Resolución del Primer Parcial de EDA (11 de Abril de 2018)

1.- Diseña una método en la clase `LEGListaConPI` que, accediendo únicamente a los atributos de la clase, elimine todos los elementos que hay antes del Punto de Interés (PI) de una Lista y devuelva como resultado cuántos ha eliminado. El PI de la Lista deberá permanecer inalterado. **(2.5 puntos)**

```
public int metodo1() {
    if (ant == pri) return 0; // Si NO hay elementos ANTES del PI, se borran 0 elementos
    int res = 0;              // Sino, hay que contar cuántos elementos hay ANTES del PI
    for (NodoLEG<E> aux = pri; aux != ant; aux = aux.siguiente) { res++; }
    // Tras contarlos, ...
    // PASO 1: actualizar la talla
    talla = talla - res;
    // PASO 2: Borrado "lazy" de todos los elementos ANTES del que ocupa el PI,
    // que consiste en desenlazar los nodos que los contienen como sigue:
    // 2.1 - El elemento que ocupa el PI debe pasar a ser el primero de la lista.
    pri.siguiente = ant.siguiente; // El nodo cabecera pasa a apuntar al que ocupa el PI
    ant = pri; // Y el anterior al PI es ahora el nodo cabecera
    // 2.2 - Si hubiera que borrar todos los elementos de la Lista, ult se debe actualizar
    if (talla == 0) { ult = pri; }
    return res;
}
```

2.- Diseñar un método estático Divide y Vencerás que, dado un array `v` de `Integer` ordenado ascendentemente y sin elementos repetidos y un `Integer` `e`, devuelva el sucesor de `e` en `v`, o null si no existe tal sucesor en `v`. Recuerda que el sucesor de un elemento `e` es el menor de todos los elementos mayores que `e`. **(2.5 puntos)**

```
public static Integer metodo2(Integer[] v, Integer e) {
    return metodo2(v, e, 0, v.length - 1);
}
private static Integer metodo2(Integer[] v, Integer e, int i, int f) {
    Integer res = null; // Caso base implícito: si se alcanza, el sucesor de e NO está en v[i, f]
    if (i <= f) { // Caso general: el sucesor de e PUEDE estar en v[i, f]
        int m = (i + f) / 2;
        if (v[m].compareTo(e) > 0) {  // el sucesor de e PUEDE ser v[m]
            if (m == 0 || v[m - 1].compareTo(e) <= 0) { res = v[m]; } // v[m] ES el sucesor de e
            else { res = metodo2(v, e, i, m - 1); } // el sucesor de e PUEDE estar en v[i, m - 1]
        }
        else { res = metodo2(v, e, m + 1, f); } // el sucesor de e PUEDE estar en v[m + 1, f]
    }
    return res;
}
```

a) Indica la talla del problema: x = `f - i + 1`, o `v.length` en la llamada más alta

b) Escribe las Relaciones de Recurrencia para el Mejor y Peor de los casos.

En el caso general, cuando x > 0:

$T^{M}_{metodo2}(x) = k$

$T^{P}_{metodo2}(x) = 1 * T^{P}_{metodo2}(x / 2) + k'$

c) Indica el coste Temporal del método que has diseñado utilizando la notación asintótica (O y Ω o bien Θ).

$T_{metodo2}(x) \in \Omega(1)$ y $T_{metodo2}(x) \in O(\log x)$

3.- Diseña un método estático que, dados un array v de `Integer` y un `Integer` e, devuelva un `String` en el que se indique, con el formato que se muestra en el siguiente ejemplo, los pares de elementos de v que sumen e. Por ejemplo, si v = [1, 4, 6, 3, 8, 9, 5, 2} y e = 10, el resultado del método es el `String` resultante que figura a continuación: **(2.5 puntos)**

```
v[1] + v[2] = 4 + 6 = 10
v[0] + v[5] = 1 + 9 = 10
v[4] + v[7] = 8 + 2 = 10
```

A la hora de diseñar este método debes tener en cuenta las siguientes consideraciones:

- El array v no debe contener elementos repetidos. Así que, en cuanto tu método detecte un repetido debe devolver como resultado el `String` `"Error: hay elementos repetidos"`.
- Si `v[i] + v[j]` está en el resultado, no hay que incluir también `v[j] + v[i]` (por la propiedad conmutativa).
- El método debe tener el menor coste temporal posible, debiendo evitar recorrer el array v más de una vez. Considera pues el uso de un Map como estructura de datos auxiliar.

```
public static String metodo3(Integer[] v, Integer e) {
     String res = "";
     // Map: clave = elemento de v, valor = posición de la clave en v
     Map<Integer, Integer> mapAux = new TablaHash<Integer, Integer>(v.length);
     for (int j = 0; j < v.length; j++) {
          Integer i = mapAux.recuperar(e - v[j]);
          if (i != null) { // v[j] forma par con v[i] (v[i] + v[j] = e) → res se actualiza
               res += "v[" + i + "] + v[" + j + "] = " + v[i] + " + " + v[j] + " = " + e + "\n";
          }
          // Como hay que controlar la aparición de repetidos en v, tanto si i != null como si
          // no lo es hay que insertar la Entrada de clave v[j] y valor j en mapAux
          Integer jRepetido = mapAux.insertar(v[j], j);
          if (jRepetido != null) { return "Error: hay elementos repetidos"; }
          // Observa también que insertando v[j] tras haber actualizado res permite también
          // controlar que si v[i] + v[j] está en el resultado, v[j] + v[i] NO lo estará
     }
     return res;
}
```

4. Diseña un método en la clase `MonticuloBinario` que devuelva el número de apariciones de un elemento dado e en un Montículo. Aprovecha la propiedad de orden del Montículo para que el coste Temporal efectivo de tu método solo dependa del número de sus elementos que sean menores o iguales que e. **(2.5 puntos)**

```
public int metodo4(E e) { return metodo4(e, 1); }
private int metodo4(E e, int i) {
    int res = 0; // Caso base implícito Heap vacío
    if (i <= talla) { // Caso general: por definición, solo pueden haber iguales a e en el Heap
                      // si e es mayor o igual que su raíz → Mejor caso: e es menor que su raíz
        int resC =  elArray[i].compareTo(e);
        if (resC <= 0) {
            if (resC == 0) { res++; }
            res += metodo4(e, 2 * i) + metodo4(e, 2 * i + 1);
        }
    }
    return res;
}
```

a) Indica la talla del problema: x = Tamaño del Heap con raíz en `i`, o `this.talla` en la llamada más alta

b) Escribe las Relaciones de Recurrencia para el Mejor y Peor de los casos.

En el caso general, cuando x > 0: $T^M_{metodo4}(x) = k$ y $T^P_{metodo4}(x) = 2 * T^P_{metodo4}(x / 2) + k'$

c) Indica el coste Temporal del método que has diseñado utilizando la notación asintótica (O y Ω o bien Θ).

$T_{metodo4}(x) \in \Omega(1)$ y $T_{metodo4}(x) \in O(x)$

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();
}
```

Las clases `NodoLEG` y `LEGListaConPI` del paquete `lineales`.

```
class NodoLEG<E> {
    E dato;
    NodoLEG<E> siguiente;
    NodoLEG(E e, NodoLEG<E> s) { dato = e; siguiente = s; }
    NodoLEG(E dato) { this(dato, null); }
}

public class LEGListaConPI<E> implements ListaConPI<E> {
    protected NodoLEG<E> pri, ant, ult;
    protected int talla;
    ...
}
```

La clase `MonticuloBinario` del paquete `jerarquicos`.

```
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {
    protected static final int CAPACIDAD_INICIAL = 50;
    protected E[] elArray;
    protected int talla;
    ...
}
```

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 $a>c$, $f(x) \in \Theta(x^{\log_c a})$;