P1 Práctica. (2 puntos) Supongamos que se levanta la restricción de que cada académico sólo pueda realizar una tesis doctoral (nótese que deberán conservarse los supervisores para cada tesis de cada doctor).

a) ¿Cómo afectaría a la representación de los datos en el segundo supuesto (academia con árbol genealógico completo)? Comente los cambios necesarios, si los hubiere, sin escribir el código.

b) ¿Qué cambios habría que realizar en el método `getSiblings` para que, recibiendo la tesis como parámetro (en caso de múltiples tesis para un mismo doctor), devolviese los hermanos debidos a esa tesis para ese doctor?

1. (1.5 punto) Considérense las implementaciones de los números factorial(n) y fibonacci(n) mediante sus definiciones matemáticas recursivas (para naturales, es decir, la precondición de ambas exige *enteros no negativos*):

```
int fact(int n){
  if(n == 0) return 1;
  else return n * fact(n-1);
}
```

```
int fib(int n){
  if(n<2) return 1;
  else return fib(n-1)+fib(n-2);
}
```

   Explicar de forma razonada cuál sería su coste asintótico temporal en caso peor, y qué se puede concluir sobre la adecuación de cada una de estas implementaciones.

2. (1.5 puntos) Impleméntese un programa recursivo que compruebe si un árbol binario de enteros tiene algún valor mayor que un número dado k. Razónese cual es su coste asintótico temporal y compárese con el de la implementación directa de la función de Fibonacci considerada en el ejercicio anterior.

3. Se desea abordar el problema de la ordenación de enteros cuando el conjunto de entrada tiene muchas repeticiones y una cantidad reducida de números diferentes (por ejemplo, cuando en una oficina de correos se organizan los sobres por código postal; hay un número reducido de códigos postales que llegan a esa oficina, y un número grande de envíos que corresponden a cada uno de los códigos). Llamaremos `PCollection` a un conjunto de enteros con estas características. Su interfaz puede ser:

   `PCollectionIF`:

```
/* Representa una bolsa de números con pocos elementos distintos          *
 * posiblemente muy repetidos.                                            */
public interface PCollectionIF {
  /* Consulta la frecuencia con que aparece un entero num en la colección. *
  /* @param el entero num                                                  *
  /* @returns el número de veces que aparece num en la colección.         */
  public int frequency (int num)

  /* Consulta el tamaño de la colección.                                   *
  /* @returns el número total de enteros en la colección (contando los     *
              repetidos.                                                   */
  public int size()

  /* Consulta los valores distintos que aparecen en la colección.          *
   * @returns lista de valores diferentes que aparecen en la colección,    *
              ordenados de menor a mayor.                                  */
  public ListIF getValues ()

  /* Ordena los valores en la colección de menor a mayor.                  *
  /* @returns la lista completa de enteros de la colección, con            *
```

```
            repeticiones, ordenados de menor a mayor.                    */
  public ListIF sort ()

  /* Añade un nuevo número a la colección.                                  */
  public void addNumber (int num)
}
```

Una forma eficiente de ordenar un conjunto de estas características es haciendo analogía con el reparto de cartas en buzones. En este caso, el equivalente de echar la carta en su buzón consiste en disponer de un espacio reservado para cada valor posible, y en ese espacio ir contando el número de veces que aparece ese valor según se va incrementando la colección.

Contemplaremos dos supuestos diferentes:

A) Se conoce de antemano la lista de valores posibles que toman los enteros de la colección. Un ejemplo serían las notas obtenidas por una serie de estudiantes en una o varias asignaturas (siempre que sean enteras), ya que sabemos que los valores serán siempre enteros del 0 al 10.

B) Se desconoce a priori cual es la lista de valores posibles.

Se pide:

a) (1'5 puntos) Implementar los métodos de la interfaz `PCollectionIF` en el supuesto (A) mediante una clase `PCollectionA`. Se requiere que el método size tenga un coste asintótico temporal en $\mathcal{O}(1)$.

b) (1'5 puntos) Implementar los métodos `frequency` y `getValues` de la interfaz `PCollectionIF` en el supuesto (B) mediante una clase `PCollectionB`, con la misma restricción sobre el método `size`.

c) (1 punto) Razonar sobre el coste asintótico temporal en caso peor del método `frequency` en cada uno de los supuestos.

d) (1 punto) Razonar sobre el coste asintótico temporal en caso peor del método `sort` en cada uno de los supuestos. Se sabe que el problema general de ordenación de n enteros arbitrarios puede resolverse, en el mejor de los casos, con un coste asintótico temporal en caso peor de $\mathcal{O}(n \log n)$. Comparar esta cota con los costes calculados y establecer conclusiones.

CollectionIF (Colección)

```
/* Representa una colección de elementos. Una colección no          *
 * tiene orden.                                                     */
public interface CollectionIF<E> {
    /* Devuelve el número de elementos de la colección.             *
     * @return: cardinalidad de la colección.                       */
    public int size ();
    /* Determina si la colección está vacía.                        *
     * @return: size () == 0                                        */
    public boolean isEmpty ();
    /* Determina la pertenencia del parámetro a la colección        *
     * @param: el elemento cuya pertenencia se comprueba.           *
     * @return:param \in self                                       */
    public boolean contains (E e);
    /* Elimina todos los elementos de la colección.                 */
    public void clear ();
    /* Devuelve un iterador sobre la colección.                     *
     * @return: un objeto iterador para los elementos de            *
     * la colección.                                                */
    public IteratorIF<E> iterator ();
}
```

SetIF (Conjunto)

```
/* Representa un conjunto de elementos. Se trata del concepto       *
 * matemático de conjunto finito (no tiene orden).                  */
public interface SetIF<E> extends CollectionIF<E> {
    /* Devuelve la unión del conjunto llamante con el parámetro     *
     * @param: el conjunto con el que realizar la unión             *
     * @return: self \cup @param                                    */
    public SetIF<E> union (SetIF<E> s);
    /* Devuelve la intersección con el parámetro.                   *
     * @param: el conjunto con el que realizar la intersección.     *
     * @return: self \cap @param                                    */
    public SetIF<E> intersection (SetIF<E> s);
    /* Devuelve la diferencia con el parámetro (los elementos       *
     * que están en el llamante pero no en el parámetro).           *
     * @param: el conjunto con el que realizar la diferencia.       *
     * @return: self \setminus @param                               */
    public SetIF<E> difference (SetIF<E> s);
    /* Determina si el parámetro es un subconjunto del llamante.    *
     * @param: el posible subconjunto del llamante.                 *
     * @return: self \subseteq @param                               */
    public boolean isSubset (SetIF<E> s);
}
```

ListIF (Lista)

```
/* Representa una lista de elementos.                               */
public interface ListIF<E> extends CollectionIF<E>{
    /* Devuelve el elemento de la lista que ocupa la posición       *
     * indicada por el parámetro.                                   *
     * @param pos la posición comenzando en 1.                      *
     * @Pre: 1 \leq pos \leq size()                                 *
     * @return el elemento en la posición pos.                     */
    public E get (int pos);
    /* Modifica la posición dada por el parámetro pos para que      *
```

```
     * contenga el valor dado por el parámetro e.                    *
     * @param pos la posición cuyo valor se debe modificar,          *
     *  comenzando en 1.                                             *
     * @param e el valor que debe adoptar la posición pos.           *
     * @Pre: 1 \leq pos \leq size()                                  */
    public void set (int pos, E e);
    /* Inserta un elemento en la Lista.                              *
     * @param elem El elemento que hay que añadir.                   *
     * @param pos  La posición en la que se debe añadir elem,        *
     *  comenzando en 1.                                             *
     * @Pre: 1 \leq pos \leq size()+1                                */
    public void insert (E elem, int pos);
    /* Elimina el elemento que ocupa la posición del parámetro       *
     * @param pos la posición que ocupa el elemento a eliminar,      *
     *  comenzando en 1                                              *
     * @Pre: 1 \leq pos \leq size()                                  */
    public void remove (int pos);
}
```

StackIF (Pila)

```
/* Representa una pila de elementos.                                */
public interface StackIF <E> extends CollectionIF<E>{
    /* Obtiene el elemento en la cima de la pila                     *
     * @Pre !isEmpty ();                                             *
     * @return la cima de la pila.                                   */
     public E getTop ();
    /* Incluye un elemento en la cima de la pila. Modifica el        *
     * tamaño de la misma.                                          *
     * @param elem el elemento que se quiere añadir en la cima       */
     public void push (E elem);
    /* Elimina la cima de la pila. Modifica el tamaño de la          *
     * pila.                                                        *
     * @Pre !isEmpty ();                                             */
     public void pop ();
}
```

QueueIF (Cola)

```
/* Representa una cola de elementos.                                */
public interface QueueIF <E> extends CollectionIF<E>{
    /* Devuelve el primer elemento de la cola.                       *
     * @Pre !isEmpty()                                               *
     * @return la cabeza de la cola (su primer elemento).            */
    public E getFirst ();
    /* Incluye un elemento al final de la cola. Modifica el          *
     * tamaño de la misma (crece en una unidad).                    *
     * @param elem el elemento que debe encolar (añadir).            */
    public void enqueue (E elem);
    /* Elimina el primer elemento de la cola. Modifica la            *
     * tamaño de la misma (decrece en una unidad).                  *
     * @Pre !isEmpty();                                              */
    public void dequeue ();
}
```

TreeIF (Árbol general)

```
/* Representa un árbol n-ario de elementos, donde el número de    *
```

```
 * hijos de un determinado nodo no está determinado de antemano  *
 * (fan-out no prefijado, no necesariamente igual en cada nodo). */
public interface TreeIF<E> extends CollectionIF<E>{
    public int PREORDER  = 0;
    public int POSTORDER = 1;
    public int BREADTH   = 2;
    /* Obtiene la raíz del árbol (único elemento sin antecesor). *
     * @Pre: !isEmpty ();                                        *
     * @return el elemento que ocupa la raíz del árbol.          */
     public E getRoot ();
    /* Modifica la raíz del árbol.                               *
     * @param el elemento que se quiere poner como raíz del      *
     * árbol.                                                    */
     public void setRoot (E e);
    /* Obtiene los hijos del árbol llamante.                     *
     * @Pre: !isEmpty ();                                        *
     * @return la lista de hijos del árbol (en el orden en que   *
     * están almacenados en el mismo).                           */
     public ListIF <TreeIF <E>> getChildren ();
    /* Obtiene el hijo que ocupa la posición dada por parámetro. *
     * @param pos la posición del hijo que se desea obtener,     *
     * comenzando en 1.                                          *
     * @Pre 1 \leq pos \leq getChildren().size() && !isEmpty()   *
     * @return el árbol hijo que ocupa la posición pos.          */
     public TreeIF<E> getChild (int pos);
    /* Inserta un árbol como hijo en la posición pos.            *
     * @param pos la posición que ocupará el árbol entre sus     *
     * hermanos, comenzando en 1.                                *
     * Si pos == getChildren ().size () + 1, se añade como       *
     * último hijo.                                              *
     * @param e el hijo que se desea insertar.                   *
     * @Pre 1 \leq pos \leq getChildren().size()+1 && !isEmpty() */
     public void addChild (int pos, TreeIF<E> e);
    /* Elimina el hijo que ocupa la posición parámetro.          *
     * @param pos la posición del hijo con base 1.               *
     * @Pre 1 \leq pos \leq getChildren().size() && !isEmpty()   */
     public void removeChild (int pos);
    /* Determina si el árbol llamante es una hoja.               *
     * @Pre: !isEmpty (); (un arbol vacio no se considera hoja)  *
     * @return el árbol es una hoja (no tiene hijos).            */
     public boolean isLeaf ();
    /* Obtiene un iterador para el árbol.                        *
     * @param traversal el tipo de recorrido indicado por las    *
     * constantes PREORDER (preorden o profundidad), POSTORDER   *
     * (postorden) o BREADTH (anchura)                           *
     * @return un iterador según el recorrido indicado           */
     public IteratorIF<E> iterator (int traversal);
}
```

BTreeIF (Árbol Binario)

```
/* Representa un arbol binario de elementos                     */
public interface BTreeIF<E> extends CollectionIF<E>{
  public int PREORDER  = 0;
  public int POSTORDER = 1;
  public int BREADTH = 2;
```

```
  public int INORDER   = 3;
  public int RLBREADTH = 4;
  /* Obtiene la raíz del árbol (único elemento sin antecesor).    *
   * @Pre: !isEmpty ();                                           *
   * @return el elemento que ocupa la raíz del árbol.             */
  public E getRoot ();
  /* Obtiene el hijo izquierdo del árbol llamante o un árbol      *
   * vacío en caso de no existir.                                 *
   * @Pre: !isEmpty ();                                           *
   * @return un árbol, bien el hijo izquierdo bien uno vacío      *
   * de no existir tal hijo.                                      */
  public BTreeIF<E> getLeftChild ();
  /* Obtiene el hijo derecho del árbol llamante o un árbol        *
   * vacío en caso de no existir.                                 *
   * @Pre: !isEmpty ();                                           *
   * @return un árbol, bien el hijo derecho bien uno vacío        *
   * de no existir tal hijo.                                      */
  public BTreeIF<E> getRightChild ();
  /* Modifica la raíz del árbol.                                  *
   * @param el elemento que se quiere poner como raíz del         *
   * árbol.                                                       */
  public void setRoot (E e);
  /* Pone el árbol parámetro como hijo izquierdo del árbol        *
   * llamante. Si ya había hijo izquierdo, el antiguo dejará de   *
   * ser accesible (se pierde).                                   *
   * @Pre: !isEmpty ();                                           *
   * @param child el árbol que se debe poner como hijo izquierdo.*/
  public void setLeftChild (BTreeIF <E> child);
  /* Pone el árbol parámetro como hijo derecho del árbol          *
   * llamante. Si ya había hijo izquierdo, el antiguo dejará de   *
   * ser accesible (se pierde).                                   *
   * @Pre: !isEmpty ();                                           *
   * @param child el árbol que se debe poner como hijo derecho.   */
  public void setRightChild (BTreeIF <E> child);
  /* Elimina el hijo izquierdo del árbol.                         *
   * @Pre: !isEmpty ();                                           */
  public void removeLeftChild ();
  /* Elimina el hijo derecho del árbol.                           *
   * @Pre: !isEmpty ();                                           */
  public void removeRightChild ();
  /* Determina si el árbol llamante es una hoja.                  *
   * @Pre: !isEmpty (); (un arbol vacio no se considera hoja)     *
   * @return true sii el árbol es una hoja (no tiene hijos).      */
  public boolean isLeaf ();
  /* Obtiene un iterador para el árbol.                           *
   * @param traversal el tipo de recorrido indicado por las       *
   * constantes PREORDER (preorden o profundidad), POSTORDER      *
   * (postorden), BREADTH (anchura), INORDER (inorden) o          *
   * RLBREADTH (anchura de derecha a izquierda).                  *
   * @return un iterador según el recorrido indicado.            */
  public IteratorIF<E> iterator (int traversal);
}
```

ComparatorIF (Comparador)

```
/* Representa un comparador entre elementos respecto a una       *
```

```
 * relación de (al menos) preorden.                                    */
public interface ComparatorIF<E>{
  /* Sean a, b elementos de un conjunto dado y \sqsubset la           *
   * relación que establece un preorden entre ellos (nótese           *
   * que \sqsupset sería la relación recíproca, es decir, en          *
   * sentido opuesto a \sqsubset):                                    */
  public static int LT = -1; // Less than: a \sqsubset b
  public static int EQ = 0;  // Equals: !(a \sqsubset b) &&
                             // && !(a \sqsupset b)
  public static int GT = 1;  // Greater than: a \sqsupset b
  /* Compara dos elementos respecto a un preorden e indica su         *
   * relación respecto al mismo, es decir, cuál precede al            *
   * otro mediante esa relación.                                      *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return LT sii a \subsesq b;                                     *
   *         EQ sii !(a \subsetsq b) && !(a \sqsupset b)              *
   *         GT sii a \sqsupset b                                     */
  public int compare (E a, E b);
  /* Determina si el primer parámetro precede en el preorden          *
   * al segundo (a < b).                                              *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return a \sqsubset b;                                           */
  public boolean lt (E a, E b);
  /* Determina si el primer parámetro es igual al segundo en          *
   * el preorden.                                                     *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return a EQ b sii !(a \sqsubset b) && !(a \sqsupset b)          */
  public boolean eq (E a, E b);
  /* Determina si el primer parámetro sucede en el preorden           *
   * al segundo (b > a).                                              *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return a GT b sii b \sqsupset a                                 */
  public boolean gt (E a, E b);
}
```

IteratorIF (Iterador)

```
/* Representa un iterador sobre un Tipo Abstracto de Datos.          */
public interface IteratorIF<T>{
   /* Obtiene el siguiente elemento de la iteración.                 *
    * @Pre: hasNext ();                                              *
    * @return el siguiente elemento de la iteración,                 */
   public T getNext ();
   /* Comprueba si aún quedan elementos por iterar.                  *
    * @return true sii el iterador dispone de más elementos.        */
   public boolean hasNext ();
   /* Vuelve la posición del iterador al principio. Esto             *
    * permite reutilizar un iterador evitando crear otro nuevo.     */
   public void reset ();
}
```