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 **22 de Noviembre 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**.

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

```
static <E> PositionList<Boolean> estaRepetidoEnListas
                                        (PositionList<PositionList<E>> listas,
                                         E element)
```

que recibe como parámetros una lista de listas `listas` y un elemento `element`, y debe devolver una lista de *Boolean*, del mismo tamaño que `listas`, en la que la posición *i* de la lista resultado contiene **`true`** si la lista que ocupa la posición *i* contiene `element` al menos dos veces y **`false`** en caso contrario.

La lista de entrada `listas` y las listas contenidas en `listas` nunca serán **`null`**. Por el contrario, tanto `element`, como los elementos contenidos en las listas, podrán ser **`null`** y el método debe tratar los elementos **`null`** como un valor más.

Por ejemplo, considerando `listas = [[1,2,7,1],[7,3],[1,null,null,1]]` dadas las siguientes llamadas obtendríamos los siguientes resultados:

`estaRepetidoEnListas(listas, 2)` debe devolver `[false,false,false]`,
`estaRepetidoEnListas(listas, 1)` debe devolver `[true,false,true]`,
`estaRepetidoEnListas(listas, null)` debe devolver `[false,false,true]`,
`estaRepetidoEnListas([[],[]], 3)` debe devolver `[false, false]`.

Solución:

```
public static <E> PositionList<Boolean> estaRepetidoEnListas (PositionList<Positi
  PositionList<Boolean> res = new NodePositionList<>();

  Position<PositionList<E>> cursor = listas.first();

  while (cursor!= null) {
    res.addLast(estaRepetido(cursor.element(), element));
    cursor = listas.next(cursor);
  }

  return res;
}

private static <E> boolean estaRepetido(PositionList<E> list, E element) {

  Position<E> cursor = list.first();
  int counter = 0;
  while (cursor != null && counter < 2) {
```

```
      if (eqNull(cursor.element(), element)) {
        counter ++;
      }
    }

    return counter == 2;
  }

  public static boolean eqNull (Object o1, Object o2) {
    return o1 == o2 || (o1 != null && o1.equals(o2));
  }
```

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

`public Iterable<Integer> multiplicaPares (Iterable<Integer> iterable)`

que recibe como parámetro un `Iterable` de `Integer` y debe devolver una nueva estructura `Iterable` que contenga el producto de cada par de valores consecutivos según el orden devuelto por `iterable`. Los elementos devueltos por el iterable podrían ser **`null`** y deben ser ignorados, es decir, dos valores que estén separados por elementos **`null`** deben ser considerados consecutivos. Por ejemplo, dada la siguiente secuencia de elementos: $v_0, null, v_1, v_2, null, v_3, ..., v_{n-1}, null, v_n$, el iterable resultante debe devolver $v_0 * v_1, v_1 * v_2, v_2 * v_3, ..., v_{n-1} * v_n$.

El iterable `iterable` podrá ser **`null`**, 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, si `iterable` devuelve los elementos `1,2,3,4`, la llamada `multiplicaPares(iterable)` debe devolver un iterable que contenga los elementos `2,6,12`. Si `iterable` devuelve `1,`**`null,null,`**`3,5`, la llamada `multiplicaPares(iterable)` debe devolver un iterable que contenga `3,15`; si `iterable` devuelve **`null,`**`1,`**`null`**, entonces `multiplicaPares(iterable)` debe devolver un iterable sin elementos; y si `iterable` devuelve `1`, entonces `multiplicaPares(iterable)` también debe devolver un iterable sin elementos.

Solución:

```
  public static Iterable<Integer> multiplicaPares (Iterable<Integer> iterable) {
    if (iterable == null) {
      throw new IllegalArgumentException();
    }
    Iterator<Integer> it = iterable.iterator();
    PositionList<Integer> res = new NodePositionList<>();

    if (!it.hasNext()) {
      return res;
    }

    Integer prev = null;
    while (it.hasNext()) {
      Integer current = it.next();
      if (current != null) {
        if (prev != null) {
          res.addLast(prev*current);
        }
        prev = current;
```

```
      }
    }

    return res;
  }
```

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

```
public String getMejorGrupo (PositionList<Pair<String,Integer>> notas,
                             Map<String,String> grupos)
```

que recibe como parámetro una lista `notas`, que contiene pares de $< DNI, nota >$ (el DNI es de tipo `String`) y un *map* con entradas $< DNI, grupo >$ (el grupo también es de tipo `String`) y devuelve el identificador de uno de los grupos (`String`) cuyos alumnos tengan la suma de notas más alta (podría haber más que un grupo con la misma suma de notas). Ni la lista ni el *map* recibidos de entrada contendrán elementos **`null`** y todos los alumnos recibidos en la lista `notas` tendrán entrada en el map `grupos`.

Por ejemplo, dada la lista `notas=[<a,6>,<b,8>,<c,10>,<d,9>,<e,5>]` y el map `grupos={<a,g1>,<b,g2>,<c,g1>,<d,g2>,<e,g2>` la llamada `getMejorGrupo(notas, grupos)` devolverá el grupo `g2` dado que los alumnos de este grupo acumulan 22 puntos frente a los 16 acumulados por `g1`. Sin embargo, dado `grupos2={<a,g1>,<b,g2>,<c,g3>,<d,g1>,<e,g3>`, la llamada `getMejorGrupo(notas, grupos2)` podrá devolver el grupo `g1` o `g3` ya que ambos acumulan 15 puntos frente a los 8 acumulados por `g2`.

Para la implementación se recomienda el uso de una variable auxiliar de tipo `Map<String,Integer>` para acumular la suma de notas por grupo. Se dispone de la clase `HashTableMap<K,V>` con un constructor sin parámetros que crea un `Map` vacío.

Solución:

```
public String getMejorGrupo (PositionList<Pair<String,Integer>> notas,
                             Map<String,String> grupos) {
  Map<String,Integer> notasXGrupo = new HashTableMap<String, Integer>();
  Integer notaRes = null;
  String grupoRes = null;

  for (Pair<String,Integer> nota: notas) {
    String grupo = grupos.get(nota.getLeft());

    Integer notaAcum = notasXGrupo.get(grupo);
    notaAcum = nota.getRight() + (notaAcum==null ? 0 : notaAcum);
    notasXGrupo.put(grupo, notaAcum);
    if (notaRes == null || notaAcum > notaRes) {
      notaRes = notaAcum;
      grupo = grupoRes;
    }
  }

  return grupoRes;
}
```

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

```java
<E> void method2 (PositionList<E> l) {
  Position<E> c = l.first();
  while (c != null) {
    maux(l,l.first());
    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 $O(n^2)$