Computer Programming – ETSINF – Academic year 2018/2019 – First mid term exam April 1st, 2019 – duration 2 hours

Notice: The highest mark for this exam is 10 points, but its weight in the final grade of PRG is **3 points**.

1. $\boxed{\text{4 points}}$ Let a be an array of integers and k an integer, such that $k \geq a.length$. It is defined a condition named $keyPalindrome(k)$ as follows: an array a is $keyPalindrome(k)$ if the absolute value of the difference between its symmetrical elements is less than $k - i$, where i is the index of the left hand side element of the both ones used, that is, $|a[i] - a[a.length - i - 1]| < k \; \forall i <= a.length/2$. The empty array fulfils the condition by definition, and any array whose length is one too. **You have to design and implement a recursive method for checking if an array a is $keyPalindrome(k)$.** Let us see some examples:
   - Given the array $\{20, 15, 14, 32, 10, 7, 22\}$ and $k = 10$, the method should return `true` since:
     - $|20 - 22| < 10$,
     - $|15 - 7| < 9$,
     - $|14 - 10| < 8$ and,
     - obviously, $|32 - 32| < 7$.
   - Given the array $\{20, 15, 14, 10, 7, 22\}$ and $k = 10$, the method should `true` since:
     - $|20 - 22| < 10$,
     - $|15 - 7| < 9$,
     - $|14 - 10| < 8$.
   - Given the two previous arrays and $k = 9$ the method should return `false` because:
     - $|20 - 22| < 9$, but
     - $|15 - 7| \geq 8$.

   What to do:

   a) (0.75 points) Profile of the method with the appropriate parameters for solving the problem recursively. Add the necessary pre-conditions for the parameters.

   b) (1.25 points) Define trivial and general cases.

   c) (1.50 points) Write the Java code for the recursive method.

   d) (0.50 points) Initial call to the recursive method for processing the whole array for a given value of k. Use a wrapper method if you prefer it.

Solution:

a) Profile:

```
/** Precondition: k >= a.length, 0 <= left, right < a.length */
public static boolean keyPalindrome( int[] a, int left, int right, int k )
```

for checking the sub-array $a[left \ldots right]$ is key palindrome of key k, and assuming $0 \leq left \leq right \leq a.length \leq k$.

b)
- Trivial case: $left \geq right$: any sub-array of 0 or 1 elements is $keyPalindrome(k)$ by definition, the method returns *true*.
- General case: $left < right$: sub-array of 2 or more elements. If $|a[left] - a[right]| \geq k$ return *false*, otherwise the method must call itself to check if the sub-array $a[left + 1 \ldots right - 1]$ could be $keyPalindrome(k - 1)$.

c)

```
/** Precondition: k >= a.length, 0 <= left, right < a.length */
public static boolean keyPalindrome( int [] a, int left, int right, int k )
{
    if ( left < right ) {
        int diff = Math.abs( a[left] - a[right] );
        return diff < k && keyPalindrome( a, left + 1, right - 1, k - 1 );
    } else {
        return true;
    }
}
```

d) Initial call: `keyPalindrome( a, 0, a.length-1, k );`

```
/** Wrapper method */
public static boolean keyPalindrome( int [] a, int k )
{
    return keyPalindrome( a, 0, a.length-1, k );
}
```

2. 3 points Let a be a square matrix of real values, $below(a)$ be a square matrix with the values of a located below the main diagonal, $above(a)$ be a square matrix with the values of a located above the main diagonal, $diag(a)$ are the elements of the main diagonal.

The following method checks if $sum(diag(a)) = sum(below(a)) - sum(above(a))$

```
/** Precondition: a is a square matrix. */
public static boolean sumBelowAbove( double[][] a )
{
    double sumBelow = 0, sumDiag = 0, sumAbove = 0;

    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a.length; j++) {
            if (j < i) {
                sumBelow += a[i][j];
            } else if (j == i) {
                sumDiag += a[i][i];
            } else {
                sumAbove += a[i][j];
            }
        }
    }
    return sumBelow - sumAbove == sumDiag;
}
```

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 both best and worst cases ($T^b(n)$ and $T^w(n)$) if the answer to the previous question was yes, otherwise a unique expression of temporal cost function $T(n)$.

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) $n = a.length$

b) $j < a.length$

c) No. So, we do not need to describe best and worst cases.

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

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

3. $\boxed{\text{3 points}}$ An *upper triangular matrix* is a square matrix whose values under the diagonal are set to zero. The recursive method *isUpperTriangular*(), whose code is below, checks if the rows of m from the first row (row whose index is zero) to row *nRow* fulfil the property for being an upper triangular matrix. Then, for checking if the whole matrix is an upper triangular matrix the initial call to the method is:

```
isUpperTriangular( mat, mat.length -1 );
```

```
/** Precondition: m is a square matrix of integers, -1 <= nRow < m.length */
public static boolean isUpperTriangular( int [][] m, int nRow )
{
    boolean res = true;
    if ( nRow >= 0 ) {
        int j = 0;
        while( j < nRow && res ) {
            if ( m[nRow][j] != 0 ) { res = false; }
            else { j++; }
        }
        if (res) { res = isUpperTriangular( m, nRow - 1 ); }
    }
    return res;
}
```

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 both best and worst cases ($T^b(n)$ and $T^w(n)$) if the answer to the previous question was yes, otherwise a unique expression of temporal cost function $T(n)$. **As this is a recursive method, you have to use the substitution method**.

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) $n = nRow + 1$

b) $nRow >= 0$

c) Yes.

The best case is when the first element to be checked $m[nRow][0]$ is not equal to zero.

The worst case is when the matrix is an upper triangular matrix, that is, all the elements under the main diagonal are zero and must be checked.

d) Temporal cost function

- Best case: $T^b(n) = 1$.
- Worst case: recurrence equation:

$$T^w(n) = \begin{cases} T^w(n-1) + n & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Applying the substitution method:

$$\begin{aligned}
T^w(n) =&\ T^w(n-1) + n \\
=&\ T^w(n-2) + (n-1) + n \\
=&\ T^w(n-3) + (n-2) + (n-1) + n \\
=&\ \dots \\
=&\ T^w(n-i) + \textstyle\sum_{j=0}^{i-1}(n-j) \\
=&\ T^w(0) + \textstyle\sum_{j=0}^{n-1}(n-j) \\
=&\ 1 + \textstyle\sum_{j=0}^{n-1}(n-j) = 1 + \sum_{i=1}^{n} i \\
=&\ 1 + \tfrac{n\cdot(n+1)}{2}
\end{aligned}$$

e) $T^b(n) \in \Theta(1)$ and $T^w(n) \in \Theta(n^2) \Rightarrow T(n) \in \Omega(1) \cap O(n^2)$