# 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 **18 de Noviembre de 2022** 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:

```
public static <E> void eliminarRepeticiones (PositionList<E> list)
```

que recibe como parámetro una lista `list` y debe eliminar de ella todas las repeticiones de los elementos que la componen, sean consecutivas o no, conservando el orden de la primera aparición de los elementos en la lista. La lista de entrada podría ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. La lista podrá contener elementos **`null`**, que deben ser considerados como un elemento más de la lista, es decir, se deben eliminar sus repeticiones.

Por ejemplo, `eliminarRepeticiones([1,2,7])` debe dejar la lista `[1,2,7]`;
`eliminarRepeticiones([1,2,7,2,2])` debe dejar la lista `[1,2,7]`;
`eliminarRepeticiones([1,2,2,null,2,2,null,null])` debe dejar la lista `[1,2,null]`;
y `eliminarRepeticiones(null)` debe lanzar la excepción `IllegalArgumentException`.

**Solución:**

```
public static <E> void eliminarRepeticiones (PositionList<E> list) {
  if (list == null) {
    throw new IllegalArgumentException();
  }
  Position<E> cursor = list.first();

  while (cursor != null) {
    quitarRepeticionesDesde(list,cursor);
    cursor = list.next(cursor);
  }
}

private static <E> void quitarRepeticionesDesde (PositionList<E> list,
                                                 Position<E> cursor) {
  E e = cursor.element();
  cursor = list.next(cursor);
  while (cursor != null) {
    Position<E> next = list.next(cursor);
    if (eqNull(e, cursor.element())) {
      list.remove(cursor);
    }
    cursor = next;
  }
}
```

```
private static <E> boolean eqNull(E o1, E o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
}
```

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

`Iterable<Integer> sumarTrios (Iterable<Integer> iterable)`

que recibe como parámetro un `Iterable` que devuelve una secuencia de números. El método devuelve otro `Iterable` con la suma de todos los tríos de elementos **consecutivos** recibidos del iterable de entrada. Tanto el parámetro `iterable` como los elementos devueltos por este iterable nunca serán **`null`**. Si el parámetro `iterable` devuelve menos de tres elementos, el método debe devolver un `Iterable` sin elementos. Se dispone de la clase `NodePositionList<E>` y de la clase `ArrayIndexedList<E>` que implementan, respectivamente, las interfaces `PositionList<E>` e `IndexedList<E>` y que disponen de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, si el parámetro `iterable` devuelve los elementos `1,2,3,4,5`, la llamada `sumarTrios(iterable)` debe devolver la lista `[6,9,12]`, porque `6=1+2+3`, `9=2+3+4` y `12=3+4+5`; si el iterable devuelve `1,2,3`, `sumarTrios(iterable)` debe devolver `[6]`; y si el iterable devuelve `1,2`, la llamada `sumarTrios(iterable)` debe devolver `[]`.

**Solución:**

```
public static Iterable<Integer> sumarTrios (Iterable<Integer> datos) {
   PositionList<Integer> res = new NodePositionList<>();
   Iterator<Integer> it = datos.iterator();
   if (!it.hasNext()) {
     return res;
   }
   Integer prev1 = it.next();
   if (!it.hasNext()) {
     return res;
   }
   Integer prev2 = it.next();

   while (it.hasNext()) {
     Integer current = it.next();
     res.addLast(prev1+prev2+current);
     prev1 = prev2;
     prev2 = current;
   }
   return res;
 }
```

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

```
public static <E> boolean compruebaSec (PositionList<E> sec, Map<E,E> mapa)
```

que recibe como parámetros una lista `sec` y un `map`. El parámetro `map` almacena secuencias de elementos de forma que el valor asociado a una clave es el siguiente elemento en la secuencia. El método debe devolver **`true`** si la secuencia en `sec` se puede encontrar en `map`, es decir, si para cada par de elementos consecutivos `[...,e1,e2,...]` en `sec`, la clave `e1` existe, y su valor es `e2`, y **`false`** en caso contrario. Ni `sec` ni `map` serán **`null`**, ni contendrán elementos **`null`**. Si la lista `sec` contiene menos de dos elementos el método debe lanzar la excepción `IllegalArgumentException`.

Por ejemplo, dado el map `map=[<1,2>,<3,5>,<5,7>,<2,6>,<4,7>,<7,10>,<6,1>]`, la llamada `compruebaSec([3,5,7],map)` debe devolver **`true`**; la llamada `compruebaSec([3,5,6,10],map)` debe devolver **`false`**; la llamada `compruebaSec([6,1,2],map)` debe devolver **`true`**; la llamada `compruebaSec([4,7],map)` debe devolver **`true`**; la llamada `compruebaSec([6],map)` debe lanzar la excepción `IllegalArgumentException`.

**Solución:**

```
public static <E> boolean compruebaSec(PositionList<E> sec,
                                       Map<E,E> mapa) {
  if (sec.size() < 2) throw new IllegalArgumentException();

  boolean okSec = true;
  E prev = sec.first().element();
  if (!mapa.containsKey(prev))
    return false;

  Position<E> cursor = sec.next(sec.first());

  while (cursor != null && okSec) {
    okSec = cursor.element().equals(mapa.get(prev));
    prev = cursor.element();
    cursor = sec.next(cursor);
  }

  return okSec;
}
```

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

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

```
<E> int method2 (PositionList<E> l) {
  Position<E> c = l.last();
  int count = 0;
  if (l.size() % 2 == 0) {
    while (c != null) {
      count ++;
      c = l.prev(c);
    }
  }
  else {
    count = -1;
  }
  return count;
}
```

**Solución:**

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