In this exercise we study divide and conquer algorithms.

Excercise 4a. (Implement MinMax algorithm, 0,5p)

Implement class for finding both the positions of minimum and maximum element of the given collection (or array), e.g.

```
public class MinMax {
    MinMax(Comparable[] array);
    public int getMax();
    public int getMin();
    public int getComparisons();
    public void minmax();// min and max element locations (indexes) are returned
    public void minmax2();
}
```

where getMin() gives the **index** of the minimum element in the given array (in constructor), getMax() gives the **index** of maximum element is to be stored, in constructor we given the integer array where those min and max elements are to be found.

Your first implementation, minmax2() should be a straightforward for-loop method using two if-statements with calls to compareTo() method and if necessary updating the min and max storages.

Then implement the divide-and-conquer version of the minmax() method (=copy it from the slides). Check that this algorithm works. Then calculate the number of comparisons needed for this divide-and-conquer version of the algorithm. What is the relationship between the number of comparisons needed to find min and max values and the size of the array? And compare that number of comparisons to the simple minmax implementation.

After you have implemented your minmax functions, you can test it using the following test bench.

```
public class Lab04 {
    static private Integer[]    collection;
    static private SecureRandom rnd = new SecureRandom();

    /* create a random unsigned value array with lenght of n */
    static void initTestSeq(int n) {
        /* first create the collection of strings */
        collection = new Integer[n];
        for (int i = 0; i < n; i++) {
            collection[i] = rnd.nextInt();
        }
    }

    /* print out the sequence */
    static void printTestSeq() {
        for (int i = 0; i < collection.length; i++) {
            System.out.format("%1$02d: %2$11d\n", i, collection[i]);
        }

        System.out.println();
    }

    private final static int N = 10;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int max, min;

        initTestSeq(N); printTestSeq();

        MinMax minmax = new MinMax(collection);
        minmax.minmax2();
        System.out.println("Brute Force minmax search");
        System.out.println("Min index " + minmax.getMin() + ", max index " + minmax.getMax());
        System.out.println("Number of comparisions " + minmax.getComparisons());

        minmax = new MinMax(collection);
        minmax.minmax();
        System.out.println("Divide and Conquer minmax search");
        System.out.println("Min index " + minmax.getMin() + ", max index " + minmax.getMax());
        System.out.println("Number of comparisions " + minmax.getComparisons());
    }
}
```

Operation count is one measure of the complexity if we assume that the comparison is the most complex operation of the algorithm. Your simple implementation should be able to cope with 2*n comparisons (n is the length of the input array). Verify that by counting the number of comparisons needed to find min and max values related to the size of the array (n).

Exercise 4b. (Optional, Implement divide-and-conquer integer multiplication algorithm, 0,5p)

Complete the given class Bignum which is able to multiply two positive integers together. Numbers in the given Bignum class are represented by a byte array. Therefore there is no practical limitation to the length of the number represented. We use byte data type in order to same memory space without any compromise on execution speed.

Your task is to implement the `mulBigNum()` method (using the divide-and-conquer method). Every time you multiply two (single-digit numbers, the base case), you need to increment class variable `mulCounter` in order to check the total number of multiplications. You can use the following test application to test the operation of your implementation

```
public class Lab04b {
    static final String AB = "0123456789";
    static SecureRandom rnd = new SecureRandom();


    /* create a random string array with lenght of n */
    static String createTestNumber(int n) {
        StringBuilder sb = new StringBuilder(n);
        for (int i = 0; i < n; i++) {
            int c;
            if (i > 0)  // ensure that the first digit is not '0'
                c = rnd.nextInt(AB.length());
            else {
                c = 1 + rnd.nextInt(AB.length()-1);
            }
            sb.append(AB.charAt(c));
        }
        return sb.toString();
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Bignum  n1, n2, result;
        int     n;
        String  s1, s2;

        try {
            for (int i = 1; i < 30; i++) {
                s1 = createTestNumber(i); n1 = new Bignum(s1); System.out.format("%1$30s * ", n1.toString());
                s2 = createTestNumber(i); n2 = new Bignum(s2); System.out.format("%1$30s = ", n2.toString());

                result = n1.mulBigNum(n2);
                n = result.rclMulCounter();

                System.out.format("%1$60s (%2$d)\n", result, n);
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Check that your implementation produces correct results, you can use web calculator https://defuse.ca/big-number-calculator.htm to check the correctnes of the results.

Excercise 4c. (Optional, Implement defective chessboard tiling, 0,5p)

Implement function

```
void tileBoard(int tr, int tc, int dr, int dc, int size)
```

where tr and tc mark the left-upper point (row, column) of the chessboard and size = 2^k, and dr and dc give the row and column index of the defect. The number of tiles needed to tile the defective chessboard is $(size^2-1)/3$. Function `tileBoard()` represents these tiles by the integers 1 through $(size^2-1)/3$ and labels each nondefective square of the chessboard with the number of the tile that covers it.

Implement the `tileBoard()` function and test it with the initial call `tileBoard(0,0,dr,dc,size)` where `dr` and `dc` marks the defective tile and size = 2^k where k=1…6.

Output of the program should be something like this

```
Enter k, board size is 2^k
k should be in the range 0 through 6
3
Enter location of defect
2 2
   3   3   8   8  24  24  29  29
   3   0   2   8  24  23  23  29
  13   2   2  18  34  34  23  39
  13  13  18  18   1  34  39  39
  45  45  50   1   1  66  71  71
  45  44  50  50  66  66  65  71
  55  44  44  60  76  65  65  81
  55  55  60  60  76  76  81  81
```