Resolución del Primer Parcial de EDA (2 de Marzo de 2015) - Puntuación 1.2

1.- La clase `LEGLPIDeComparables` hereda de `LEGListaConPI` e implementa una `ListaConPI` de elementos Comparables. Se pide:

(a) Escribir la cabecera de la clase. **(0.2 puntos)**

```
public class LEGLPIDeComparables<E extends Comparable<E>>
extends LEGListaConPI<E> implements ListaConPI<E> { … }
```

(b) Completar su método `insertar`, que aparece a continuación, tal como se indica en su especificación y utilizando única y exclusivamente los métodos de la interfaz `ListaConPI` (ver Anexo). **(0.3 puntos)**

```
/** Inserta e detrás cada uno de los elementos de una ListaConPI que sean menores que
 * él; si en la Lista no existe ningún elemento menor que e, inserta e en su final.
 * Así, por ejemplo: si la Lista está vacía, inserta e; si la Lista no está vacía
 * PERO no contiene elementos menores que e, inserta e en su final; si la Lista
 * contiene 2 elementos menores que e, inserta e tras cada uno de ellos; etc. */
 public void insertar(E e) {
     inicio(); int menoresQueE = 0;
     while (!esFin()) {
          if (recuperar().compareTo(e) < 0) {
             siguiente(); super.insertar(e); menoresQueE++;
          }
          else siguiente();
     }
     if (menoresQueE == 0)  super.insertar(e);
 }
```

2.- Una fábrica produce monedas de un determinado peso; pero, por un defecto de fabricación, entre ellas existe una moneda de peso inferior al de las demás. Suponiendo que un array `v` contiene las referencias de todas las monedas fabricadas (`String`) y que se dispone de un método `balanza(v, i, j, k, l)` tal que, en tiempo constante, dados dos (sub)array de `v` de igual talla `v[i, j]` y `v[k, l]` ...

- devuelve 0 si las monedas de `v[i, j]` pesan lo mismo que las de `v[k, l]` ;
- devuelve un valor negativo si las monedas de `v[i, j]` pesan menos que las de `v[k, l]`;
- devuelve un valor positivo si las monedas de `v[i, j]` pesan más que las de `v[k, l]`;

Se pide escribir un método Divide y Vencerás que, con coste logarítmico en el peor caso, devuelva el índice del array `v` donde se encuentra la referencia de la moneda defectuosa. **(0.4 puntos)**

NOTA: comprueba que tu método funciona bien tanto si el número de monedas de `v` es par como si es impar.

```
public static int farsaMonea(String[] v) { return farsaMonea(v, 0, v.length - 1); }
private static int farsaMonea(String[] v, int i, int j) {
     if (i == j) return i;
     int  m = (i + j) / 2;
     if ((j - i + 1) % 2 == 0) { //si la talla es par
        if (balanza(v, i, m, m + 1, j) < 0) return farsaMonea(v, i, m);
        else return farsaMonea(v, m + 1, j);
     }
     else { // si la talla es impar
         int resC = balanza(v, i, m - 1, m + 1, j);
         if (resC == 0) return m;
         if (resC < 0) return farsaMonea(v, i, m - 1);
         return farsaMonea(v, m + 1, j);
     }
 }
```

3.- Se desea analizar el coste Temporal del siguiente método recursivo: **(0.3 puntos)**

```
private static int metodoR(int[] v, int i, int f) {
   if (i < f) {
     int mitad = (i + f) / 2;
     if (v[mitad] <= 0) {
       if (v[mitad + 1] > 0) return mitad;
       else return metodoR(v, mitad + 1, f);
     }
     else return metodoR(v, i, mitad);
   }
   else return -1;
}
```

Para ello **se pide**:

a) Expresar la talla x del problema en función de los parámetros del método. **(0.05 puntos)**

x = f – i + 1

b) Para una talla x dada, marcar con una cruz la casilla que se considere correcta y rellenar el recuadro en blanco que tenga asociado. **(0.1 puntos)**

☒ Sí existen instancias significativas

Mejor Caso: v[mitad] negativo o cero y v[mitad + 1] positivo Peor Caso: los datos del array v son, o bien todos positivos, o bien todos negativos

☐ No existen instancias significativas

porque

c) Escribir las Relaciones de Recurrencia que requiera la respuesta dada en el apartado **b)**; resolverlas y acotar su solución usando los teoremas de coste (ver Anexo). **(0.1 puntos)**

Mejor Caso: $T_{metodoR}^{M}(x > 1) = k$. Por tanto, $T_{metodoR}^{M}(x) \in \Theta(1)$ Peor Caso: $T_{metodoR}^{P}(x > 1) = 1 * T_{metodoR}^{P}(x / 2) + k'$. Por tanto, por Teorema 3 con a=1, c=2 y sobrecarga constante, $T_{metodoR}^{M}(x) \in \Theta(\log x)$

d) A partir de las cotas obtenidas en el apartado c), escribir el coste Temporal Asintótico del método.

(0.05 puntos)

$T(x) \in \Omega(1)$ y $T(x) \in O(\log x)$

ANEXO

La interfaz `ListaConPI` del paquete `modelos`.

```
public interface ListaConPI<E> {
   void insertar(E e);
   /** SII !esFin() */ void eliminar();
   void inicio();
   /** SII !esFin() */ void siguiente();
   void fin();
   /** SII !esFin() */ E recuperar();
   boolean esFin();
   boolean esVacia();
   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 $a>c$, $f(x) \in \Theta(x^{\log_c a})$;

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 $a>1$;

Primer Parcial de EDA - 18 de Abril de 2016 - Duración: 2h 30' - Puntuación 3

APELLIDOS, NOMBRE	GRUPO

1.- La Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA) ha decidido evaluar a los profesores que quieran optar a una promoción en base al impacto de su labor investigadora. A cada profesor se le asocia entonces un índice de impacto (`int`). La ANECA solo deja optar a la promoción a aquellos profesores cuyo índice de impacto sea mayor que un cierto umbral `u` (`int`).
Se dispone de un array `h` en el que `h[i]` es el índice de impacto del profesor `i`. Este array está ordenado de manera no decreciente por índice de impacto; específicamente, `h[i]` ≤ `h[i+1]` (0 ≤ `i` < `h.length-1`). Se pide:

(a) Diseña un método estático Divide y Vencerás que, dado el array ordenado `h` y el umbral `u`, devuelva el número de profesores cuyo índice de impacto es mayor que el umbral `u`. **(0.75 puntos)**

```
public static int promocion(int h[], int u) {
      return promocion(h, u, 0, h.length - 1);
}

private static int promocion(int h[], int u, int ini, int fin) {
  if (ini > fin) return 0;
  else {
    int mitad = (ini + fin) /2;
    if (h[mitad] <= u) return promocion(h, u, mitad + 1, fin);
    // h[mitad] > u --> todos los profesores con índice > mitad también tienen índice de impacto mayor que u
    return promocion(h, u, ini, mitad - 1) + (fin - mitad + 1);
  }
}
```

(b) Estudia el coste del método diseñado **(0.25 puntos)**

Talla: `n = fin - ini + 1`

Instancias significativas: `no hay`

Ecuaciones de recurrencia:

$T_{promocion}(n = 0) = k1$

$T_{promocion}(n > 0) = 1 * T_{promocion}(n/2) + k2$

Coste temporal asintótico:

$T_{promocion}(n) \in \Theta(\log_2 n)$

2.- Dos jugadores se enfrentan en una partida de un videojuego desde lugares distintos, usando cada uno de ellos una pantalla física distinta. De todas las posiciones de la pantalla que cada jugador visita durante la partida solo algunas le proporcionan un cierto número de puntos, en función del momento en el que las visita. Para guardar dichas posiciones y los puntos en ellas obtenidas, el videojuego dispone de un `Map<Posicion, Integer>`; en el apartado (a) de este ejercicio se darán los detalles de la clase `Posicion`.

Así mismo, para establecer cuál de los dos jugadores ha ganado la partida, el videojuego almacena en la `ListaConPI<Posicion> optima` la secuencia temporal de posiciones a visitar para obtener la máxima puntuación; de esta forma, gana la partida aquel jugador que haya acumulado un mayor número de puntos. Es importante señalar lo siguiente:

- El ganador no obtiene la máxima puntuación si no ha seguido la secuencia temporal óptima de visita.
- Cuando un jugador no ha visitado una posición de la lista `optima`, sus puntos se decrementan en una cantidad fija `penalty`.
- Dos jugadores pueden empatar si ambos obtienen la misma puntuación tras la partida.

(a) Si `Map<Posicion, Integer>` se implementa mediante una Tabla Hash Enlazada, completa el cuerpo de la siguiente clase con los métodos imprescindibles para que pueda ser la clase de la clave de dicho Map. **(0.3 puntos)**

```
public class Posicion {
    private int x, y;
    public Posicion(int vX, int vY) { x = vX; y = vY; }
    public int getX() { return x; }
    public int getY() { return y; }

    /*COMPLETAR*/

    public boolean equals(Object otra) {
        return otra instanceof Posicion
               && this.x == ((Posicion) otra).x
               && this.y == ((Posicion) otra).y;
    }

    public int hashCode() {
        return (x + "," + y).hashCode();
    }


}
```

(b) Completa el cuerpo del siguiente método que, dados los Map con los pares (posición, puntos) obtenidos por los jugadores 1 y 2 durante una partida, `mJ1` y `mJ2`, un `penalty` y la lista `optima`, devuelva el número del jugador (1 o 2) que ha ganado la partida, o 0 si han empatado. **(0.7 puntos)**

```
public int ganador(Map<Posicion, Integer> mJ1, Map<Posicion, Integer> mJ2,
                   int penalty, ListaConPI<Posicion> optima) {

    int puntosJ1 = 0, puntosJ2 = 0;
    for (optima.inicio(); !optima.esFin(); optima.siguiente()) {
        Posicion p = optima.recuperar();
        Integer aux = mJ1.recuperar(p);
        if (aux != null) { puntosJ1 += aux; }
        else { puntosJ1 -= penalty;}
        aux = mJ2.recuperar(p);
        if (aux != null) { puntosJ2 += aux; }
        else { puntosJ2 -= penalty;}
    }
    if (puntosJ1 == puntosJ2) { return 0; }
    else if (puntosJ1 > puntosJ2) { return 1; }
    else { return 2; }

}
```

3.- En un ABB en el que se admiten elementos repetidos, los datos del subárbol izquierdo de un nodo son menores o iguales que el que contiene dicho nodo, mientras que los de su subárbol derecho son siempre mayores. Se pide:

(a) Diseña en la clase `ABB` un método que devuelva el número de veces que aparece un elemento dado `e` en un ABB. **(0.75 puntos)**

```
public int numApariciones(E e) {
  return numApariciones(e, raiz);
}

private int numApariciones(E e, NodoABB<E> actual) {
  int res = 0;
  if (actual != null) {
    int cmp = actual.dato.compareTo(e);
    if (cmp > 0) { res = numApariciones(e, actual.izq); }
    else if (cmp < 0) { res = numApariciones(e, actual.der); }
    else { res = 1 + numApariciones(e, actual.izq); }
  }
  return res;
}
```

(b) Estudia el coste del método diseñado para un ABB Equilibrado. **(0.25 puntos)**

Talla: n = tamaño del nodo actual

Instancias significativas: No hay

Ecuaciones de recurrencia:

$T_{numApariciones}(n = 0) = k1$

$T_{numApariciones}(n > 0) = 1 * T_{numApariciones}(n/2) + k2$

Coste temporal asintótico:

$T_{numApariciones}(n) \in \Theta(\log_2 n)$

ANEXO

Las clases `ListaConPI`, `Map` y `ABB` del paquete `modelos`.

```
public interface ListaConPI<E> {
  void insertar(E e);
  /** SII !esFin() */ void eliminar();
  void inicio();
  /** SII !esFin() */ void siguiente();
  void fin();
  /** SII !esFin() */ E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}
```

```
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();
}
```

```
public class ABB<E extends Comparable<E>> {
       protected NodoABB<E> raiz;
       public ABB() {…}
       public boolean esVacio() {…}
       public int talla() {…}
       public E recuperar(E e) {…}
       public void insertar(E e) {…}
       public void eliminar(E e) {…}
       public E eliminarMin() {…}
       public E recuperarMin() {…}
       public E eliminarMax() {…}
       public E recuperarMax() {…}
       public E sucesor(E e) {…}
       public E predecesor(E e) {…}
       public String toStringInOrden() {…}
       public String toStringPreOrden() {…}
       public String toStringPostOrden() {…}
       public String toStringPorNiveles() {…}
       …
}
```

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 $a>1$;

Resolución del Primer Parcial de EDA (7 de Abril de 2017)

1.- Se quiere ampliar la Jerarquía **`ListaConPI`** para incluir el método **`sucesor`** en su funcionalidad. Para ello:

a) En el paquete **`modelos`**, se define la subinterfaz **`LPIComparable`** reutilizando vía Herencia la interfaz **`ListaConPI`**. Completa su cabecera. **(0. 5 puntos)**

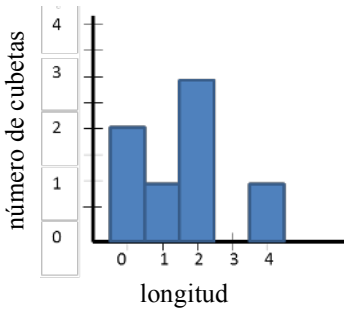
```
public interface LPIComparable <E extends Comparable<E>> extends ListaConPI<E> {
    /** SII !esvacia(): devuelve la posición del sucesor de e o -1 si no está */
    public int sucesor(E e);
}
```

b) En el paquete **`lineales`**, se define la clase **`LEGLPIComparable`** que implementa **`LPIComparable`** reutilizando vía Herencia **`LEGListaConPI`**, la clase que implementa **`ListaConPI`**. Escribe su cabecera. **(0.5 puntos)**

```
public class LEGLPIComparable<E extends Comparable<E>>
    extends LEGListaConPI<E>
    implements LPIComparable<E> { … }
```

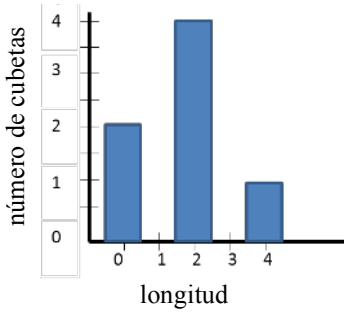
2.- Sea el siguiente el histograma de ocupación de una Tabla Hash:

a) Indica el número de elementos y el de cubetas que tiene la Tabla. **(0.5 puntos)**



11 elementos y 7 cubetas.

b) Indica en qué cubeta de la Tabla se debe insertar un nuevo elemento para que su histograma de ocupación pase a ser el siguiente: **(0.5 puntos)**



Se debe insertar en la (única) cubeta de longitud 1; así, el nº de cubetas de longitud 2 de la Tabla pasará a ser 4 y el de cubetas de longitud 1 pasará a ser cero.

3.- Diseña en la clase **ABB** el método **addMinimo** para añadir un dato **e** menor que todos los del **ABB**. No puedes utilizar el método **insertar** de la clase **ABB**. **(2 puntos)**

```
public void addMinimo(E e) {
    if (raiz == null) { raiz = new NodoABB<E>(e); }
    else { this.raiz = addMinimo(e, raiz); }
}
//SII actual != null: devuelve el Nodo resultado de insertar e en actual
protected NodoABB<E> addMinimo(E e, NodoABB<E> actual) {
    NodoABB<E> res = actual;
    if (actual.izq == null) { res.izq = new NodoABB<E>(e); }
    else {
        res.izq = addMinimo(e, actual.izq);
        res.talla++;
    }
    return res;
}
```

Una solución iterativa equivalente sería como sigue:

```
public void addMinimo(E e) {
    NodoABB<E> nuevo = new NodoABB<E>(e);
    if (raiz == null) { raiz = nuevo; }
    else {
        NodoABB<E> actual = raiz;
        while (actual.izq != null) {
            actual.talla++;
            actual = actual.izq;
        }
        actual.izq = nuevo;
    }
}
```

Suponiendo que el ABB está equilibrado, indica la talla del problema, x, y el coste Temporal del método **addMinimo** que has diseñado, $T_{addMinimo}(x)$, utilizando la notación asintótica (O y Ω o bien Θ). **(0.5 puntos)**

Talla del problema x = número de nodos del ABB, o talla de su nodo Raíz.

Coste Temporal Asintótico: el método realiza un Recorrido del camino que une la Raíz del ABB con su nodo más a la izquierda; como el ABB está equilibrado, este camino tiene una longitud del orden de log x y, por tanto, $T_{addMinimo}(x) \in \Theta(\log x)$

4.- La Dirección General de Tráfico tiene en un **Map dgt** la información de cada coche, concretamente su matrícula (clave) y el año de matriculación (valor). Se pide diseñar un método estático **cochesPorAnyo** que dado el **Map dgt** devuelva un **Map** que tenga, para cada año, el número de coches matriculados. **(2 puntos)**

```
public static Map<Integer, Integer> cochesPorAnyo(Map <String, Integer> dgt) {
    Map<Integer, Integer> res = new TablaHash<Integer, Integer>(dgt.talla());
    ListaConPI <String> matriculas = dgt.claves();
    for (matriculas.inicio(); !matriculas.esFin(); matriculas.siguiente()) {
        String matricula = matriculas.recuperar();
        Integer anyo = dgt.recuperar(matricula);
        Integer cont = res.recuperar(anyo);
        if (cont == null) { res.insertar(anyo, 1); }
        else { res.insertar(anyo, ++cont); }
    }
    return res;
}
```

Indica la talla del problema, x, y el coste Temporal del método **cochesPorAnyo** que has diseñado, $T_{cochesPorAnyo}(x)$, utilizando la notación asintótica (O y Ω o bien Θ). **(0.5 puntos)**

Talla del problema x = dgt.talla().

Coste temporal asintótico: $T_{cochesPorAnyo}(x) \in \Theta(x)$.

5.- Sea **v** un array de enteros, de longitud mayor que 0, que contiene una secuencia de valores estrictamente creciente hasta un cierto índice **p** y estrictamente decreciente desde **p** hasta el final. Diseña un método Divide y Vencerás para encontrar este índice **p** con el mejor coste posible. **(2 puntos)**

Por ejemplo, el resultado **p** sería 5 para el siguiente array:

30	40	80	100	110	160	50	10	4	2
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

```
public static int posicion(int[] v) {
    return posicion(v, 0, v.length - 1);
}
private static int posicion(int[] v, int i, int f) {
    // Búsqueda con garantía de éxito
    if (i == f) { return i; } // subarray con 1 elemento
    else if (i + 1 == f) { // subarray con 2 elementos
        if (v[i] > v[f]) { return i; }
        else { return f; }
    }
    else {
        // subarray con, mínimo, 3 elementos; de ellos, o el central cumple la
        // condición, o forma parte de la secuencia creciente o de la decreciente
        int m = (i + f) / 2;
        if (v[m - 1] < v[m]) {
            if (v[m] > v[m + 1]) { return m; } // v[m] cumple la condición
            else { return posicion(v, m + 1, f); } // v[m] en secuencia creciente
        }
        else { return posicion(v, i, m - 1); } // v[m] en secuencia decreciente
    }
}
```

Estudia el coste Temporal del método (recursivo) que has diseñado. En concreto, ... **(1 punto)**

Indica la talla del problema, x, en función de los parámetros del método: x = `f - i + 1`.

Para una talla x dada, indica si existen instancias significativas; si las hubiera, indica cuáles son y por qué:

Mejor caso: `p = m`, la posición central del subarray `v[i, f]`.

Peor caso: `p = i` o `p = f`, respectivamente la primera posición o la última del subarray `v[i, f]`.

Escribe las Relaciones de Recurrencia <u>que requiera tu respuesta en el punto anterior</u>; luego, usa los Teoremas de Coste para resolverlas y acotarlas:

$T^{P}_{posicion}(x > 2) = 1 * T^{P}_{posicion}(x / 2) + k.$

Luego, por Teorema 3 (sobrecarga constante), con a = 1 y c = 2, $T^{P}_{posicion}(x > 2) \in \Theta(\log_2 x)$.

<u>A partir de tu respuesta en el punto anterior</u>, escribe el coste Temporal Asintótico del método, utilizando la notación asintótica (O y Ω o bien Θ):

$T_{posicion}(x) \in \Omega(1)$ y $T_{posicion}(x) \in O(\log_2 x)$.

ANEXO

La interfaz `ListaConPI` del paquete `modelos`.

```
public interface ListaConPI<E> {
      void insertar(E e);
      /** SII !esFin() */ void eliminar();
      void inicio();
      /** SII !esFin() */ void siguiente();
      void fin();
      /** SII !esFin() */ E recuperar();
      boolean esFin();
      boolean esVacia();
      int talla();
}
```

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 `ABB` y `NodoABB` del paquete `jerarquicos`.

```
public class ABB<E extends Comparable<E>> {
    protected NodoABB<E> raiz;
    public ABB() { this.raiz = null; }
    …
}

class NodoABB<E> {
    protected E dato;
    protected NodoABB<E> izq, der;
    int talla;
    NodoABB(E e) {
        this.dato = e;
        this.izq = null; this.der = null;
        talla = 1;
    }
}
```

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 $a>1$;

Resolución de la Recuperación del Primer Parcial de EDA (20 de Junio de 2016) - Puntuación: 3

1.- Una fábrica dispone de una máquina que prepara bombones y los almacena en cajas con un número par de unidades. Aunque la máquina fabrica a veces un bombón defectuoso, que tiene un peso mayor que el resto, se puede asegurar que no puede haber más de uno de estos bombones en cada caja. A nivel de programación, una caja `c` se puede representar como un array de objetos de tipo `Bombon` (`Bombon[] c`); además, es posible conocer el peso de un subconjunto de bombones de una caja (subarray `c[i, j]`) en tiempo constante mediante el siguiente método:

```
// SII i >= 0 && i <= j && j < caja.length: devuelve el peso de los bombones situados entre
// las posiciones i y j, ambas inclusive, de una caja c
public static float peso(Bombon[] c, int i, int j) { … }
```

Diseña un método estático Divide y Vencerás que devuelva la posición que ocupa el bombón defectuoso en una caja `c`, o -1 si la caja no contiene tal bombón. El método diseñado debe tener un coste temporal O(log `c.length`). **(1 punto)**

```
public static int bombonDefectuoso(Bombon[] c) {

    // Al ser par el nº de bombones por caja...
    // si las 2 mitades de la caja c pesan lo mismo, no hay bombón defectuoso;
    // sino, buscar el bombón defectuoso **con garantía de éxito**
    int m = c.length / 2 - 1;
    if (peso(c, 0, m) == peso(c, m+1, c.length - 1)) { return -1; }
    else { return bombonDefectuoso(c, 0, c.length - 1); }
}

// Búsqueda con garantía de éxito: seguro que hay un bombón defectuoso en c[i, j]
private static int bombonDefectuoso(Bombon[] c, int i, int j) {

    // Caso base: si c tiene un único bombón, seguro que es defectuoso
    if (i == j) { return i; }

    // Caso general: si c tiene 2 o más bombones, usar DyV para buscar el defectuoso

    // (a) DIVIDIR la caja c[i, j] en 2 mitades
    int m = (i + j) / 2, talla = j - i + 1;
    float pesoIzq, pesoDer;

    // (b) VENCER, en base al peso que tenga cada mitad de c[i, j]
    //     OJO: según la paridad de la talla de c[i, j], c[m] está
    //          en una de las mitades a pesar o las separa
    if (talla % 2 == 0) {
    // talla de c[i, j] es par → c[m] es el último bombón de la 1era mitad a pesar
        pesoIzq = peso(c, i, m);
        pesoDer = peso(c, m + 1, j);
        if (pesoIzq > pesoDer) { return bombonDefectuoso(c, i, m); }
        else { return bombonDefectuoso(c, m + 1, j); }
    }
    else {
    // talla de c[i, j] es impar → c[m] es el bombón que separa las mitades a pesar
        pesoIzq = peso(c, i, m - 1);
        pesoDer = peso(c, m + 1, j);
        if (pesoIzq == pesoDer) { return m; }
        else if (pesoIzq > pesoDer) { return bombonDefectuoso(c, i, m - 1); }
        else { return bombonDefectuoso(c, m + 1, j); }
    }
}
```

2.- Diseña un método estático que, dado un array genérico `E[] v`, devuelva una *Lista con Punto de Interés* con aquellos elementos de `v` cuya frecuencia de aparición sea menor que un umbral dado `u` (`int`). Para lograr un coste temporal O(`v.length`) tu método debe usar un *Map* implementado mediante una *Tabla Hash*. **(0.75 puntos)**

```
public static ListaConPI<E> menorQue(E[] v, int u) {
    Map<E, Integer> map = new TablaHash<E, Integer>(v.length);
    ListaConPI<E> res = new LEGListaConPI<E>();
    for (int i = 0; i < v.length; i++) {
        Integer frec = map.recuperar(v[i]);
        if (frec == null) frec = 0;
        map.insertar(v[i], frec + 1);
    }
    ListaConPI<E> l = map.claves();
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        Integer frec = map.recuperar(l.recuperar());
        if (frec < u) res.insertar(l.recuperar());
    }
    return res;
}
```

3.- Sea un *ABB* sin elementos repetidos.

a) Implementa en la clase `ABB` un método recursivo que, con el mejor coste Temporal posible, devuelva una *Lista con Punto de Interés* con los elementos del *ABB* que hay en el camino desde su raíz hasta un cierto elemento `x`, o `null` si dicho elemento no está en el *ABB*. **(0.75 puntos)**

```
public ListaConPI<E> camino(E x) {
    ListaConPI<E> res = new LEGListaConPI<E>();
    return camino(raiz, x, res);
}

protected ListaConPI<E> camino(NodoABB<E> actual, E x, ListaConPI<E> res) {
    if (actual == null) { return null; }
    res.insertar(actual.dato);
    int cmp = actual.dato.compareTo(x);
    if (cmp > 0) { return camino(actual.izq, x, res); }
    else if (cmp < 0) { return camino(actual.der, x, res); }
    return res;
}
```

(b) Estudia el coste del método diseñado para un *ABB* Equilibrado. **(0.5 puntos)**

Talla: **n =** tamaño del nodo `actual`, el del ABB en la llamada más alta.

Instancias significativas:

Mejor caso: en la llamada más alta, el dato del nodo `actual` es igual a `x`.

Peor caso: `x` no está en el *ABB*.

Ecuaciones de Recurrencia:

$T_{camino}^{P}(n = 0) = k2$; $T_{camino}^{P}(n > 0) = 1 * T_{camino}(n/2) + k3$

Coste Temporal Asintótico:

$T_{camino}(n) \in \Omega(1)$ y $T_{amino}(n) \in O(\log_2 n)$

ANEXO

```
public interface ListaConPI<E> {
    void insertar(E e);
    /** SII !esFin() */ void eliminar();
    void inicio();
    /** SII !esFin() */ void siguiente();
    void fin();
    /** SII !esFin() */ E recuperar();
    boolean esFin();
    boolean esVacia();
    int talla();
}
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();
}
public class TablaHash<C,V> implements Map<C,V> {
    …
    public TablaHash(int inicial) {…}
    …
}
```

```
public class ABB<E extends Comparable<E>> {
    protected NodoABB<E> raiz;
    public ABB() {…}
    public boolean esVacio() {…}
    public int talla() {…}
    public E recuperar(E e) {…}
    public void insertar(E e) {…}
    public void eliminar(E e) {…}
    …
}
class NodoABB<E> {
    E dato;
    int talla;
    NodoABB<E> izq, der;
    NodoABB(E e) {…}
    NodoABB(E e, NodoABB<E> i, NodoABB d) {…}
}
```

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 $a>1$;

Resolución de la Recuperación del Primer Parcial de EDA (6/6/2012)

1.-La clase *LEGPila* es una implementación de la interfaz *Pila* mediante una lista enlazada. Se ha diseñado una nueva clase, *LEGPilaDeEnteros*, a la que se quiere añadir un nuevo método, *moverPrimeroAlFinal*, que sitúa al final de una pila de enteros su primer nodo siempre que el dato que éste contenga sea menor que el del último nodo. El método devuelve *true* si se ha movido el primer nodo y *false* en caso contrario o si la pila está vacía.

Se pide completar el código del método *moverPrimeroAlFinal*. **(3.5 puntos)**

```
public class LEGPilaDeEnteros extends LEGPila<Integer> {
  public boolean moverPrimeroAlFinal(){
     boolean mover = false; NodoLEG<Integer> aux = tope;
     if ( aux!=null ){
        while ( aux.siguiente!=null ) aux = aux.siguiente;
        if ( tope.dato.compareTo(aux.dato)<0 ){
            mover = true;
            aux.siguiente = tope; tope = tope.siguiente;
            aux.siguiente.siguiente = null;
        }
     }
     return mover;
  }
}
```

```
public classLEGPila<E>
implements Pila<E>{
  protected NodoLEG<E> tope;
  …
}
classNodoLEG<E>{
  E dato;
  NodoLEG<E> siguiente;
  …
}
```

2.-Sea *v* un array de *int* ordenado ascendentemente y sin elementos repetidos; se quiere comprobar si el par de *int* *x* e *y*, tal que *x<y*, ocupa posiciones consecutivas dentro del array *v*. Para ello **se pide** completar el código del método recursivo *hayPar* usando de la estrategia Divide y Vencerás para obtener una implementación lo más eficiente posible. **(3.5 puntos)**

```
public static boolean hayPar(int[] v,int x,int y){
       return buscarPar(v,x,y,0,v.length-1);
}
private static boolean hayPar(int[] v, int x, int y, int izq, int der){
     if ( izq>=der ) return false;
     else{  int mitad= (izq+der)/2;
            if ( v[mitad]==x )    return ( v[mitad+1]==y );
            else if ( v[mitad]<x )
                                  return hayPar(v, x, y, mitad+1, der);
                 else             return hayPar(v, x, y, izq, mitad);
     }
}
```

3.- Se desea analizar el coste Temporal del siguiente método recursivo: **(3 puntos)**

```
private static boolean comparar(int[] a, int[] b, int inicio, int fin){
   if ( inicio>fin ) return true;
   int mitad = (fin+inicio)/2;
   boolean res = ( a[mitad]==b[mitad] );
   if (res){
      res = comparar(a, b, inicio, mitad-1);
      if (res) res = comparar(a, b, mitad+1, fin);
   }
   return res;
}
```

Para ello **se pide**:

a) Expresar la talla del problema en función de los parámetros del método: **(0.5 puntos)**

x = fin-inicio+1

b) Indicar si existen instancias significativas para una talla dada y por qué; en caso y por qué; en caso afirmativo describir cada una de ellas**:** **(0.5 puntos)**

Sí hay instancias significativas, pues 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* a comparar **no** son iguales ya en la llamada más alta (con *inicio=0* y *fin=a.length-1=b.length-1*). Así, en tiempo constante *res* se evalúa a *false*.

Peor de los Casos: los arrays *a* y *b* a comparar **sí** son iguales, i.e. contienen ya en la llamada más alta 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 las relaciones de recurrencia que definan el coste del método: **(1 punto)**

Mejor de los Casos: no hay Relación de Recurrencia asociada al no producirse llamada alguna en el cuerpo de *comparar*. $T_{comparar}^{M}(x>0) = k1$

Peor de los Casos: $T_{comparar}^{P}(x=0) = k2$; $T_{comparar}^{P}(x>0) = 2 * T_{comparar}^{P}(x/2) + k3$

d) Resuelve las relaciones de recurrencia del apartado anterior y escribe el coste temporal asintótico del método: **(1 punto)**

Peor de los Casos: por T3 con a=c=2 y sobrecarga constante, $T_{comparar}^{P}(x) \in \Theta(x)$

Mejor de los Casos: $T_{comparar}^{M}(x) \in \Theta(1)$

Por tanto, $T_{comparar}(x) \in O(x)$ y $T_{comparar}(x) \in \Omega(1)$

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

1.- Las siguientes clases implementan un ABB en Java:

```
public class ABB <E extends Comparable <E>> {
   protected NodoABB<E> raiz;
   protected int talla;
   public ABB(){ raiz = null; talla = 0; }
   //Resto de métodos de la clase
   …
}
class NodoABB<E> {
   E dato;
   NodoABB<E> izq, der;
   public NodoABB(E e){ dato = e; izq = null; der = null; }
   public NodoABB(E e, NodoABB<E> izq, NodoABB<E> der){
       dato = e; this.izq = izq; this.der = der;
   }
}
```

Se pide: **(4 puntos)**

(a) Diseñar un método en la clase *ABB* que devuelva el tamaño de aquel de los nodos de un ABB que contenga a *e* como dato, *0* si *e* no está en el ABB; suponer que no hay elementos repetidos en el ABB. **(3 puntos)**

Nota: el tamaño de un nodo se define como el número de descendientes que tiene, él mismo incluido, y el de un árbol como el de su nodo raíz.

```
public int tamañoNodo(E e){
    return tamanyo(recuperar(e, this.raiz));
}
protected NodoABB<E> recuperar(E e, NodoABB<E> actual){
    NodoABB<E> res = actual;
    if ( actual!=null ){
        int resC = actual.dato.compareTo(e);
        if       ( resC<0 )  res = recuperar(e, actual.der);
        else  if ( resC>0 )  res = recuperar(e, actual.izq);
    }
    return res;
}
protected int tamanyo(NodoABB<E> actual){
    if ( actual==null ) return 0;
    else return 1 + tamanyo(actual.izq) + tamanyo(actual.der);
}
```

(b) Indicar el coste Temporal del método diseñado, justificándolo adecuadamente. **(1 punto)**

Suponiendo que x es el tamaño del ABB, x=*talla*:

En el Mejor de los Casos *e* no está en el ABB y, además, es mayor (o menor) que todos los elementos del ABB Completamente Degenerado por la Izquierda (o Derecha) sobre el que se aplica el método *tamañoNodo*, pues entonces su coste se reduce al de *recuperar* un elemento de un nodo de *tamanyo* igual a *cero* (el hijo Izquierdo o Derecho del nodo raíz del ABB). Por tanto, $\mathbf{T^M_{tamaño}(x) \in \Theta(1)}$ **y** $\mathbf{T_{tamaño}(x) \in \Omega(1)}$.

En el Peor de los Casos *e* es el dato que ocupa la raíz del ABB sobre el que se aplica el método *tamañoNodo*, pues entonces su coste es el de *recuperar* un elemento del nodo raíz de un ABB de *tamanyo* igual a *talla*, independiente de su grado de Equilibrio. Por tanto, $\mathbf{T^P_{tamaño}(x) \in \Theta(x)}$ **y** $\mathbf{T_{tamaño}(x) \in O(x)}$.

2.- Supóngase que se ha modificado la clase genérica *TablaHash* para permitir la inserción de entradas con la misma clave. **Se pide** diseñar en esa clase un método *recuperarIguales* que, con coste mínimo, obtenga una Lista con Punto de Interés con los valores de todas las entradas de una Tabla Hash cuya clave sea *c*. **(3 puntos)**

```
public ListaConPI<V> recuperarIguales(C c){
    ListaConPI<V> res = new LEGListaConPI<V>();
    // todas las Entradas de la Tabla con clave c sólo pueden
    // estar en una cubeta: elArray[posTabla(c)]
    ListaConPI<EntradaHash<C,V>> lpi = elArray[posTabla(c)];
    for ( lpi.inicio(); !lpi.esFin(); lpi.siguiente() ){
          EntradaHash<C,V> e =  lpi.recuperar();
          if ( e.clave.equals(c) ) res.insertar(e.valor);
    }
    return res;
}
```

```
public interface
  ListaConPI<E> {
  void insertar(E e);
  void eliminar();
  E recuperar();
  void inicio();
  void siguiente();
  boolean esFin();
  boolean esVacia();
  void fin();
  int talla();
}
```

3.-Supóngase que los *talla*>0 elementos que contiene el atributo *elArray* de la clase *MonticuloBinari*o se han introducido usando el siguiente método, y no usando el método *insertar*:

```
public void introducir(E e){
   if ( talla==elArray.length-1 ) duplicarArray();
   elArray[++talla] = e;
}
```

Nótese entonces que, al menos, hay que comprobar si los elementos de *elArray* cumplen la Propiedad de Ordenación de un Min-Heap; si no lo hacen, como ya es sabido, será necesario invocar la ejecución del método *arreglar*. Para ello, s**e pide**: **(3 puntos)**

(a) Diseñar en la clase *MonticuloBinario* un método eficiente *incumplePO* tal que devuelva la posición del primer elemento de *elArray*, entre *1* y *talla*, que **in**cumpla la propiedad de ordenación del Min-Heap, *elArray.length* si no existe tal elemento. **(2 puntos)**

```
public int incumplePO(){
    for ( int i=2; i<=talla; i++ )
      if ( elArray[i].compareTo(elArray[i/2])<0 ) return i;
    return elArray.length;
}
```

(b) Indicar el coste Temporal del método diseñado, justificándolo adecuadamente. **(1 punto)**

Suponiendo que x es el tamaño del Min-Heap, x=*talla*:

En el Mejor de los Casos el Hijo Izquierdo de la raíz del Min-Heap, que ocupa la posición *2* de *elArray*, ya incumple la propiedad de ordenación (elArray[2]<elArray[1]). Así, $\mathbf{T^{M}_{incumplePO}(x) \in \Theta(1)}$ **y** $\mathbf{T_{incumplePO}(x) \in \Omega(1)}$.

En el Peor de los Casos, tras *talla* comparaciones, ninguno de los elementos introducidos en *elArray* incumple la propiedad de ordenación del Min-Heap. Así, $\mathbf{T^{P}_{incumplePO}(x) \in \Theta(x)}$ **y** $\mathbf{T_{incumplePO}(x) \in O(x)}$.

Resolución del Primer parcial de EDA (12/3/2012)

1.- La siguiente interfaz extiende la funcionalidad de una Lista con Punto de Interés para añadir un nuevo método, *borrarTodos*:

```
public interface ListaConPIPlus<E> extends ListaConPI<E> {
intborrarTodos(Ex);
}
```

Este método elimina todos los elementos de la lista que sean iguales a *x*, y devuelve el número de elementos que se han borrado.**Se pide** completar el método *borrarTodos* en la siguiente clase, haciendo uso únicamente de los métodos de la interfaz *ListaConPI*: **(2puntos)**

```
public classImplementacionListaConPIPlus<E>
extends ImplementacionListaConPI<E> implements ListaConPIPlus<E> {
   public intborrarTodos(Ex) {
      int n = 0;
      inicio();
      while ( !esFin() )
            if ( recuperar().equals(x) ){
                  eliminar();
                  n++;
            }
            else siguiente();
      return n;
   }
}
```

```
public interface
ListaConPI<E> {
voidinsertar(E e);
voideliminar();
 E recuperar();
voidinicio();
voidsiguiente();
booleanesFin();
booleanesVacia();
voidfin();
inttalla();
}
```

2.-La clase *LEGCola* es una implementación de la interfaz *Cola* mediante una lista enlazada. Se ha diseñado una nueva clase, *LEGColaDeEnteros*, a la que se quiere añadir un nuevo método, *borrarPrimero*, que elimine el primer elemento de la cola sí y sólo sí el valor del primer elemento es mayor que la suma de todos los demás. El método devuelve *true* si se ha realizado el borrado y *false* en caso contrario.

Se pide completar el método *borrarPrimero* asumiendo que hay, al menos, dos elementos en la cola: **(2 puntos)**

```
public class LEGColaDeEnteros extends LEGCola<Integer> {
  public booleanborrarPrimero(){
      int suma = 0;
      NodoLEG<Integer> aux = primero.siguiente;
      while ( aux!=null ){
            suma += aux.dato;
            aux = aux.siguiente;
      }
      if ( primero.dato.compareTo(suma)>0 ){
            primero = primero.siguiente;
            return true;
      }
      else return false;
  }
}
```

```
publicclassLEGCola<E>
implements Cola<E>{
protectedNodoLEG<E>
primero, fin;
 …
}
classNodoLEG<E>{
 E dato;
NodoLEG<E>siguiente;
 …
}
```

3.- Se desea analizar el coste temporal del siguiente método recursivo: **(3 puntos)**

```
public int metodo(int v[], int i, intj){
      if ( i>j )return 0;
      else{
            int medio = ( i + j )/2;
            int n1 = metodo(v, i, medio - 1);
            int n2 = metodo(v, medio + 1, j);
            return n1 + v[medio] + n2;
      }
}
```

Para ello **se pide**:

a) Expresar la talla del problema en función de los parámetros del método: **(0.5 puntos)**

x = j–i+1

b) Indica de forma justificada si existen instancias significativas para una talla dada**:** **(0.5 puntos)**

No hay instancias significativas pues, en cualquier caso, se debe recorrer el (sub) array v[i…j]

c) Escribir las Relaciones de Recurrencia que definan el coste del método: **(1punto)**

$$T_{metodo}(x) = \begin{cases} k' & si\ x = 0 \\ T_{metodo}\left(\frac{x}{2}\right) + k & si\ x > 0 \end{cases}$$

d) Resuelve las relaciones de recurrencia del apartado anterior y escribe el coste temporal asintótico del método: **(1 punto)**

Por Teorema 3 con a =c=2, $T_{metodo}(x) \in \Theta(x)$

4.- Dado un array *v* de componentes de tipo *int*, ordenado de forma creciente y sin elementos repetidos, se quiere determinar si existe alguna componente de *v* que represente el mismo valor que el de su posición en *v* (y obtener dicha posición). En el caso de que no haya ninguna, se devolverá -1. **(3 puntos)**

Ejemplo:

0	1	2	3	4	5	6
-5	-4	-2	1	4	7	8

Para ello **se pide**: completar el método recursivo *igualAPosicion* haciendo uso de la estrategia Divide y Vencerás para obtener una implementación lo más eficiente posible.

```
public static int igualAPosicion(int v[]){
      return igualAPosicion(v, 0, v.length-1);
}

private static int igualAPosicion(int v[], int izq, int der){
      if ( izq <= der){
          int mitad = (izq + der)/2;
          if ( v[mitad]==mitad ) return mitad;
          if ( v[mitad]< mitad ) return igualAPosicion(v, mitad+1, der);
          else return igualAPosicion(v, izq, mitad-1);
      }
      else return -1;
}
```

Resolución del Segundo Parcial de EDA (7/7/2012)

1.- En la clase `ABB`, diseña un método `numNodos2Hijos` que devuelva el número de nodos con dos hijos que hay en un Árbol Binario de Búsqueda. **(2 puntos)**

```
public int numNodos2Hijos(){ return numNodos2Hijos(this.raiz); }
protected int numNodos2Hijos(NodoABB<E> actual){
   int res = 0;
   if ( actual!=null )
     if ( actual.izq!=null && actual.der!=null )
          res += 1 + numNodos2Hijos(actual.izq)+numNodos2Hijos(actual.der);
     else res +=     numNodos2Hijos(actual.izq)+numNodos2Hijos(actual.der);
   return res;
}
```

```
public class ABB <E extends Comparable <E>> {
   protected NodoABB<E> raiz;
   protected int talla;
   public ABB(){ raiz = null; talla = 0;}
   //Resto de métodos de la clase
   …
}
class NodoABB<E> {
   E dato;
   NodoABB<E> izq, der;
   public NodoABB(E e){ dato = e; izq = null; der = null; }
   public NodoABB(E e, NodoABB<E> izq, NodoABB<E> der){
         dato = e; this.izq = izq; this.der = der;
    }
}
```

2.- Para representar un ABB de palabras se ha definido la clase `ABBDeString`, una derivada de `ABB` cuyos elementos son de tipo `String`:

```
public class ABBDeString extends ABB<String> { … }
```

Diseña en esta clase un método que, con el menor coste temporal posible, devuelva el número de palabras de un ABB que empiecen por una letra dada (representada mediante un `String` de longitud 1): `public int numPalabrasCon(String letraPrefijo)` **(2 puntos)**

Nota: es recomendable que uses el método `boolean startsWith(String prefijo)` de la clase `String` para comprobar si una palabra del ABB empieza por `letraPrefijo`.

```
public int numPalabrasCon(String letraPrefijo){
   return numPalabrasCon(letraPrefijo, this.raiz);
}
protected int numPalabrasCon(String lP, NodoABB<String> n){
   if ( n==null ) return 0;
   else{
      if ( n.dato.startsWith(lP) )
       return 1 + numPalabrasCon(l, n.izq) + numPalabrasCon(lP, n.der);
      else if ( n.dato.compareTo(lP)>0 ) return numPalabrasCon(lP, n.izq);
           else                          return numPalabrasCon(lP, n.der);
}
```

3.- Diseña un método recursivo que devuelva el número de elementos menores que uno dado `e` contenidos en un Montículo Binario Minimal (*Min-Heap*). Este método debe aprovechar la propiedad de ordenación del *Min-Heap* para que su coste temporal sea el menor posible. Indica también su coste asintótico en el mejor y el peor caso. **(3 puntos)**

```
public int numMenoresQue(E e){
  return numMenoresQue(e, 1);
}
protected int numMenoresQue(E e, int actual){
  if ( actual>talla ) return 0;
  else{  int resC = elArray[actual].compareTo(e);
         if ( resC<0 )
     return 1+numMenoresQue(e, 2*actual)+numMenoresQue(e, 2*actual+1);
         else return 0;
   }
}
```

Análisis del coste Temporal Asintótico: sea x=`talla`, el tamaño del problema.

En el Mejor Caso, cuando el dato que ocupa la Raíz del *Min-Heap* ya es mayor o igual que `e`, se tiene que $T^{M}_{numMenoresQue}(x) \in \Theta(1)$. Por tanto, $\mathbf{T_{numMenoresQue}(x) \in \Omega(1)}$.

En el Peor Caso, cuando los x=`talla` datos del *Min-Heap* son menores que `e`, se tiene que $T^{P}_{numMenoresQue}(x) \in \Theta(x)$. Por tanto, $\mathbf{T_{numMenoresQue}(x) \in O(x)}$.

4.- Recordando que en una Tabla Hash Enlazada si dos Entradas están en la misma cubeta es porque han colisionado, diseña un método consultor en la clase `TablaHash` que devuelva el número total de colisiones que se han producido tras insertar un cierto número de Entradas distintas, con claves distintas. **(1 punto)**

```
public int numColisiones(){
   int nTotalColisiones = 0;
   for ( int i=0; i<elArray.length; i++ ){
       int nColCubetaI = 0;
       for ( EntradaHash<C, V> e = elArray[i]; e!=null; e = e.siguiente )
                 nColCubetaI++;
      if ( nColCubetaI>0 ) nTotalColisiones += nColCubetaI-1;
   }
   return nTotalColisiones;
}
```

5.- En un sistema de información se dispone de dos `Map<Clave, Valor>` `m1` y `m2` implementados con sendas Tablas Hash. Como resultado de la actualización de dicho sistema se desea obtener un tercer `Map<Clave, Valor>` que sea la diferencia de `m1` y `m2`, es decir que contenga solo aquellas Entradas de `m1` que no estén también en `m2`. Por ejemplo, si las Entradas de `m1` son `{("uno", 1) ("dos", 2) ("tres", 3) ("cuatro", 4) ("seis", 6)}` y las de `m2` son `{("tres", 3) ("cuatro", 4) ("siete", 7) ("ocho", 8)}`, las Entradas del `Map` diferencia serán `{("uno", 1), ("dos", 2),("seis", 6)}`.

Diseña un método (estático) `diferencia` que devuelva el `Map` diferencia de dos `Map` `m1` y `m2` dados; ten en cuenta que para ello solo podrás utilizar los métodos definidos en la interfaz `Map` y el método constructor de `TablaHash`, que figuran al final de esta hoja. **(2 puntos)**

Nota: para simplificar, asume que si una Entrada de `m1` tiene la misma clave que otra de `m2` entonces los valores de ambas Entradas serán también iguales.

```
public static Map<Clave,Valor> diferencia(Map<Clave,Valor> m1, Map<Clave,Valor> m2){
   Map<Clave,Valor> res = new TablaHash<Clave,Valor>(m1.talla());
   Object[] clavesM1 = m1.claves();
   for ( int i=0; i<clavesM1.length; i++ ){
     Clave c = (Clave)clavesM1[i];
     Valor v = m2.recuperar(c);
     if ( v==null ) res.insertar(c, m1.recuperar(c));
   }
   return res;
}
```

```
public interface Map<C, V> {

  /** inserta la Entrada (c,v) en un Map y devuelve null; si ya
   *  existe una Entrada de Clave c en el Map, devuelve su valor
   *  asociado y lo substituye por v (o actualiza la Entrada)*/
  V insertar(C c, V v);

  /** elimina la Entrada con Clave c de un Map y devuelve su
   *  valor asociado, null si no existe una Entrada con dicha clave*/
  V eliminar(C c);

  /** devuelve el valor asociado a la clave c en un Map,
   *  null si no existe una Entrada con dicha clave*/
  V recuperar(C c);

  /** comprueba si un Map está vacío */
  boolean esVacio();

  /** devuelve la talla, o número de Entradas, de un Map */
  int talla();

  /** devuelve un array que contiene las talla() Claves de un Map */
  C[] claves();
}
```

Método constructor de la clase `TablaHash<C, V>`:

```
public TablaHash(int tallaMaximaEstimada){…}
```

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

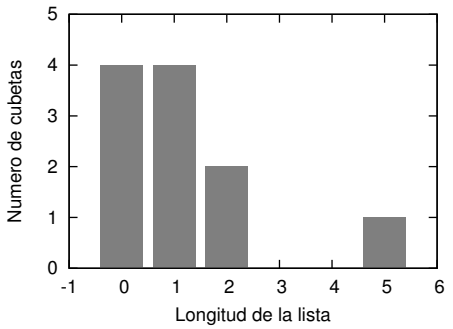
Resolución de la recuperación del segundo parcial de EDA
Escola Tècnica Superior d'Enginyeria Informàtica
18 de junio de 2014 – Duración 2 horas

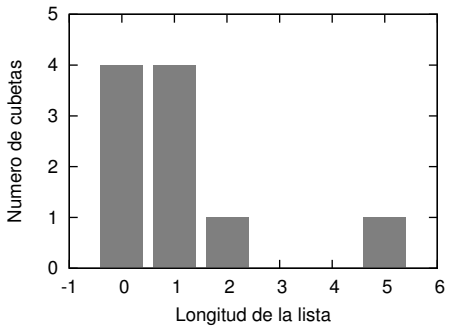
Pregunta 1 (2.5 puntos) Se dispone de una tabla de dispersión con resolución de colisiones por encadenamiento separado y $N = 10$ cubetas. La tabla guarda números enteros. La función de dispersión es $f(x) = x \mod N$. Se pide:

a) **(1.0 puntos)** Dibuja el histograma de ocupación de la tabla tras realizar las siguientes inserciones sobre una tabla de dispersión inicialmente vacía: 31, 34, 33, 53, 14, 13, 20, 56, 71, 15, 23, 3, 62.

b) **(0.75 puntos)** ¿Es posible realizar inserciones en la tabla del apartado a) para que su histograma de ocupación sea el del histograma que aparece a la izquierda? En caso afirmativo, da valores concretos de elementos tales que, al insertarlos, se obtiene una tabla con este histograma. En caso negativo, razona la respuesta.

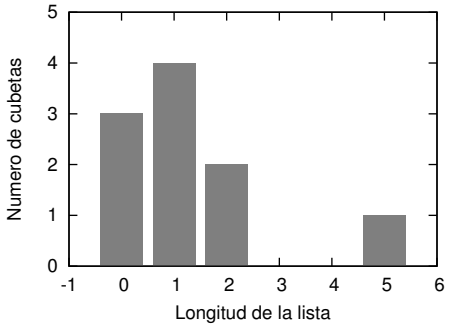
c) **(0.75 puntos)** ¿Es posible realizar borrados en la tabla del apartado a) para que su histograma de ocupación sea el del histograma que aparece a la derecha? En caso afirmativo, da valores concretos de elementos tales que, al borrarlos, se obtiene una tabla con este histograma. En caso negativo, razona la respuesta.





Solución

a)



b) No es posible puesto que al añadir elementos el número de cubetas vacías no se puede incrementar.

c) Si es posible, eliminando cualquiera de estos valores: 20, 62, 15 y 56; y cualquiera de estos: 31, 71, 34 y 14 o también eliminando el 31 y el 71 o el 34 y el 14..

Pregunta 2 (2.5 puntos) Un sistema guarda para cada usuario el momento de su última entrada expresada en milisegundos desde el dia 1 de enero de 1070 (lo que devuelve System.currentTimeMillis()). Esta información la tenemos guardada en una tabla hash *TablaHash <String,Long>tabla* (atributo de la clase *Aplicacion*). Queremos borrar aquellos usuarios que no hayan entrado en el sistema desde una fecha determinada. Para ello queremos añadir una nueva funcionalidad a la clase *Aplicacion* con el siguiente perfil *public void purgar(Long fecha)* que reciba una fecha expresada en milisegundos y elimine de la tabla todas aquellas entradas anteriores.

Solución

```
1  public class Aplicacion {
2      //atributos
3      TablaHash<String,Long> tabla;
4
5      //metodos
6      ...
7      ...
8      public void purgar(Long fecha){
9          ListaConPI<String> claves = tabla.claves();
10         for (claves.inicio(); !claves.esFin(); claves.siguiente()) {
11             String e = claves.recuperar();
12             Long t = tabla.recuperar(e);
13             if (t < fecha) tabla.eliminar(e);
14         }
15     }
16 }
```

Pregunta 3 (2.5 puntos) Un proceso industrial que se inicia diariamente genera millones de valores al dia (N valores) y en cualquier momento se desea consultar los K mayores valores generados hasta ese instante. Dada la magnitud de N, no es posible mantener todos los N valores en una estructura de datos y ordenarlos en cada consulta. Se pide implementar en la clase `MonticuloBinario`, un método eficiente `insertarHastaK` que reciba como parámetro un valor y lo inserte en el montículo manteniendo fija su talla en K. Es importante destacar que puesto que el método se invoca diariamente, el montículo inicialmente no tiene ningún elemento. Puedes suponer que la clase `MonticuloBinario` está definida como sigue:

```
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {
  protected E elArray[];
  protected static final int CAPACIDAD_POR_DEFECTO = 40;
  protected int talla;

  public MonticuloBinario(){talla=0; elArray=(E[]) new Comparable[CAPACIDAD_POR_DEFECTO];}
  public MonticuloBinario(int K) {talla=0; elArray=(E[]) new Comparable[K+1];}
  public boolean esVacia(){ ... }
  public E recuperarMin(){ ... }
  public void insertar(E x){ ... }
  public E eliminarMin{ ... }
  protected void hundir(int hueco){ ... }
  public void arreglarMonticulo{ ... }
}
```

Solución

```
public void insertarHastaK(E e) {
  if (talla < elArray.length) {
    elArray[++talla] = e;
    if (talla == elArray.length-1) arreglarMonticulo();
  }
  else {
    if (elArray[1].compareTo(e) < 0) {
      elArray[1] = e;
      hundir(1);
    }
  }
}
```

Pregunta 4 (2.5 puntos) Se desea determinar si un conjunto de valores reales se ajusta *aproximadamente* a una función lineal creciente. Para ello un ingeniero informático ha pensado un algoritmo que consiste en comprobar si la diferencia entre todo par de valores consecutivos de menor a mayor está acotada por un epsilon. Escribe un método eficiente estático que recibe una cola de prioridad no vacia de valores `double` y un valor `double` epsilon positivo que implemente dicho algoritmo. El método debe devolver `true` si el conjunto de valores se ajusta aproximadamente a una función lineal creciente y `false` en caso contrario. Por ejemplo, dado un el valor epsilon 0.05 y el conjunto $C_1 = \{0.3, 0.29, 0.34\}$ el resultado del método debe ser `true`, mientras que si el conjunto es $C_1 = \{0.3, 0.29, 0.36\}$ el resultado deber ser `false`.

Solución

```
public static boolean casiLineal(ColaPrioridad<Double> q, Double epsilon) {
  double a = q.eliminarMin();
  while (!q.esVacia()) {
    double b = q.eliminarMin();
    // a<=b
    if ( b-a > epsilon ) return false;
    a = b;
  }
  return true;
}
```

Resolución del Primer Parcial de EDA (24 de Marzo de 2014)

1.- En el paquete `lineales`, se desea implementar la siguiente subinterfaz de `ListaConPI` mediante una clase que extienda `LEGListaConPI`.

```
public interface ListaConPIPlus<E> extends ListaConPI<E> {
   /** Si el Elemento e está en una Lista Con PI elimina su última aparición y devuelve
    *  true como resultado; sino, si e no está en la lista, devuelve false para advertirlo */
   public boolean eliminarUltimo(E e);
}
```

Se pide: **(3.25 puntos)**

(a) Escribir la cabecera de dicha clase. **(0.75 puntos)**

```
public class LEGListaConPIPlus<E> extends LEGListaConPI<E> implements ListaConPIPlus<E> {…}
```

(b) Implementar el método `eliminarUltimo`. Sólo se podrán utilizar los métodos de `ListaConPI` (ver Anexo). **(2 puntos)**

```
public boolean eliminarUltimo(E e) {
    this.inicio(); int pos = 0, posUltima = -1;
    while (!this.esFin()) {
       if (this.recuperar().equals(e)) posUltima = pos;
       pos++;
       this.siguiente();
    }
    if (posUltima == -1) return false;
    this.inicio(); pos = 0;
    while (pos < posUltima) {
       pos++;
       this.siguiente();
    }
    this.eliminar();
    return true;
}
```

(c) Utilizando la notación asintótica (O y Ω o bien Θ) y tomando como talla del problema el número de elementos de la `ListaConPI`, escribe el coste de `eliminarUltimo`. **(0.5 puntos)**

No hay Mejor ni Peor de los Casos porque en el primer bucle siempre se recorre toda la Lista con PI; por tanto, si x es su talla, la complejidad temporal del método es Θ(x).

2.- Se tiene un conjunto de bolas blancas y negras dispuestas sobre un array `v` de modo que al principio están todas las blancas y a continuación las negras. Considerando que una bola se representa mediante un `String` cuyo valor es "`blanca`" o "`negra`" y que al menos hay una de cada color, s**e pide:** **(3.5 puntos)**

(a) Diseñar un método Divide y Vencerás `contarBlancas` que, con coste logarítmico, devuelva el número de bolas blancas que hay en `v`. **(3 puntos)**

```
public static int contarBlancas(String[] v) {
    return contarBlancas(v, 0, v.length - 1)
}
private static int contarBlancas(String[] v, int i, int f) {
   // Búsqueda con garantía de éxito de la última bola blanca de v:
   // usando una estrategia DyV, se divide el problema en 2 subproblemas
   int mitad = (i + f) / 2;
   // Para que en mitad esté la última bola blanca, al menos v[mitad] es blanca
   if (v[mitad].equals("blanca") {
       //si v[mitad+1] es negra entonces mitad+1 es la solución
       if (v[mitad + 1].equals("negra") return mitad + 1;
       // sino, la última bola blanca solo puede estar a partir de mitad+1,
       // i.e. en el subArray v[mitad+1...f]
       else return contarBlancas(v, mitad + 1, f);
    }
    // Si v[mitad] es una bola negra, la última blanca solo puede estar antes,
    // i.e. en el subArray v[i...mitad-1]
    else return contarBlancas(v, i, mitad - 1);
}
```

(b) Justificar su coste logarítmico utilizando los teoremas de los costes. **(0.5 puntos)**

En el Peor de los Casos la talla `x` decrece geométricamente en la única llamada que se produce y el coste del resto de operaciones (la sobrecarga) es constante. Por tanto, aplicando el Teorema 3 con a = 1 y c = 2, se tiene que el coste es logarítmico, i.e. $T(x) \in O(\log x)$.

3.- Se dispone de los siguientes métodos `examen`: **(3.25 puntos)**

```
public int examen(int[] v) { return examen(v, 0, v.length-1); }
protected int examen(int[] v, int p, int f) {
  if (p > f) return 0;
  else if (p == f) return v[p];
  else {   int c = (p + f) / 2;
           int sol1 = examen(v, p, c);  int sol2 = examen(v, c + 1, f);
           if (sol1 == sol2) return sol1;
           if (sol1 > 0) if (test(v, p, f, sol1)) return sol1;
           if (sol2 > 0) if (test(v, p, f, sol2)) return sol2;
           return 0;
  }
}
```

Sabiendo que todos los elementos de `v` son mayores o iguales que cero y que el método `test` tiene siempre un coste lineal con la talla del subarray sobre el que se aplica, **se pide**:

(a) Si `v.length=8`, completar los siguientes enunciados sobre el método `protected examen`: **(0.5 puntos)**

- La llamada `examen(v, 6, 6)` es la penúltima que se genera y `examen(v, 7, 7)` la última.
- La llamada `examen(v, 0, 0)` es la primera que se resuelve y `examen(v, 0, 7)` la última.

(b) Marcar con una cruz la casilla correspondiente al tamaño o talla, `x`, del problema que resuelve el método `protected examen`. **(0.25 puntos)**

☐ `x = v.length` ☐ `x = (p + f) / 2` ☒ `x = f - p + 1` ☐ `x = p - f + 1`

☐ Según sea el Peor o Mejor de los Casos, `x=v.length` o $x \leq 1$ respectivamente.

(c) Para una talla `x` dada, marcar aquellas que entre las siguientes (una o más) sean las instancias significativas que presenta el método `protected examen`. **(0.5 puntos)**

- ☐ No existen instancias significativas, pues siempre se genera el mismo número de llamadas recursivas.
- ☐ El Mejor de los Casos se da cuando el subarray v[p...f] que se le pasa como argumento en la llamada principal tiene cero o un elemento, i.e. cuando `x=0` o `x=1` respectivamente.
- ☒ El Mejor de los Casos se da cuando el subarray v[p...f] que se le pasa como argumento en la llamada principal tiene todos sus elementos iguales.
- ☐ El Peor de los Casos se da cuando el subarray v[p...f] que se le pasa como argumento en su llamada principal tiene todos sus elementos iguales.
- ☒ El Peor de los Casos se da cuando en todas las llamadas sucede que sol1 es distinto de sol2.

(d) En función de la respuesta dada en el apartado anterior, completar y resolver las Ecuaciones de Recurrencia que expresan la Complejidad Temporal del método, indicando su solución con notación asintótica. Si no hubieran instancias significativas, usa el espacio reservado al Peor de los Casos en el siguiente recuadro. **(1.5 puntos)**

Caso base: si $x \leq 1$ entonces $T(x) = k$

Mejor de los Casos, caso general: Si $x > 1$ entonces $T^M(x) = 2 \cdot T^M(x/2) + k'$. Por tanto, por Teorema 3 con $a = c = 2$ y sobrecarga constante, $T^M(x) \in \Theta(x)$

Peor de los Casos, caso general: Si $x > 1$ entonces $T^P(x) = 2 \cdot T^P(x/2) + k'' \cdot x$. Por tanto, por Teorema 4 con $a = c = 2$ y sobrecarga lineal, $T^P(x) \in \Theta(x \cdot \log x)$

(e) En función del resultado del apartado (d), indicar el coste Temporal Asintótico de dicho método. **(0. 5 puntos)**

$T(x) \in \Omega(x)$ y $T(x) \in O(x \cdot \log x)$

ANEXO

La interfaz `ListaConPI` del paquete `modelos`.

```
public interface ListaConPI<E> {
	void insertar(E e);
	/** SII !esFin() */ void eliminar();
	void inicio();
	/** SII !esFin() */ void siguiente();
	void fin();
	/** SII !esFin() */ E recuperar();
	boolean esFin();
	boolean esVacia();
	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 $a>c$, $f(x) \in \Theta(x^{\log_c a})$;

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 $a=1$;

Resolución del Primer Parcial de EDA (26 de Marzo de 2019)

1.- ***(2 puntos)*** El siguiente método de la clase **ArrayCola** invierte el orden de los elementos de una Cola, usando una **ListaConPI** auxiliar. Escribe en cada recuadro el número de la opción (ver listado a la derecha) que le corresponde.

```
public void invertir() {
    ListaConPI<E> lpi = [6] ();
    while ( !this.esVacia() ) {
        [2] ( [7] () );
        [4] ();
    }
    while ( !lpi.esVacia() ) {
        [1] ( [8] () );
        [3] ();
    }
}
```

① **this**.encolar
② lpi.insertar
③ lpi.eliminar
④ lpi.inicio
⑤ **new** ListaConPI<E>
⑥ **new** LEGListaConPI<E>
⑦ **this**.desencolar
⑧ lpi.recuperar

2.- ***(2 puntos)*** Estudia el coste Temporal del siguiente método:

```
/* Precondición: v ordenado ascendentemente AND a < b */
private static int metodo(int[] v, int ini, int fin, int a, int b) {
    if (ini > fin) { return 0; }
    int mitad = (ini + fin) / 2, res = 0;
    if (v[mitad] >= a) { res += metodo(v, mitad + 1, fin, a, b); }
    if (v[mitad] <= b) { res += metodo(v, ini, mitad - 1, a, b); }
    if (v[mitad] >= a && v[mitad] <= b) { res++; }
    return res;
}
```

a) Expresa la talla del problema **x** en función de los parámetros del método. **x =** fin – ini + 1 ***(0.2 puntos)***

b) Indica si hay instancias significativas para una talla dada y por qué. En caso afirmativo, descríbelas**.** ***(0.6 puntos)***

Sí hay instancias significativas:

- **Mejor caso:** todos los elementos de *v* son menores que *a* (o mayores que *b*)
- **Peor caso:** todos los elementos de *v* están en el intervalo [a, b]

c) En base a tu apartado **b)**, escribe la(s) Relación(es) de Recurrencia que expresa(n) el coste del método. ***(0.6 puntos)***

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

$T_{metodo}^{M}(x) = 1 * T_{metodo}^{M}(x / 2) + k$

$T_{metodo}^{P}(x) = 2 * T_{metodo}^{P}(x / 2) + k$

d) Resuelve la(s) Relación(es) de Recurrencia del apartado **c)**, indicando el(los) Teoremas de Coste que usas y escribiendo el coste Temporal del método en notación asintótica (O y Ω o bien Θ). ***(0.6 puntos)***

Por Teorema 3 con a = 1 y c = 2, $T_{metodo}(x) \in \Omega(\log_2 x)$.

Por Teorema 3 con a = 2 y c = 2, $T_{metodo}(x) \in O(x)$.

3.- ***(3 puntos)*** Sea una aplicación de gestión de notas en la que se usa un **`Map<Alumno, Double>`**, cada una de cuyas Entradas representa a un alumno y la nota que este ha obtenido en una asignatura.

a) Implementa un método cuyo perfil sea el mostrado en el siguiente recuadro, donde el parámetro **`m`** es el Map de las notas de todos los alumnos de una asignatura. El método debe realizar las siguientes acciones: ***(2.25 puntos)***

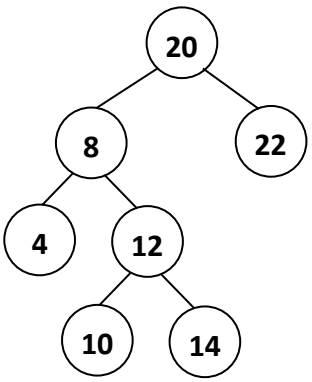
- Devolver un (nuevo) Map que contenga únicamente las Entradas de **`m`** que corresponden a alumnos aprobados (con nota mayor o igual que 5.0).
- Eliminar del Map **`m`** todas las Entradas que corresponden a alumnos aprobados. Es decir, al terminar la ejecución del método, **`m`** debe contener únicamente las Entradas correspondientes a alumnos suspendidos.

```
public static Map<Alumno, Double> obtenerAprobados(Map<Alumno, Double> m) {
    Map<Alumno, Double> aprobados = new TablaHash<Alumno, Double>(m.talla());
    ListaConPI<Alumno> l = m.claves();
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        Alumno alumno = l.recuperar();
        Double nota = m.recuperar(alumno);
        if (nota >= 5.0) {
            aprobados.insertar(alumno, nota);
            m.eliminar(alumno);
        }
    }
    return aprobados;
}
```

b) Suponiendo que el Map **`m`** se ha implementado eficientemente mediante una **`TablaHash`**, indica para el método diseñado: la talla del problema **x** que resuelve, en función de sus parámetros; las instancias significativas que presenta, si las hubiera; su coste Temporal en notación asintótica (O y Ω o bien Θ). ***(0.75 puntos)***

- Talla del problema, en función de los parámetros del método: **x** = `m.talla()`.
- Instancias significativas: no hay, pues se trata de un método de Recorrido.
- Utilizando la notación asintótica, $T_{obtenerAprobados}(x) \in \Theta(x)$.

4.- ***(3 puntos)*** El Ascendiente Común "Más Bajo", o *Lowest Common Ancestor* (*LCA*), de dos elementos **e1** y **e2** de un Árbol se define como el elemento situado en el nodo "más bajo" (a mayor profundidad o distancia de la raíz) del que los nodos que contienen a **e1** y **e2** son descendientes, pudiendo ser un nodo descendiente de él mismo. Así, por ejemplo, en el ABB de la figura a tu derecha: el *LCA* de 10 y 14 es 12, el de 8 y 14 es 8, el de 10 y 22 es 20 y el de 8 y 22 es 20.



En la clase **ABB**, implementa un método público **`lCA`** que, en tiempo lineal con su altura, devuelva el *LCA* de **e1** y **e2** en un ABB Equilibrado, no vacío y sin elementos repetidos. Asume además que **e1** y **e2** son dos elementos del ABB, por lo que su *LCA* siempre existe, y que **e1** es menor que **e2**.

```
/ ** SII ABB no es vacío, e1 y e2 están en el ABB y e1 < e2 */
public E lCA(E e1, E e2) { return lCA(this.raiz, e1, e2).dato; }

// Búsqueda con garantía de éxito del Nodo del LCA de e1 y e2
protected NodoABB<E> lCA(NodoABB<E> n, E e1, E e2) {

    // Como e1 < e2, si n.dato > e2, también n.dato > e1
    if (n.dato.compareTo(e2) > 0) { return lCA(n.izq, e1, e2); }

    // Sino, si n.dato < e1, también n.dato < e2
   if (n.dato.compareTo(e1) < 0) { return lCA(n.der, e1, e2); }

   // Sino, o n.dato = e1, o n.dato = e2, o n.dato > e1 y n.dato < e2
   return n;

}
```

ANEXO

Las interfaces `Map` y `ListaConPI` 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();
}
```

```
public interface ListaConPI<E> {
   void insertar(E e);
   void eliminar();
   void inicio();
   void siguiente();
   void fin();
   E recuperar();
   boolean esFin();
   boolean esVacia();
   int talla();
}
```

Las clases `NodoABB` y `ABB` del paquete `jerarquicos`.

```
class NodoABB<E> {
     E dato;
     NodoABB<E> izq, der;
     int talla;
     NodoABB(E dato) {…}
}
```

```
public class ABB<E extends Comparable<E>> {
     protected NodoABB<E> raiz;
     protected int talla;
     public ABB() {…}
     …
}
```

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 $T(x) = a \cdot T(x/b) + \Theta(x^k)$, con $a \geq 1$ y $b>1$ es:

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

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

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

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 $a>1$;

Resolución del examen de Recuperación del Primer Parcial de EDA (12 de Junio de 2019)

1.- ***(3 puntos)*** En la clase **ABB**, implementa un método público que, con el menor coste temporal posible, elimine el nodo que contiene el máximo de un ABB.

```
public void eliminarMax() {
    if (this.raiz != null) { this.raiz = eliminarMax(this.raiz); }
}
//SII actual != null: devuelve el nodo actual tras eliminar su máximo
protected NodoABB<E> eliminarMax(NodoABB<E> actual) {
    NodoABB<E> res = actual;
    if (actual.der != null) {
        res.der =  eliminarMax(actual.der);
        res.talla--; // o res.talla = 1 + talla(res.izq) + talla(res.der);
    }
    else { res = actual.izq; }
    return res;
}
```

Una vez diseñado el método, y suponiendo que se aplica sobre un ABB Equilibrado de N nodos y altura H, ...

a) Indica la talla del problema que resuelve **x =** N, o bien H = $\log_2$ N ***(0.1 puntos)***

b) Indica si hay instancias significativas para una talla dada y por qué. En caso afirmativo, descríbelas. ***(0.25 puntos)***

No hay instancias significativas, pues el método realiza siempre un **Recorrido** en Pre-Orden del Camino más a la derecha del ABB hasta alcanzar su último nodo, lo que siempre tiene el mismo coste en un ABB equilibrado: del orden de la altura del ABB, o del logaritmo de su talla. Además, el borrado del nodo más a la derecha del ABB no añade coste ni casos, pues como solo puede ser una Hoja o un nodo con hijo Izquierdo, se realizará en tiempo constante.

c) Indica su coste Temporal utilizando la notación asintótica (O y Ω o bien Θ). ***(0.4 puntos)***

$T_{eliminarMax}(N) \in \Theta(\log_2 N)$, o bien $T_{eliminarMax}(H) \in \Theta(H = \log_2 N)$

2.- ***(3 puntos)*** Escribe un método estático Divide y Vencerás que, dados un array **v** de **`int`** ordenado ascendentemente y sin elementos repetidos y un **`int x`**, devuelva el elemento de **v** con valor más cercano al de **x**. Para ello...

- Puedes suponer que **v** tiene como mínimo tres elementos y que, de ellos, el elemento de valor más cercano al de **x** no está NI en la primera posición de **v** NI en la última.
- Puedes usar en tu código un método **`comparar`** que, como su nombre indica y en el orden de una constante, compara con **x** dos elementos **eV1** y **eV2** de **v** y devuelve aquel de ellos con valor más cercano al de **x**. Su perfil es: **`int comparar(int[] v, int eV1, int eV2, int x)`**.

```
public static int buscarMasCercano(int[] v, int x) {
    return buscarMasCercano(v, x, 0, v.length - 1);
}
protected static int buscarMasCercano(int[] v, int x, int ini, int fin) {
    int m = (ini + fin) / 2;
    if (v[m] == x) { return v[m]; }
    if (v[m] < x) {
        if (v[m + 1] > x) { return comparar(v, v[mitad], v[mitad + 1], x); }
        else { return buscarMasCercano(v, x, m + 1, fin); }
    }
    else if (v[m - 1] < x) { return comparar(v, v[m - 1], v[m], x); }
    else { return buscarMasCercano(v, x, ini, m - 1); }
}
```

Una vez diseñado el método, estudia su coste temporal del método recursivo que lanza. En concreto:

a) Expresa la talla del problema **x** en función de sus parámetros **x =** `fin - ini + 1` ***(0.2 puntos)***

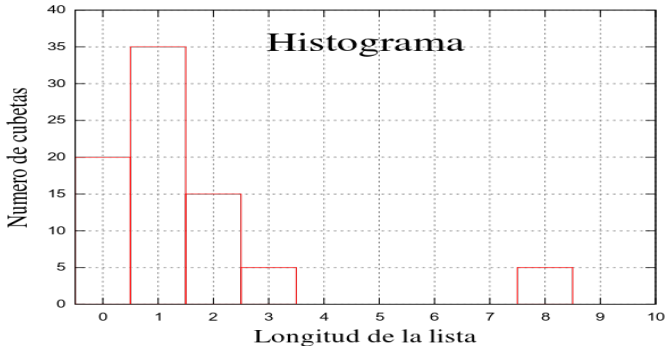
b) Escribe la(s) Relación(es) de Recurrencia que expresa(n) su coste. ***(0.4 puntos)***

En el caso general, **cuando x >** **2** , $T_{buscarMasCercano}^{M}(x) = k$ y $T_{buscarMasCercano}^{P}(x) = 1 * T_{buscarMasCercano}^{P}(x / 2) + k'$.

c) Resuelve la(s) Relación(es) de Recurrencia del apartado **b)**, indicando el(los) Teoremas de Coste que usas y escribiendo el coste Temporal del método en notación asintótica (O y Ω o bien Θ). ***(0.4 puntos)***

$T_{buscarMasCercano}(x) \in \Omega(1)$ y, por Teorema 3 con a = 1 y c = 2, $T_{buscarMasCercano}(x) \in O(\log_2 x)$.

3.- ***(2 puntos)*** Dado el siguiente histograma de ocupación de una Tabla Hash (con Hashing Enlazado):



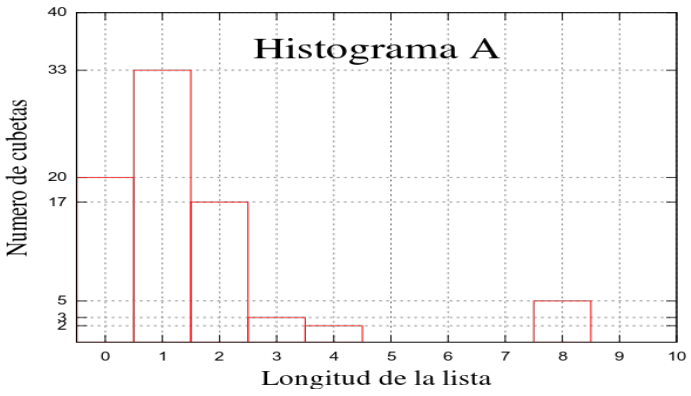
a) Indica el número de cubetas y el de elementos que tiene la Tabla, dejando indicadas las operaciones que has realizado para calcular dichos valores. ***(0.5 puntos)***

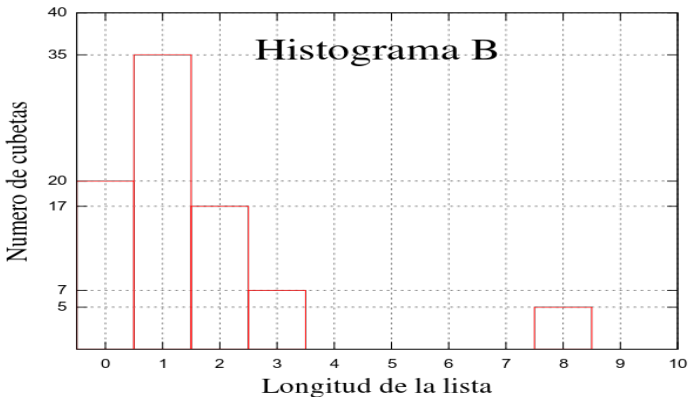
Número de cubetas, o `elArray.length` = 20 + 35 + 15 + 5 + 5 = 80.
Número de elementos de la Tabla, o `talla` = 20 * 0 + 35 * 1 + 15 * 2 + 5 * 3 + 5 * 8 = 120.

b) Calcula el Factor de Carga de la Tabla y la desviación típica de las longitudes de sus cubetas, dejando indicadas las operaciones que has realizado para calcular dichos valores. ***(0.5 puntos)***

FC = `talla / elArray.length` = 120 / 80 = 1.5.
σ = sqrt((20 * (0-1.5)² + 35 * (1-1.5)² + 15 * (2-1.5)² + 5 * (3-1.5)² + 5 * (8-1.5)²) / 80).

c) Supón que se insertan cuatro elementos distintos y que no estaban en la Tabla. Indica cuál de los siguientes es el histograma de ocupación de la Tabla tras estas inserciones. Es imprescindible que razones tu respuesta. ***(1 punto)***





El histograma de ocupación resultante es el A porque en él hay dos cubetas más de 2 elementos que en el anterior (tras insertar 2 nuevos elementos en 2 de sus cubetas de longitud uno) y, consecuentemente, dos cubetas menos de 1 elemento que en la anterior. Además, también hay en él dos cubetas más de 4 elementos que en el anterior (tras insertar 2 nuevos elementos en 2 de sus cubetas de longitud tres).
Por otra parte, el histograma resultante no puede ser el B porque tras insertar nuevos elementos en la Tabla no puede aumentar el número de cubetas de 1 elemento SIN disminuir el número de las que tienen 0 o 1.

4.- ***(2 puntos)*** El siguiente método de la clase **`ArrayColaExt<E extends Comparable<E>>`**, que extiende de **`ArrayCola<E>`**, ordena ascendentemente los elementos de una Cola usando una **`ListaConPI`** como estructura auxiliar. Escribe en cada recuadro el número de la opción (ver listado a la derecha) que le corresponde.

```
public void ordenar() {
  [6] lpi = new [5] ();
  while (!this.esVacia()) {
    E e = [2] ();
    for (lpi.inicio(); [3] && [1] ; [8] );
      [7] (e);
  }
  lpi.inicio();
  while (!lpi.esVacia()) {
    [10] ( [9] ());
    [4] ();
  }
}
```

① `lpi.recuperar().compareTo(e) <= 0`
② `this.desencolar`
③ `!lpi.esFin()`
④ `lpi.eliminar`
⑤ `LEGListaConPI<E>`
⑥ `ListaConPI<E>`
⑦ `lpi.insertar`
⑧ `lpi.siguiente()`
⑨ `lpi.recuperar`
⑩ `this.encolar`

ANEXO

Las clases `NodoABB` y `ABB` del paquete `jerarquicos`.

```
class NodoABB<E> {
    E dato;
    NodoABB<E> izq, der;
    int talla;
    NodoABB(E dato) {…}
}
```

```
public class ABB<E extends Comparable<E>> {
    protected NodoABB<E> raiz;
    protected int talla;
    public ABB() {…}
    …
}
```

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 $T(x) = a \cdot T(x/b) + \Theta(x^k)$, con $a \geq 1$ y $b>1$ es:

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

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

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