Recovering of first quiz – PRG – ETSInf
Date: June 19th, 2012 – 2 hours

1. (2 points) Let `a` be an array of integer numbers, and let `x` be an integer number. **We ask you** the design of a recursive method for checking whether the sequence of values stored in `a` form a geometric progression of ratio `x`, i.e., `a[i+1] = a[i]*x` $\forall$ i $\in [0, \texttt{a.length} - 2]$.

   For example: given `a` = {`3,6,12,24,48`} and x=2, the method must return `true`, but if `a` = {`3,6,12,33,48`} for the same value of `x`, then it must return `false`. If the length of `a` is one, then it is considered as a geometric progression whatever the value of `x`.

   You must also write the first call to the method.

Solution:

```
/** Checks if all the values in a[i..a.length-1], 0<=i<a.length,
 *  form a geometric progression of ratio x
 */
public static boolean isGeometricProgression( int[] a, int i, int x )
{
    if ( i == a.length-1 )
        return true;
    else if ( a[i+1] == a[i]*x )
        return isGeometricProgression( a, i+1, x );
    else
        return false;
}
```

The first call to the method should be `isGeometricProgression( a, 0, x )`

2. (2 points) Write a recursive method with an integer number as a parameter, $n \geq 0$, for writing, on standard output and in a unique line, the values:

$$-n \quad -(n-1) \quad \cdots \quad \texttt{-2 -1 0 1 2} \quad \cdots \quad (n-1) \quad n$$

   For $n = 3$ the output will be:

```
-3 -2 -1 0 1 2 3
```

Solution:

```
/** n>=0 */
public static void write( int n )
{
    if ( n==0 )
        System.out.print( 0 );
    else {
        System.out.print( -n + " " );
        write( n-1 );
        System.out.print( " " + n );
    }
}
```

3. (2.5 points) Given the following method:

```
/** n>=0, 1<=x<=9 */
public static boolean searchX( int n, int x )
{
    if ( n > 0 ) {
        if ( n%10 == x )
            return true;
        else
            return buscarX( n/10, x );
    } else
        return false;
}
```

We as you:

a) Indicate what is the input size of the problem and the expression that defines it.

b) Determine whether there are significant instances. If so, identify which instances best represent the best case and the worst case of the algorithm.

c) Write the recurrence equation that defines the temporal cost as a function of the input size. Two equations, one for the best case and another for the worst case if there are significant instances, or a unique equation for the general case if there aren't significant instances.

   The substitution method must be used in any case.

d) Use the asymtotic notation for identifying the typical function that best fits the behavior of the temporal cost function(s) obtained in the previous step.

Solution:

a) The input size is defined by the value of the parameter of the method, that is, `n`.

b) Yes, there exist significant instances. The best case is when the least significant digit of `n` is equal to `x`. The worst case is when any digit of `n` is equal to `x`.

c) Best case: $T^m(n) \in \Theta(1)$. Worst case: the recurrence equation for the cost, using program steps as unit of measure, is:

$$T^p(n) = \begin{cases} T^p(n/10) + 1 & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

   Applying the substitution method:

   $T^p(n) = T^p(n/10) + 1 = T^p(n/10^2) + 2 = \ldots = T^p(n/10^k) + k.$

   The trivial case is reached when $k = 1 + \lfloor log_{10} n \rfloor$, then $T^p(0) = 1$. So $T^p(n) = 2 + \lfloor log_{10} n \rfloor$.

d) Using the asymtotic notation: $T^m(n) \in \Theta(1)$ and $T^p(n) \in \Theta(\log n)$.

   So, $T(n) \in \Omega(1) \cap O(\log n)$.

4. (3.5 points) Let `a` be an array of `double`, $\{\mathtt{a}_0, \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_{n-1}\}$, for representing the coefficients of a polynom: $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$.

   For computing the value of $P(x)$ we have available the following two methods:

   First method:

```
/** a.length>=1 */
public static double polynom1( double[] a, double x )
{
    double result = a[0];

    for( int i=1; i<a.length; i++ ) {
        double pot = 1;
        for( int k=1; k<=i; k++ ) pot = pot*x;
        result += a[i]*pot;
    }
    return result;
}
```

   Second method:

```
/** a.length>=1 */
public static double polynom2( double[] a, double x )
{
    double result = a[a.length-1];

    for( int i=a.length-2; i>=0; i-- )
        result = result*x + a[i];

    return result;
}
```

   We ask you:

   a) **For each method:**
      1. Indicate what is the input size of the problem and the expression that defines it.
      2. Determine whether there are significant instances. If so, identify which instances best represent the best case and the worst case of the algorithm.
      3. Choose a unit of measure for estimating the temporal cost function: program step or critical instruction. Then, write the mathematical expression of the temporal cost as a function of the input size and as accurate as possible. Two expressions, one for the best case and another for the worst case if there are significant instances, or a unique expression for the general case if there aren't significant instances.
      4. Use the asymtotic notation for identifying the typical function that best fits the behavior of the temporal cost function(s) obtained in the previous step.

   b) Which method has the best performance? Why?

Solution:

a) **First method.**

   1. The input size is $n =$`a.length`, i.e., the amount of items in `a`.

2. There are not significant instances. So, we have to compute just one temporal cost function, $T(n)$.
3. Using the program step as unit of measure,

$$T(n) = 1 + \sum_{i=1}^{n-1} (1+i) = 1 + (2+3+\ldots+n) = \frac{(n+1)\cdot n}{2} \in \Theta(n^2)$$

4. $T(n) \in \Theta(n^2)$.

Second method.

1) As for the first method.
2) As for the first method.
3) Using the program step as unit of measure,

$$T(n) = 1 + \sum_{i=1}^{n-1} 1 = n \in \Theta(n)$$

4) $T(n) \in \Theta(n)$.

b) The first method has a quadratic growth depending on the input size. The second one has a linear growth. So, we can conclude without discussion that the second method is more efficient than the first one.

We can observe that the first method computes `x` raised to `i` at every iteration of the outer loop. This operation is more and more expensive as the value of `i` increases. The second method avoids this extra computation thanks that the powers of `x` are calculated progressively at every iteration of the loop, and strategically accumulated in the variable `result`.