Examen de Programación I
(Ingeniería Informática)

Septiembre 2008

Primera parte (4 puntos, 40% nota del examen)

1) Escribir un diseño iterativo para la siguiente especificación:

método desviacionEstandar (real[1..N] x, real media) retorna real
{Pre: N>=1}
calcula la desviación estándar
{Post: valor retornado= $\sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x[i] - media)^2}$ }
fmétodo

Para calcular el sumatorio el método hace un recorrido de la tabla usando una variable de control *i* que varía entre 1 y *N* (el bucle finaliza para *i*=*N*+1). Usa una variable llamada *suma* que va acumulando el valor del sumatorio para los términos hasta el *i*-ésimo. Utilizar para el diseño iterativo el siguiente invariante:

1<=*i*<=*N*+1, *suma* contiene la suma de los términos $(x[j] - media)^2$ para todos los valores de la tabla *x* desde el 1 hasta el *i*-1

Indicar los diferentes pasos del diseño iterativo (inicializaciones, condición de permanencia en el bucle, cuerpo del bucle, y cota) así como las operaciones posteriores al bucle.

2) Se dispone de una clase descrita mediante la siguiente especificación:

```
clase  Articulo
  atributos
     real precioConIVA  //en euros
     real precioSinIVA   //en euros
  fatributos
fclase
```

Escribir la especificación y diseño de un método de esa clase que calcula el tipo de IVA del artículo y retorna un carácter que será:

- 'S': si el IVA es *superreducido* (4%)
- 'R': si el IVA es *reducido* (7%)
- 'G': si el IVA es *general* (16%)
- 'E': en otros casos

El tipo de IVA es *superreducido* si el valor absoluto de la diferencia entre el `precioConIVA` y el `precioSinIVA`*1.04 es menor que un céntimo de euro (0.01). Es *reducido* si se cumple la condición anterior pero usando un factor 1.07 para multiplicar el `precioSinIva`. Y es *general* si se cumple la condición usando un factor 1.16.

3) Utilizando el esquema de búsqueda en tablas visto en clase, escribir una especificación y un diseño para un método al que se le pasa una tabla con números reales, y que retorna el índice de la primera casilla de la tabla que cumple la propiedad de que su valor tiene un

signo diferente al de la casilla precedente y a la posterior. Si ninguna casilla de la tabla cumple esa propiedad, retornar -1. Suponer que el valor cero tiene signo positivo.

Observar que, con respecto al esquema de búsqueda, hay que modificar el rango de casillas visitadas de la tabla, para recorrer sólo las casillas de la segunda a la penúltima. Para la primera y última casillas la propiedad mencionada no tiene sentido.

4) Diseñar un método recursivo que permita determinar si un texto que se le pasa como parámetro es simétrico, es decir que se puede leer de la misma manera de izquierda a derecha o de derecha a izquierda (por ejemplo, `"abccba"`). Para este diseño el caso directo ocurre cuando el texto es de longitud menor que 2, y se considera siempre simétrico. En el caso recursivo se considera simétrico aquel texto cuyo primer y último caracteres son iguales, y en el que el fragmento de texto formado por el resto de caracteres (entre el segundo y el penúltimo) es simétrico. Esta última condición se obtiene llamando recursivamente al método.

   Suponer que el texto es un objeto que dispone de la operaciones:

   método car(entero n) retorna caracter: retorna el carácter n del texto (suponer que el primer caracter es el n=0)
   método longitud() retorna entero: retorna el número de caracteres del texto
   método fragmento(entero i, entero j) retorna texto: retorna un texto nuevo cuyos caracteres son iguales a los comprendidos entre el i (incluido) y el j (excluido) en el texto original.

Examen de Programación I
(Ingeniería Informática)

Febrero 2008

Segunda parte (6 puntos, 60% nota del examen)

Se desea escribir una clase que proporcione información sobre una ruta de carretera realizada entre diferentes localidades. Para ello se dispone de dos clases ya realizadas. La clase `Tramo` describe un tramo de carretera entre dos localidades y contiene métodos para saber la localidad de origen, la localidad de destino y la distancia a recorrer:

```
public class Tramo
{
    /**
     * Retorna el origen
     */
    public String origen() {...}

    /**
     * Retorna el destino
     */
    public String destino() {...}

    /**
     * Retorna la distancia
     */
    public double distancia() {...}
}
```

La clase `Ruta` contiene una ruta descrita mediante una secuencia de tramos de carretera (objetos de la clase `Tramo`), y sigue la interfaz de la secuencia vista en clase:

```
public class Ruta {
    /**
     * Constructor que crea la ruta vacía, con un máximo de tramos indicado
     */
    public Ruta(int max)   {...}

    /**
     * Reinicia
     */
    public void reinicia()   {...}

    /**
     * Obtiene el Tramo actual
     */
    public Tramo actual()  {...}

    /**
     * Avanza al siguiente tramo
     */
    public void avanza()  {...}

    /**
     * Fin de secuencia
     */
    public boolean fds()  {...}

    /**
     * Inserta un tramo al final de la ruta
     * Retorna true si lo ha podido hacer y false si no cabe el tramo
     */
    public boolean inserta(Tramo t)  {...}
}
```

Lo que se pide es escribir la clase `Viaje` con operaciones que trabajan con una `Ruta`, y que responda a la siguiente interfaz:

```
public class Viaje {
    // el atributo es una ruta
    private Ruta ruta;

    /**
     * Constructor al que se le pasa la ruta
     */
    public Viaje(Ruta r)
    {
        ruta = r;
    }

    /**
     * Retorna si la ruta almacenada es correcta o no. La ruta se considera
     * correcta cuando tiene al menos un tramo y el destino de cada tramo
     * coincide con el origen del siguiente
     */
    public boolean esCorrecto() {...}

    /**
     * Indica si la ruta pasa por la localidad indicada
     * Para ello se comprueba si ciudad es el origen
     * o destino de alguno de los tramos almacenados en la ruta
     */
    public boolean pasaPor(String ciudad) {...}

    /**
     * Retorna la distancia recorrida en la ruta
     * Se calcula sumando las distancias de cada tramo almacenado
     */
    public double distancia() {...}

    /**
     * Retorna la distancia recorrida para llegar por la ruta almacenada
     * desde la localidad origen a la localidad destino. Se calcula sumando
     * las distancias de cada tramo almacenado desde la localidad origen a
     * la localidad destino. Si la ruta no pasa por el origen o no pasa
     * por el destino, retorna -1.0
     */
    public double distanciaEntre(String origen, String destino) {...}

}
```

Hay que escribir los métodos que están incompletos, teniendo en cuenta lo siguiente:

- `pasaPor` (1 punto): Usar el esquema de búsqueda en secuencias.
- `distancia` (1 punto): Usar el esquema de recorrido en secuencias.
- `esCorrecto` (2 puntos): almacenar en una variable llamada `anterior` el destino de la localidad anterior en la ruta; inicializar esa variable al destino del primer tramo de la ruta; luego, recorrer los tramos desde el segundo al último comprobando que el origen del tramo actual coincide con el destino del tramo anterior y actualizando a cada paso la variable `anterior`. Finalizar el recorrido si se encuentra un tramo incorrecto.
- `distanciaEntre` (2 puntos): Primero comprobar si el origen y el destino están en la ruta. En caso afirmativo, recorrer la secuencia y usar una variable booleana llamada `encontradoOrigen` para saber si ya hemos encontrado el origen o no. Así sabremos si debemos ir sumando distancias de tramos. Finalizar el recorrido al llegar al destino.

Nota: para comparar localidades usar el método `equalsIgnoreCase` de la clase `String`, que no diferencia mayúsculas de minúsculas al comparar.