Algoritmos y Estructuras de Datos: Examen 1 Rec (Solución)

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Grado en Ingeniería Informática, Grado en Matemáticas e Informática y
Doble Grado en Informática y Administración y Dirección de Empresas

- Este examen dura **100 minutos** y consta de **4 preguntas** que puntúan hasta **10 puntos**.
- **Las preguntas 1 y 2 deben contestarse en la misma hoja.**
- **Las preguntas 3 y 4 deben contestarse en la misma hoja (distinta de la hoja de las preguntas 1 y 2).**
- Todas las hojas entregadas deben indicar, en la parte superior de la hoja, **apellidos**, **nombre**, **DNI/NIE** y **número de matrícula**.
- Las calificaciones provisionales de este examen se publicarán el **25 de Enero de 2024** en el Moodle de la asignatura junto con la fecha de la revisión.
- **NOTA:** Recordad que el uso de **métodos auxiliares SÍ** está permitido y que **NO está permitido modificar las estructuras de datos recibidas como parámetro, salvo que el enunciado indique lo contrario**.

(3½ puntos) 1. **Se pide:** Implementar en Java el método:

```
public static Pair<PositionList<Integer>, PositionList<Integer>>
  split(int splitter, PositionList<Integer> l)
```

que recibe como parámetros una lista `l` y un número `splitter` y debe devolver dos listas, una que contiene los elementos de `l` que son $<$ que `splitter` y otra que contiene los elementos de `l` que son $\geq$ que `splitter`. Los elementos en las dos listas deberian estar ordenados en el mismo orden que tenían en `l`. La lista de entrada `l` podría ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. Los elementos contenidos en `l` podrían ser **`null`** en cuyo caso no se deben incluir en ninguna de las listas. Para devolver las listas debéis crear un objeto de tipo `Pair`. La clase `Pair` dispone del siguiente constructor `Pair(Object o1, Object o2)` que permite crear un `Par` que contiene `o1` y `o2`. La clase `NodePositionList` implementa el interfaz `PositionList` y dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, obtendríamos los siguientes resultados:

`split(3,[1,2,null,7,1,3])` debe devolver el par `([1,2,1],[7,3])` o la llamada `split(8,[1,2,7,1,3, null])` debe devolver el par `([1,2,1,7,3],[])`.

Solución:

```
static Pair<PositionList<Integer>,PositionList<Integer>>
  split(int splitter, PositionList<Integer> l) {
    if (l == null) throw new IllegalArgumentException();
    PositionList<Integer> lt = new NodePositionList<>();
    PositionList<Integer> geq = new NodePositionList<>();
    Pair<PositionList<Integer>,PositionList<Integer>> pair =
      new Pair<>(lt,geq);
    for (Integer i : l) {
      if (i != null) {
        if (i < splitter) lt.addLast(i);
        else geq.addLast(i);
      }
    }
    return pair;
  }
```

(2½ puntos) 2. **Se pide:** implementar la clase

```
public class Sequence implements Iterator<Integer> {
  public Sequence (int min, int max) {...}
  ...
  // Implementar el resto de  métodos
  ...
}
```

La clase `Sequence` debe implementar un iterador que recibe en el constructor dos parámetros, `min` y `max` y debe devolver todos los números contenidos entre `min` y `max` en orden ascendiente, ambos incluídos. Cuando ya no queden elementos por devolver, el método `hasNext` devolverá **`false`** y el método `next` lanzará la excepción `NoSuchElementException`.

Por ejemplo, el iterador `Sequence(1,5)`, debe devolver en las sucesivas llamadas a `next`, los números `1,2,3,4 y 5`; el iterador `Sequence(-1,6)` debe devolver los elementos `-1,0,1,2,3,4,5 y 6`; y el iterador `Sequence(7,2)` no devolverá ningún elemento.

Notad de que es permitido añadir atributos nuevos a la clase `Sequence`, siempre cuando sus tipos son primitivos (por ejemplo, un **`boolean`**, **`char`** o un **`int`**) y no de una clase, array o interfaz. Por ejemplo, no esta permitido definir un atributo de tipo `PositionList` o `NodePositionList`, etc.

Solución:

```
public class Sequence implements Iterator<Integer> {
  private int curr;
  private int max;
  public Sequence (int min, int max) {
    this.curr = min;
    this.max = max;
  }
  public boolean hasNext() {
    return curr <= max;
  }
  public Integer next() {
    if (curr > max) throw new NoSuchElementException();
    else {
      int result = curr;
      curr = curr+1;
      return result;
    }
  }
}
```

(3 puntos) 3. **Se pide:** Implementar la clase

```
public class RegistroContaminacionAED implements RegistroContaminacion {
  private Map<String,Integer> diasContaminados;
  private int limite;
  ...
  public RegistroContaminacionAED (int limite) {
    // Implementar el constructor
  }
  ...
  // Implementar el resto de métodos
}
```

dado el interfaz:

```
public interface RegistroContaminacion {
  // Recibe el lugar y el valor de la contaminación registrado para un día
  public void registrarDia(String lugar, int valorContaminacion);
```

```
    // Devuelve el número de días donde la contaminación sobrepasa el límite
    // Si el lugar no dispone de datos el método debe lanzar la
    // excepción LugarNoRegistradoException
    public int diasQueSobrepasaLimite(String lugar)
                                              throws LugarNoRegistradoException;
}
```

Los valores del parametro `lugar` (en ambos métodos) nunca van a ser **`null`**.

Por ejemplo, al final de la siguiente secuencia de llamadas:

```
RegistroContanimacion reg = new RegistroContaminacionAED(10);
reg.registrarDia("Madrid", 5);
reg.registrarDia("Barcelona", 12);
reg.registrarDia("Madrid", 13);
reg.registrarDia("Barcelona", 4);
reg.registrarDia("Madrid", 16);
reg.registrarDia("Barcelona", 16);
reg.registrarDia("Madrid", 11);
reg.registrarDia("Barcelona", 10);
```

La llamada `reg.diasQueSobrepasaLimite("Madrid")` debe devolver 3, la llamada `reg.diasQueSobrepasaLimite("Barcelona")` debe devolver 2, o la llamada `reg.diasQueSobrepasaLimite("Alicante")` lanzará la `LugarNoRegistradoException`.

Se dispone de la clase `HashTableMap<K,V>` con un constructor sin parámetros que crea un `Map` vacío.

IMPORTANTE: NO está permitido añadir atributos nuevos a la clase `RegistroContaminacionAED`.

Solución:

```
public class RegistroContaminacionAED implements RegistroContanimacion {
  private int limite;
  private Map<String,Integer> diasContaminados;

  public RegistroContaminacionAED (int limite) {
    this.limite = limite;
    this.diasContaminados = new HashTableMap<>();
  }

  public void registrarDia(String lugar, int valorContaminacion) {
    if (valorContaminacion > limite) {
      Integer dias = diasContaminados.get(lugar);
      dias = (dias == null) ? 1 : dias+1;
      diasContaminados.put(lugar,dias);
    }
  }

  public int diasQueSobrepasaLimite(String lugar) throws LugarNoRegistradoExceptio
    Integer dias = diasContaminados.get(lugar);
    if (dias == null) throw new LugarNoRegistradoException();
    else return dias;
  }
}
```

(1 punto) 4. **Se pide:** Indicar la complejidad computacional de los métodos `method1` y `method2` en función de la longitud de la lista de entrada (n):

```java
<E> void method1 (PositionList<E> l) {
  Position<E> c = l.first();
  while (c != null) {
    maux(l);
    c = l.next(c);
  }
}

<E> void maux(PositionList<E> l){
  Position<E> c = l.first();
  while (c != null) {
    Position<E> c2 = l.first();
    while (c2 != null) {
      c2 = l.next(c2);
    }
    c = l.next(c);
  }
}
```

```java
<E> void method2 (PositionList<E> l) {
  Position<E> c = l.first();
  while (c != null) {
    c = l.next(c);
  }
  Position<E> c2 = l.first();
  while (c2 != null) {
    c2 = l.next(c2);
  }
}
```

Solución:

- El método `method1` tiene complejidad $O(n^3)$
- El método `method2` tiene complejidad $O(n)$