Solución al parcial de EDA de febrero de 2012

1. Eligiendo algoritmos:
   Los algoritmos A y B son equivalentes cuando $207 + 4n^2 = 3n^4$; restando, tenemos $3n^4 - 4n^2 - 207 = 0$; se puede sacar una solución (con valores fraccionarios para n) usando una variable auxiliar $x = n^2$; pero es más fácil substituir valores por prueba y error (ya que B crece mucho, mucho más rápido que A):
   Con n = 2: $(207 + 4 \cdot 2 \cdot 2 = 207 + 16)$ > $(3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 3 \cdot 16)$
   Con n = 3: $(207 + 4 \cdot 3 \cdot 3 = 207 + 36)$ = $(3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 3 \cdot 81)$
   Con lo cual, para $n > 3$, es mejor usar A; y para $n < 3$ es mejor usar B; cuando $n = 3$, son equivalentes

2. Comparando complejidades:
   1. El logaritmo crece mucho más lento que la raíz cuadrada; por lo tanto, $n\,log\,n \in O(n\sqrt{n})$
   2. Expandiendo, tenemos $n^2 + 2n + 1$ y $n^2 - 2n + 1$, ambas en $O(n^2)$: $(n+1)^2 \in \Theta(n-1)^2$ (y viceversa). Si buscamos constantes c_1 y c_2 para que se cumpla esta relación, pueden valer $c_1 = ½$ y $c_2 = 2$.
   3. Expandiendo la primera, tenemos $(n+1)! = (n+1) \cdot n!$; por tanto, $n! \in O((n+1)!)$
   4. El crecimiento de a^n es exponencial, mientras que el de n^aes polinómico. Por tanto, $n^a \in O(a^n)$; ambas se cruzarán cuando $a^n = n^a$; y a partir de cierto momento, siempre será mayor la exponencial. Por ejemplo, si a=1.5, entonces $n \geq 8$ resultará en que $a^n > n^a$

   NOTA: no hacía falta dar explicaciones, pero se incluyen para facilitar la comprensión de las soluciones.

3. Número exacto de ejecuciones:
   Para n=0, {A} se ejecutará 0 veces
   Para n=1, {A} se ejecutará con i=0, j=0: 0 + 1 = 1 veces
   Para n=2, {A} se ejecutará con i=0, j=0; i=1, j=0; i=1, j=1: 1 + 2 = 3 veces
   Y, en general, si *A(n)* es el número de ejecuciones de {A} para una *n*, entonces $A(n) = A(n-1) + n$, que se puede escribir como $\sum_{i=0}^{n} n = \frac{n^2+n}{2}$
   Si {A} tiene complejidad O(n), y dado que {A} se está llamando $O(n^2)$ veces, podemos afirmar que el bucle exterior tiene complejidad $O(n^3)$

4. Postcondiciones ($n \geq 2$ elementos)
   1. $b = (1 = (\#\, p : 0 \leq p < n : v[p] = 3p))$
   2. $s = (\max i, j : 0 \leq i < j < n \wedge v[i] \neq v[j] : v[i] + v[j])$

5. Algoritmo iterativo para encontrar pico, dado un vector montaña con $n \geq 1$

```
// pre: n >= 1 y existe p : 0 <= p < n : cumbre(p, v, n)
int posCumbre(const int v[], int n) {

    int p=0;
    while (p<n-1 && v[p]<v[p+1]) {
        p ++;
    }
    return p;
}
// post: 0 <= p < n y cumbre(p, v, n)
```

Usamos el predicado auxiliar $cumbre(p,v,n) = \forall\, i,j,k,l : 0 \leq i < j \leq p \leq k < l < n :$
$v[j] < v[k] \wedge v[k] > v[l]$ (hay otras muchas formas de especificarlo; por ejemplo, usando predicados auxiliares "creciente" y "decreciente")

Como **invariante**, usamos

$I = 0 \leq p < n \wedge \exists\, i : p \leq i < n : cumbre(i,v,n)$ - Es decir, nuestra invariante es que el índice *p* todavía no ha *sobrepasado* la cumbre (pero puede *ser* la cumbre).

Demostramos $P \implies I$, es decir, que **la invariante se cumple antes** de entrar en el bucle:

$P = \exists\, i : 0 \leq i < n : cumbre(i,v,n)$
$\{P\}\ p = 0\,\{I\}$ ya que $pmd(p = 0; I) \Leftrightarrow I_p^0 \Leftrightarrow 0 \leq 0 < n \wedge \exists\, i : 0 \leq i < n : cumbre(i,v,n) \Leftarrow P$

Demostramos $\{I \wedge B\}A\,\{I\}$, es decir, la i**nvariante se cumple tras cada ejecución** del bucle

$\{I \wedge B\}\ p = p + 1\ \{I\} \Leftrightarrow \{I \wedge (p < n - 1 \wedge v[p] < v[p+1])\}\, p = p + 1\,\{I\}$
$\Leftrightarrow\ I_p^{p+1}\ \Leftrightarrow\ 0 \leq p + 1 < n \wedge \exists\, i : p + 1 \leq i < n : cumbre(i,v,n)$
$\Leftrightarrow 0 \leq p < n - 1 \wedge \exists\, i : p + 1 \leq i < n : cumbre(i,v,n)$
$\Leftarrow I \wedge (p < n - 1 \wedge v[p] < v[p+1])$
Ya que, si hay una cumbre en *p* o más allá de *p*, y *p* no es cumbre ($v[p] < v[p+1]$), entonces la cumbre tiene que seguir existiendo en algún índice mayor que *p*.

Demostramos $\{I \wedge \neg B\}A\,\{Q\}$, es decir, **al salir del bucle se cumple la postcondición**:

$Q =\ 0 \leq p < n \wedge cumbre(p,v,n)$
$\{I \wedge \neg B\} \Rightarrow \{Q\}$
$\big((p = n - 1) \vee (v[p] < v[p+1])\big) \wedge \big(0 \leq p < n \wedge \exists\, i : p \leq i < n : cumbre(i,v,n)\big)$
$\Rightarrow 0 \leq p < n \wedge cumbre(p,v,n) \Rightarrow Q$ – Ya que sólo salimos cuando el siguiente es menor o-igual (como existe cumbre, tiene que ser menor, y éste es por tanto la cumbre) o no hay siguiente. En ambos casos, *p* contiene el índice de la cumbre.

Finalmente, una buena **cota** es $n - p - 1$, que, mientras se ejecuta el bucle, **es positiva**
(al entrar en el bucle tenemos $p < n - 1$, y por tanto $n - p - 1\ > 0\ \Leftrightarrow n - 1 > p$: cierto
Siempre decrece, ya que *p* aumenta en cada iteración; es decir, $(n - p - 1)_p^{p+1} < n - p - 1$
Y, cuando la **cota llega a cero** (aunque no hay *obligación* de que llegue a cero), se sale del bucle:
$n - p - 1 = 0\ \Leftrightarrow p = n - 1 \Rightarrow \neg(p < n - 1 \wedge v[p] < v[p+1]) \equiv \neg B$

Finalmente, el algoritmo tiene complejidad $O(n)$, ya que, a lo sumo, el bucle se ejecutará exactamente $n - 1$ veces.

6. Algoritmo recursivo de orden logarítmico para encontrar si existen elementos que coinciden con su índice en un vector estrictamente creciente con $n \geq 0$

// $P = n \geq 0 \wedge creciente(v,n)$, con $creciente(v,n) = \forall i,j : 0 \leq i < j < n : v[i] < v[j]$

```
bool especial(const int v[], int n) {
    return aux(v, 0, n);
}
```
// $Q = (b = \exists i : 0 \leq i < n : v[i] = i)$

Que se pida un orden logarítmico hace pensar en usar una *búsqueda binaria*. Efectivamente, basta con una búsqueda binaria donde, en lugar de buscar una 'x' fija, buscamos, en cada i, el valor $v[i]$.

// $P_{aux} = a \leq b \wedge creciente(v,n) \wedge (\forall i,j : 0 \leq i < a \vee b < j < n : v[i] < i \wedge v[j] > j)$

(es decir, si existe un índice con $v[i] = i$, debe de estar comprendido entre *a* y *b*)

```
bool aux(const int v[], int a, int b) {
    int m = (a+b)/2;
    if (a+1 > b) {              // d1; necesario cuando a == b, que sólo ocurre con n=0
        return false;           // g1
    } else if (a+1 == b) {      // d2
        return v[m] == m;       // g2
    } else if (v[m] <= m) {     // no-d1: a+1 < b y v[m] <= m
        return aux(v, m, b);    // c1 y s1: si existe ese índice, está en la mitad >= m
    } else {                    // no-d2: a+1 < b y v[m] > m
        return aux(v, a, m);    // c2 y s2: si existe ese índice, está en la mitad < m
    }
}
```
// $Q_{aux} = (r = \exists i : 0 \leq i < n : v[i] = i)$
Verificaremos el algoritmo auxiliar, que es el que realmente hace el trabajo.

1. Se cubren todos los casos, ya que siempre se entra por alguna de las 4 ramas del 'if' (que corresponde, respectivamente a los casos directos e indirectos).
2. El caso base verifica la post-condición:
    a. Si entramos por *g1*, entonces es que *a* y *b* se han cruzado; como, por la pre-condición, si i no está entre *a* y *b* no puede cumplir $v[i] = i$, devolvemos falso.
    b. Si entramos por *g2*, entonces es que *a* y *b* se han encontrado. Basta con verificar si el único elemento que puede cumplir $v[i] = i$ lo cumple.
3. Los argumentos de la llamada recursiva satisfacen su precondición: en ambos casos (c1 y c2) se cumple que $a \leq \left\lfloor \frac{a+b}{2} \right\rfloor \leq b$, con lo cual $a \leq m$ y $m \leq b$; sóo resta la parte que dice $(\forall i,j : 0 \leq i < a \vee b < j < n : v[i] < i \wedge v[j] > j)$, que se verifica como sigue:
    a. $v[m] \leq m \Rightarrow (\forall i,j : 0 \leq i < m \vee b < j < n : \boldsymbol{v[i]} < \boldsymbol{i} \wedge v[j] > j)$, ya que para todos los índices menores que m, los valores serán todavía menores que para v[m] (el vector es creciente).
    b. $v[m] > m \Rightarrow (\forall i,j : 0 \leq i < a \vee m < j < n : v[i] < i \wedge \boldsymbol{v[j]} > \boldsymbol{j})$, ya que el vector es creciente, y los valores de todos los índices mayores que m serán siempre mayores que sus índices.
4. El paso de inducción es correcto, ya que cuando se descarta una mitad del vector (ya sea mediante c1 ó c2) es porque el elemento buscado no puede estar dentro de esa mitad (ver punto 3); y por tanto, de existir, tiene que existir en la mitad restante.
5. La cota es $t(v,a,b) = b - a$, que es positiva porque $a \leq b$, y decrece a cada iteración $\left(t(s_1(v,a,b)) = b - \left\lfloor \frac{a+b}{2} \right\rfloor < b - a\right) \wedge \left(t(s_2(v,a,b)) = \left\lfloor \frac{a+b}{2} \right\rfloor - a < b - a\right)$, ya que $a \leq \left\lfloor \frac{a+b}{2} \right\rfloor \leq b$; además, cuando $b - a = 0$, estamos en una de los casos base y salimos de la función.