Examen de Programación II
(Ingeniería Informática)

Septiembre 2009

1) Lenguaje C (1.5 puntos)

Escribir el módulo "Nombres" (ficheros `nombres.h` y `nombres.c`) que defina el tipo de datos:

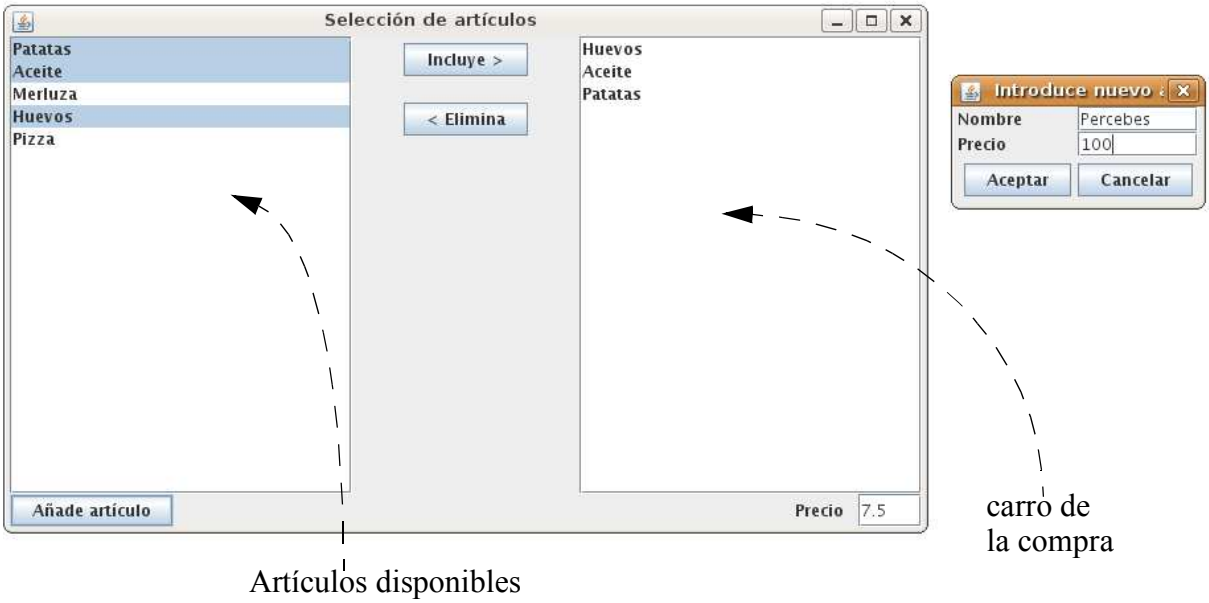
```
typedef struct {
  char * nombre;
  char * primer_apellido;
  char * segundo_apellido;
} nombre_persona_t;
```

Junto con ese tipo de datos el módulo deberá definir una función que reciba como parámetro un puntero a una variable de tipo `nombre_persona_t` y retorne un string que contenga el nombre completo de la persona, es decir, su nombre y sus dos apellidos separados por espacios.

Escribir un programa principal muy sencillo (que no pida datos al usuario) que utilice la función del módulo "Nombres".

Indicar cual sería el comando que permitiría compilar y enlazar el programa principal desarrollado y crear un ejecutable llamado `"nombres.exe"`.

2) (3.5 puntos) Se pretende realizar una aplicación gráfica formada por la ventana `VentanaSelección` y el diálogo `DiálogoIntroduceArtículo` que tienen la apariencia mostrada a continuación:



La ventana `VentanaSelección` muestra dos listas. La de la izquierda contiene todos los artículos disponibles en una tienda y la de la derecha los artículos incluidos en el carro de la compra. Además, el campo de texto etiquetado como "Precio" muestra el precio total de los artículos del carro.

Ambas listas permiten selección múltiple. La pulsación del botón "Incluye" añade al carro de la compra todos los artículos seleccionados en la lista de artículos disponibles. La pulsación del botón "Elimina" elimina del carro de la compra los artículos seleccionados en esa lista. Cada vez que cambian los artículos incluidos en el carro de la compra, el campo de texto "Precio" debe cambiar para reflejar el precio total actual.

La pulsación del botón "Añade artículo" provoca la aparición del diálogo `DiálogoIntroduceArtículo` que permite introducir un nuevo artículo con su nombre y su precio. La pulsación del botón "Aceptar" después de haber introducido los datos en el diálogo provocará la aparición del nuevo artículo en la lista de artículos disponibles.

Escribir todo el código correspondiente a la aplicación descrita, salvo la parte correspondiente a la distribución de los componentes en la ventana y el diálogo.

3) Esquemas algorítmicos

Una inmobiliaria mantiene una lista de posibles compradores y otra de pisos a la venta. Cada comprador ha informado a la inmobiliaria sobre las características del piso que desearía comprar (balcón, ascensor, orientación sur y vistas al mar). Así, por ejemplo, un comprador puede querer un piso con balcón y vistas al mar, pero no le importa si tiene ascensor o está orientado al sur.

La inmobiliaria desea realizar una asignación de pisos a compradores de forma que se maximice la suma total de las características satisfechas a todos los compradores. Por ejemplo, dado el siguiente conjunto de compradores:

Compradores	Balcón	Ascensor	Orientación Sur	Vistas al Mar
C1	Sí	No	No	Sí
C2	No	Sí	Sí	Sí

Y los siguientes pisos:

Pisos	Balcón	Ascensor	Orientación Sur	Vistas al Mar
P1	Sí	No	Sí	Sí
P2	Sí	No	No	No
P3	No	Sí	No	Sí

La solución optima sería asignar el piso P1 al comprador C1 (2 características satisfechas) y el piso P3 al comprador C2 (también 2 características satisfechas), lo que hace un total de 4 características satisfechas que es el máximo valor que se puede obtener para estos conjuntos de compradores y pisos.

3.a) (2 puntos) Implementar un algoritmo basado en "Ramificación y Poda" que permita resolver el problema planteado. Escribir también el código correspondiente a los métodos y/o atributos que es necesario añadir a la clase `AsignaciónParcial` (mostrada más adelante). El método que implemente el algoritmo deberá tener la siguiente cabecera:

```
/**
 * Realiza la asignación de pisos a compradores
 * @param pisos todos los pisos de la inmobiliaria
 * @param compradores todos los compradores de la inmobiliaria
 * @return la asignación de pisos a compradores que maximiza la
 * suma total de características satisfechas
 */
public static AsignaciónParcial asignaPorRyP(
    ArrayList<Piso> pisos,
    ArrayList<Comprador> compradores) {
```

3.b) (1.5 punto) Diseñar e implementar un algoritmo heurístico que permita obtener una buena asignación de pisos a compradores. Escribir también el código correspondiente a los métodos y/o atributos que es necesario añadir a la clase `AsignaciónParcial` (mostrada más adelante). Explicar con lenguaje natural el criterio de selección utilizado.

3.c) (0.5 puntos) Explicar como el algoritmo basado en "Ramificación y Poda" podría sacar ventaja del uso del heurístico.

Para implementar los algoritmos solicitados, se dispone de la clase enumerada `CaracterísticaPiso`:

```
/**
 * Características que un comprador puede desear en un piso
 */
public enum CaracterísticaPiso {
  BALCÓN, ASCENSOR, ORIENTACIÓN_SUR, VISTAS_MAR
}
```

También se dispone de las clases ya implementadas `Piso` y `Comprador` de las que sólo se muestra la documentación correspondiente a los métodos necesarios para el problema planteado:

- Interfaz de la clase `Comprador`:

`public boolean quiereCaracterística(CaracterísticaPiso c)`

```
Indica si una característica es deseada o no por este comprador

Parameters:
    c característica
Returns:
    true si la característica indicada está entre las deseadas por
    el comprador y false en caso contrario
```

public int características Coincidentes(Piso p)

Retorna el número de características coincidentes entre este comprador y el piso. Es decir, el número de características deseadas por el comprador que posee el piso

Parameters:
p piso
Returns:
número de características coincidentes entre el comprador y el piso

- Interfaz de la clase `Piso`:

public boolean tieneCaracterística(CaracterísticaPiso c)

Indica si este piso posee la característica indicada

Parameters:
c característica
Returns:
true si el piso posee la característica y false en caso contrario

Además se dispone de la clase `AsignaciónParcial`, que permite representar cada una de las soluciones parciales o nodos del espacio de soluciones del problema. Esta clase se encuentra parcialmente implementada, a continuación se muestran sus atributos y los métodos ya implementados (no aparece su código para ahorrar espacio). A esta clase el alumno puede añadir los atributos y métodos que crea conveniente (al menos deberá añadir los métodos "hijo" y "coste" para el algoritmo de "Ramificación y Poda"):

```
public class AsignaciónParcial {

  /**
   * Cada AsignaciónParcial contiene referencias a todos los
   * pisos y a todos los compradores existentes en la inmobiliaria
   * para poder utilizarlas en sus métodos
   */
  ArrayList<Piso> todosLosPisos;
  ArrayList<Comprador> todosLosCompradores;

  /**
   * Asignación de pisos a compradores. El piso i-ésimo
   * se encuentra asignado al comprador i-ésimo.
   */
  private ArrayList<Piso> pisos = new ArrayList<Piso>();
  private ArrayList<Comprador> compradores =
    new ArrayList<Comprador>();
```

```
    /**
     * Crea una asignación parcial vacía (no incluye ninguna
     * asignación entre compradores y pisos)
     * @param todosLosPisos
     * @param todosLosCompradores
     */
    public AsignaciónParcial(ArrayList<Piso> todosLosPisos,
        ArrayList<Comprador> todosLosCompradores) { ... }

    /**
     * Indica si esta asignación parcial es ya una solución final:
     * no es posible hacer más asignaciones puesto que ya se han
     * asignado todos los pisos o todos los compradores tienen piso
     * asignado.
     * @return true si es una solución final y false en caso
     * contrario
     */
    public boolean esSoluciónFinal() { ... }

    /**
     * Retorna si el piso se encuentra ya asignado a algún
     * comprador en esta asignación parcial
     * @param p piso
     * @return true si el piso ya se encuentra asignado y false en
     * caso contrario
     */
    public boolean yaAsignado(Piso p) { ... }

    /**
     * Retorna si el comprador ya tiene asignado algún piso en esta
     * asignación parcial
     * @param c comprador
     * @return true si el comprador ya tiene asignado piso y false
     * en caso contrario
     */
    public boolean yaAsignado(Comprador c) { ... }

    /**
     * Retorna el número de características satisfechas en las
     * asignaciones comprador-piso ya existentes en esta asignación
     * parcial
     * @return número de características satisfechas en esta
     * asignación parcial
     */
    public int característicasSatisfechas() { ... }

}
```

4) (1 punto) Escribir una función Haskell (junto con su cabecera) que implemente el algoritmo de "búsqueda binaria". La función deberá retornar la posición en la que se encuentra el elemento buscado o la posición en que debería insertarse el elemento buscado en el caso de que éste no esté en la lista. Ejemplos de invocación

```
busqueda_binaria 1 [1,2,3,8,9,12] ----------------------> 0
busqueda_binaria 9 [1,2,3,8,9,12] ----------------------> 4
busqueda_binaria 0 [1,2,3,8,9,12] ----------------------> 0
busqueda_binaria 4 [1,2,3,8,9,12] ----------------------> 3
```

Para su implementación podría ser necesario utilizar el operador '!!' (visto en teoría) y las funciones estándar Haskell:

- `div`: división entera
  `div 5 2 --------------------> 2`
- `take`: crea una lista con los primeros `n` elementos de la lista original
  `take 4 [1,2,3,8,9,12] ------> [1,2,3,8]`
- `drop`: crea una lista descartando los primeros `n` elementos de la lista original
  `drop 4 [1,2,3,8,9,12] ------> [9,12]`