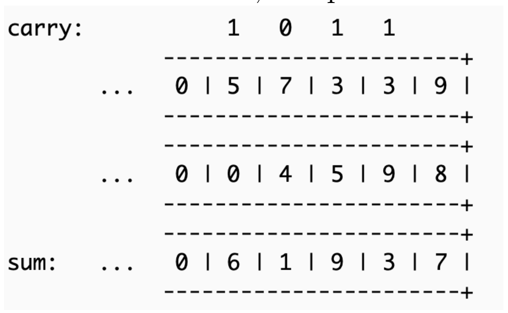
  should print: 255553

Here again, the method should **NOT** modify the original objects, and you should throw an IllegalArgumentException if the parameter is null and an ArithmeticException if the result overflows.

Since I am a nice person, here are some algorithm hints for you.

The algorithms for manipulating these big integers are simply computational versions of the procedures that we would use to add, compare, or multiply integers "on paper".

For example, to add two arrays of digits, we must go from **right to left** (i.e. **least significant digit to most significant digit)** adding the digits and keeping track of the carry. For example, to add 57339 to 4598, the process looks something like this:



```
carry:          1   0   1   1
                ----------------------+
      ...    0 | 5 | 7 | 3 | 3 | 9 |
                ----------------------+
                ----------------------+
      ...    0 | 0 | 4 | 5 | 9 | 8 |
                ----------------------+
                ----------------------+
sum:  ...    0 | 6 | 1 | 9 | 3 | 7 |
                ----------------------+
```

Note that addition will result in overflow (creating a number with more than 20 digits) if you need to add the digits in position 0 of the two arrays and if their sum plus any incoming carry value produces a value that is greater than or equal to 10.

We encourage you to:

- Go through a similar analysis using concrete cases to determine the algorithms for comparing and multiplying BigInt objects.

- Consider how you could use the numSigDigits field to assist you in the execution of your algorithms.

- For the sake of efficiency, consider writing a private helper method that checks if a BigInt object represents the number 0. If you know that one of the operands in a multiplication or addition expression is zero, how would that help you?