COMPUTER PROGRAMMING – ETSINF – THEORY – Academic year 2016/2017
Retake first mid term exam – June 26, 2017 – Duration 2 hours

Notice: The maximum mark of this exam is 10, but his weight in the final grade is **3 points**.

1. $\boxed{\text{3 points}}$

Triangular numbers are those which can be represented by an equilateral triangle. Consider 0 is the first triangular number. The first triangular numbers are: 0, 1, 3, 6, 10, 15, 21, 28, 36, 45.
Triangular numbers can be computed by means of the following recursive equation:

$$t(n) = \begin{cases} 0 & n = 0 \\ n + t(n-1) & n > 0 \end{cases}$$

where $t(n)$ is the n-*th* triangular number and $t(n-1)$ its predecessor.
You have to implement an algorithm by using recursion when required for filling in an array of n elements with the first n triangular numbers. The solution implies to implement two methods:

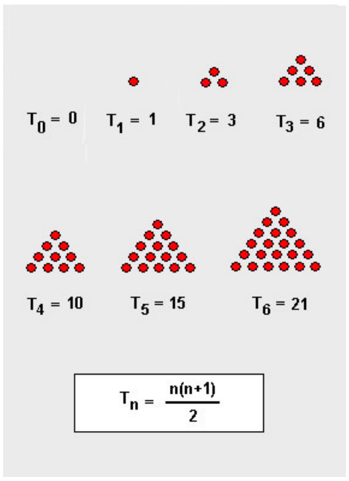

Figure 1. Graphical representation of the first seven triangular numbers.

- A wrapper method, non recursive, with one integer parameter, n, a positive integer. This method should create the array and call the second method for filling in the array with the first n triangular numbers.
- A recursive method with two parameters, the array of size n and another parameter useful for the recursion. This method should compute the triangular numbers with the proper strategy and store each one in his position.

You also have to specify the preconditions of both methods.

For instance, if call the wrapper method with $n = 5$ it should return the array $\{0, 1, 3, 6, 10\}$, and if we call it with $n = 1$ it should return the array $\{0\}$.

Solution:

A possible solution is to compute each triangular number by means of the equation. $t(n) = n + t(n-1)$.

```
/** Precondition: n>0 */
public static int[] triangularNumbers( int n )
{
    int[] a = new int[n];

    triangularNumbers( a, n-1 );

    return a;
}

/** Precondition: n>=0 && n < a.length */
private static void triangularNumbers( int[] a, int n )
{
    if (n == 0) {
        a[n] = 0;
    } else {
        triangularNumbers( a, n-1 );
        a[n] = n + a[n-1];
    }
}
```

Another possible solution consists in compute each triangular number by using the equation

$$t(n) = n * (n+1)/2$$

shown in Figure 1. In this case it is presented a recursive solution based on ascending decomposition.

```
/** Precondition: n>0 */
public static int[] triangularNumbers( int n )
{
    int[] a = new int[n];
    triangularNumbers( a, 0 );
    return a;
}

/** Precondition: n>=0  &&  n < a.length */
private static void triangularNumbers( int[] a, int n )
{
    if ( n < a.length ) {
        a[n] = (n * (n + 1)) / 2;
        triangularNumbers( a, n+1 );
    }
}
```

2. $\boxed{\text{4 points}}$ Given a sorted array in ascending order, the following method returns how many times it is repeated the first repeated value, the smallest repeated number. Or zero if there are not repeated numbers. As an example, given the array `a = {2, 5, 5, 5, 8, 8, 9}` the method returns `3` because the smallest repeated number is the five that appears three times. In the case of `a = {2, 5, 8, 9}` the method returns `0`.

```
/** Precondition: a is sorted in ascending order */
public static int countOcurrencesOfFirstRepeatedNumber( int [] a )
{
    int counter = 0, i = 0;
    boolean repeated = false;
    while( i < a.length-1  &&  !repeated ) {
        int j = i + 1;
        repeated = (a[j] == a[i]);
        while( j < a.length && a[j] == a[i] ) { j++; }
        if ( repeated ) { counter = j - i; }
        i++;
    }
    return counter;
}
```

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) (0.75 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) (0.50 points) Use the asymptotic notation for expressing the behaviour of the temporal cost function for large enough values of the input size.

f) (1.00 point) Considering the *insertion sort* algorithm studied in class, that has a linear temporal cost in the best case $T^b(n) \in \Theta(n)$, i.e. when the array to be sorted is already sorted, and a n-squared temporal cost in the worst case $T^w(n) \in \Theta(n^2)$. What is the global temporal cost of the operation composed by (1) sorting the array `a` by using the *insertion sort* algorithm and then (2) using the method `countOcurrencesOfFirstRepeatedNumber(int[])`?

Solution:

a) The input size is the number of elements in the array `a`, and the expression is `a.length`.

From now on $n \equiv$ `a.length`.

b) The critical instruction is the condition of the most inner loop: `j < a.length && a[j] == a[i]`

c) Yes, the best case is when the first value stored in the array is repeated once: `a[0] == a[1]`.

In this case the most outer loop finishes its execution after its first iteration. The most inner loop iterates once, because it finishes as soon as j is equal to 2.

The worst case is dual, when no repeated values appear in the array `a` and when all the values in the array are the same `a[i] == a[0]` $\forall i$. In both possible worst cases the temporal cost is linear.

d) Using the program step as the measure unit:

- Best case: $T^b(n) = 3$ *program steps*
- Worst case: $T^w(n) = 1 + \sum_{i=0}^{n-1} 1 = n + 1$ *program steps*

And using the critical instruction as the measure unit:

- Best case: $T^b(n) = 2$ *critical instructions*
- Worst case: $T^w(n) = \sum_{i=0}^{n-2} 1 = n - 1$ *critical instructions*

e) Using the asymptotic notation: $T^b(n) \in \Theta(1)$ and $T^w(n) \in \Theta(n) \Rightarrow T(n) \in \Omega(1) \cap O(n)$.

f) When using the composed operation of sorting by using the *insertion sort* algorithm and then invoking the method `countOcurrencesOfFirstRepeatedNumber()`, the complexity is the following:

$$T_{composed}(n) = T_1(n) + T_2(n)$$

- Best case: array sorted in ascending order and `a[0] == a[1]` and `a[1] != a[2]`.

$$T_{composed}(n) \in \Theta(n) + \Theta(1) \;\Rightarrow\; T_{composed}(n) \in \Theta(n)$$

- Worst case: array sorted in descending order and the array does not contain repeated values.

$$T_{composed}(n) \in \Theta(n^2) + \Theta(n) \;\Rightarrow\; T_{composed}(n) \in \Theta(n^2)$$

3. $\boxed{\text{3 points}}$ The following method computes the sum of the bits of an positive integer `n`. Recall that in integer variables the values are stored in base 2, i.e. in binary format.

```
/** Precondition: n >= 0 */
public static int bitSum( int n )
{
    if ( n <= 1 ) {
        return n;
    } else {
        int lastBit = n % 2;
        return bitSum( n / 2 ) + lastBit;
    }
}
```

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) (0.75 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) (0.50 points) Use the asymptotic notation for expressing the behaviour of the temporal cost function for large enough values of the input size.

Solution:

a) The input size is just n, the parameter of the method.

b) The critical instruction is the condition of the `if` that distinguishes between trivial and general case: `n <= 1`.

c) The recurrence equation is defined as:

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

   Applying the substitution method:
   $T(n) = T(n/2) + 1 = T(n/4) + 2 = T(n/8) + 3 = \ldots = T(n/2^i) + i.$
   If $n/2^i = 1 \rightarrow i = log_2 n$, so $T(n) = T(n/2^{log_2 n}) + log_2 n = T(1) + log_2 n = 1 + log_2 n$ *program steps*

d) Using the asymptotic notation: $T(n) \in \Theta(log_2 n)$

An alternative solution exists when using as input size the number of figures of n in base 2. Which is the same to say the minimum number of bits needed for representing the value of n. In this case:

a) $m =$ number of figures (or digits) of the representation of `n` in base 2.

b) The critical instruction is the condition of the `if` that distinguishes between trivial and general case: `n <= 1`.

c) The recurrence equation is defined as:

$$T(m) = \begin{cases} T(m-1) + 1 & \text{if } m > 1 \\ 1 & \text{if } m \leq 1 \end{cases}$$

   Applying the substitution method:
   $T(m) = T(m-1) + 1 = T(m-2) + 2 = T(m-3) + 3 = \ldots = T(m-i) + i.$
   When $m - i = 1 \rightarrow i = m - 1$, so $T(m) = T(1) + m - 1 = 1 + m - 1 = m$ *program steps*

d) Using the asymptotic notation: $T(m) \in \Theta(m)$

   It can be observed that the temporal cost is the same despite the obtained final expressions are different. We get a logarithmic temporal cost when n is used as the input size, and a linear cost when the input size is defined in terms of logarithms, $m \equiv figures(n) = 1 + \lfloor log_2 n \rfloor$.