Algoritmos y Estructuras de Datos: Examen 1b (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**.

($3\frac{1}{2}$ puntos) 1. **Se pide:** Implementar en Java el método:

```
static <E> boolean hayListaEnRango (PositionList<PositionList<Integer>> list,
                                    int a, int b)
```

que recibe como parámetros una lista de listas `list` y dos números `a` y `b` y debe devolver **`true`** si existe una lista contenida en `list` en la que todos sus elementos estén en el rango `[a,b]`. Decimos que x está en el rango `[a,b]` si $a \leq x \leq b$. La lista de entrada `list`, las listas contenidas `list` y los elementos contenidos en ellas serán siempre distintos de **`null`**. Asimismo, las listas contenidas en `list` nunca serán vacías.

Con las llamadas que de incluyen a continuación obtendríamos los siguientes resultados:
`hayListaEnRango([[1,2,7],[1,3]], 3,4)` debe devolver **`false`**,
`hayListaEnRango([[1,2,7],[1,3],[1,2]], 1,4)` debe devolver **`true`**,
`hayListaEnRango([[1,2,7],[1,3],[1]], 2,3)` debe devolver **`false`**,
`hayListaEnRango([],1,2)` debe devolver **`false`**.

Solución:

```
  static <E> boolean hayListaEnRango (PositionList<PositionList<Integer>> list,
                                      int a, int b){

    Position<PositionList<Integer>> cursor = list.first();
    while (cursor != null && !todosEnRango(cursor.element(),a,b)) {
      cursor = list.next(cursor);
    }
    return cursor != null;
  }

  private static boolean todosEnRango(PositionList<Integer> list,
                                      int a, int b) {
    Position<Integer> cursor = list.first();

    while (cursor != null && cursor.element() >= a && cursor.element() <= b) {
      cursor = list.next(cursor);
    }
    return cursor == null;
  }
}
```

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

`Iterable<Integer> filtrarEnRango(Iterable<Integer> iterable, Integer a, Integer b)`

que recibe como parámetro un `Iterable` de `Integer` y debe devolver un nuevo `Iterable` que contenga los números devueltos por `iterable` estén dentro del rango `[a,b]`.

El `iterable` podrá ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. De la misma forma, el `iterable` podrá devolver elementos **`null`**, que deben ser ignorados y no incluidos en el iterable resultado. 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 `filtrarEnRango(iterable,1,3)` debe devolver un iterable que contenga únicamente el elemento `3`; si `iterable` devuelve `3,4,9,11,17`, la llamada `filtrarEnRango(iterable,3,7)` debe devolver un iterable que contenga `3,4`; si `iterable` devuelve `3,7,9`, entonces `filtrarEnRango(iterable,10,12)` debe devolver un iterable sin elementos; y si `iterable` devuelve **`null,null`**, entonces `filtrarEnRango(iterable,2)` debe devolver un iterable sin elementos.

Solución:

```
boolean filtrarEnRango(Iterable<Integer> iterable, Integer a, Integer b)
  if (iterable == null) {
    throw new IllegalArgumentException();
  }

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

  for (Integer e: iterable) {
    if (e != null && e >= a && e <= b) {
      res.addLast(e);
    }
  }

  return res;
}
```

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

```
static void imprimirPorSuma (PositionList<PositionList<Integer>> list)
```

que recibe como parámetro `list` una lista de listas que contienen `Integer` y debe imprimir todas las listas contenidas en `list` clasificadas por la suma de sus elementos. El orden en el que éstas se impriman no es relevante.

Por ejemplo, la llamada `imprimirPorSuma([[1,2,3],[1,3],[6],[4,5],[],[1,-1]])` debe imprimir:

```
6 -> [[1,2,3], [6]]
4 -> [[1,3]]
9 -> [[4,5]]
0 -> [[],[1,-1]]
```

Para ello se recomienda el uso de una variable auxiliar de tipo `Map<Integer,PositionList<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 interfaz `PositionList` dispone de un método `toString` que devuelve un `String` que contiene todos los elementos contenidos en la lista para poder imprimirlos. El parámetro `list` nunca será **`null`** ni contendrá valores **`null`**.

Solución:

```
static void imprimirPorSuma (PositionList<PositionList<Integer>> list) {
  Map<Integer,PositionList<PositionList<Integer>>> map = new HashTableMap<>();

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

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


private static Integer suma(PositionList<Integer> list) {
  int suma = 0;
  for (Integer e: list) {
    suma += e;
  }
  return suma;
}
```

(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:

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

```java
<E> void method2 (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) {
    maux2(l)
    c = l.next(c);
  }
}
<E> void maux2(PositionList<E> l){
  Position<E> c = l.first();
  while (c != null) {
    c = l.next(c);
  }
}
```

Solución:

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