PRG - ETSInf. Lab Practises. Academic Year 2013-14. First partial exam. April 14th, 2014. Duration: 1 hour

1. $\boxed{\text{2.5 points}}$ The following implementation of the recursive algorithm for the Hanoi towers can be compiled with no errors, and no errors appear during its execution.

```
public static void hanoi( int disks, String origin, String target, String temporary )
{
    if ( disks == 1 ) {
        moveDisk( origin, target );
    } else {
        hanoi( disks-1, origin, temporary, target );
        moveDisk( origin, target );
        hanoi( discos-2, origin, target, temporary );
    }
}
```

However, there are logic errors, i.e., the result of its execution is not correct. The version we studied in the lab practises was right.

You have to: detect where the errors are, explain them and correct the code in order to obtain the same solution studied in the lab sessions.

Solution: The second recursive call `hanoi( disks-2, origin, target, temporary)` is wrong due to the following reasons:

- After moving one disk they remain `disks-1` in the temporary tower instead of `disks-2`.
- In the second recursive call the `temporary` tower should act as the `origin` one and vice-versa.

The error could be corrected by changing the second recursive call by:

```
hanoi( disks-1, temporary, target, origin );
```

2. $\boxed{\text{2.5 points}}$ A method with the profile `boolean isSuffix( String a, String b )` is available. It returns *true* if `a` is suffix of `b` and *false* otherwise.

To be done: write a recursive method that returns *true* if a given string `s` is substring of another string `t` by means of using the method `boolean isSuffix( String, String )`. The profile of the method must be:

```
public static boolean isSubstring( String s, String t )
```

Remind, `s.substring(i,j)` is an available method of the class `String` that returns an object of the class `String` that is a substring of `s` containing the characters from position `i` up to the `j-1`.

Solution:

```
public static boolean isSubstring( String s, String t )
{
    if ( s.length() > t.length() ) return false;
    else if ( isSuffix( s, t ) ) return true;
    else return isSubstring( s, t.substring( 0, t.length()-1 ) );
}
```

3. [2.5 points] The static method `sort( int [] )` of the class `MeasurableAlgorithms` has, in the average case, a quadratic temporal cost if we consider the length of the array given as parameter as the input size of the problem.

To be done: complete the code of the following method in order to perform the experimental or *a posteriori* analysis of the algorithm in the average case. 50 repetitions should be performed for every value of `t`, i.e., the input size. The following methods can be used:

- `public static int[] fillRandomArray( int size )`, which returns an array of length `size` with random values.
- `public static long nanoTime()`, from the class `java.lang.System`, which returns the current value of the time in nanoseconds.

```
public static void sortTemporalCostMeasurement()
{
    System.out.printf( "# Input size    Measured time \n" );
    System.out.printf( "#----------------------------\n" );
    long time1 = 0, time2 = 0, totalTime = 0; double averageTime = 0;
    for( int t=10000; t<=100000; t+=10000 ) {

        //  TO BE COMPLETED

        System.out.printf( "%8d  %8d\n", t, averageTime/1000 );
    }
}
```

Solution:

```
public static void sortTemporalCostMeasurement()
{
    System.out.printf( "# Input size    Measured time \n" );
    System.out.printf( "#----------------------------\n" );
    long time1 = 0, time2 = 0, totalTime = 0; double averageTime = 0;
    for( int t=10000; t<=100000; t+=10000 ) {
        totalTime=0;
        for( int r=0; r < 50; r++ ) {
            int[] a = fillRandomArray(t);
            time1 = System.nanoTime();       // Starting time
            MeasurableAlgorithms.sort(a);
            time2 = System.nanoTime();       // Ending time
            totalTime += (time2-time1);      // Accumulates the lapse of time
        }
        averageTime = (double)totalTime/50;  // Average measured temporal cost
        System.out.printf( "%8d  %8d\n", t, averageTime/1000 );
    }
}
```

4. [2.5 points] Assuming that the results of the previous experimental analysis were stored in the file `results.out`, with two columns, the first one with the input size and the second one with the measured temporal cost, you have you do the following tasks:

a) (1.75 points) To define the function in order to fit the shape defined by the theoretical or *a priori* analysis, and to obtain the best fit by means of the `fit` command of Gnuplot. The syntax of this command is:

`fit` *function file-name* `using` *i*:*j* `via` *parameters*

where:

- *function*: is the name of the function to be adjusted, defined previously.
- *file-name*: the name of the file with the empirical results, specified between double quotes.
- `using` *i*:*j*: specifies the columns from the file to be used for the fitting command, the first one for the X axis and the second one for the Y axis.
- `via` *parameters*: specifies the parameters of the function to be fitted delimited by commas.

b) (0.75 points) Once the fit has been done, how would you estimate the temporal cost (or time) for the method when the input size of the problem is 25000?

Solution:

a) `f(x) = a*x*x + b*x + c;`   `fit f(x) "results.out" using 1:2 via a,b,c`

b) Once `a`, `b` and `c` are known thanks to the `fit` command, then we only need to compute `f(25000)`.