Estrategias de Programación y Estructuras de Datos

Dpto. de Lenguajes y Sistemas Informáticos

Material permitido: NINGUNO. Duración: 2 horas

Alumno:	**D.N.I.:**
C. Asociado en que realizó la Práctica Obligatoria:	

Este documento detalla una posible solución propuesta por el Equipo Docente con intención didáctica antes que normativa. Nótese que el nivel de detalle de este documento no es el pretendido para los ejercicios realizados por los alumnos en la solución de sus exámenes.
Por brevedad, no se incluyen las interfaces de los tipos, que el alumno puede consultar en la documentación de la asignatura.

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.

En lugar de mantener una única lista de supervisores asociada a cada Doctor, sería necesario mantener una lista de pares <`Tesis`, `ListIF`< `DoctorIF`>>, de manera que se pueda conservar la relación de supervisión asociada a cada una de las Tesis Doctorales realizadas por el Doctor. Nótese que la lista de Doctores supone la colección de supervisores del Doctor autor de esa tesis.

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?

Teniendo en cuenta los cambios sugeridos en el apartado anterior, la diferencia con respecto a la estructura utilizada en la Práctica es que la lista de supervisores del Doctor ahora depende de la Tesis Doctoral que se considere, mientras que en la Práctica se podía obtener de forma directa.

Dado que para obtener los hermanos académicos del Doctor es necesario operar con dicha lista de supervisores, será necesaria una primera operación adicional consistente en buscar la lista de los doctores que fueron los supervisores de esa Tesis (recibida como parámetro por el método `getSiblings`) en primer lugar.

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.

En primer lugar, hay que establecer cuál es el tamaño del problema. En ambos casos el tamaño del problema se puede fijar en n, el parámetro natural que recibe cada una de las funciones.

A continuación hay que identificar cuál es la forma de las funciones: las dos son funciones recursivas y en ambos casos la reducción del problema se realiza mediante sustracción.

Por cada ejecución de la función `fact` que no lleva a un caso trivial se genera una única llamada recursiva que reduce el problema en una unidad, mientras que por cada ejecución de la función `fib` que no lleva a un caso trivial se generan dos llamadas recursivas que reducen el problema en una y dos unidades respectivamente.

Llamando a al número de llamadas recursivas que se generan por cada ejecución de las funciones y b a la reducción de tamaño que realizan dichas llamadas recursivas, tenemos que en el caso de la función `fact` tenemos que $a = 1$ y $b = 1$, mientras que para la función `fib` tenemos que $a = 2$ y $b = 1$, ya que como estamos calculando el coste asintótico temporal **en el caso peor** debemos escoger la menor reducción de las dos.

En ambas funciones, el coste del caso base es constante respecto al tamaño del problema, ya que la comprobación de si el tamaño del problema conduce al caso trivial no depende del valor de n y las dos funciones devuelven 1 como resultado del caso trivial. El coste de todas esas operaciones es constante respecto al valor de n.

En cuanto a las operaciones no recursivas realizadas en cada llamada, también son de coste constante con respecto al tamaño del problema, ya que sólo involucran la comprobación de si n conduce o no a un caso trivial (comprobación que será falsa en este caso) y operaciones como sumas y multiplicaciones. Todo ello con un coste constante respecto al valor de n.

Así pues, aplicando las reglas prácticas para el cálculo del orden al que pertenece cada función obtenemos que `fact` $\in \mathcal{O}(n)$, mientras que `fib` $\in \mathcal{O}(2^n)$. Podemos afirmar que esta última implementación no es adecuada, dado que es *innecesariamente ineficiente*.

Se pueden encontrar más explicaciones sobre la ineficiencia de la implementación de `fib` en el video sobre recursividad preparado por el equipo docente y disponible en el entorno virtual de la asignatura.

Errores Comunes: En primer lugar y como suele suceder en este tipo de ejercicio, no detallar el tamaño del problema. Este error invalida cualquier respuesta subsiguiente.

Además, en caso de tener que determinar el coste de un algoritmo, procede realizar un cálculo preciso, no una descripción vaga. Si bien esto no presupone necesariamente la aplicación de una regla de cálculo, esta forma garantiza la corrección y la precisión. De otra manera, puede incurrirse en multitud de imprecisiones que marran la solución.

Por último, si el algoritmo es recursivo, calcular el número de *vueltas* (léase, iteraciones) que realiza el código del método equivale a declarar que no se conoce la materia preguntada. Nótese que estos errores son comunes a los ejercicios 3c y 3d, por lo que obviaremos su mención en ellos para ahorrar redundancia innecesaria.

---

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.

   Crearemos un método **`boolean`** `elemGT(`**`int`** `k)` que realizará la tarea pedida: comprobar si el árbol binario de enteros llamante tiene, o no, un elemento mayor que k.

   Para ello, en primer lugar comprobaremos si el árbol es vacío, en cuyo caso devolveremos **`false`**. En caso contrario, comprobaremos si la raíz del árbol es mayor que k, devolviendo **`true`** en tal caso.

   Si la raíz del árbol no es mayor que k tendremos que realizar una búsqueda recursiva en los dos hijos del árbol (siempre que éstos existan) y devolver el resultado de dicha búsqueda:

```
public boolean elemGT(int k) {
  if (isEmpty()) return false;
  if (getRoot() > k) return true;
  return getLeftChild().elemGT(k) || getRightChild().elemGT(k);
```

```
}
```

Nótese que si cualquiera de los hijos fuese vacío, en la siguiente llamada se devolvería **`false`**.

Para calcular su coste asintótico temporal es necesario, en primer lugar, definir cuál es el tamaño del problema. Dado que estamos trabajando con un árbol (binario) podemos fijar el tamaño del problema al número n de nodos que tenga el árbol. En segundo lugar, se puede ver que el método `elemGT` se ha implementado de forma recursiva y que la reducción del problema se realiza mediante división, ya que cada ejecución del método genera llamadas recursivas sobre árboles con un número de nodos en el orden de $n \underline{\mathbf{div}}\, 2$. Por lo tanto, $b = 2$.

En el caso peor se explorará todo el árbol y siempre se realizarán las dos llamadas recursivas, por lo que $a = 2$.

En cuanto al coste asintótico temporal en el caso peor de los casos triviales, se puede ver fácilmente que es constante con respecto al número de nodos que contenga el árbol llamante, ya que sólo involucra la comprobación de si el árbol es vacío y la obtención de la raíz del mismo en caso contrario, además de devolver expresiones booleanas elementales.

Por último, el coste asintótico temporal en el caso peor de las operaciones no recursivas también es constante con respecto al número de nodos del árbol, ya que consisten en obtener los hijos izquierdo y derecho y comprobar si son o no vacíos, operaciones que pueden realizarse en un tiempo constante independientemente del número de nodos que tengan ambos hijos.

Por lo tanto, aplicando las reglas prácticas para el cálculo del coste de programas recursivos con una reducción mediante división, podemos ver que el coste de `elemGT` está en $\mathcal{O}(n^{log_2 2})$, o sea $\mathcal{O}(n)$. Es decir: en el caso peor se deberán explorar todos los nodos del árbol binario.

La relación con el coste asintótico temporal de la función `fib` del apartado anterior, está en que en ambos casos se está realizando una recursividad múltiple, con la diferencia de que en el caso de `fib`, la reducción del problema mediante sustracción lleva a un coste exponencial, mientras que en el caso de `elemGT`, la reducción del problema mediante división lleva a un coste lineal.

Si se hubiera elegido como tamaño del problema el número k de niveles del árbol binario, el coste de `elemGT` sería $\mathcal{O}(2^k)$, ya que cada llamada recursiva reduciría en 1 el número de niveles de los árboles a explorar (en consecuencia $b = 1$). Tendríamos, así, el mismo coste que en el caso de `fib`.

Dicho de otra forma, la diferencia entre `fib` y `elemGT` radica en que, en el primer caso, la ejecución de un problema de tamaño n da lugar (innecesariamente) a un árbol de llamadas recursivas de tamaño exponencial respecto a n en el primer caso. Mientras que en `elemGT` el árbol con los elementos por inspeccionar es la entrada del problema (y por tanto su tamaño es el tamaño n del problema), y en el peor de los casos hay que recorrerlo entero (y, por tanto, el coste es lineal con respecto a n, dado que la visita a cada elemento no depende de dicho tamaño).

Errores Comunes: El error más habitual en este ejercicio ha consistido en la suposición de que el árbol binario sobre el que había que buscar el elemento era un árbol de búsqueda binaria (equilibrado o AVL en muchos casos). Esto conlleva una simplificación del problema por lo que no es aceptable (implica la anulación de todo el ejercicio: el enunciado dice "árbol binario", no "árbol de búsqueda binaria").

Otros errores típicos han consistido en no comprobar la vacuidad (o el carácter de *hoja* del árbol donde procediese, incluir un parámetro de tipo árbol en el método (que se define cual función en un lenguaje *no orientado a objetos*) en lugar de utilizar el árbol llamante a tal efecto o, por último, en utilizar métodos no presentes en las interfaces del curso o con parámetros distintos a los de los perfiles de las mismas.

---

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<int> 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<int> sort ();

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

Nota: En el enunciado del examen y por un error tipográfico aparecía `ListIF` en lugar de `ListIF<int>`.

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 $\mathcal{O}(1)$. La exigencia de que el método `size` tenga un coste asintótico temporal constante implica que dicho método no puede realizar ningún tipo de cálculo, es decir, no puede contar cuántos elementos hay cada vez que es invocado. Para ello lo mejor es que exista un atributo: **int** `numElems;` que se inicialice a 0 en el constructor de la clase y que el código del método `size()` sea:

```
public int size() {
  return numElems;
}
```

Esto requiere que el atributo `numElems` sea actualizado (incrementando su valor en 1) cada vez que se añada un nuevo número a la colección, tarea que realiza el método `addNumber`.

Bajo el supuesto (A) sabemos a priori la lista de posibles valores que toman los enteros de la colección.

Hay dos opciones principales: una primera, con la que trabajaremos en esta solución, consiste en que el constructor reciba la lista de valores. Esta lista podría venir ordenada ascendentemente (impuesta esta restricción en la precondición) o ser responsabilidad del constructor ordenarla de esta manera (no se solicita el constructor, por lo que el enunciado no requiere que se diseñe dicha ordenación, que escapa del ámbito de esta solución). Otra opción sería que el constructor no reciba ningún parámetro y que las inserciones se realicen manteniendo el orden de la lista por valores de forma ascendente. En todo caso, la lista debe almacenarse en un atributo que declararemos `ListIF<int> values;` y mantenerse siempre en orden para su consulta. El código del método `getValues()` quedaría como sigue:

```
public ListIF<int> getValues() {
  return values;
}
```

Por simplicidad se devuelve `values`, aunque lo correcto sería devolver una copia de la lista para evitar que ésta sea modificada desde el exterior.

Para implementar los métodos restantes es necesario detallar la estructura de datos interna donde se van a almacenar los enteros de la colección.

Bajo el supuesto (A) se conocen a priori los posibles valores de los elementos de la colección, pero no se establece ningún tipo de restricción sobre ellos, por lo que pueden ser negativos o no consecutivos. Esto descarta la posibilidad de usar directamente un `Array` de enteros cuyos índices sean esos valores[1].

Una posible opción es utilizar una lista cuyos elementos sean pares `<valor,frecuencia>`, consideremos una clase interna:

```
private class element {

  int value;
  int freq;

  // Constructor de un nuevo par, inicializa frecuencia a 0
  public element(int num) {
    value = num;
    freq = 0;
  }
  ...
}
```

[1]La situación sería diferente, y más sencilla, si supiéramos que todos los números son enteros positivos por debajo de un cierto número máximo m. En ese caso se podría inicializar un vector de tamaño m y utilizar la posición i-ésima para guardar la frecuencia con que aparece el número i en la colección. En ese caso añadir un elemento nuevo i supondría aumentar en uno la posición i-ésima del vector (coste constante), y consultar la frecuencia de un número i supondría devolver la posición i-ésima del vector (coste constante). Puede comprobarse que el coste de sort sería lineal con m (hay que recorrer todo el vector para generar la secuencia de números) y el tamaño de la colección de números (hay que generar toda la secuencia de repeticiones a partir del vector).

Una solución intermedia sería que en el constructor se prepare una asignación previa entre los (n) enteros del problema y los n primeros naturales. En ese caso, cada vez que se añadiera un número se llamaría a una función de mapeado `private int map(int num)` y a partir de ahí se trabajaría en el espacio de los naturales entre 1 y n, con lo que se podrían almacenar las frecuencias de cada número en un vector (`frequencies`) de tamaño n y acceder a cada una de ellas en tiempo constante. De esta forma, añadir un número nuevo *num* consistiría en aumentar en uno el valor de la casilla `frequencies(map(num))`. El coste de modificar una posición del vector sería ahora constante, pero el coste total dependería ahora de la operación de asignación `map(num)`.

y asumamos la existencia de getters y setters para ambos atributos.

Dado que el constructor de la clase `PCollectionA` conoce la lista de posibles valores que toman los enteros, aparte de inicializar el atributo `values` con dicha lista ordenada crecientemente, también puede inicializar un atributo:

`ListIF<element> elements;`

de forma que dicho atributo contenga una lista de pares `<valor,frecuencia>` ordenados crecientemente por valor y con todas las frecuencias inicializadas a 0.

Sobre esa lista de pares se deben realizar dos operaciones: una de modificación (`addNumber(int num)`) y otra de consulta (`frequency(int num)`). Ambos reciben un valor entero y deben, respectivamente, incrementar en 1 o devolver la frecuencia asociada a dicho valor.

Así pues, parece lógico que exista un método que realice la búsqueda dentro de la lista de elementos de un par cuyo valor coincida con uno dado. Por lo tanto:

```
private element getElement(int num) {
  int pos = 1;
  while (pos <= elements.size() ) {
    element myElement = elements.get(pos);
    if (myElement.getValue() == num ) return myElement;
    if (myElement.getValue() > num ) return null;
    pos++;
  }
  return null;
}
```

Este método examina la lista `elements` y devuelve un objeto de la clase `element` cuyo atributo `value` coincide con el parámetro recibido. Si en la lista no existiese un par con esas características, devolverá `null`. Además, dado que la lista de pares está ordenada crecientemente por valores, cuando un par contiene un valor superior al número buscado, ya sabemos que no está en la lista y devolvemos, directamente, `null`.

De esta forma, el código de los métodos `addNumber(int num)` y `frequency(int num)` quedaría como sigue:

```
public void addNumber(int num) {
  element myElement = getElement(num);
  myElement.setFreq(myElement.getFreq()++);
  numElems++
}

public int frequency(int num) {
  element myElement = getElement(num);
  if (myElement == null ) return 0;
  return myElement.getFreq();
}
```

Nótese que, al estar bajo el supuesto (A), para todos los números recibidos por `addNumber` se garantiza la existencia de un par en la lista `elements`, por lo que nunca se recibiría un `null` tras la llamada a `getElement`. Esto no es cierto para el método `frequency`, que devuelve un 0 si el número consultado no está en la colección.

Además, el método `addNumber` se encarga de incrementar en 1 el número total de elementos de la colección para garantizar que el atributo `numElems` siempre es correcto cuando es consultado por `size()`.

Sólo nos queda el método `sort()`, que debe devolver todos los elementos de la colección ordenados de menor a mayor. Dado que en la lista `elements` ya tenemos los elementos en ese orden y sabemos la frecuencia de cada uno de ellos, basta con recorrer dicha lista e ir añadiendo a la lista de salida tantos elementos de cada valor como indique su número de repeticiones.

```
public ListIF<int> sort() {
  ListIF<int> output = new ListIF<int>;
  int pos = 1;
  while (pos <= elements.size() ) {
    element myElement = elements.get(pos);
    int v = myElement.getValue();
    int f = myElement.getFreq();
    for (int aux = 1 ; aux <= f ; aux++ ) {
      output.insert(v,output.size()+1);
    }
    pos++
  }
  return(output);
}
```

Los recorridos que se han utilizado en este apartado (`getElement` y `sort`) son las versiones simples de uso de las listas, que se han introducido para ilustrar los sobrecostes en los que incurre esta estructura. Serían mejorables sin más que utilizar iteradores.

Nótese que existe otra opción de representación interesante, consistente en almacenar los objetos `element` en un Árbol AVL ordenados por valor. La consulta a esta estructura sería más eficiente (lo veremos en el apartado correspondiente) para valores puntuales. La devolución de la lista de valores requeriría un recorrido completo del árbol. La ordenación de dicha estructura se lleva a cabo por construcción y consiste en insertar en su lugar cada elemento, es decir, buscar su posición y reequilibrar el AVL si procede. Sería un ejercicio interesante reescribir la solución utilizando este tipo de representación.

Errores Comunes: En este tipo de ejercicio, es imprescindible comentar o complementar el código que se incluye con una descripción mínima siquiera de lo que se está intentando llevar a cabo. Nótese que no se solicitaba la inclusión del constructor de la clase, por lo que, cualquier responsabilidad que recayese sobre éste y que influyese en cualquier otro método debería haber sido explicada previamente a la exposición del código de ese segundo método.

Además, un error que ha aparecido recurrentemente consiste en suponer que los números, que en este apartado se decían conocidos de antemano, eran positivos y contiguos (veáse la nota al pie de la página 5). Nada de ello se decía en el enunciado ni se desprendía del mismo, por lo que supone una simplificación inaceptable e implica un resultado que no satisface las condiciones solicitadas.

---

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`.

La implementación del método `size` para la clase `PCollectionA` es válida para la clase `PCollectionB` siempre y cuando se mantenga un atributo `numElems` que sea actualizado de manera conveniente por el método `addNumber`.

Bajo el supuesto (B) no se conoce a priori la lista de valores posibles, por lo que el constructor de la clase no puede recibir esa lista y, por tanto, no puede almacenarla para que sea utilizada por el método `getValues()`.

Por el mismo motivo, tampoco es posible crear a priori la lista de objetos de la clase `element` en el atributo `elements` para almacenar los valores de la colección, por lo que su construcción y mantenimiento deberá delegarse en el método `addNumber`, quien deberá buscar el par dentro de la lista (para lo cual puede utilizarse el método `getElement` del apartado anterior) e incrementar la frecuencia del número si ya existía o añadir un nuevo par (manteniendo el orden) con frecuencia 1 si no existía previamente.

Una posibilidad es que el método `addNumber` también mantenga un atributo `values` con la lista de valores, lo cual implica que si no existía el par con el elemento, aparte de actualizar la

lista contenida en `elements` también debería insertar el valor en la lista `values`. En este caso, el código de `getValues()` sería idéntico al del apartado anterior.

Otra posibilidad es que no exista ese atributo, para lo cual el método `getValues()` debería recorrer la lista `elements` y devolver una lista con los valores de cada par:

```
public ListIF<int> getValues() {
  ListIF<int> output = new ListIF<int>;
  int pos = 1;
  while (pos <= elements.size() ) {
    output.insert(elements.get(pos).getValue(),output.size());
    pos++
  }
  return(output);
}
```

En cuanto al método `frequency`, la implementación dada para el supuesto (A) es también válida para el supuesto (B).

Errores Comunes: En este apartado, el error más habitual ha consistido en no marcar las diferencias (o justificar la similitud) con la solución del ejercicio anterior o dejar de incluir información, a veces, en la forma de metodos necesarios como `addNumber` o, cuando menos, una descripción de las modificaciones que éste requería para el nuevo caso de valores no conocidos de antemano.

---

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

En ambos supuestos la implementación del método `frequency` es idéntica, por lo que el razonamiento sobre su coste no va a depender del supuesto bajo el que estemos.

En primer lugar vamos a fijar cuál es el tamaño del problema. En un principio puede parecer lógico establecer este valor al número n de elementos añadidos a la colección. Sin embargo en este caso es más razonable medir el coste con respecto al número k de diferentes valores que pueden tomar los números de la colección.

Según el código del método `frequency`

```
1  public int frequency(int num) {
2    element myElement = getElement(num);
3    if (myElement == null ) return 0;
4    return myElement.getFreq();
5  }
```

vemos que las instrucciones de las líneas 3 y 4 tienen un coste constante con respecto a k, por lo que el coste del método será el coste de la llamada a `getElement(num)`.

En el peor de los casos, el método `getElement(num)` provocará que se consulten todos los elementos de la lista `elements`. Dado que el acceso a cada uno de ellos es (en el caso peor) lineal con respecto al tamaño de la lista (que es, precisamente, el tamaño del problema), tenemos que el coste de este método está en $\mathcal{O}(k^2)$.

Por tanto, el coste del método `frequency` es cuadrático con respecto al número de valores diferentes almacenados en la colección.

Si `getElement` utilizase un iterador, este coste podría reducirse en un factor lineal (quedaría como $\mathcal{O}(k)$).

Nótese además que este coste podría reducirse utilizando la opción de representar los valores en un AVL, lo que dejaría el anterior cálculo en $\mathcal{O}(\log k)$.

Errores Comunes: Tanto en este apartado como en el siguiente y excluyendo los errores ya comentados en el ejercicio 1 en la página 2, ha sido muy común que los costes no se calculasen sino que se

describiesen informalmente, ítem más, dando por hecho que un recorrido implica *necesariamente* un coste lineal y, habitualmente, olvidando incluir y reseñar el coste que añaden las operaciones de acceso a una lista por posición en la interfaz `ListIF`.

---

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.

En ambos supuestos el método `sort` es idéntico, por lo que el cálculo de su coste no depende del supuesto en el que estemos.

Dado que este método devuelve una lista con todos los números añadidos a la colección, es lógico que se mida su comportamiento asintótico con respecto a este valor. Por lo tanto, definimos el tamaño del problema a n, que representa el número total de elementos en la colección (recordemos que es el valor del atributo `numElems`).

Volviendo al código del método `sort`:

```
1   public ListIF<int> sort() {
2     ListIF<int> output = new ListIF<int>;
3     int pos = 1;
4     while (pos <= elements.size()) {
5       element myElement = elements.get(pos);
6       int v = myElement.getValue();
7       int f = myElement.getFreq();
8       for (int aux = 1 ; aux <= f ; aux++) {
9         output.insert(v,output.size()+1);
10      }
11      pos++
12    }
13    return(output);
14  }
```

vemos que en la línea 9 se añade un elemento a la lista de salida por cada elemento que existe en la colección, por lo que cabría esperar que el coste del método estuviera en $\mathcal{O}(n)$.

Sin embargo, hay que puntualizar esto, ya que vemos que en la línea 5 se efectúa una llamada al método `get` para recuperar un elemento de la lista de pares. Dado que el coste del método `get` depende linealmente del tamaño de esa lista, es necesario tener en cuenta ese valor (al que habíamos llamado k en el apartado anterior) para calcular el coste.

Sin embargo, el supuesto principal del problema es que la colección tiene un número reducido de diferentes valores y muchas repeticiones de éstos. Esto significa que n es mucho mayor que k, por lo que el coste de recuperar un elemento de la lista de pares puede considerarse despreciable con respecto al número de números almacenados en la colección.

Así pues, podemos concluir que el coste del método `sort` está en $\mathcal{O}(n)$, es decir, es lineal con respecto al número de números presentes en la colección.

Nótese que ni la representación en AVL ni el uso de iteradores reducirían este coste, ya que hay que devolver todos los elementos.

Errores Comunes: El error más habitual, más allá de los citados anteriormente, en este ejercicio ha sido el suponer costes tanto para la ordenación en general como para el método que se hubiera implementado a tal efecto sin tener en consideración ni el tamaño del problema ni las particularidades de la solución, como se comentaba en los errores del anterior apartado.
