Grado en Ingeniería Informática y Doble Grado en Informática y Administración de Empresas	***MODELO A*** Asignatura Estructura de Datos y Algoritmos **30 de Junio de 2014.** **CONVOCATORIA EXTRAORDINARIA**

Nombre: .....

Apellidos: .....

Grupo: .....

LEA ATENTAMENTE ESTAS INSTRUCCIONES ANTES DE COMENZAR LA PRUEBA

1. Es necesario poner todos los datos del alumno en el cuadernillo de preguntas (este documento). Use un bolígrafo para rellenarlos.
2. El examen está compuesto por 5 Preguntas.
3. Solamente se evaluará la contestación en este cuadernillo de preguntas.
4. Cuando finalice la prueba, se deben entregar el enunciado del examen y cualquier hoja que haya empleado.
5. No está permitido salir del aula por ningún motivo hasta la finalización del examen.
6. Desconecten los móviles durante el examen.
7. La duración del examen es de **2 horas**.

NO PASE DE ESTA HOJA hasta que se le indique el comienzo del examen

Problema 1 (3.5 puntos).

Data la siguiente especificación formal del TAD Sucesión Geométrica:

```
/**
 * Especificación del TAD Sucesión Geométrica.
 * En una sucesión geométrica, cada término se obtiene multiplicando
 * el anterior por una cantidad constante que se llama factor
 * @author isegura
 */
public interface ISucGeometrica {
	/** devuelve el factor de la sucesión*/
	public int getFactor();
	/** calcula y devuelve el siguiente término de la sucesión*/
	public int sigTermino();
	/** muestra la sucesión */
	public void mostrar();
	/** muestra la sucesión en sentido inverso*/
	public void mostrarInv();
}
```

El objetivo de este ejercicio es desarrollar una clase, **SucGeometrica**, que implemente el TAD ISucGeometrica. Dicha clase debe permitir almacenar los términos de la sucesión así como su factor. El alumno deberá implementar su propia clase que implemente una SucGeométrica almacenando los términos en una lista enlazada. Dicho de otra forma, **no se permite el uso de clases que implementen listas enlazadas (es decir, no se permite el uso de clases como SList, DList, ArrayList , LinkedList o Vector). Tampoco se puede utilizar arrays.** Sin embargo, se recomienda el uso de la clase SNode o DNode.

a) (0.1 puntos) Escribe un método constructor que permita crea una sucesión vacía.
b) (0.25 puntos) Escribe un método constructor que reciba como parámetro el factor de la sucesión y el valor del primer término de la sucesión.
c) (0,25 puntos) ¿Qué atributos debemos añadir que permitan representar correctamente un objeto de la clase?
d) (0.1 puntos) Escribe el método getFactor() que devuelva el factor de la sucesión geométrica.

```
public class SucGeometrica implements ISucGeometrica {


	public SucGeometrica(int factor, int primTerm) {



	}

	public int getFactor() {

	}
}
```

Solución:

```
public class SucGeometrica implements ISucGeometrica {
	/** Nodo que almacenará el primer elemento de la sucesión*/
```

```
SNode<Integer> primero=null;
/** Factor de la sucesión geométrica*/
int factor;
/** Almacena el valor del primer término*/
int primTerm;

/** Constructor que guarda el factor y crea el primer término de
la sucesión */
public SucGeometrica(int factor, int primTerm) {
	this.factor=factor;
	this.primTerm=primTerm;
	primero=new SNode<Integer>(primTerm);
}

/** Devuelve el factor de la sucesión */
public int getFactor() {
	return factor;
}
```

e) (1 punto) Escribe el método sigTermino(). Este método calcula y almacena el siguiente término de la sucesión. Dicho término se obtiene multiplicando el anterior término por el factor. Es recomendable que el método considere el caso en el cual la sucesión no ha sido inicializada aún (es decir, cuando *primero* no apunta aún a ningún nodo)

```
public int sigTermino() {




}
```

Solución:

```
public int sigTermino() {
	SNode<Integer> anterior = null;
	for (SNode<Integer> nodeIt = primero;
			nodeIt != null; nodeIt = nodeIt.nextNode) {

		anterior = nodeIt;
	}
	int sigTerm=-1;
	if (anterior == null) {
		sigTerm=primTerm;
		primero=new SNode<Integer>(primTerm);
	} else {
		int antValue=anterior.getElement();
		sigTerm=antValue*factor;
		SNode<Integer> newNode=new SNode<Integer>(sigTerm);
		anterior.nextNode = newNode;
	}
	return sigTerm;
}
```

f) (0.3 puntos) Escribe un segundo método constructor que además de recibir como parámetro el factor y el primer término de la sucesión, cree los primeros n términos de la sucesión.

```
public SucGeometrica(int factor, int primTerm, int n) {
```

```
}
```

Solución:

```
public SucGeometrica(int factor, int primTerm, int n) {
	this.factor=factor;
	this.primTerm=primTerm;
	for (int i=0; i<=n;i++) sigTermino();
}
```

g) (0. 5 puntos) Escribe el método mostrar().

```
public void mostrar() {



}
```

Solución:

```
public void mostrar() {
		System.out.print("Sucesión : ");

		for (SNode<Integer> nodeIt = primero;
				nodeIt != null; nodeIt = nodeIt.nextNode) {
			System.out.print(nodeIt.getElement()+" ");
		}
		System.out.println();
	}
```

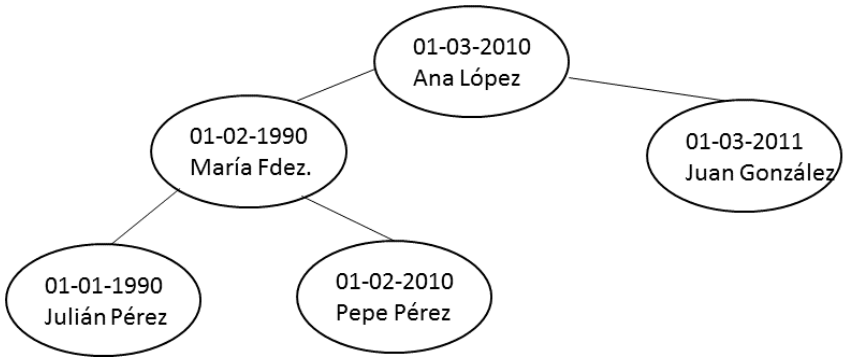
h) (1 puntos) Escribe el método mostrarInv() que muestra en sentido inverso la sucesión geométrica. Nota: se recomienda el uso de tipos abstractos de datos como pilas, colas, etc. en caso necesario.

```
public void mostrarInv() {



}
```

Solución:

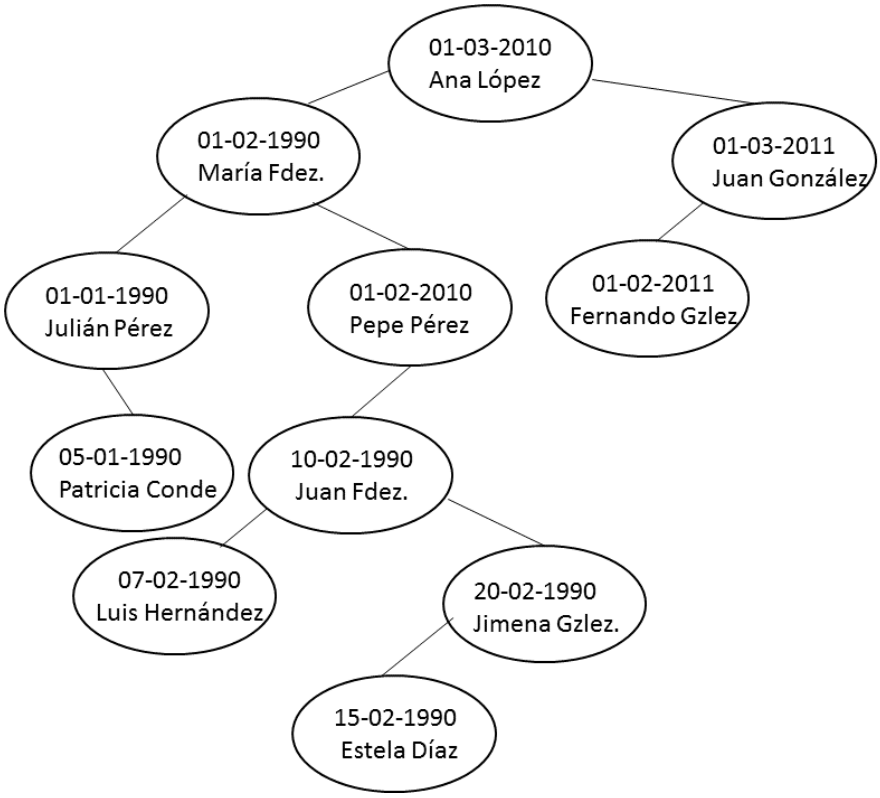
```
public void mostrarInv() {
	SStack<Integer> pila=new SStack<Integer>();
	for (SNode<Integer> nodeIt = primero;
			nodeIt != null; nodeIt = nodeIt.nextNode) {
		pila.push(nodeIt.getElement());
	}
	System.out.print("Sucesión en orden inverso: ");
	while (!pila.isEmpty()) {
		System.out.print(pila.pop()+ " ");
	}
	();
}
```

Problema 2 (3,5 puntos) Mi familia es un poco especial, en lugar de recordar a mis ancestros en forma de árbol genealógico, este año ha preferido almacenar toda la información en forma de Árbol Binario de Búsqueda (ABB o BST en inglés), y es que se han dado cuenta de que así les es más sencillo realizar búsquedas de los familiares que vivieron en un determinado periodo. En la siguiente figura hemos dibujado parte del árbol genealógico de mi familia, donde la clave es la fecha de nacimiento y el elemento de cada nodo es el nombre propio de la persona:



a) **(0,5 puntos)** Dibuja el árbol resultante de insertar los siguientes familiares en el árbol de la figura. IMPORTANTE: es necesario hacerlo en el orden que aparecen a continuación:
    - <10-02-1990, Juan Fdez.>
    - <20-02-1990, Jimena González>
    - <15-02-1990, Estela Díaz>
    - <05-01-1990, Patricia Conde>
    - <01-02-2011, Fernando González>
    - <07-02-1990, Luis Hernández>

    Solución:



b) **(0,5 puntos)** Escribe la cabecera de la clase "ArbolFamiliar" que permite almacenar este árbol binario de búsqueda. Para ello, será necesario hacer uso de la librería sobre Árboles Binarios de Búsqueda proporcionada durante el curso (en particular, la librería BSTree). Las fechas serán de tipo "Date" y los nombres se definirán como "String".

Solución:

```
public class ArbolFamiliar extends BSTree<Date, String> {

}
```

c) **(1 puntos)** Implementa el algoritmo recursivo "primerFamiliar" que devuelve el nombre de la primera persona que nació en mi familia (según la información almacenada en el árbol).
NOTA IMPORTANTE: Para las implementaciones que existen en este ejercicio (esta u otras secciones), recuerda que existen los métodos:

- Clase BSTNode
  - *nodo.**hasLeftChild()***
  - *nodo.**hasRigtChild()***

Solución:

```
public String primerFamiliar(){
      return primerFamiliar (root);
}

public String primerFamiliar(BSTNode<Date, String> nodo){
```

```
        if (nodo != null){
                if (nodo.hasLeftChild())
                        return primerFamiliar(nodo.getLeftChild());
                else return nodo.getElement();
        }
        return null;
}
```

d) **(1.5 punto)** Implementa el algoritmo recursivo "ancestros" que imprime por pantalla el nombre de todos los familiares que nacieron antes que una fecha determinada, **ordenados** de la fecha más reciente a la más antigua. Recuerda que puedes utilizar el método ***<date>.compareTo(<date>)***, donde dadas dos fechas *f1 y f2*, f1.***compareTo(****f2****)*** devolverá:

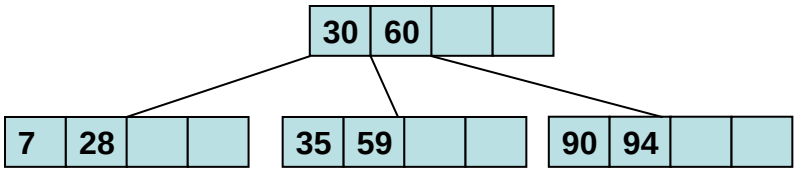
- ***a.*** un número negativo si f1<f2;
- ***b.*** 0 si f1=f2
- ***c.*** un número >0 si f1>f2.

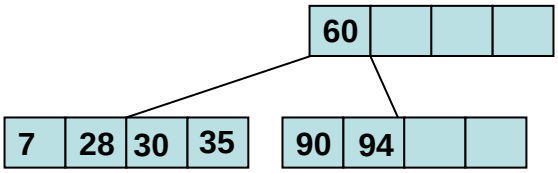
Solución:

```
// ancestros
    public void ancestros(Date fecha) {
            ancestros(root, fecha);
    }


    public void ancestros(BSTNode<Date, String> nodo, Date fecha) {
            if (nodo != null) {
                    // recorrido POST-ORDER

                    // hijo derecho
                    If ((nodo.hasRightChild()) &&
                            (nodo.getKey().compareTo(fecha) < 0))
                                    ancestros(nodo.getRightChild(), fecha);

                    // nodo
                    if (nodo.getKey().compareTo(fecha) < 0)
                                    System.out.println(nodo.getElement());

                    // hijo izquierdo
                    if ((nodo.hasLeftChild()) &&
                            (nodo.getKey().compareTo(fecha) < 0))
                                    ancestros(nodo.getLeftChild(), fecha);
            }
    }
```

Problema 3 (1 punto).

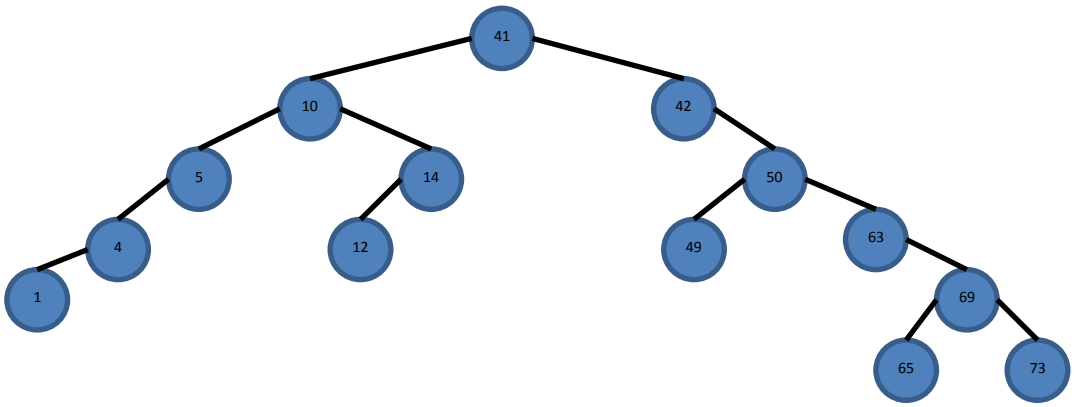
a) **(0,30 puntos)** Para el siguiente árbol B, dibuja el árbol resultante de borrar la clave 59.



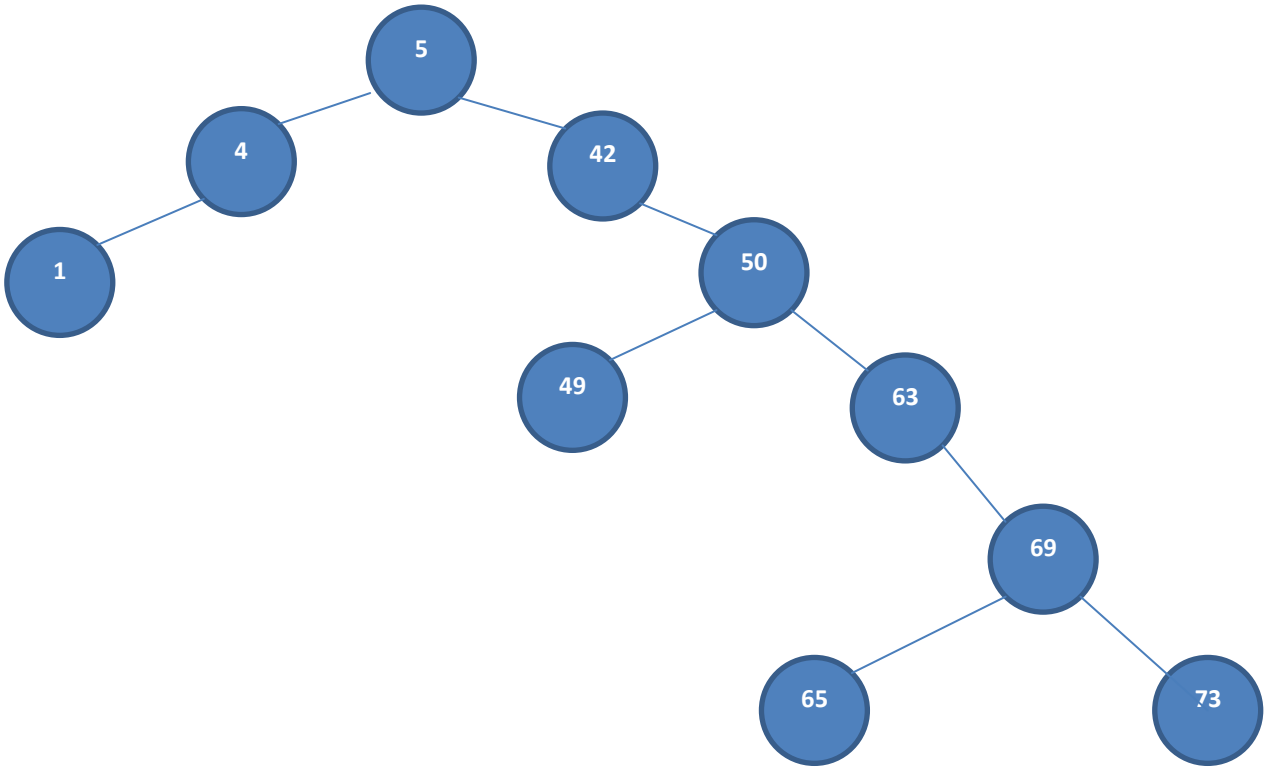


b) **(0,40 puntos)** Utilizando como base el árbol binario de búsqueda siguiente (donde la clave es igual al propio elemento almacenado), elimina cuatro veces la raíz del árbol. Dibuja a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadros).

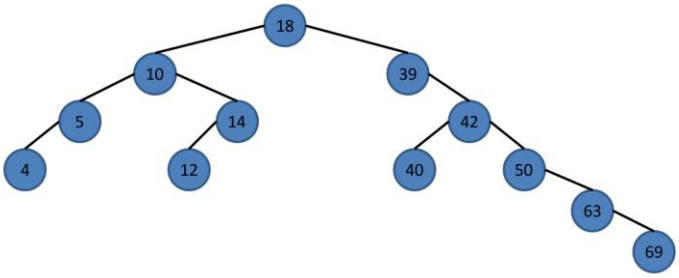
r

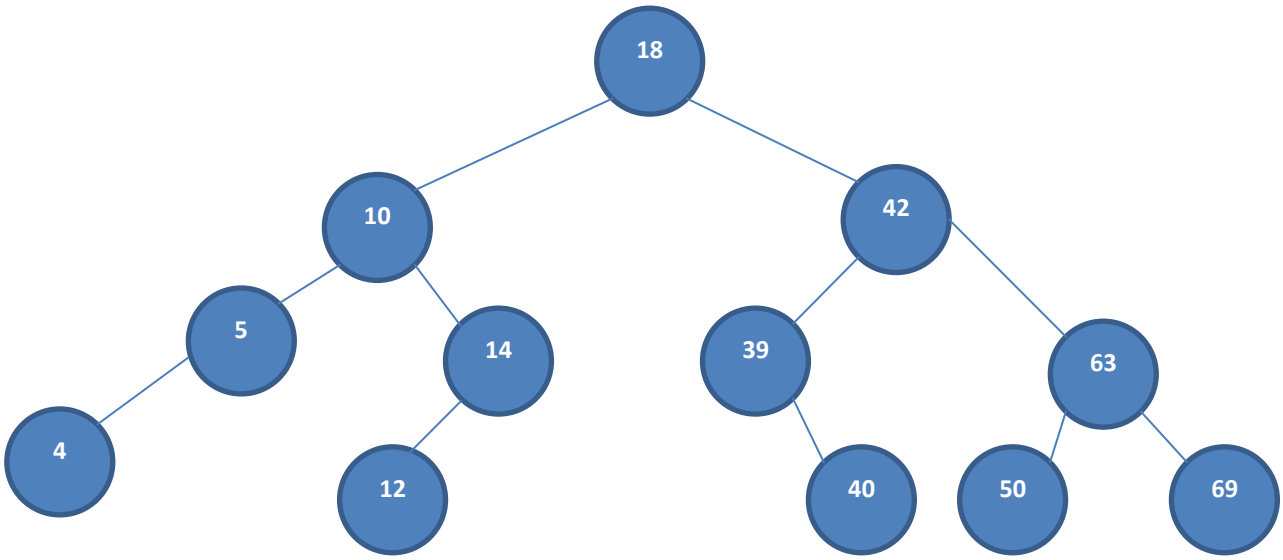


Nota: En la operación de borrado, siempre optaremos por la elección del predecesor.



c) (0, 40 puntos) Realiza el equilibrado en tamaño del árbol binario de búsqueda siguiente. Dibuja a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadros).





Problema 4 (1 punto). Dado un árbol binario de números enteros, escribe un método recursivo que recorra el árbol e imprima por pantalla aquellos elementos cuyo abuelo tengan un valor par.

```
	public static<E> void mostrarAbuelosPar(BinTree<Integer>tree) {
		mostrarAbuelosPar(tree.rootNode);
}

	public static<E> void mostrarAbuelosPar(BinTreeNode<Integer>node){

		if (node==null) return;
```

```
        if (node.parent!=null&&node.parent.parent!=null&&
node.parent.parent.getElement()%2==0) {

                        System.out.println(node.getElement());
                }

        mostrarAbuelosPar(node.leftChild);
        mostrarAbuelosPar(node.rightChild);
        }
```

Problema 5 (1 punto)

Para cada una de estas afirmaciones, indique si son ciertas o falsas. Justifique su respuesta:

1. Para definir una relación de herencia con una clase dada es necesario utilizar la palabra reservada 'extends'.

   Solución: [X] Verdadero [ ] Falso

   Justificación:

   Efectivamente, para definir una relación de herencia es necesario indicar que la subclase hereda de la superclase. Por ejemplo:

   public class B extends A {...}

2. Los métodos estáticos son aquellos que requieren la creación de un objeto mediante la palabra reservada new para poder ser invocados.

   Solución: [ ] Verdadero [X] Falso

   Justificación:

   Es justamente al contrario, la invocación de un método estático no requiere la creación previa de un objeto. Los métodos estáticos no están ligados a una instancia concreta de una clase.

3. La sobrecarga de un operador puede realizarse únicamente si se define una relación de herencia con una clase padre.

   Solución: [ ] Verdadero [X] Falso

   Justificación:

   No es así, la sobrecarga supone la definición de varias cabeceras para un mismo método. El único requisito a cumplir es que esas cabeceras tengan distinto número de parámetros y/o distintos tipos de datos asociados.

4. Una lista simplemente enlazada está compuesta por una secuencia de nodos. Cada nodo almacena referencias o punteros al nodo que le precede y al que le sucede.

   Solución: [ ] Verdadero [X] Falso

   Justificación:

   En una lista simplemente enlazada, cada nodo guarda únicamente una referencia al nodo que le sucede.

5. Un algoritmo recursivo debe definirse siempre como estático (mediante la palabra reservada 'static') para poder invocarlo desde una clase independiente de aquella en la que está definido.

   Solución: [ ] Verdadero [X] Falso

   Justificación:

   No es cierto, que un método sea o no estático no tiene que ver con que sea recursivo o no.

6. Que la complejidad temporal de un algoritmo, O, sea de orden n, depende, entre otras cosas, de la existencia de bucles que recorran secuencialmente los datos de entrada del algoritmo.

   Solución: [X] Verdadero [ ] Falso

   Justificación:

   Efectivamente, si un algoritmo incluye un bucle podemos asegurar que al menos tiene un orden O(n). Si incluyese dos bucles anidados, alcanzaría una complejidad de orden cuadrático, si los elementos de entrada se recorren secuencialmente.

7. La complejidad temporal de un algoritmo de búsqueda sobre un árbol de búsqueda binaria tiene complejidad O(n)

   Solución: [ ] Verdadero [X] Falso

   Justificación:

   En una búsqueda sobre un árbol de búsqueda binario el espacio de búsqueda se divide en 2, en cada iteración o llamada recursiva para realizar la búsqueda. La sucesión 1/2, 1/4, 1/8... converge a log n.

8. Para que un método recursivo funcione correctamente basta con que exista un caso recursivo que llama al mismo método.

   Solución: [ ] Verdadero [X] Falso

   Justificación:

Es necesario incluir también al menos un caso base que permita terminar el proceso recursivo.

9. Suponiendo que se ha definido una interfaz IList, el siguiente código Java debería producir un error de compilación:

```
IList<Integer> mylist = new IList<Integer>();
```

Solución: [X] Verdadero [ ] Falso

Justificación:

Efectivamente, es imposible instanciar una interfaz. La interfaz se define exclusivamente como una 'plantilla' con los métodos que debe incluir aquella clase que quiera implementar esa interfaz.

10. Dado el siguiente algoritmo recursivo para obtener el factorial de un número almacenando los factores en una lista simplemente enlazada:

```
static long recFactorial (int n) {

      SList<Integer> list = new SList<Integer>();

      if (n == 0) {
         list.add(n);
         return 1;
      }else {
         list.add(n);
         return n * recFactorial(n - 1);
      }
}
```

Al alcanzar el caso base la lista 'list' tendrá almacenados todos los factores utilizados. Por ejemplo, para el valor n=5, la lista 'list' contendrá los valores 5,4,3,2,1,0.

Solución: [ ] Verdadero [X] Falso

Justificación:

Es necesario darse cuenta de que en cada llamada se está creando una nueva lista 'list', cuyo ámbito se limita al de la llamada en cuestión y que, por tanto, no existe como tal al finalizar el proceso recursivo.