PRG - ETSInf. TEORÍA. Curso 2014-15. Parcial 1.
27 de abril de 2015. Duración: 2 horas.

1. 3 puntos Dado un array `a` de `int` con al menos un elemento, escribir un método **recursivo** que compruebe si todos los valores del array son pares y están almacenados en orden creciente.

   Se pide:

   a) (0.5 puntos) Perfil del método, con los parámetros adecuados para resolver recursivamente el problema.

   > **Solución:** Una posible solución consiste en definir un método con el siguiente perfil:
   >
   > ```
   > public static boolean paresYCrecientes(int[] a,int pos)
   > ```
   >
   > siendo `a.length`>0 y 0≤`pos`<`a.length`.

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

   > **Solución:**
   >
   > - Caso base, `pos==a.length-1`: Subarray de un elemento. Devuelve `true` si `a[pos]` es par y `false` en caso contrario.
   > - Caso general, `pos<a.length-1`: Subarray de más de un elemento. Devuelve `true` sii `a[pos]` es par y `a[pos] <= a[pos+1]`, y los elementos del subarray `a[pos+1..a.length-1]` son pares y están ordenados crecientemente. En otro caso, devuelve `false`.

   c) (1 punto) Implementación en Java.

   > **Solución:**
   >
   > ```
   > /** Comprueba si todos los elementos del array a son pares
   >  *  y están ordenados crecientemente.
   >  *  Precondición: a.length>0 y 0<=pos<a.length.
   >  */
   > public static boolean paresYCrecientes(int[] a, int pos) {
   >     if (pos == a.length-1) return a[pos]%2 == 0;
   >     else return a[pos]%2 == 0 && a[pos] <= a[pos+1] && paresYCrecientes(a, pos+1);
   > }
   > ```

   d) (0.3 puntos) Llamada inicial para que se verifique la propiedad sobre todo el array.

   > **Solución:** Para un array `a`, la llamada `paresYCrecientes(a,0)` resuelve el problema del enunciado.

2. 7 puntos Para determinar cuántos elementos de un array `a` son menores que un valor dado `x`, se proponen las dos soluciones siguientes en Java, donde la primera de ellas supone que el array está ordenado ascendentemente.

   - Solución 1

   ```
   /** Devuelve el número de elementos del array a menores que x
    *  Precondición: a está ordenado ascendentemente
    */
   public static int contarMenoresX1(int[] a, int x) {
       int i = a.length - 1;
       while (i >= 0 && a[i] >= x) i--;
       return i + 1;
   }
   ```

- Solución 2

```
/** Devuelve el número de elementos del array a menores que x
 *  Precondición: 0 <= pos <= a.length
 *  Llamada inicial:  contarMenoresX2(a, x, 0);
 */
public static int contarMenoresX2(int[] a, int x, int pos) {
    if (pos == a.length) return 0;
    else if (a[pos] < x) return 1 + contarMenoresX2(a, x, pos + 1);
    else return contarMenoresX2(a, x, pos + 1);
}
```

Se pide:

1. (3 puntos/propuesta) Para cada solución propuesta:
   a) (*0.25 puntos*) Indicar cuál es el tamaño o talla del problema, así como la expresión que la representa.
   b) (*0.5 puntos*) Indicar si existen diferentes instancias significativas para el coste temporal del algoritmo e identificarlas si es el caso.
   c) (*1.5 puntos*) En el caso del método iterativo, elegir una unidad de medida para la estimación del coste (pasos de programa, instrucción crítica) y de acuerdo con ella, obtener una expresión matemática, lo más precisa posible, del coste temporal del programa, a nivel global o en las instancias más significativas en caso de haberlas.
   En el caso del método recursivo, 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 hubiera un caso. Hay que resolverla por sustitución.
   d) (*0.75 puntos*) Expresar el resultado anterior utilizando notación asintótica.

Solución:

Análisis del método de la Solución 1:

a) El tamaño o talla del problema es el número de elementos del array `a` y la expresión que la representa es `a.length`. De ahora en adelante, denominaremos a este número `n`. Esto es, `n`= `a.length`.

b) Se trata de un problema de búsqueda (iterativa descendente) y, por lo tanto, para una misma talla sí que presenta instancias distintas.

   Como el array `a` está ordenado ascendentemente, el caso mejor se da cuando en la última posición del array encontramos un valor menor que `x` (`a[a.length-1]` $<$ `x`), esto es, todos los elementos del array son menores que `x`. El caso peor se da cuando todos los elementos del array son mayores o iguales que `x` ($\forall$`i, 0` $\leq$ `i` $<$ `a.length, a[i]` $\geq$ `x`).

c) Eligiendo como unidad de medida el paso de programa, y considerando una pasada del bucle como un paso de programa y el resto de operaciones como un único paso de programa, la función de coste temporal en el caso mejor será $T^m(n) = 1$ *p.p.* y en el caso peor $T^p(n) = 1 + \sum_{i=0}^{n-1} 1 = n + 1$ *p.p.*

   Eligiendo como unidad de medida la instrucción crítica, la única instrucción que podemos considerar como tal es la guarda del bucle: `i=>0 && a[i]>=x` (se ejecuta siempre una vez más que cualquiera de las del cuerpo del bucle). En el caso mejor sólo se ejecutará una vez. En el caso peor se repetirá tantas veces como elementos tenga el array más una (cuando se hace falsa). La función de coste temporal, considerando la instrucción crítica de coste unitario, en el caso mejor será $T^m(n) = 1$ *i.c.* y en el caso peor $T^p(n) = \sum_{i=0}^{n} 1 = n + 1$ *i.c.*

d) En notación asintótica: $T^m(n) \in \Theta(1)$ y$T^p(n) \in \Theta(n)$. Por lo tanto, $T(n) \in \Omega(1)$ y $T(n) \in O(n)$, es decir, el coste temporal está acotado inferiormente por una función constante y superiormente por una función lineal con la talla del problema.

Análisis del método de la Solución 2:

a) El tamaño o talla del problema es el número de elementos del array `a` en consideración, y la expresión que la representa es `a.length - pos`. De ahora en adelante, denominaremos a este número `n`. Esto es, `n = a.length - pos`.

b) Se trata de un problema de recorrido (recursivo ascendente) y, por lo tanto, para una misma talla no presenta instancias distintas.

c) Para obtener el coste del método, planteamos la ecuación de recurrencia:

$$T(n) = \begin{cases} T(n-1)+k & \text{si } n > 0 \\ k' & \text{si } n = 0 \end{cases}$$ siendo k y constantes k' positivas, en alguna unidad de tiempo.

Resolviendo por sustitución: $T(n) = T(n-1)+k = T(n-2)+2k = \ldots = T(n-i)+ik$. Al llegar al caso base, para talla 0, $n - i = 0 \rightarrow n = i$. Con lo que $T(n) = k' + nk$.

d) En notación asintótica: $T(n) \in \Theta(n)$, es decir, el coste temporal depende linealmente de la talla del problema.

II. (1 punto) Teniendo en cuenta que el algoritmo de ordenación por *inserción directa* visto en clase tiene un coste lineal en el mejor caso y cuadrático en el peor, y que para la primera solución habría que ordenar el array previamente, comparar el coste temporal total de contar los valores menores que `x` en un array `a` no necesariamente ordenado, cuando se resuelve el problema mediante los siguientes algoritmos:

- Algoritmo 1: Primero ordenar `a` por inserción directa y después aplicar el método `contarMenoresX1(int[], int)`.
- Algoritmo 2: Aplicar directamente al array `a` el método `contarMenoresX2(int[], int, int)`.

Solución:
La resolución del problema aplicando el algoritmo 1 tendrá las siguientes cotas de complejidad:

- En el caso mejor, el array está ordenado ascendentemente y todos sus elementos son menores que `x`. Aplicar la ordenación por inserción directa tendrá un coste $\Theta(n)$ y contar el número de elementos menores que `x` será $\Theta(1)$. Por lo tanto, el coste en el caso mejor será lineal con la talla del problema.
- En el caso peor, los elementos del array están ordenados descendentemente y todos ellos son mayores o iguales que `x`. El coste de la ordenación por inserción directa será $\Theta(n^2)$ y contar el número de elementos menores que `x` será $\Theta(n)$. Por lo tanto, el coste en el caso peor será cuadrático con la talla del problema.

La resolución del problema aplicando el algoritmo 2, sea cual sea el array `a`, tendrá un coste $\Theta(n)$.
Por lo tanto, podemos concluir que el algoritmo 2 es más eficiente cuando el problema no está restringido a arrays ordenados.