Resolución de la Recuperación del Primer Parcial de EDA (20 de Junio de 2018)

1.- Analiza el coste del siguiente método recursivo, que comprueba si dos subarrays del mismo tamaño, no vacíos y de tipo genérico son idénticos, i.e. contienen los mismos elementos ocupando las mismas posiciones. **(1.5 puntos)**

```
private static <E> boolean sonIguales(E[] a, E[] b, int inicio, int fin) {
    if (inicio > fin) return true;
    int mitad = (fin + inicio) / 2;
    boolean res = a[mitad].equals(b[mitad]);
    if (res) {
        res = sonIguales(a, b, inicio, mitad – 1);
        if (res) { res = sonIguales(a, b, mitad + 1, fin);
    }
    return res;
}
```

Para ello:

a) Expresa la talla del problema **x** en función de los parámetros del método. **(0.2 puntos)**

x = `fin – inicio + 1`

b) Indica si existen instancias significativas para una talla dada y por qué; en caso afirmativo descríbelas**.** **(0.4 puntos)**

Sí hay instancias significativas: se comprueba si a y b son iguales con una Búsqueda de la primera componente de a que, ocupando idéntica posición que una de b, **no** sea igual a ella.

- **Mejor de los Casos:** las componentes **centrales** de a y b **no** son iguales ya en la primera llamada al método. Así, basta un tiempo constante para determinar que `res` es `false` y, por tanto, a y b **no** son iguales.
- **Peor de los Casos: los** arrays a y b **sí** son iguales, i.e. contienen las mismas componentes en el mismo orden. Así, `res` siempre se evalúa a `true` y se alcanza el caso base del método tras haber realizado tantas llamadas recursivas como componentes tenga a (o b).

c) Escribe la(s) Relación(es) de Recurrencia que expresan el coste del método, en consonancia con tu apartado **b)**. **(0.4 puntos)**

En el caso general, **cuando x >** **0** **,**

- **Mejor de los Casos:** como no se produce llamada alguna en el cuerpo del método, $T^M_{sonIguales}(x) = k1$.
- **Peor de los Casos:** $T^P_{sonIguales}(x) = 2 * T^P_{sonIguales}(x/2) + k3$.

d) Resuelve la(s) Relación(es) de Recurrencia de tu apartado **c)**, indicando el(los) Teorema(s) de Coste que usas y los valores de sus coeficientes. Hecho esto, escribe el coste temporal del método usando la notación asintótica (O y Ω o bien Θ). **(0.5 puntos)**

- **Peor de los Casos:** por T3 con a=c=2 y sobrecarga constante, $T^P_{sonIguales}(x) \in \Theta(x)$.
  Por tanto, $T_{sonIguales}(x) \in O(x)$
- **Mejor de los Casos:** $T^M_{sonIguales}(x) \in \Theta(1)$.
  Por tanto, $T_{sonIguales}(x) \in \Omega(1)$

2. Se dispone de dos arrays no vacíos de tipo genérico a y b, idénticos salvo por una sola cosa: a contiene un elemento más que b. Dos ejemplos concretos (en los que se ha instanciado el tipo genérico) de estos arrays serían:

- a = [2, 4, 16, 8, **9**, 3] y b = [2, 4, 16, 8, 3]. Nota que a tiene el elemento adicional en la posición 4.
- a = ["sol", "bar", "col", **"mar"**, "pez", "rayo"] y b = ["sol", "bar", "col", "pez", "rayo"]. Nota que a tiene el elemento adicional en la posición 3.

Precisamente para obtener la posición que ocupa en a el elemento adicional se ha diseñado el siguiente método DyV. Escribe el código que creas oportuno donde se te indica para que el coste temporal del método sea, como máximo, logarítmico con la talla del problema y, como mínimo, del orden de una constante (independiente de la talla). **(2.5 puntos)**

IMPORTANTE: nota que los arrays **NO están ordenados** y, por tanto, sus elementos **NO** tienen por qué ser `Comparable`.

```
public static <E> int posicionAdicional(E[] a, E[] b) {

    // Hipótesis: el elemento adicional es el último de a
    int res = a.length - 1;
    if (!a[b.length - 1].equals(b[b.length - 1])) {
        // Si la hipótesis NO es cierta, se busca la posición
        // del elemento adicional en 2 subarrays de igual tamaño
        // COMPLETAR:
        res = posicionAdicional(a, b, 0, b.length – 1);
    }
    return res;
}

// COMPLETAR:
protected static <E> int posicionAdicional(E[]a, E[]b, int inicio, int fin) {
    // Búsqueda con garantía de éxito
    int mitad = (inicio + fin) / 2;
    if (!a[mitad].equals(b[mitad])) {
        if (mitad == 0 || a[mitad - 1].equals(b[mitad - 1])) { return mitad; }
        else { return posicionAdicional(a, b, inicio, mitad - 1); }
    }
    else { return posicionAdicional(a, b, mitad + 1, fin); }
}
```

3- Dado un array de `Integer` v, diseña un método estático que, en tiempo O(`v.length`), compruebe si existe en v un subarray cuyos elementos suman `0`. A continuación figuran varios ejemplos de lo que debe hacer este método:

- Si v = [1, 2, 3] o [-5, 4] o [6] devolverá `false`, pues en ninguno de los tres casos existe un subarray de v cuyos elementos sumen `0`.
- Si v = [1, 0, 3] o [-5, 5] o [0] devolverá `true`, pues suman 0, respectivamente, el subarray [0] (la segunda componente de v), [-5, 5] y [0] -nota que en los dos últimos casos, el subarray de v que suma 0 es el propio v.
- Si v = [15, -2, 2, -8, 1, 7, 10, 23] o [-5, 4, 0] o [5, 2, -4, 3, -6, 1] devolverá `true`, pues suman 0, respectivamente, los elementos de los subarrays [-2, 2],[0] (la tercera componente de v) y los de [5, 2, -4, 3, -6].

PISTAS: para diseñar este método debes usar un Map auxiliar en el que almacenar las sumas sucesivas de los elementos del array; dicho Map está implementado mediante una Tabla Hash. También ten en cuenta que si la suma de los `i` primeros elementos de v es igual a la de sus `j` primeros elementos (con `i`≠`j`), entonces todos los elementos desde `i` + 1 hasta `j` suman cero. **(3 puntos)**

```
public static boolean metodoMap(Integer[] v) {
    Map<Integer, Integer> aux = new TablaHash<Integer, Integer>(v.length);
    int sumaHastaI = 0;
    for (int i = 0; i < v.length; i++) {
        sumaHastaI += v[i];
        Integer valor = aux.recuperar(sumaHastaI);
        if (sumaHastaI != 0 && valor == null) {
            aux.insertar(sumaHastaI, sumaHastaI);
        }
        else { return true; } // o sumaHastaI es 0 o valor != null
    }
    return false;
}
```

4.- Diseña un método genérico, estático e iterativo `fusion` que dadas `cP1` y `cP2`, dos Colas de Prioridad implementadas mediante Heaps, devuelva una Lista Con PI que contiene los datos de `cP1` y `cP2` ordenados ascendentemente; dicho método no puede hacer uso de ninguna estructura de datos auxiliar para calcular su resultado y, además, `cP1` y `cP2` deben estar vacías al concluir su ejecución. El coste temporal del método diseñado deberá ser Θ(x·logx), siendo x la talla de la Lista resultado. **(3 puntos)**

```
public static <E extends Comparable<E>> ListaConPI<E> fusion(ColaPrioridad<E> cP1,
                                                             ColaPrioridad<E> cP2) {
    ListaConPI<E> res = new LEGListaConPI<E>();
    while (!cP1.esVacia() && !cP2.esVacia()) {
        if (cP1.recuperarMin().compareTo(cP2.recuperarMin()) < 0) {
            res.insertar(cP1.eliminarMin());
        }
        else { res.insertar(cP2.eliminarMin()); }
    }
    while (!cP1.esVacia()) { res.insertar(cP1.eliminarMin()); }
    while (!cP2.esVacia()) { res.insertar(cP2.eliminarMin()); }
    return res;
}
```

ANEXO

La interfaz `Map` del paquete `modelos`.

```
public interface Map<C, V> {
    V insertar(C c, V v);
    V eliminar(C c);
    V recuperar(C c);
    boolean esVacio();
    int talla();
    ListaConPI<C> claves();
}
```

La interfaz `ColaPrioridad` del paquete `modelos`.

```
public interface ColaPrioridad<E extends Comparable<E>> {
    void insertar(E e);
    /** SII !esVacia() */ E  eliminarMin();
    /** SII !esVacia() */ E  recuperarMin();
    boolean esVacia();
}
```

Teoremas de coste:

Teorema 1: $f(x) = a \cdot f(x - c) + b$, con $b \geq 1$

- si $a=1$, $f(x) \in \Theta(x)$;
- si $a>1$, $f(x) \in \Theta(a^{x/c})$;

Teorema 2: $f(x) = a \cdot f(x - c) + b \cdot x + d$, con b y $d \geq 1$

- si $a=1$, $f(x) \in \Theta(x^2)$;
- si $a>1$, $f(x) \in \Theta(a^{x/c})$;

Teorema 3: $f(x) = a \cdot f(x/c) + b$, con $b \geq 1$

- si $a=1$, $f(x) \in \Theta(\log_c x)$;
- si $a>1$, $f(x) \in \Theta(x^{\log_c a})$;

Teorema 4: $f(x) = a \cdot f(x/c) + b \cdot x + d$, con b y $d \geq 1$

- si $a<c$, $f(x) \in \Theta(x)$;
- si $a=c$, $f(x) \in \Theta(x \cdot \log_c x)$;
- si $a>c$, $f(x) \in \Theta(x^{\log_c a})$;

Teoremas maestros:

Teorema para recurrencia divisora: la solución a la ecuación $T(n) = a \cdot T(n/b) + \Theta(n^k)$, con $a \geq 1$ y $b>1$ es:

- $T(n) = O(n^{\log_b a})$ si $a>b^k$;
- $T(n) = O(n^k \cdot \log n)$ si $a=b^k$;
- $T(n) = O(n^k)$ si $a<b^k$;

Teorema para recurrencia sustractora: la solución a la ecuación $T(n) = a \cdot T(n-c) + \Theta(n^k)$ es:

- $T(n) = \Theta(n^k)$ si $a<1$;
- $T(n) = \Theta(n^{k+1})$ si $a=1$;
- $T(n) = \Theta(a^{n/c})$ si $a>1$;