Algoritmos y Estructuras de Datos: Examen 1a (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 **21 de Noviembre 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 las estructuras de datos recibidas como parámetro, salvo que el enunciado indique lo contrario**.

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

```
static <E> boolean todasOrdenadas (PositionList<PositionList<E>> list,
                                   Comparator<E> cmp)
```

que recibe como parámetros una lista de listas `list` y un comparador `cmp` y debe devolver **`true`** si todas las listas contenidas en `list` están ordenadas ascendentemente según el criterio de comparación establecido por el comparador `cmp`. La lista de entrada `list`, las listas contenidas en `list`, los elementos contenidos en ellas y el comparador `cmp` serán siempre distintos de **`null`**.

Por ejemplo, asumiendo que disponemos de un comparador `IntegerCmp`, que implementa la comparación habitual de los números enteros (por ejemplo, 5 es menor que 7) obtendríamos los siguientes resultados:

`todasOrdenadas([[1,2,7],[1,3],[]], new IntegerCmp())` debe devolver **`true`**,
`todasOrdenadas([[1,2,7],[7,3],[1,2]], new IntegerCmp())` debe devolver **`false`**,
`todasOrdenadas([[1,2,7],[1,3],[1]], new IntegerCmp())` debe devolver **`true`**,
`todasOrdenadas([], new IntegerCmp())` debe devolver **`true`**,
`todasOrdenadas([[],[]], new IntegerCmp())` debe devolver **`true`**.

Solución:

```
public static <E> boolean todasOrdenadas (PositionList<PositionList<E>> list,
                                          Comparator<E> cmp){

  Position<PositionList<E>> cursor = list.first();
  while (cursor != null && estaOrdenada(cursor.element(),cmp)) {
    cursor = list.next(cursor);
  }
  return cursor == null;
}

private static <E> boolean estaOrdenada(PositionList<E> list, Comparator<E> cmp)
  if (list.size() < 2) {
    return true;
  }
  Position<E> prev = list.first();
  Position<E> cursor = list.next(prev);

  while (cursor != null && cmp.compare(prev.element(), cursor.element()) <= 0) {
    prev = cursor;
    cursor = list.next(cursor);
```

```
    }

    return cursor == null;
  }
```

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

```
static Iterable<Integer> filtrarMultiplos(Iterable<Integer> input,
                                          Integer n)
```

que recibe como parámetro un `Iterable` de `Integer` y debe devolver una nueva estructura `Iterable` que contenga los número devueltos por `input` que sean múltiplos de `n`. El iterable resultante debe devolver los elementos en el mismo orden relativo que tienen en `input`.

El iterable `input` podrá ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. De la misma forma, el iterable `input` podrá devolver elementos **`null`**, que deben ser ignorados y no incluidos en el iterable resultado. El valor de `n` podría ser ≤ 0, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. Recordad que algunas de las estructuras de datos lineales que se han usado extensivamente durante el curso implementan la interfaz `Iterable`. Por ejemplo, las clases `NodePositionList` y `ArrayIndexedList` implementan las interfaces `PositionList` e `IndexedList` respectivamente y ambas disponen de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado un `iterable` que devuelve los elementos `3,4,null,11,18`, la llamada `filtrarMultiplos(iterable,2)` debe devolver un iterable que contenga los elementos `4,18`. Si `iterable` devuelve `3,4,9,11,17`, la llamada `filtrarMultiplos(iterable,3)` debe devolver un iterable que contenga `3,9`; si `iterable` devuelve `3,7,9`, entonces `filtrarMultiplos(iterable,4)` debe devolver un iterable sin elementos; y si `iterable` devuelve **`null,null`**, entonces `filtrarMultiplos(iterable,2)` debe devolver un iterable también sin elementos.

Solución:

```
  static Iterable<Integer> filtrarMultiplos (Iterable<Integer> input,
                                             Integer n) {
    if (input == null || n <= 0) {
      throw new IllegalArgumentException();
    }

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

    for (Integer e: input) {
      if (e != null && e % n == 0) {
      res.addLast(e);
    }
  }

  return res;
}
```

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

```
static <E> void imprimirPorModulo (PositionList<Integer> list, Integer n)
```

que recibe como parámetro una lista `list` y un número `n` y debe imprimir todos los números contenidos en `list` clasificados por el resultado de la operación *%n*. El orden en el que éstos se impriman no es relevante. El valor de `n` nunca será **`null`**, pero podría ser ≤ 0, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. El parámetro `list` nunca será **`null`** ni contendrá valores **`null`**.

Por ejemplo, la llamada `imprimirPorModulo([0,2,4,3,1,4],3)` debe imprimir

```
1 -> [4,1,4]
0 -> [0,3]
2 -> [2]
```

o la llamada `imprimirPorModulo([0,2,4,1,4],4)` debe imprimir

```
2 -> [2]
1 -> [1]
0 -> [0,4,4]
```

Para ello se recomienda el uso de una variable auxiliar de tipo `Map<Integer,PositionList<Integer>>`. Se dispone de la clase `HashTableMap<K,V>` con un constructor sin parámetros que crea un `Map` vacío. Asimismo disponéis de la clase `NodePositionList`, que implementa la interfaz `PositionList` y que dispone de un constructor sin parámetros para crear una lista vacía. Recordad que la clase `NodePositionList` implementa la interfaz `PositionList` y dispone de un método `toString()` que devuelve un `String` que contiene todos los elementos contenidos en la lista para poder imprimirlos.

Solución:

```
static <E> void imprimirPorModulo (PositionList<Integer> list,
                                   Integer n) {

  if (n <= 0) {
    throw new IllegalArgumentException();
  }

  Map<Integer,PositionList<Integer>> map = new HashTableMap<>();

  for (Integer e: list) {
    Integer mod = e % n;
    PositionList<Integer> l = map.get(mod);
    if (l == null) {
      l = new NodePositionList<>();
      map.put(mod, l);
    }
    l.addLast(e);
  }

  for (Entry<Integer,PositionList<Integer>> entry: map) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
  }
}
```

(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> int method1 (PositionList<E> l) {
  Position<E> c = l.first();
  int count = 0;
  while (c != null && count < 3) {
    count ++;
    c = l.next(c);
  }
  Position<E> c2 = c;
  while (c2 != null) {
    count ++;
    c2 = l.next(c2);
  }
  return count;
}
```

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

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

Solución:

- El método `method1` tiene complejidad $O(n)$
- El método `method2` tiene complejidad $O(n^2)$