PRG - ETSInf. TEORÍA. Curso 2012-13. Parcial 1.
22 de abril de 2013. Duración: 2 horas.

1. $\boxed{\text{4 puntos}}$ Dado un array `a` de `int` y un entero `m`, escribir un método **recursivo** que compruebe si existe una pareja `a[i]` y `a[f]`, 0≤`i`≤`f`<`a.length`, de componentes *simétricas* (la distancia de `i` a 0 es la misma que la distancia de `f` a `a.length`−1) que sumen `m`. Si existe la pareja, se debe devolver el índice en el que se encuentra (el índice `i`, el más bajo de la pareja), −1 en caso contrario.

   Por ejemplo, para `m`=4 y `a`={1,4,5,9,6,0,-8} el método en la llamada inicial que se extienda sobre todo el array ha de devolver 1, para `m`=18 y `a` ={1,4,5,9,6,0,2} ha de devolver 3, para `m`=25 y `a`={1,3,2,5,4,6} ha de devolver −1.

   Se pide:

   a) Perfil del método que se va a escribir, añadiendo el/los parámetros adecuados para resolver recursivamente el problema.

   > **Solución:** Una posible solución consiste en definir el método como
   >
   > `public static int parejaSim(int[] a,int ini,int fi,int m)`
   >
   > de manera que siendo 0≤`ini`, `fi<a.length`, se delimita a buscar una pareja de elementos simétricos en el subarray `a[ini..fi]`.

   b) Caso base y caso general.

   > **Solución:**
   >
   > - Caso base `ini>fi`: No se encuentra la pareja buscada, se tiene que devolver -1.
   > - Caso general, `ini<=fi`. Si `a[ini]+a[fi]` vale `m`, se tiene que devolver `ini`, sino la búsqueda se reduce a `a[ini+1..fi-1]`.

   c) Implementación en Java.

   > **Solución:**

```
/** Busca el índice de la primera pareja de elementos simétricos en a[ini..fi],
 * 0<=ini, fi<a.length, que sumados dan m. Si no existe, devuelve -1.
 */
  public static int parejaSim(int[] a,int ini,int fi,int m){
        if (ini>fi) return -1;
        else if (a[ini]+a[fi]==m) return ini;
        else return parejaSim(a,ini+1,fi-1,m);
  }
```

   d) Llamada inicial.

   > **Solución:** Para un array `a` y un entero `m`, la llamada `parejaSim(a,0,a.length-1,m)` resuelve el problema del enunciado.

2. $\boxed{\text{3 puntos}}$ Dada cierta matriz `m` cuadrada y un array `a`, de enteros, los dos con la misma dimensión (esto es, `m.length == a.length`), el siguiente método **iterativo** devuelve la posición (número de fila) donde se encuentra el array `a` en `m`, caso de que se encuentre, o devuelve -1 si no fuera así:

```
/**
 * Devuelve la posición en que el array "a" se encuentra como fila en la matriz
 * cuadrada "m", o -1 si no se encuentra.
 * PRECONDICION: m es cuadrada y m.length == a.length
 */
public static int buscaPosFila(int[][] m, int[] a) {
    boolean enc = false;
    int i = 0;
    while (i<m.length && !enc) {
        enc = true;
        for (int j=0; j<m.length && enc; j++)
            enc = (m[i][j]==a[j]);
        if (!enc) i++;
      }
    if (enc) return i; else return -1;
}
```

Se pide el estudio del coste temporal:

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

> **Solución:** La talla del problema es $n =$ `m.length`, es decir, la dimensión de la matriz.

b) Identifica, caso de que las hubiere, las instancias del problema que representan el caso mejor y peor del algoritmo.

> **Solución:** El método es un problema de búsqueda y, por tanto, para una misma talla sí que presenta instancias distintas.
>
> *Caso mejor*. Por una parte, si las n componentes de `a` aparecen en la primera fila de la matriz, el bucle exterior solo ejecuta una pasada, en la que se comprueba en n pasadas que las `n` sucesivas componentes de `a[0..n-1]` coinciden con las correspondientes componentes de `m[0]`.
>
> Por otra parte, se puede dar que el bucle interno sea lo más corto posible para todas las filas si el primer elemento de cada fila es diferente de `a[0]`. En ese caso el bucle exterior completaría las n pasadas (una por cada fila de la matriz), y todas ellas con el coste de hacer una sola comprobación.
>
> *Caso peor*. Ocurre cuando para todas las filas se hacen el máximo de comprobaciones posibles: las componentes de `a[0..n-2]` coinciden con las correspondientes de la fila, pero `a` no aparece completo en ninguna fila de `m` (excepto quizás la última).

c) Elige una unidad de medida para la estimación del coste (pasos de programa, instrucción crítica) y acorde con ella, obtén una expresión matemática, lo más precisa posible, del coste temporal del programa, a nivel global o en las instancias más significativas si las hay.

Solución:

- Si optamos por escoger como unidad de medida el paso de programa, se obtiene:

  En el caso mejor, $T^m(n) = 1 + \sum_{i=0}^{n-1} 1 = n + 1$ pasos.

  En el caso peor, $T^p(n) = 1 + \sum_{i=0}^{n-1}(1 + \sum_{j=0}^{n-1} 1) = 1 + \sum_{i=0}^{n-1}(1 + n) = 1 + n + n^2$ pasos.

- Si optamos por escoger la instrucción crítica como unidad de medida para la estimación del coste, se puede considerar como tal la comparación: `(m[i][j] == a[j])`.

  En el *caso mejor* se ejecutará n veces y la función de coste temporal será $T^m(n) = n$.

  En el *caso peor* se repetirá el número máximo de veces posible y la función de coste será $T^p(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = n \cdot n = n^2$.

d) Expresa el resultado anterior utilizando notación asintótica.

Solución: $T(n) \in \Omega(n)$, $T(n) \in O(n^2)$.

3. $\boxed{\text{2 puntos}}$ El siguiente método comprueba si cierto subarray `a[i..f]` de `int` está o no ordenado ascendentemente, subdividiéndose previamente y comprobando que cada una de sus dos partes lo esté, así como la relación entre ellas:

```
/** Devuelve true sii a[i..f] está ordenado ascendentemente. Precondición: i<=f */
 public static boolean estaOrdenado(int[] a, int i, int f) {
     if (i==f) return true;
     else {
         int m = (i+f)/2;
         return estaOrdenado(a,i,m) && estaOrdenado(a,m+1,f) && a[m]<=a[m+1];
     }
 }
```

Se pide el estudio del coste temporal:

a) Indica cuál es la talla del problema y qué expresión la define.

Solución: La talla, n, es el tamaño del subarray comprendido entre las posiciones i y f. Esto es, $n = f - i + 1$.

b) Determina si existen instancias significativas. Si las hay, identifica las que representan los casos mejor y peor del algoritmo.

Solución: El mejor caso es aquel en el que siempre se ejecuta solo la primera llamada, esto es, cuando el primer elemento está fuera de orden (cuando inicialmente `a[i]` $>$ `a[i+1]`). El peor caso es aquel en el que se han de ejecutar todas las llamadas recursivas. Esta situación se da, por ejemplo, cuando inicialmente el subarray `a[i..f]` está ordenado.

c) 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 por sustitución.

Solución:

Mejor Caso:

$$T^m(n) = \begin{cases} k & n = 1 \\ T^m(\frac{n}{2}) + k' & n > 1 \end{cases}$$

$$\begin{array}{lll} 1) & T^m(n) = & T^m(\frac{n}{2}) + k' \\ 2) & & T^m(\frac{n}{2^2}) + 2k' \\ 3) & & T^m(\frac{n}{2^3}) + 3k' \\ .. & & ... \\ i) & & T^m(\frac{n}{2^i}) + ik' \end{array}$$

$\frac{n}{2^i} = 1; \quad n = 2^i; \quad log_2 n = i;$

$T^m(n) = k + k' log_2 n$

Peor Caso:

$$T^p(n) = \begin{cases} k & n = 1 \\ 2T^p(\frac{n}{2}) + k' & n > 1 \end{cases}$$

$$\begin{array}{lll} 1) & T^p(n) = & 2T^p(\frac{n}{2}) + k' \\ 2) & & 2^2 T^p(\frac{n}{2^2}) + k'(1+2) \\ 3) & & 2^3 T^p(\frac{n}{2^3}) + k'(1+2+2^2) \\ .. & & ... \\ i) & & 2^i T^p(\frac{n}{2^i}) + k' \sum_{j=0}^{i-1} 2^j \end{array}$$

$\frac{n}{2^i} = 1; \quad n = 2^i;$

$T^p(n) = 2^i k + (2^i - 1)k' = nk + (n-1)k' = n(k+k') - k'$

d) Expresa el resultado anterior usando notación asintótica.

Solución:

$$T(n) \in \Omega(log\ n)$$
$$T(n) \in O(n)$$

4. $\boxed{\text{1 punto}}$ Es posible modificar una única instrucción del algoritmo anterior para que su coste temporal sea $\Omega(1)$.

¿Qué instrucción se tendría que modificar y de qué forma?

Solución: Hay que modificar la instrucción que tiene las llamadas recursivas para que se ejecute en primer lugar la comprobación, esto es, habría que sustituirla por:

```
return a[m]<=a[m+1] && estaOrdenado(a,i,m) && estaOrdenado(a,m+1,f);
```