There are 1 questions for a total of 100 points.

---

(100) 1. The first programming assignment will involve working with large binary numbers and understanding running time behaviour of various algorithms operating on these numbers.

As you know by now that Java is an Object Oriented Programming Language and one important feature of object oriented design is **encapsulation**. Here, we can create program components where data can be bundled together with methods that act on the data and the internal working of the component is private to the component. Since our goal in the first assignment will be to do manipulations on large binary numbers, we will use encapsulation and define a java class related to large binary numbers. Let us call this java class **myBinaryNumber**. In order to maintain the spirit of encapsulation, the following rules will be strictly enforced in this assignment:

1. You are not allowed to make any changes in the file **myBinaryNumber.java**.
2. All binary numbers in your code (including any intermediate binary numbers) should be stored using the **myBinaryNumber** class. No exceptions will be allowed.

Here is the description of the class **myBinaryNumber**. You will also find some of the description as comments in the code.

- Constructor *myBinaryNumber(int n)*: This initialises an all 0's n-bit binary number.
- Constructor *myBinaryNumber(String S)*: This initialises a binary numbers represented by the string S. So, the length of the number is the size of the string.
- Public method *getSize()*: This returns the size of the binary number. That is the number of bits.
- Public method *getBit(int p)*: This returns the bit at position p from the least significant bit (LSB) side. That is if $p = 0$, it returns the LSB.
- Public method *setBit(int p, int b)*: This method sets the bit at position p (from LSB side) as b. Note that this method throws an exception if the inputs are not valid. The programmer using this method should make sure to handle the exception.
- Public method *printNumber()*: This prints the binary number.

We would like to do operations on binary numbers. So, basically do operations on objects of the type **myBinaryNumber**. For this, we have written an *abstract* class named **binaryOperations**. We have already given an implementation of binary addition as a helper code. However, there is an abstract method defined for multiplication and your main coding job in this assignment is to extend this abstract class into multiple classes that give implementation of the multiplication method. You are given four java files, each of which extends **binaryOperations**. What you are expected to write in these four files is given below:

1. **Mult1.java**: In this file, the multiplication method should be an implementation of the long multiplication idea discussed in class.
2. **Mult2.java**: In this file, the multiplication method should be an implementation of the naïve $O(n^2)$ Divide and Conquer algorithm discussed in class.
3. **Mult3.java**: In this file, the multiplication method should be an implementation of the $O(n^{\log_2 3})$-time Karatsuba algorithm discussed in class.
4. **Mult4.java**: In this file, the multiplication method should be an implementation of the $O(n^{\log_2 3})$-time Karatsuba algorithm discussed in class. However, the difference with Mult3.java is that the base case of the recursive program should not be when the numbers are single bits but when the number are of size ≤ 100. Moreover, when the numbers are of size ≤ 100, then the long multiplication algorithm should be called on those numbers to multiply them. You may reuse your code from Mult1.java in this file.

There is java file named **Test.java** that is provided to you to get started. You may use this to test your code. You will also need to experiment with very large inputs and examine the running time behaviour of these multiplication algorithms. Some of the questions that you should examine while experimenting are as follows:

1. We saw that long multiplication takes $O(n^2)$ time. That is, the running time should exhibit quadratic behaviour. Is this true in practice? If you plot running time (i.e., the actual system time) with respect to n, then do you observe a quadratic curve? Repeat these experiments with the other multiplication algorithms and even the addition algorithm that is provided.
2. Both long multiplication and naïve Divide and Conquer has $O(n^2)$ asymptotic running time. However, we mentioned that there are overheads associated with recursive programs which might make the Divide and Conquer recursive algorithm slower *in practice*. Do you indeed observe this behaviour?
3. The asymptotic running time analysis suggested that the Karatsuba algorithm should be better than the long multiplication algorithm. Do you observe this behaviour in practice? The asymptotic analysis suggests that there should be a large n after which Karatsuba should be better than long multiplication. Try finding this n experimentally.
4. Which is the best algorithm for multiplication in practice and why?

The above experimental work will be evaluated through a viva. So, please be prepared to give evidence that you have given adequate thought to these questions and performed appropriate experiments.

Evaluation: Evaluation of homework consists of the following two components:

1. Submission component (75 points): Your code will be checked for correctness and running time. This will partially be done using an automated scripts. So, please make sure that you strictly follow these instructions:
   - Write all your code in a directory named ⟨Your entry number⟩.
   - For submission, create a zip file named ⟨Your entry number⟩.zip of your directory and submit. Details of where to submit will be sent in some time.
2. Viva component (25 points): We will hold all the labs during an evaluation week and you will be expected to attend the lab. During the lab, the following will be done:
   - You will be asked to make a small addition in your program to check your understanding.
   - The TA will evaluate the experimental part described at the top of this page.