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 **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:

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

que recibe como parámetros una lista `list` y un comparador `cmp` y debe devolver una **nueva** lista que contenga los mismos elementos que `list`, pero ordenados en orden ascendiente acorde al criterio de comparación establecido por el comparador `cmp`. La lista de entrada y el comparador podrían ser **`null`**. En caso de que alguno de los dos sea **`null`**, el método debe lanzar la excepción `IllegalArgumentException`. La lista de entrada no contendrá elementos **`null`**. Se dispone de la clase `NodePositionList<E>`, que implementa la interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

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:
`ordenar([1,2,7], new IntegerCmp())` debe devolver `[1,2,7]`,
`ordenar([7,2,1], new IntegerCmp())` debe devolver `[1,2,7]`,
`ordenar([1,3,2,3,9], new IntegerCmp())` debe devolver `[1,2,3,3,9]` y
`ordenar([],new IntegerCmp())` debe devolver `[]`.

Si ahora asumimos que contamos con un comparador `IntAbsCmp` que ordena los números de forma ascendiente por su valor absoluto (`IntAbsCmp`), la llamada `ordenar([-6,2,-4,10], new IntAbsCmp())`, debe devolver la lista `[2,-4,-6,10]` y la llamada `ordenar([-6,10,-2,-4], new IntAbsCmp())` debe devolver `[-2,-4,-6,10]`.

Las llamadas `ordenar(null,new IntegerCmp())` y `ordenar([...],null)` deben lanzar `IllegalArgumentException`.

Solución:

```
public static <E> PositionList<E> ordenar (PositionList<E> list,
                                           Comparator<E> cmp){
  if (list == null || cmp == null) {
    throw new IllegalArgumentException();
  }

  PositionList<E> res = new NodePositionList<>();
  for (E e: list) {
    insertarEnOrden(res,cmp,e);
  }
  return res;
}

private static <E> void insertarEnOrden(PositionList<E> list,
```

```
                                        Comparator<E> cmp,
                                        E e) {
  Position<E> cursor = list.first();

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

  if (cursor != null) {
    list.addBefore(cursor, e);
  }
  else {
    list.addLast(e);
  }
}
```

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

`boolean esSerieFibonacci(Iterable<Integer> iterable)`

que recibe como parámetro un `Iterable` de `Integer` y debe devolver **`true`** si cada número devuelto por el iterable, sin contar los dos primeros, es la suma de los dos números inmediatamente anteriores. Tanto el `Iterable`, como los elementos devueltos por el `Iterable` nunca serán **`null`**. Si el iterable devuelve menos de tres elementos, el método debe devolver **`true`**.

Por ejemplo, dado un `iterable` que devuelve los elementos `3,4,7,11,18`, la llamada `esSerieFibonacci(iterable)` debe devolver **`true`**, si `iterable` devuelve `3,4,7,11,17`, la llamada `esSerieFibonacci(iterable)` debe devolver **`false`**; si `iterable` devuelve `1,2,3,5,8,13`, entonces `esSerieFibonacci(iterable)` debe devolver **`true`**; y si `iterable` devuelve `1,2`, entonces `esSerieFibonacci(iterable)` debe devolver **`true`**.

Solución:

```
  public static boolean esSerieFibonacci(Iterable<Integer> datos)  {
    Iterator<Integer> it = datos.iterator();
    boolean esFib = true;
    if (!it.hasNext()) {
      return esFib;
    }
    Integer prev1 = it.next();
    if (!it.hasNext()) {
      return esFib;
    }
    Integer prev2 = it.next();

    while (it.hasNext() && esFib) {
      Integer current = it.next();
      esFib = current.equals(prev1+prev2);
      prev1 = prev2;
      prev2 = current;
    }
    return esFib;
  }
```

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

```
static <E> Iterable<E> obtenerSecuencia(Map<E,E> map, E pos)
```

que recibe como parámetro un `map` y un elemento inicial `pos`. 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 un objeto iterable que contenga, en el orden correcto, la secuencia de elementos empezando con el parametro `pos`.

La secuencia de elementos puede terminar de dos formas: cuando se obtiene un elemento que no está como clave en `map` o bien cuando se recibe un elemento que ya se ha añadido al objeto iterable. En ambos casos se debe añadir el elemento al objeto iterable como último elemento de la secuencia. 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. Asimismo, se dispone de la clase `HashTableMap<K,V>` con un constructor sin parametros que crea un `Map` vacío. El parámetro `map` nunca será **`null`** ni contendrá valores **`null`**.

Por ejemplo, dado `map=[<1,2>,<3,5>,<5,7>,<2,6>,<7,10>,<6,1>,<8,9>,<9,4>,<4,9>]`, la llamada
`obtenerSecuencia(map,3)` debe devolver `[3,5,7,10]`,
`obtenerSecuencia(map,2)` debe devolver `[2,6,1,2]`,
`obtenerSecuencia(map,8)` debe devolver `[8,9,4,9]`,
`obtenerSecuencia(map,12)` debe devolver `[12]` y
`obtenerSecuencia(map,10)` debe devolver `[10]`.

Solución:

```
public static <E> Iterable<E> obtenerSecuencia(Map<E,E> mapa, E pos) {
  PositionList<E> res = new NodePositionList<>();
  Map<E,E> mapAux = new HashTableMap<>();

  while (mapa.containsKey(pos) && !mapAux.containsKey(pos)) {
    mapAux.put(pos, pos);
    res.addLast(pos);
    pos = mapa.get(pos);
  }

  res.addLast(pos);

  return res;
}
```

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

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

```java
<E> void method2 (PositionList<E> l) {
  Position<E> c = l.first();
  int count = 0;
  while (c != null && count < 3) {
    count ++;
    c = l.next(c);
  }
  maux(l,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 $O(n^2)$
- El método `method2` tiene complejidad $O(n)$