PRG - ETSInf. THEORY. Academic Year 2015-2016. First partial exam recovery.
GII – GIINF – GADE
June 17th, 2016. Duration: 2 hours.

1. [4 points] Let `a` be an array of type `int` whose length is greater than or equal to 1 (`a.length` ≥ 1).

   It is requested to write a **recursive** method for checking if the elements of the array are the Fibonacci succession, i.e. each value is equal to the sum of the two previous ones in the sequence. The two initial values should be 0 and 1. An example is {`0,1,1,2,3,5,8,13,21,...`}.

   a) (0.75 points) Profile of the method with the proper arguments for solving the problem recursively.

   b) (1.25 points) Describe the trivial and general cases for the solution.

   c) (1.50 points) Body of the method in Java.

   d) (0.50 points) Initial call to the method, i.e. how to invoke it from other methods for solving the problem.

Solution:

a) A possible solution consists in defining a method with the following profile:

```
/** Precondition: a.length >= 1  and  0 <= pos <= a.length - 1. */
public static boolean isFibonacci( int[] a, int pos )
```

for checking whether the elements in `a[0..pos]` are the initial `pos+1` values of the Fibonacci succession, where `0` $\leq$ `pos`.

b)
- Trivial case, `pos = 0`: check if `a[0] = 0`
- Trivial case, `pos = 1`: check if `a[1] = 1` and `a[0] = 0`
- General case, `pos` $>$ `1`: check if `a[pos] = a[pos - 1] + a[pos - 2]`

c)
```
/** Check if the values in a[0..pos] are the Fibonacci succession.
  * Precondition: a.length >= 1  and  0 <= pos <= a.length - 1. */
public static boolean isFibonacci( int[] a, int pos )
{
    if ( pos == 0 ) { return a[0] == 0; }
    else if ( pos == 1 ) { return a[1] == 1 && a[0] == 0; }
    else { return a[pos] == a[pos-1] + a[pos-2] && isFibonacci( a, pos-1 ); }
}
```

d) For a an `a` the correct initial call is `isFibonacci( a, a.length-1 )`.

2. [3 points] By using the following method you can check if the sum of all the values stored in each row of a matrix `m` are equal.

   `m` is a squared ($N \times N$) matrix of type `double` with at least two rows and columns ($N \geq 2$).

```
/** PRECONDITION: m is a NxN matrix with N >= 2. */
public static boolean allRowsAddUpTheSameTotal( double [][] m )
{
    // Computes the sum of the first row.
    int sum0 = 0;
    for( int i = 0; i < m.length; i++ ) { sum0 += m[0][i]; }

    boolean equalSums = true;
    // Computes the sum of each row.
    for( int i=1; i < m.length && equalSums; i++ ) {
        int rowSum = 0;
```

```
            for( int j=0; j < m.length; j++ ) { rowSum += m[i][j]; }
            equalSums = (sum0 == rowSum);
        }

        return equalSums;
    }
```

It is requested:

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

b) (0.25 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) (1.50 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 for large enough values of the input size.

Solution:

a) The input size is the dimension of the matrix `m`, i.e. `m.length`. For now on

$$n = \texttt{m.length}$$

b) The best critical instruction is `j < m.length`, the condition of the most inner loop.

c) The algorithm is sensible to different instances of the input data.

It is a search problem and the algorithm stops when it is found a row whose values add up to a different value of the sum of the values in the first row.

The best case is when the sum of `m[1]` is not equal to the sum of `m[0]`, i.e. when the sum of values in the second row differs from the sum of the values in the first row.

The worst case is then all the rows add up to the same value. In this case all the values of the matrix are going to be visited for computing the sum in all rows.

d) By choosing the program step as the measure unit, the temporal cost function is:

- Best case:

$$T^b(n) = 1 + \sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} 1 = 2n + 1 \; program \; steps$$

- Worst case:

$$T^w(n) = 1 + \sum_{i=0}^{n-1} 1 + \sum_{i=1}^{n-1} (1 + \sum_{j=0}^{n-1} 1) = 1 + n + \sum_{i=1}^{n-1} (1+n) = 1 + n + (n-1)(n+1) = n^2 + n \; program \; steps$$

If we choose as unit measure the critical instruction and consider `sum0 += m[0][i];` from the first loop and `rowSum += m[i][j];` from the second loop, both with constant running time, we get the following expressions:

- Best case:

$$T^b(n) = \sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} 1 = 2n$$

- Worst case:

$$T^w(n) = \sum_{i=0}^{n-1} 1 + \sum_{i=1}^{n-1}\sum_{j=0}^{n-1} 1 = n + \sum_{i=1}^{n-1} n = n + n(n-1) = n^2$$

e) Using the asymptotic notation, we get $T^b(n) \in \Theta(n)$ and $T^w(n) \in \Theta(n^2)$ so

$$T(n) \in \Omega(n) \cup O(n^2)$$

3. 3 points Given the following implementation of the recursive algorithm for computing the *Russian product* of two natural numbers `a` and `b`:

```
/** PRECONDITION: a >= 0  and  b >= 0. */
public static int russianProduct( int a, int b )
{
    if (b == 0) { return 0; }
    else {
        if (b % 2 == 0) { return russianProduct( a * 2, b / 2 ); }
        else { return a + russianProduct( a * 2, b / 2 ); }
    }
}
```

It is requested to analyse the temporal cost of this algorithm.

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

b) (0.25 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) (1.50 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. As the algorithm is recursive use the substitution method for obtaining the mathematical expression for the temporal cost function. Using the recurrence relation will help you to apply the substitution method.

e) (0.50 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 just `b`, but from now on we are going to use n for referencing it as the input size. The integer `a` does not have influence in the number of recursive calls. Neither in the temporal cost of the non recursive instructions. So it can be ignored for obtaining the temporal cost of the algorithm.

b) The critical instruction is the condition of the conditional instruction for distinguishing among trivial and general cases. In this case there is one trivial case only, but two general cases.

c) This algorithm always goes through the sequence of values $b, \frac{b}{2}, \frac{b}{4}, \frac{b}{8} \ldots$ So the algorithm is not sensible to different instances of the input data.

d) The running time of the instructions that are executed in the trivial case is constant, $c_0 \in \Re^+$. The running time of the instructions that are executed in the general cases are also constant but the recursive call, $c_1 \in \Re^+$.

In the general case the recursive call reduces the input size of the problem dividing by 2. As a consequence the recurrence relation to be used for getting the temporal cost function by applying the substitution method is the following:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 + T(\frac{n}{2}) & \text{if } n > 0 \end{cases}$$

So

$$T(n) = c_1 + T(\frac{n}{2^1}) = 2c_1 + T(\frac{n}{2^2}) = 3c_1 + T(\frac{n}{2^3}) = \ldots = kc_1 + T(\frac{n}{2^k})$$

As the input size in the trivial case is equal to zero, for simplifying the will use as the trivial case when $\frac{n}{2^k} = 1$, then $k = \log_2 n$. Replacing k by $\log_2 n$ in the expression $k * c_1 + T(\frac{n}{2^k})$ we get

$$T(n) = c_1 \log_2 n + T(1) = c_1 \log_2 n + (c_1 + T(0)) = c_1 \log_2 n + c_1 + c_0$$

e) Using the asymptotic notation $T(n) \in \Theta(\log n)$, i.e. the temporal cost function of this algorithm has a logarithmic dependency on the input size.