Algoritmos y Estructuras de Datos: Examen 1 (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 **6 de Julio de 2023** 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 ninguna de las estructuras de datos recibidas como parámetro, salvo que el enunciado indique lo contrario.

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

```
<E extends Comparable<E>> PositionList<E> getMaximosRelativos (PositionList<E> list) {
```

que recibe como parámetro una lista de números y debe devolver una nueva lista con los *máximos relativos* encontrados en la misma. Un elemento e_i de la lista, que ocupa la posición i, será un máximo relativo cuando éste sea mayor o igual que sus dos elementos vecinos, es decir, se cumpla la condición $e_{i-1} \leq e_i \leq e_{i+1}$. Por simplicidad, el primer y el último elemento de la lista no se incluirán en la lista resultado. Podría darse el caso de que no haya máximos relativos en una lista. Recordad que `<E extends Comparable<E>>` indica que el genérico `E` siempre implementará el interfaz `Comparable<E>`.

La lista de entrada nunca podrá ser **`null`**, pero si la lista tiene menos de tres elementos el método debe lanzar la excepción `IllegalArgumentException`. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, `getMaximosRelativos([1,2,4,2,3,1])` deberá devolver la lista `[4,3]`,
`getMaximosRelativos([3,3,2,4,5,2,7])` deberá devolver `[3,5]`,
`getMaximosRelativos([1,2,3,4,5])` deberá devolver `[]`,
`getMaximosRelativos([1,2,1,2,1])` deberá devolver `[2,2]` y
`getMaximosRelativos([1,2])` deberá lanzar la excepción `IllegalArgumentException`

Solución:

```
<E extends Comparable<E>> PositionList<E> getMaximosRelativos (PositionList<E> list) {

  PositionList<E> res = new NodePositionList<>();

  if (list.size() < 3) {
    throw new IllegalArgumentException();
  }

  Position<E> anterior = list.first();
  Position<E> cursor = list.next(anterior);
  Position<E> siguiente = list.next(cursor);
  while (siguiente != null) {
    if (cursor.element().compareTo(anterior.element()) >= 0 &&
    cursor.element().compareTo(siguiente.element()) >= 0 ) {
      res.addLast(cursor.element());
    }
    anterior = cursor;
    cursor = siguiente;
    siguiente = list.next(cursor);
  }
```

```
  return res;
}
```

(3 puntos) 2. **Se pide:** implementar el método

```
Iterable<Integer> getMaximosRelativosIterador (Iterable<Integer> list)
```

que tiene la misma especificación que el método del ejercicio anterior, con una diferencia, que tanto el parámetro como el resultado devuelto son de tipo `Iterable<Integer>`. El `Iterable` nunca será **`null`** y siempre devolverá al menos 3 elementos. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Solución:

```
public static Iterable<Integer> getMaximosRelativosIterador (Iterable<Integer> list) {
  Iterator<Integer> it = list.iterator();
  PositionList<Integer> res = new NodePositionList<>();

  Integer anterior = it.next();
  Integer actual = it.next();
  Integer siguiente = null;

  while (it.hasNext()) {
    siguiente = it.next();
    if (actual >= anterior && actual >= siguiente) {
      res.addLast(actual);
    }
    anterior = actual;
    actual = siguiente;
  }
  return res;
}
```

(3 puntos) 3. **Se pide:** Implementar la clase `RegistroCochesMap` con la siguiente cabecera:

```
public class  RegistroCochesMap implements RegistroCoches {...}
```

El interfaz `RegistroCoches` es el siguiente:

```
interface RegistroCoches {
  // Añade un nuevo Coche al registro con la matricula y los datos del coche.
  // Devuelve false si ya existia un coche con dicha matricula (y no se añade),
  // y true en caso de haber añadido el coche
  public boolean add(String matricula, String bastidor, String marca, String modelo);

  // Borra un coche del registro; devuelve el Coche borrado si existía, y
  // null si no había ningún coche registrado con la matrícula
  public Coche remove(String matricula);

  // Cambia los datos del coche asociado a la matricula; si el coche no existía
  // la llamada no hace nada. Si existia, devuelve el Coche que estaba registrado
  // con la matrícula recibida
  public Coche update(String matricula, Coche coche);

  // Busca y devuelve un coche asociado con la matricula
  public Coche find(String matricula);

  // Devuelve todas las matriculas de los coches registrados
  public Iterable<String> getMatriculas ();
}
```

Asimismo, se dispone de la clase coche que se define:

```
class Coche {
  String numeroBastidor;
  String marca;
  String modelo;
  public Coche(String numeroBastidor, String marca, String modelo) {
    this.numeroBastidor = numeroBastidor;
    this.marca = marca;
    this.modelo = modelo;
  }
```

Es **obligatorio utilizar un atributo de tipo `Map`** para almacenar los coches registrados. Se dispone de la clase `HashTableMap<K,V>`, que implementa el interfaz `Map<K,V>` y que dispone de un constructor sin parámetros para crear un map vacía.

Solución:

```
class RegistroCochesMap implements RegistroCoches {

  private Map<String,Coche> coches =
    new HashTableMap<>();

  public boolean add(String matricula, String bastidor,
                     String marca, String modelo) {
    if (coches.containsKey(matricula)) {
      return false;
    }
    Coche c = new Coche(bastidor,marca,modelo);
    coches.put(marca, c);
    return true;
  }

  public Coche remove(String matricula) {
    return coches.remove(matricula);
  }
```

```
  public Coche update(String matricula, Coche coche) {
    if (coches.containsKey(matricula)) return null;
    else return coches.put(matricula, coche);
  }

  public Coche find(String matricula) {
    return coches.get(matricula);
  }

  public Iterable<String> getMatriculas() {
    return coches.keySet();
  }
}
```

(1 punto) 4. **Se pide:** Indicar la complejidad de los métodos `method1` y `method2`:

```
<E> int method1 (PositionList<E> l) {
  Position<E> c = l.first();
  int count = 0;
  while (c != null) {
    Position<E> c2 = l.first();
    while (c2 != null) {
      Position<E> c3 = l.first();
      while (c3 != null) {
        count ++;
        c3 = l.next(c3);
      }
      c2 = l.next(c2);
    }
    c = l.next(c);
  }
  return count;
}
```

```
<E> int method2 (int[] l){
  int counter = 0;
  for (int j=l.length; j>0; j=j/2){
    for (int i = 0;i < l.length;i++){
      counter ++;
    }
  }
  return counter;
}
```

Solución:

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