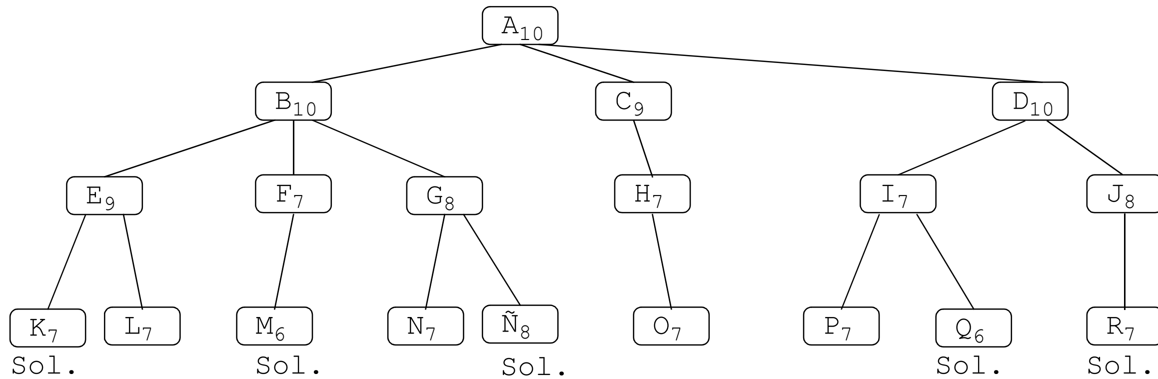
Examen de Programación II
(Ingeniería Informática)

Junio 2007

1) (2 puntos) El espacio de soluciones de un problema está representado por el árbol de expansión mostrado a continuación:



Para cada nodo, el número que aparece junto a la letra mayúscula que le identifica es su coste. Los nodos etiquetados con "Sol." son soluciones al problema.

Se dispone de un algoritmo de ramificación y poda para buscar la solución óptima al problema (solución con mayor coste). El algoritmo utiliza el coste de los nodos para su ordenación en la cola de nodos vivos y para realizar la poda. Completar las etapas que realizaría el algoritmo, indicando, para cada nodo podado, la causa de su poda. Etapas:

eNodo	Nodos hijos	Cola nodos vivos	Mejor sol.	Podado
A_{10}→	B_{10} C_9 D_{10}→	B_{10} D_{10} C_9	-	-
B_{10}→	E_9 F_7 G_8 →	D_{10} C_9 E_9 G_8 F_7	-	-
...	...	...	...	...

2) (1.5 puntos) Escribir el comentario que describe la funcionalidad de la función Haskell siguiente:

```
--  descripción ...
func      :: Integer -> [Integer] -> [Integer]
func s []        = []
func s (x:xs)
  | x<0        = func s xs
  | s-x>=0     = x:func (s-x) xs
  | s-x<0      = []
```

¿Qué resultado se obtendría de evaluar la siguiente expresión?

```
func 5 [3,2,-1,0,3,-2,2]
```

3) (2.5 puntos) Se dispone de un conjunto de cajas (cada una capaz de soportar un peso máximo determinado) y de un conjunto de objetos (cada uno caracterizado por su peso). Se desea utilizar un algoritmo basado en el esquema de "vuelta atrás" para conocer el máximo número de objetos que sería posible guardar en las cajas (no se quiere obtener la distribución óptima de objetos en cajas, sólo el número de objetos que es posible almacenar en esa distribución óptima).

Para la implementación del algoritmo se dispone de las clases `Objeto` y `Caja` ya realizadas. No va a ser necesario utilizar ningún método de la clase `Objeto`, por lo que no se proporciona su interfaz. La clase `Caja` tiene la siguiente interfaz:

```
public class Caja {
  /** crea una caja con el peso máximo indicado por pesoMáximo y
   *  con un peso disponible igual al peso máximo */
  public Caja(double pesoMáximo) { xxx }

  /** mete el objeto en la caja restando su peso del peso
   *  disponible en la caja */
  public void mete(Objeto o) { xxx }

  /** saca el objeto de la caja. El peso del objeto se
   *  restituye en el peso disponible de la caja */
  public void saca(Objeto o) { xxx }

  /** retorna true si el peso del objeto es menor o
   *  igual que el peso disponible en la caja */
  public boolean cabe(Objeto o){ xxx }
}
```

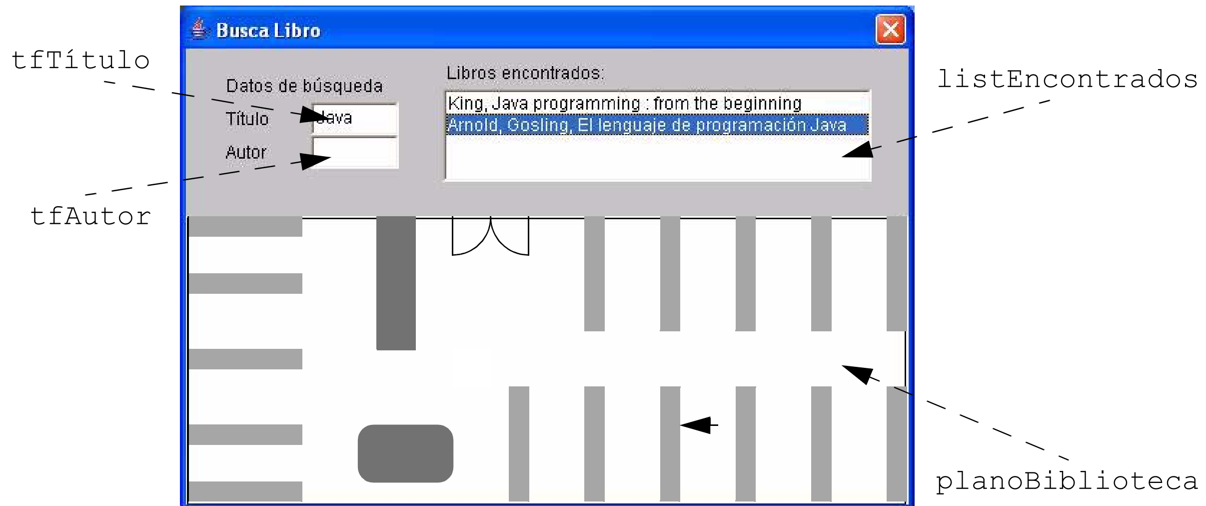
Se pide completar el código correspondiente a los métodos `maxNumObjetosEnCajas` y `maxNumObjetosEnCajas_Recursivo` mostrados a continuación que serán los que implementen el algoritmo de "vuelta atrás" que resuelve el problema citado:

```
/** método que retorna el máximo número de los objetos incluidos en
 * el array objs que pueden llegar a meterse en las cajas incluidas
 *  en el array cajas */
public static int maxNumObjetosEnCajas(Objeto[] objs,Caja[] cajas){
  ordenaDeMenorAMayorPeso(objs);

  // invoca maxNumObjetosEnCajas_Recursivo
  ...
}

/** método recursivo que implementa el algoritmo basado en
 *  "Vuelta Atrás" */
public static ... maxNumObjetosEnCajas_Recursivo(...){
  ...
}
```

4) (2.5 puntos) Se está desarrollando una GUI para gestión de los libros de una biblioteca. Uno de los diálogos que forman esta GUI (la clase `DiálogoBuscaLibro`) permite buscar libros por autor y/o título y mostrar su situación en el mapa de la biblioteca. Suponemos que el diálogo ya ha sido diseñado e implementado aunque todavía no responde a ningún evento. La apariencia del diálogo mostrando la localización del libro seleccionado sería:



Los componentes activos del diálogo son dos campos de texto para el nombre del autor (`tfAutor`) y el título del libro (`tfTítulo`), una lista que muestra los libros encontrados para el autor y/o el título indicados (`listEncontrados`) y un canvas que permite mostrar la localización de un libro sobre el mapa de la biblioteca (`planoBiblioteca`). La funcionalidad que se desea conseguir con el diálogo es la siguiente:

- cuando se pulse "enter" en cualquiera de los campos de texto deberá realizarse la búsqueda de los libros con el autor y/o el título escritos en los respectivos campos de texto. A continuación se borran los contenidos de la lista (resultado de una posible búsqueda anterior) y se muestran los libros encontrados en la búsqueda actual. También deberá borrarse la marca de localización en el mapa.
- cuando se pulsa el botón del ratón sobre el nombre de uno de los libros en la lista de libros encontrados, se deberá mostrar su localización en el mapa.

Se pide completar la clase `DiálogoBuscaLibro` para que implemente la funcionalidad descrita, añadiendo los atributos y métodos que se considere necesario y añadiendo instrucciones al constructor (al menos lo referente a la asignación de los manejadores de eventos):

```
public class DialogoBuscaLibro extends Dialog {

  otros atributos no relevantes para el problema planteado
  private List listEncontrados = xxx;
  private TextField tfAutor = xxx;
  private TextField tfTítulo = xxx;
  private PlanoBiblioteca planoBiblioteca = xxx;

  /** Constructor */
  public DialogoBuscaLibro(Frame owner) {
    super(owner);
    crea los componentes y les añade a los paneles
  }
}
```

Para implementar la funcionalidad deseada se dispone del conjunto de clases que componen la aplicación y que se suponen ya implementadas. De cada clase sólo se muestra la cabecera de los métodos relevantes para el problema planteado:

```
/** ventana principal de la aplicación, desde ella se lanza el
  * diálogo de búsqueda de libros. */
public class VentanaPrincipal extends Frame {

  atributos y métodos no relevantes para el problema planteado

  /**retorna una lista con todos los libros que contienen el string
    *"título" en su título y el string "autor" en sus autores */
  public LinkedList<Libro> buscaLibros(String título, String autor)
  { xxx }
}


/** dibujo del plano de la biblioteca. Puede utilizarse para
 *  indicar la posición de un libro (ver método "posiciónLibro")*/
public class PlanoBiblioteca extends Canvas{

  atributos y métodos no relevantes para el problema planteado

  /** indica las coordenadas en las que se quiere poner la marca
   * que indica la localización de un libro. Las coordenadas
   * (-1,-1) indican que no se quiere marcar ninguna posición.
   * La invocación de este método no implica el repintado del mapa
   * con la marca en sus nuevas coordenadas. El repintado debe
   * forzarse de forma expresa después de llamar a este método. */
  public void posiciónLibro(int x, int y) { xxx }
}


/** clase que contiene la información asociada con cada libro:
 *  título, autores y coordenada X e Y de su situación en la
 *  biblioteca */
public class Libro {

  atributos y métodos no relevantes para el problema planteado

  /** retorna una línea de texto con el nombre de los autores y el
   *  título del libro */
  public String aTexto() { xxx }

  /** retorna la coordenada x de la situación del libro en el plano
   *  de la biblioteca */
  public int coordenadaX() { xxx }

  /** retorna la coordenada y de la situación del libro en el plano
   *  de la biblioteca */
  public int coordenadaY() { xxx }

}
```

5) (1.5 puntos) Se dispone de la clase `Vector`, cuyos métodos públicos son los mostrados a continuación:

```java
public class Vector {

   /**
    * Crea un vector con los componentes indicados en el array
    * componentes.
    * La dimensión del vector se hace igual a la longitud del
    * array componentes
    * @param componentes componentes del vector a crear
    * @throws ArrayNulo si componentes es null o tiene longitud 0
    */
   public Vector(int[] componentes) throws ArrayNulo { xxx }

   /**
    * Suma v1 con el vector actual, almacenando el resultado en el
    * vector actual
    * @param v1 vector a sumar con el vector actual
    * @throws DimensionesDiferentes si la dimensión de v1 no es
    *                               igual a la del vector actual
    */
   public void suma(Vector v1) throws DimensionesDiferentes { xxx }

   /**
    * Retorna el valor de la componente c del vector
    * @param c componente del vector (numeradas de 0 a dimensión del
    *          vector-1)
    * @return valor de la componente c del vector
    * @throws ComponenteInexistente si c no está en el rango 0 a
    *                               dimensión del vector-1
    */
   public int componente(int c) throws ComponenteInexistente { xxx }

   /**
    * Retorna la dimensión del vector
    * @return dimensión del vector
    */
   public int dimensión() { xxx }
}
```

Se pretende probar la clase `Vector` por el método de "caja negra":

- indicar las particiones de equivalencia necesarias para probar todos sus métodos
- escribir un método de prueba para la herramienta JUnit que permita probar el método `suma` (suponiendo que todos los demás métodos de la clase son correctos).