PRG - ETSInf. TEORÍA. Curso 2018-19. Parcial 1.
1 de abril de 2019. Duración: 2 horas.

Nota: El examen se evalúa sobre 10 puntos, pero su peso específico en la nota final de PRG es de **3 puntos**.

1. 4 puntos Dados un array de enteros `a` y un entero `k`, $k \geq$ `a.length`, se dice que `a` es un *capicúa de clave* `k` si la diferencia en valor absoluto entre las componentes primera y última es menor que `k`, menor que $k-1$ entre las componentes segunda y penúltima, menor que $k-2$ entre la tercera y la antepenúltima, y así sucesivamente. Se entiende que el array vacío o con una sola componente son capicúas, para cualquier clave. Se desea un método recursivo que compruebe si `a` es un capicúa de clave `k`. Por ejemplo,
   - para el array $\{20, 15, 14, 32, 10, 7, 22\}$ y la clave 10, el método debe retornar `true` ya que:
     - $|20 - 22| < 10$,
     - $|15 - 7| < 9$,
     - $|14 - 10| < 8$ y
     - trivialmente, $|32 - 32| < 7$.
   - para el array $\{20, 15, 14, 10, 7, 22\}$ y la clave 10, el método también debe retornar `true` ya que:
     - $|20 - 22| < 10$,
     - $|15 - 7| < 9$,
     - $|14 - 10| < 8$.
   - para los arrays anteriores y la clave 9 el método debe retornar `false`, dado que:
     - $|20 - 22| < 9$, pero
     - $|15 - 7|$ no es menor que 8.

   Se pide:

   a) (0.75 puntos) Perfil del método recursivo, con los parámetros adecuados para resolver recursivamente el problema, y precondición relativa a dichos parámetros.

   b) (1.25 puntos) Caso base y caso general de la recursión.

   c) (1.50 puntos) Implementación en Java del método recursivo.

   d) (0.5 puntos) Llamada inicial al método para comprobar si un cierto array `data` es o no capicúa de clave `k`.

Solución:

a) Una posible solución consiste en definir un método con el siguiente perfil:

```
/** Precondición: k >= a.length, 0 <= ini, fin < a.length */
public static boolean palindromeKey(int[] a, int ini, int fin, int k)
```

tal que comprueba si el array `a[ini..fin]` es capicúa de clave `k`, siendo `k` $\geq$ `a.length`, 0 $\leq$ `ini` y `fin` $<$ `a.length`.

b)
- Caso base, `ini >= fin`: Subarray de 0 o 1 elementos. Es capicúa de clave `k` y devuelve `true`.
- Caso general, `ini < fin`: Subarray de 2 o más elementos. Si la diferencia en valor absoluto entre `a[ini]` y `a[fin]` no es menor que `k`, el array no es capicúa de clave `k` y devuelve `false`. En caso contrario (`Math.abs(a[ini] - a[fin]) < k`), el problema se reduce a comprobar si el subarray `a[ini + 1..fin - 1]` es capicúa de clave `k - 1`.

c)
```
/** Precondición: k >= a.length, 0 <= ini, fin < a.length */
public static boolean palindromeKey(int[] a, int ini, int fin, int k) {
   if (ini < fin) {
       int diff = Math.abs(a[ini] - a[fin]);
       return diff < k && palindromeKey(a, ini + 1, fin - 1, k - 1);
   }
   else { return true; }
}
```

d) La llamada inicial para comprobar si el array `data` es capicúa de clave `k` sería, siempre que `k` $\geq$ `data.length`: `palindromeKey(data, 0, data.length - 1, k)`.

2. 3 puntos Dada una matriz cuadrada `a` de reales, el siguiente método comprueba si la suma de los elementos que están por debajo de la diagonal principal menos la de los que están por arriba, coincide con la suma de los elementos de la diagonal principal.

```
/** Precondición: a es una matriz cuadrada. */
public static boolean sumBelowAbove(double[][] a) {
    double sumBelow = 0, sum = 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) { sum += a[i][i]; }
            else { sumAbove += a[i][j]; }
        }
    }
    return sumBelow - sumAbove == sum;
}
```

Se pide:

a) (0.25 puntos) Indica cuál es el tamaño o talla del problema, así como la expresión que la representa.

b) (0.75 puntos) Indica, y justifica, si existen diferentes instancias significativas para el coste temporal del algoritmo e identifícalas si es el caso.

c) (1.50 puntos) Elige una unidad de medida para la estimación del coste (pasos de programa, instrucción crítica) y de acuerdo con ella obtén una expresión matemática, lo más precisa posible, del coste temporal del método, distinguiendo el coste de las instancias más significativas en caso de haberlas.

d) (0.50 puntos) Expresa el resultado anterior utilizando notación asintótica.

Solución:

a) La talla es el número de filas (o columnas) de la matriz `a`, $n =$ `a.length`.

b) Para una talla dada n no existen instancias significativas.

c) Se puede considerar como instrucción crítica (de coste constante) la adición del elemento `a[i][j]` de la matriz sobre alguna de las variables `sum`, `sumBelow` o `sumAbove` o la evaluación de la condición del `if` del bucle más interno. Así, la función de coste se puede expresar como sigue:

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

O contando pasos de programa: $T(n) = 1 + \sum_{i=0}^{n-1}(\sum_{j=0}^{n-1} 1 + 1) = 1 + \sum_{i=0}^{n-1}(n+1) = n^2 + n + 1$ *p.p.*

d) La función de coste expresada en notación asintótica es $T(n) \in \Theta(n^2)$.

3. 3 puntos Una *matriz triangular superior* es una matriz cuadrada cuyos valores por debajo de la diagonal principal son todos iguales a 0. El método recursivo `isUpperTriangular` siguiente comprueba si las filas de la matriz `m`, desde `0` hasta `nRow` cumplen esta propiedad. Así, para comprobar si cierta matriz `mat` es triangular superior haríamos la llamada `isUpperTriangular(mat, mat.length - 1)`.

```
/** Precondición: m es una matriz cuadrada de enteros, -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;
    }
```

Se pide:

a) (0.25 puntos) Indica cuál es el tamaño o talla del problema, así como la expresión que la representa.

b) (0.75 puntos) Indica, y justifica, si existen diferentes instancias significativas para el coste temporal del algoritmo e identifícalas si es el caso.

c) (1.50 puntos) Escribe la ecuación de recurrencia del coste temporal en función de la talla para cada uno de los casos si hubiera varios, o una única ecuación si sólo hubiera un caso. Resuélvela(s) por sustitución.

d) (0.50 puntos) Expresa la función de coste utilizando notación asintótica.

Solución:

a) La talla es el número de filas en consideración de la matriz `m`, $n =$ `nRow + 1`.

b) Para una talla dada n sí existen instancias significativas: el caso mejor se da cuando el primer elemento que se analiza, `m[nRow][0]` es distinto de `0`; el caso peor corresponde a una matriz triangular superior, ya que debe recorrer todos los elementos necesarios para comprobar que efectivamente son iguales a `0`.

c) Ecuaciones de recurrencia:

- Para el caso mejor: en la única iteración del bucle `while` se cumple que `m[nRow][0]` es distinto de `0`, se pone la variable `res` a `false`, acaba el bucle y no se hace la llamada recursiva. Así el coste en el caso mejor es constante, i.e. $T^m(n) = 1$.
- Para el caso peor: se realiza una única llamada y el tamaño del problema se reduce en una unidad pero ahora el coste del resto de operaciones (el bucle de búsqueda) es lineal con n; para saber esto, se puede considerar como instrucción crítica la guarda del bucle `j < nRow && res` y ver que el número de veces que se repite es n, para los valores de `j` desde `0` hasta `nRow`. Por ello,

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

Resolviendo por sustitución:
$T^p(n) = T^p(n-1) + n = T^p(n-2) + (n-1) + n = T^p(n-3) + (n-2) + (n-1) + n = \ldots = T^p(n-i) + \sum_{j=0}^{i-1}(n-j)$. Se llega al caso base $T^p(0) = 1$ cuando $n - i = 0$, esto es, cuando $i = n$.

Así, $T^p(n) = 1 + \sum_{j=0}^{n-1}(n-j) = 1 + \sum_{j=0}^{n-1} n - \sum_{j=0}^{n-1} j = 1 + n^2 - \dfrac{n(n+1)}{2} + n = 1 + \dfrac{n(n+1)}{2}$

d) Las funciones de coste de los casos mejor y peor expresadas en notación asintótica son $T^m(n) \in \Theta(1)$ y $T^p(n) \in \Theta(n^2)$ y, si se toma la función de coste del caso mejor como cota inferior y la del caso peor como cota superior, la función de coste expresado en notación asintótica es: $T(n) \in \Omega(1), T(n) \in O(n^2)$.