PRG - ETSInf. THEORY. Academic year 2015-16. First mid term exam. April 11th, 2016. Duration: 2 hours.

1. 4 points Given an array `a` of `int` arranged in ascending order and an integer `x`, write a **recursive** method for computing the sum of all the elements in `a` lower than `x`, i.e. the sum of all `a[i]` such that `a[i]` $<$ `x`. An example, if `a` $= \{2, 2, 7, 8, 10, 10, 12, 23, 34\}$ and `x` $= 10$, then the result must be 19. For the same array and a value of `x` $= 1$ the result must be zero. The method should avoid to compare each value in `a` with `x` more than once.

   What to do:

   a) (0.75 points) Profile of the method with all the parameters needed to solve the problem recursively.

   b) (1.25 points) Description of the trivial and general cases.

   c) (1.50 points) Implementation in Java.

   d) (0.50 points) Initial call to the method for performing the sum using the whole array.

   Solution:

   a) A possible solution is the following:

   ```
   /** Precondition: a is sorted in ascending order and pos >= 0. */
   public static int sumValuesLowerThanX( int [] a, int pos, int x )
   ```

   for returning the sum of all the values lower than `x` stored in the slice `a[pos .. a.length-1]`, where `pos` ≥ 0.

   b)
   - Trivial case: `pos` $\geq$ `a.length` $\rightarrow$ Empty sub-array, zero is returned.
   - General case: `pos` $<$ `a.length` $\rightarrow$ Non-empty sub-array, if `a[pos]` $<$ `x` returns `a[pos]` plus the sum of all the values lower than `x` stored in the slice `a[pos+1 .. a.length-1]`, otherwise returns zero, because it is known no other values in `a` from `pos+1` up to the end can be lower than `x` thanks to the fact the array is sorted in ascending order. In other words, it is not necessary to perform more comparisons.

   c)
   ```
   /** Returns the sum of the values lower than x stored in a[pos..a.length-1].
    *  Precondition: a is sorted in ascending order and pos >= 0. */
   public static int sumValuesLowerThanX( int[] a, int pos, int x )
   {
       if ( pos < a.length ) {
           if ( a[pos] >= x ) { return 0; }
           else { return a[pos] + sumValuesLowerThanX(a, pos + 1, x); }
       } else { return 0; }
   }
   ```

   d) Initial call: `sumValuesLowerThanX( a, 0, x );`

2. 3 points The following method checks whether an slice of an array of integers is sorted in ascending order. The slice is defined from positions `left` up to `right` both included:

   ```
   /**
    * Returns true if a[left .. right] is sorted in ascending order,
    * false otherwise.
    */
   public static boolean sorted( int [] a, int left, int right )
   {
       if ( left > right ) {
           return true;
   ```

```
        } else {
            if ( a[left] > a[left+1] || a[right] < a[right-1] ) {
                return false;
            } else {
                return sorted( a, left+1, right-1 );
            }
        }
    }
```

What to do:

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

b) (0.50 points) Choose a critical instruction for using it as reference for counting program steps. Check if the algorithm is sensible for different instances of the input data. If the answer is yes, then describe best and worst cases.

c) (1.50 points) As it is a recursive method, write the recurrent relation for obtaining the temporal cost function by using the substitution method. If the algorithm is sensible to different instances then obtain the temporal cost function for the best case and for 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:

a) The input size of the problem is the length of the array `a`, $n =$ `a.length`. But in this case that the algorithm is recursive, we can also describe the input size as the length of the slice, so we can use $n = right - left + 1$.

b) In recursive algorithms the critical instruction is the condition of the `if` that allows us to distinguish between trivial and general cases. So it is `left` $>$ `right`.

The algorithm checks whether an array is sorted in ascending order. If a pair of consecutive values are not in the proper order it is not necessary to continue checking, so this is a search algorithm. All search algorithms are sensible to different instances of the problem for the same input size, so we have to define best and worst cases.

Best case is when either the first two consecutive values in the left end or in the right end to be checked do not fulfil the condition. Worst case is when the given slice of the array is all sorted in ascending order and the algorithm must reach the trivial case to stop.

c) We have to obtain the temporal cost function for the both best and worst cases.

- Best case: $T^b(n) = 1$ program steps
- Worst case:

$$T^w(n) = \begin{cases} T^w(n-2)+1 & \text{if } n > 1 \\ 1 & \text{if } n \leq 1 \end{cases}$$

expressed in program steps.

Applying the substitution method: $T^w(n) = T^w(n-2)+1 = T^w(n-4)+2 = \ldots = T^w(n-2\cdot i)+i$. The trivial case of the recursive algorithm is reached when $i = n/2$, so

$$T^p(n) = 1 + \frac{n}{2} \texttt{ program steps}$$

d) By using the asymptotic notation we have $T^b(n) \in \Theta(1)$ and $T^w(n) \in \Theta(n)$ so

$$T(n) \in \Omega(1) \cap O(n)$$

i.e. the temporal cost is upper bounded by a linear function depending on the input size.

3. [3 points] The following method transposes an squared matrix in memory:

```
/** Changes matrix m by its transpose in the same place.
  * Precondition: m is an squared matrix.
  */
public static void transpose( int [][] m )
{
    for( int i = 0; i < m.length; i++ ) {
        for( int j = 0; j < i; j++ ) {
            int tmp = m[i][j];
            m[i][j] = m[j][i];
            m[j][i] = tmp;
        }
    }
}
```

What to do:

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

b) (0.50 points) Choose a critical instruction for using it as reference for counting program steps.

c) (0.50 points) Is the method sensible to different instances of the problem for the same input size? In other words, is the critical instruction repeated more or less times depending on the input data for the same input size?

If the answer is yes describe best and worst cases.

d) (1.00 points) Obtain an expression of the temporal cost function for each case if the answer to the previous question was yes and a unique expression if the answer was no.

e) (0.75 points) Use the asymptotic notation for expressing the behaviour of the temporal cost for large enough values of the input size.

Solution:

a) The input size of the problem is the number of rows or columns of matrix `m`. So we can express $n = m.length$ with no ambiguity because `m` is squared.

b) The critical instruction can be `j` $<$ `i`.

c) No, this algorithm is not sensible to the way in that values are arranged in the matrix.

d)

$$T(n) = 1 + \sum_{i=0}^{n-1}(1 + \sum_{j=0}^{i-1} 1) = 1 + \sum_{i=0}^{n-1}(1+i) = 1 + \frac{(1+n)\cdot n}{2} = 1 + \frac{n}{2} + \frac{n^2}{2} \quad \texttt{program steps}$$

e) $T(n) \in \Theta(n^2)$