PRG - ETSInf. THEORY. Academic Year 2013-14. First partial exam. April 14th, 2014. Duration: 2 hours.

1. 3 points Given an array of integers named `a`, write a recursive method for adding up the matching numbers from both ends of the slice of the array defined by [`left`, `right`] (0≤`left`≤`right`<`a.length`), to the middle of the array while the numbers match. The algorithm should stop when the middle is reached of the first pair of positions with different numbers is reached. If the array length is odd the number just in the middle should be added only once. If the array has no matching numbers in symmetrical positions the result should be zero.

   Some examples:

   - The result for array `a = {1,4,1,2,4,1}` is 10 = (1+1+4+4),
   - the result for array `a = {1,4,2,2,4,1}` is 14 = (1+1+4+4+2+2),
   - the result for array `a = {1,4,1,4,1}` is 11 = (1+1+4+4+1),
   - the result for array `a = {1,4,0,0,1}` is 2 = (1+1),
   - the result for array `a = {8,4,0,0,1}` is 0.

   To be done:

   a) (0.5 points) Write the profile of the method with the proper parameters for the recursive solution of the problem.

   > **Solution:** A possible solution consists of a method with the following profile:
   >
   > `public static int matchingEndsSum( int[] a, int left, int right )`
   >
   > where 0≤`left`≤`right`<`a.length`.

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

   > **Solution:**
   >
   > - Trivial case, `left>right`: empty slice. Zero is returned, the neutral element of the sum.
   > - Trivial case, `left==right`: slice of size one, the value of `a[left]` is returned.
   > - General case, `left<right`: If both ends don't match (`a[left]!=a[right]`) zero is returned. Otherwise (`a[left]==a[right]`) the result of adding the values of both ends of the slice is returned `a[left]` and `a[right]` plus the sum of the sum of the slice `a[left+1..right-1]`.

   c) (1 point) Java implementation of the recursive method.

   > **Solution:**
   >
   > ```
   > /** Sum of the values of matching numbers at both ends of the slice
   >   * of the array considered. The slice is defined by the parameters
   >   * left and right holding the condition 0 <= left <= right < a.length.
   >   * If the slice is empty or values at both ends dint match zero is
   >   * returned.
   >  */
   > ```

```
public static int matchingEndsSum( int[] a, int left, int right )
{
    if ( left > right ) return 0;
    else if( left == right ) return a[left];
    else if ( a[left] != a[right] ) return 0;
    else return a[left] + a[right] + matchingEndsSum( a, left+1, right-1 );
}
```

d) (0.3 points) Initial call.

Solution: For a given array `a`, the initial call should be `matchingEndsSum( a, 0, a.length-1 )` for solving the stated problem.

2. 3 points Given the following recursive method written in Java for checking if all the values stored in an array `a[pos .. a.length-1]` form an arithmetic progression with difference `d`:

```
/** Returns 'true' if for all couple of consecutive elements a[i] and a[i+1]
  * in a[pos .. a.length-1] the condition (a[i+1] == a[i]+d) holds.
  * Precondition: a.length >= 1 && 0 <= pos < a.length
  */
public static boolean arithmeticProgression( int[] a, int d, int pos )
{
    if ( pos == a.length-1 ) return true;
    else return ( a[pos+1] == a[pos]+d ) && arithmeticProgression( a, d, pos+1 );
}
```

To be done:

a) (0.25 points) Determine the input size of the problem and write the expression that represents it.

Solution: The input size is the amount of elements in the slice of the array considered at each call. The expression which best defines it is `n` ≡ `a.length - pos`.

b) (0.5 points) Identify the critical instruction in the algorithm that is useful for counting program steps. Identify if there exist significant instances. In the affirmative case indicate the best and worst cases.

Solution: There exist different instances because this is a search problem in an array.

The best case is when the condition (`a[i+1] == a[i]+d`) doesn't hold for the first pair of values. In this case *false* is returned and none recursive call is performed.

The worst case is when the condition holds for all pairs of consecutive values in the array, i.e., the condition (`a[i+1] == a[i]+d`) holds for all values of `i` from zero to `a.length-2`. *True* is returned when the trivial case is reached after the maximum of recursive calls have been executed.

c) (1.5 points) Obtain the mathematical expression as precise as be possible for the temporal cost function $T(n)$. If there exist significant instances then both functions must be obtained, one for the best case $T^b(n)$ and another for the worst case $T^w(n)$. Advice: use the substitution method.

Solution:

For the best case $T^b(n) = k$, where k is a positive constant measured using a certain time unit.

For the worst case

$$T^p(n) = \begin{cases} k_0 & \text{if } n = 1 \\ k_1 + T^w(n-1) & \text{if } n > 1 \end{cases}$$

where k_0, k_1 are positive constants measured using a certain time unit. Applying the substitution method:

$$\begin{aligned} T^w(n) &= k_1 + T^w(n-1) = 2 \cdot k_1 + T^w(n-2) = 3 \cdot k_1 + T^w(n-3) = \cdots = \\ &= i \cdot k_1 + T^w(n-i) = \cdots = \\ &(\textit{trivial case}: \ n - i = 1, \ i = n-1) \\ &= k_1 \cdot (n-1) + T^w(1) = k_1 \cdot n + (k_0 - k_1) \end{aligned}$$

d) (0.75 points) Express the results using asymptotic notation.

Solution:

$$T^b(n) \in \theta(1), \ T^w(n) \in \theta(n).$$

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

3. [4 points] The following method named `triangle(int)` determines, by writing them, the number of right triangles with sides of integer length and hypotenuse `h`:

```
/** This method counts, by writing them, all the right triangles
 *  with sides of integer length and hypotenuse h. */
public static int triangle( int h )
{
    int cont = 0;
    for( int c1 = 4; c1 < h; c1++ )
        for( int c2 = 3; c2 < c1; c2++ )
            if ( c1*c1 + c2*c2 == h*h ) {
                cont++;
                System.out.println( "c1= " + c1 + ", c2= " + c2 + ", h= " + h );
            }
    return cont;
}
```

To be done:

a) (0.5 points) Determine the input size of the problem and write the expression that represents it.

> **Solution:** The input size n of the problem is the parameter h, i.e., the hypotenuse of all the possible right triangles we look for.

b) (0.5 points) Identify the critical instruction in the algorithm that is useful for counting program steps. Identify if there exist significant instances. In the affirmative case indicate the best and worst cases.

> **Solution:** This algorithm doesn't present significant instances, the number of times loops iterate only depend on the input size.

c) (2 points) Obtain the mathematical expression as precise as be possible for the temporal cost function $T(n)$. If there exist significant instances then both functions must be obtained, one for the best case $T^b(n)$ and another for the worst case $T^w(n)$.

> **Solution:**
> By counting program steps:
>
> $$T(n) = 1 + \sum_{i=4}^{n-1}(1 + \sum_{j=3}^{i-1} 1) = 1 + \sum_{i=4}^{n-1}(i-2) = 1 + \frac{(n-1)\cdot(n-4)}{2} \; p.s.$$
>
> If we take into consideration the evaluation of the condition `c1*c1 + c2*c2 == h*h`, which is equivalent to ignore terms of lower order:
>
> $$T(n) = \sum_{i=4}^{n-1}\sum_{j=3}^{i-1} 1 = \sum_{i=4}^{n-1}(i-3) = \frac{(n-3)\cdot(n-4)}{2} \; \textit{critical instructions}$$

d) (1 point) Express the results using asymptotic notation.

Solution: $T(n) \in \theta(n^2)$.