First mid term lab exam – PRG – ETSINF – Academic Year 2016-17
April 10th, 2017 – Duration: 1 hour

Notice: The maximum mark of this exam is 10 points, but its weight in the final grade is **0.8 points**.

NAME:	LAB GROUP:

1. [4 points] The method `isPrefix( String, String )` implemented in the second lab practice returns `true` if the first string is prefix of the second one, otherwise it returns `false`.

   What to do: complete the following recursive method `countAppearances( String, String )` for returning how many times a non-empty string `a` appears as a substring in another string `b`. Some examples:

```
countAppearances( "ab", "aaba" ) should return 1
countAppearances( "aa", "abac" ) should return 0
countAppearances( "aa", "aaaa" ) should return 3
```

```
/** Precondition: a.length() > 0 */
public static int countAppearances( String a, String b )
{
    if ( a.length() <= b.length() ) {
        if ( isPrefix( a, b ) ) {
            return /* TO BE COMPLETED */;
        } else {
            return /* TO BE COMPLETED */;
        }
    } else {
        return 0;
    }
}
```

   Remember that the method `substring(int)` of the class `String` from the Java library returns an object of the class `String` representing a substring of an existing one. For instance, if `str` is an existing string, then `str.substring(i)` returns another object of the class `String` containing the characters from the one at position `i` in `str` to the end of `str`.

Solution:

```
/** Precondition: a.length() > 0 */
public static int countAppearances( String a, String b )
{
    if ( a.length() <= b.length() ) {
        if ( isPrefix( a, b ) ) {
            return 1 + countAppearances( a, b.substring(1) );
        } else {
            return countAppearances( a, b.substring(1) );
        }
    } else {
        return 0;
    }
}
```

2. [2 points] **What to do:** complete the following method to attain it returns an array of integers of size `n` representing an instance for the **worst case** of the *insertion sort* algorithm. But with the additional condition that the difference between any two consecutive values should be greater than or equal to 2.

```
private static int[] worstInsertionCase( int n )
{
    int[] a = new int[n];
    for( /* TO BE COMPLETED */ ) {
        a[i] = /* TO BE COMPLETED */;
    }
    return a;
}
```

Solution:

Worst case for the *insertion sort* algorithm is when the values in the array to be sorted are sorted in the reverse order, i.e., if the goal is to sort the array in ascending order then the values are in descending order, and viceversa. Next you have two possible solutions for completing the `for` loop. The array `a` is the generated array returned by the method.

The following conditions are fulfilled. $\forall$`i`, `0` $\leq$ `i` $<$ `a.length - 1`, `a[i] - a[i + 1]` $\geq$ `2`.

```
// First possible solution
for( int i = 0; i < a.length; i++ ) {
   a[i] = t - (2 * i) - 1;
}

// Second possible solution
for( int i = 0, j = t - 1; i < a.length; i++, j -= 2 ) {
    a[i] = j;
}
```

3. [4 points] The *insertion sort* algorithm is available as a method in the class `MeasurableAlgorithms` with the following profile:

```
public static void insertionSort( int[] a )
```

The following methods for creating arrays are implemented in the class `MeasuringSortingAlgorithms`:

- `private static int[] createRandomArray( int n )`
  that returns an array of `n` randomly selected integers in the range $[0, n-1]$.
- `private static int[] createArraySortedInAscendingOrder( int n )`
  that returns an array of `t` integers sorted in ascending order.
- `private static int[] createArraySortedInDescendingOrder( int n )`
  that returns an array of `t` integers sorted in descending order.

Given the following incomplete code snippet from class `MeasuringSortingAlgorithms` corresponding to method `measuringWorstCaseOfInsertionSort()`, a method created specifically for measuring empirically the temporal behaviour of the *insertion sort* algorithm:

```
// Constants for measuring
public static final int MAX_SIZE = 10000, MIN_SIZE = 1000;
public static final int STEP_OF_SIZE = 1000, REPETITIONS = 200;
public static final double NMS = 1e3;  // ratio microseconds / nanoseconds

public static void measuringInsertionSort()
{
    long ti = 0, // Initial timestamp
         tf = 0; // Final timestamp
```

```
        // Print header
        System.out.printf( "# InsertionSort. Time in microseconds\n" );
        System.out.printf( "#   Size        Worst \n" );
        System.out.printf( "#------------------------\n" );

        int[] a;

        double worst_time;

        for( /* TO BE COMPLETED */ ) {

            /* TO BE COMPLETED */

            worst_time /= REPETITIONS;
            System.out.printf( Locale.US, "%8d %11.3f\n", size, worst_time / NMS);
        }
    }
```

What to do: complete the method `measuringWorstCaseOfInsertionSort()` in order to get the running times expressed in microseconds for the **worst case** of the *insertion sort* algorithm.

Remember that the method `static long nanoTime()`, from package `java.lang.System`, returns the current time in nanoseconds with the maximum possible accuracy.

Solution:

```
    public static void measuringInsertionSort()
    {
        long ti = 0, // Initial timestamp
             tf = 0; // Final timestamp

        // Print header
        System.out.printf( "# InsertionSort. Time in microseconds\n" );
        System.out.printf( "#   Size        Worst \n" );
        System.out.printf( "#------------------------\n" );

        int[] a;

        double worst_time;

        for( int size = MIN_SIZE; size <= MAX_SIZE; size += STEP_OF_SIZE ) {

            for( int r=0; r < REPETITIONS; r++ ) {
                a = createArraySortedInDescendingOrder( size );
                ti = System.nanoTime();
                MeasurableAlgorithms.insertionSort(a);
                tf = System.nanoTime();
                worst_time += tf - ti;
            }

            worst_time /= REPETITIONS;
            System.out.printf( Locale.US, "%8d %11.3f\n", size, worst_time / NMS);
        }
    }
```