PRG - ETSInf. THEORY. Academic Year 2014-15. Mid term exam 1. April 27th, 2015. Duration: 2 hours.

1. 3 points Given an array `a` of integers whose size is ≥ 1, write a **recursive** method for checking if all the values stored in the array are even numbers and are stored in ascending order.

   What to do:

   a) (0.5 points) Profile of the method with the corresponding parameters for recursively solve the problem.

   Solution:

   ```
   public static boolean evenNumbersAndSorted( int [] a, int pos )
   ```

   where `a.length`>0 and $0\leq$`pos`$<$`a.length`.

   b) (1.2 points) Describe the trivial and general cases.

   Solution:

   - Trivial case, `pos==a.length-1`: Array of one component. Returns `true` if `a[pos]` is an even number, otherwise `false`.
   - General case, `pos<a.length-1`: Array with more than one component. Returns `true` if and only if `a[pos]` is an even number and `a[pos] <= a[pos+1]` all the elements from `a[pos+1]` up to `a[a.length-1]` are even numbers and are stored in ascending order. Otherwise returns `false`.

   c) (1 point) Write the Java code for the method.

   Solution:

   ```
   /** Checks if all the elements in the array <code>a</code> are even numbers
    *  and are sorted in ascending order.
    *  Precondition: a.length>0 and 0<=pos<a.length.
    */
   public static boolean evenNumbersAndSorted( int[] a, int pos )
   {
     if ( pos == a.length-1 )
       return a[pos]%2 == 0;
     else
       return a[pos]%2 == 0 && a[pos] <= a[pos+1] && evenNumbersAndSorted( a, pos+1 );
   }
   ```

   d) (0.3 points) Initial call for verifying the condition over the full array.

   Solution: For an array `a`, the initial call `evenNumbersAndSorted(a,0)` solves the posed problem.

2. 7 points For counting how many elements of an array `a` are lower than a given value `x`, we propose two solutions, algorithms written in Java, where the first one assumes the array is sorted in ascending order.

   - Solution 1

   ```
   /** Returns the number of elements in the array <code>a</code> lower than <code>x</code>.
    *  Precondition: <code>a</code> is sorted in ascending order.
    */
   public static int countLowerThanX_1( int[] a, int x )
   {
       int i = a.length - 1;
       while( i >= 0 && a[i] >= x ) i--;
   ```

```
    return i + 1;
}
```

- Solution 2

```
/** Returns the number of elements in the array <code>a</code> lower than <code>x</code>.
 *  Precondition: 0 <= pos <= a.length
 *  Initial call:  countLowerThanX_2( a, x, a.length-1 );
 */
public static int countLowerThanX_2( int[] a, int x, int pos )
{
    if ( pos < 0 )         return 0;
    else if ( a[pos] < x ) return countLowerThanX_2( a, x, pos-1 ) + 1;
    else                   return countLowerThanX_2( a, x, pos-1 );
}
```

What to do for each proposed solution:

i. (3 points for each proposed solution)

a) (*0.25 points*) Describe the input size of the problem and give an expression for it.

b) (*0.5 points*) Check if the algorithm is sensible for different instances of the input data. If the answer is yes, then describe the best case and the worst case.

c) (*1.5 points*) For the iterative method choose a critical instruction as a reference for counting program steps. Then, obtain the corresponding mathematical expression for the temporal cost function. If the algorithm is sensible to different instances then obtain the temporal cost function for the best case and the worst case.

For the recursive algorithm use the substitution method. Write the recursive equation for the temporal cost and develop it for obtaining the corresponding mathematical expression of the temporal cost function. If the algorithm is sensible to different instances then obtain the temporal cost function for the best case and the worst case.

d) (*0.75 points*) Use the asymptotic notation for expressing the set of functions the obtained temporal cost function belongs to.

Solution:

Analysis of the method of the solution 1:

a) The input size is the number of elements in the array, i.e., the size of the array.

$n \equiv$ `a.length`

b) This is a search problem. Search problems are always sensible to different instances of input data, i.e., to different ways data can be stored in the array. So for a fixed input size the algorithm can behave differently.

As the array `a` is sorted in ascending order, **the best case** is when the value in the last position of the array is lower than `x`, `a[a.length-1]` $<$ `x`. In this case all the values stored in the array are lower than `x`.

The worst case is when all the values in the array are greater than or equal to `x`: $\forall$`i,0` $\leq$ `i` $<$ `a.length, a[i]` $\geq$ `x`.

c) By choosing the condition of the while loop as the critical instruction, `i >= 0 && a[i] <= x`, we are going to obtain an expression for the temporal cost function for this algorithm that gives an approach of the number of program steps in function of the input size. In other words, the number of times the critical instruction is repeated.

$T^m(n) = 1$

$T^p(n) = 1 + \sum_{i=0}^{n-1} 1 = n + 1$

In the best case the critical instruction is executed once, because the condition is evaluated to `false` the first time. In the worst case the condition fails only when `i` is -1, so the the critical instruction is repeated $n + 1$ times.

d) $T^m(n) \in \Theta(1)$, $T^p(n) \in \Theta(n)$, so $T(n) \in \Omega(1) \cap O(n)$

The lower bound of the temporal cost function is the set of functions whose behaviour is constant $\Omega(1)$, and the upper bound is the set of functions whose behaviour is linear $O(n)$.

Analysis of the method of the solution 2:

a) The input size is the number of elements in the array to be considered, i.e., the remaining elements from `0` up to `pos`, $n \equiv$ `pos` $+ 1$

b) This is not a search problem, it is a traversal one, so for a fixed input size the behaviour of the algorithm will be always the same.

c) The temporal cost function is defined by means of a recursive equation:

$$T(n) = \begin{cases} T(n-1) + k & \text{if } n > 0 \\ k' & \text{if } n = 0 \end{cases}$$

where k and k' are positive constants measured in a time unit.

Applying the substitution method

$$T(n) = T(n-1) + k = T(n-2) + 2k = \ldots = T(n-i) + ik$$

When the trivial case is reached, $n - i = 0 \equiv$ `pos` $= -1$, $T(0) = k'$, then $T(n) = k' + nk$

d) $T(n) \in \Theta(n)$, the temporal cost depends linearly on the input size.

ii. (1 point) We have two possibilities for counting how many values lower than a given value `x` are stored in an array.

If we use the insertion sort algorithm for sorting the array before calling the method `countLowerThanX_1()`, we must know that $T_{is}(n) \in \Omega(n) \cap O(n^2)$.

- Possibility 1: First sort `a` by using the insertion sort algorithm, then call the method `countLowerThanX_1( int [], int )`.
- Possibility 2: Call the method `countLowerThanX_2( int [], int )`.

You have to compare the temporal cost of the two possible ways for counting values in an array that are lower than another value `x` taking into account that the array `a` could be not in ascending order.

Solution:

The best case for possibility 1 is $T_1^b(n) \in \Theta(n) + \Theta(n) \rightarrow T_1^b(n) \in \Theta(n)$.

The worst case for possibility 1 is $T_1^w(n) \in \Theta(n^2) + \Theta(n) \rightarrow T_1^w(n) \in \Theta(n^2)$.

The temporal cost of using possibility 2 is $T_2(n) \in \Theta(n)$.

So, using directly the method `countLowerThanX_2( int [], int )` is more efficient when the problem is not restricted to arrays sorted in ascending order.