PRG - ETSInf. TEORÍA. Curso 2013-14. Parcial 1.
14 de abril de 2014. Duración: 2 horas.

1. 3 puntos Dado un array `a` de `int`, escribir un método **recursivo** que sume los números capicúa desde los extremos del array, definidos por el intervalo [`ini`,`fin`] (0≤`ini`,`fin`<`a.length`), hacia el centro hasta sumarlos todos o encontrar algún par de números que no sean iguales. En el caso de que el array tenga una cantidad impar de elementos, el elemento central del intervalo sólo debe sumarse una vez. Si el array no tiene ningún elemento capicúa en los extremos del intervalo, el resultado es cero.

   Algunos ejemplos:

   - para el array `a` = {`1,4,1,2,4,1`}, el resultado de la suma es 10 (1+1+4+4),
   - para el array `a` = {`1,4,2,2,4,1`}, el resultado de la suma es 14 (1+1+4+4+2+2),
   - para el array `a` = {`1,4,1,4,1`}, el resultado de la suma es 11 (1+1+4+4+1),
   - para el array `a` = {`1,4,0,0,1`}, el resultado de la suma es 2 (1+1),
   - para el array `a` = {`8,4,0,0,1`}, el resultado de la suma es 0.

   Se pide:

   a) (0.5 puntos) Perfil del método, añadiendo el/los parámetros adecuados para la resolución recursiva del problema.

   > **Solución:** Una posible solución consiste en definir el método con el siguiente perfil:
   >
   > `public static int sumaCapicua(int[] a, int ini, int fin)`
   >
   > siendo 0≤`ini`, `fin`<`a.length`.

   b) (1.2 puntos) Caso base y caso general.

   > **Solución:**
   >
   > - Caso base, `ini>fin`: Subarray vacío. Se devuelve el neutro de la suma 0.
   > - Caso base, `ini==fin`: Subarray de un elemento. Se devuelve `a[ini]`.
   > - Caso general, `ini<fin`: Si los extremos no son capicúa (`a[ini]!=a[fin]`), se devuelve el neutro de la suma 0. En otro caso (`a[ini]==a[fin]`), el resultado es la suma de los valores de los extremos `a[ini]` y `a[fin]` más la suma de los valores capicúa desde los extremos del subarray `a[ini+1..fin-1]`.

   c) (1 punto) Implementación en Java.

   > **Solución:**
   >
   > ```
   > /** Suma de los valores capicúa desde los extremos del subarray
   >  *  definido por las posiciones ini y fin, 0<=ini, fi<a.length.
   >  *  Si el subarray está vacío o no existen capicúa en los extremos,
   >  *  el resultado es cero.
   >  */
   > public static int sumaCapicua(int[] a, int ini, int fin) {
   > ```

```
    if (ini>fin) return 0;
    else if(ini==fin) return a[ini];
    else if (a[ini]!=a[fin]) return 0;
    else return a[ini] + a[fin] + sumaCapicua(a, ini+1, fin-1);
}
```

d) (0.3 puntos) Llamada inicial.

Solución: Para un array `a`, la llamada `sumaCapicua(a,0,a.length-1)` resuelve el problema del enunciado.

2. 3 puntos Considerar el siguiente método recursivo en Java que comprueba si todos los elementos del subarray `a[pos..a.length-1]` aparecen formando una progresión aritmética de diferencia `d`:

```
/** Devuelve true si para toda pareja a[i],a[i+1] en a[pos..a.length-1]
 *  se cumple que a[i+1] = a[i]+d, y false en caso contrario.
 *  Precondición:
 *  a.length>= 1 && 0<=pos<=a.length-1
 */
public static boolean progAritmetica(int[] a, int d, int pos) {
    if (pos==a.length-1) return true;
    else return (a[pos+1]==a[pos]+d) && progAritmetica(a, d, pos+1);
}
```

Se pide:

a) (0.25 puntos) Indicar cuál es la talla del problema, y qué expresión la define.

Solución: La talla es el número de elementos del array en consideración en cada llamada. La expresión que la define es `a.length-pos`, que llamaremos n.

b) (0.5 puntos) Identificar, caso de que las hubiere, las instancias del problema que representan el caso mejor y peor del algoritmo.

Solución: Existen diferentes instancias porque es un problema de búsqueda en un array.

El *caso mejor* se presenta cuando la condición no se verifica para el primer par de elementos considerados, es decir, cuando la diferencia entre las dos primeras componentes del subarray considerado no es `d` (es decir, `a[pos+1]`$\neq$`a[pos]+d`), de modo que se devuelve `false` sin realizar ninguna llamada recursiva.

El *caso peor* ocurre cuando la condición se verifica para todos los pares de elementos, es decir, cuando todos los pares de componentes consecutivas del subarray considerado cumplen que su diferencia es `d`, de modo que se devuelve `true` cuando se alcanza el caso base, después de realizar el número máximo de llamadas recursivas.

c) (1.5 puntos) Escribir la ecuación de recurrencia del coste temporal en función de la talla para cada uno de los casos si hay varios, o una única ecuación si sólo hay un caso. Resolverla(s) por sustitución.

Solución:

En el caso mejor, para cualquier talla se tiene $T^m(n) = k$, siendo k una constante positiva, en alguna unidad de tiempo.

En el caso peor, el coste se expresa recurrentemente como:

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

siendo k_0, k_1 constantes positivas, en alguna unidad de tiempo. Resolviéndola por sustitución:

$$\begin{aligned} T^p(n) &= k_1 + T^p(n-1) = \ 2 \cdot k_1 + T^p(n-2) = \ 3 \cdot k_1 + T^p(n-3) = \cdots = \\ &= i \cdot k_1 + T^p(n-i) = \cdots = \\ &(caso\ base: \ n - i = 1, \ i = n-1) \\ &= k_1 \cdot (n-1) + T^p(1) = k_1 \cdot n + (k_0 - k_1) \end{aligned}$$

d) (0.75 puntos) Expresar el resultado anterior utilizando notación asintótica.

Solución:

$$T^m(n) \in \theta(1), \ T^p(n) \in \theta(n).$$

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

3. 4 puntos El siguiente método, `triangulo(int)`, determina, escribiéndolos, el número de triángulos rectángulos de lados enteros e hipotenusa `h`:

```
/** El método cuenta, escribiéndolos, todos los triángulos
 *  rectángulos de lados enteros e hipotenusa h. */
public static int triangulo(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;
}
```

Se pide:

a) (0.5 puntos) Determinar la talla del problema.

Solución: La talla n del problema es el parámetro `h`, es decir, la hipotenusa de los triángulos buscados.

b) (0.5 puntos) Indicar si existen diferentes instancias significativas para el coste temporal del algoritmo, e identificarlas si es el caso.

Solución: No existen diferentes instancias, el número de pasadas que realizan los bucles sólo depende de la talla.

c) (2 puntos) Elegir una unidad de medida para el coste (pasos de programa, instrucción crítica) y acorde con ella, obtener una expresión lo más precisa posible del coste temporal del programa (para el caso mejor y el caso peor si es el caso).

Solución:

En pasos de programa:

$$T(n) = 1 + \sum_{i=4}^{n-1}(1 + \sum_{j=3}^{i-1} 1) = 1 + \sum_{i=4}^{n-1}(i-2) = 1 + \frac{(n-1)\cdot(n-4)}{2} \; p.p.$$

Si tomamos como instrucción crítica la evaluación de la condición `c1*c1 + c2*c2 == h*h`, lo que equivale a despreciar términos de orden inferior:

$$T(n) = \sum_{i=4}^{n-1}\sum_{j=3}^{i-1} 1 = \sum_{i=4}^{n-1}(i-3) = \frac{(n-3)\cdot(n-4)}{2} \; \textit{instrucciones críticas}$$

d) (1 punto) Expresar el resultado anterior en notación asintótica.

Solución: $T(n) \in \theta(n^2)$.