Retake First Lab Partial Exam – PRG – ETSInf – Academic year 2013/14
June 23rd, 2014 – Duration: 50 minutes

1. $\boxed{\text{2 points}}$ Complete the following method for solving the problem of the Hanoi Towers:

```
public static void hanoi( int disks, String origin, String target, String temporary ) {
   if( /* TO BE COMPLETED */ )          // Trivial case
      moveDisk( origin, target );
   else {                               // General case
      hanoi( /* TO BE COMPLETED */ );
      moveDisk( /* TO BE COMPLETED */ );
      hanoi( /* TO BE COMPLETED */ );
   }
}
```

Solution:

```
public static void hanoi( int disks, String origin, String target, String temporary ) {
   if( disks == 1 )           // Trivial case
      moveDisk( origin, target );
   else {                     // General case
      hanoi( disks-1, origin, temporary, target );
      moveDisk( origin, target );
      hanoi( disks-1, temporary, target, origin );
   }
}
```

2. $\boxed{\text{3 points}}$ Write a **RECURSIVE** method with the following profile:

```
public static int countAppearances( String a, String b )
```

that returns how many times the `String a` is contained in the `String b`. For example, if `a=`*coc* and `b=`*coca de cocochas cocinadas con coco*, `a` is contained five times in `b`. It is supposed that the length of `a` will be greater than or equal to one.

You `should` solve this problem by using the method implemented in lab practise 2 whose profile is:

```
public static boolean isPrefix( String a, String b )
```

Solution:

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

3. $\boxed{\text{5 points}}$ It is available a method whose profile is `public static void algorithm( int n )` that implements an algorithm whose temporal cost function, $T(n)$, has just one parameter, the input size of the problem, n. You have to complete the method:

```
public static void measuringAlgorithm( int initialSize, int finalSize,
                                       int incrementOfSize, int numberOfRepetitions )
{
    System.out.printf("# Input Size     Average Time (seconds)\n");
    System.out.printf("#-----------------------------------\n");

                   /* TO BE COMPLETED */

}
```

in order to measuring the empirical temporal cost of `algorithm()` by executing it for different values of the input size. Input size should range from `initialSize` to `finalSize` increased by steps of `incrementOfSize` units. For improving the estimation of the measured time, the method should be executed repeatedly for each value of the input size. The number of repetitions is given by the parameter `numberOfRepetitions`. For each value of the input size it should be shown the value of the input size itself and the average time of all the repetitions expressed in seconds.

You can make use of the method `public static long nanoTime()` of the class `java.lang.System`, that returns the current value of the timer in nanoseconds (1 nanosecond is equal to 10^{-9} seconds).

NOTE: Both methods `algorithm()` and `measuringAlgorithm()` belong to the same Java class.

An output example for `measuringAlgorithm( 10000, 20000, 1000, 10 )` could be:

```
# Input Size    Average Time (seconds)
#---------------------------------------
     10000            5.32
     11000            6.28
     12000            8.61
      ...             ...
     20000           30.45
```

Solution:

```
public static void measuringAlgorithm( int initialSize, int finalSize,
                                       int incrementOfSize, int numberOfRepetitions )
{
    System.out.println( "# Input Size    Average Time (seconds)" );
    System.out.println( "#-------------------------------------" );

    long time1=0, time2=0, totalTime=0; double averageTime = 0;
    for( int size=initialSize; size <= finalSize; t += incrementOfSize ) {
        totalTime=0;                        // Accumulated time reset
        for( int r=0; r < numberOfRepetitions; r++ ) {
            time1=System.nanoTime();      // Instant time before execution
            algorithm(t);
            time2=System.nanoTime();      // Instant time after execution
            totalTime+=(time2-time1);     // Updating accumulated time
        }
        averageTime=(double)totalTime/numberOfRepetitions; // Computing the average time
        System.out.printf( " %8d    %10.6f \n", t, averageTime*1.0e-9 );
    }
}
```