Examen de Programación I
(Ingeniería Informática)

Septiembre 2009

Primera parte (5 puntos, 50% nota del examen)

1) Escribir un *diseño iterativo* para un método que calcula el arco tangente de x de acuerdo al desarrollo en serie que aparece en la postcondición de esta especificación:

método arcoTangente (real x, entero n) retorna real
{Pre: |x|<1, n>0}
Calcula el arco tangente
{Post: valor retornado= $\sum_{i=0}^{n} \frac{(-1)^i}{2i+1} x^{2i+1}$ }
fmétodo

Indicar para este diseño lo siguiente:

- el invariante
- las variables a crear y sus inicializaciones
- la condición de permanencia en el bucle
- el cuerpo del bucle
- la función de cota, razonando sobre si el bucle termina
- el estado a la terminación, con las instrucciones a realizar después del bucle

Poner además el pseudocódigo completo. Para una mayor eficiencia, intentar evitar el uso de la función x^y.

2) Escribir la *especificación* de dos métodos que calculen y retornen las soluciones a un sistema de dos ecuaciones y dos incógnitas x,y (un método retorna la solución de x y otro la de y). El sistema de ecuaciones es:

$$ax + by = e$$
$$cx + d = f$$

Y las soluciones son:

$$x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

3) Escribir un *diseño recursivo* para un método que presente en pantalla las sucesivas raices cuadradas de un número real x superior a la unidad que se le pasa como parámetro, hasta llegar a un número inferior a `1.000001`. En el caso directo, que es cuando x<`1.000001`, basta mostrar x en la pantalla. En el caso recursivo se muestra el valor de x en la pantalla y luego se invoca recursivamente al mismo método con un parámetro igual a la raíz cuadrada de x. La especificación del método es:

método muestraRaices(real x)
{Pre: x>1}
muestra raices

{Post: se han mostrado en pantalla x, $\sqrt{x}$, $\sqrt{\sqrt{x}}$, ... hasta que el último de estos valores es <1.000001}
fmétodo

4) Escribir el diseño del siguiente método, que busca la primera casilla de la secuencia *sec* que cumple que la diferencia de esa casilla con la anterior es superior a *d*:

```
método buscaDiferenciaSuperior(Secuencia<real> sec, real d)
        retorna encontrado:booleano
{Pre:}
  Busca una diferencia entre la casilla actual y la anterior superior a d
{Post: encontrado= algún elemento de sec posterior al primero
                   cumple la propiedad P; siendo
                   P=la diferencia entre el valor de la casilla y la anterior >d,
      si encontrado entonces ningún elemento de
           pi(sec) posterior al primero cumple P y sec.actual() cumple P,
      si no encontrado entonces pd(sec)=vacia; }
```

Nota: Usar una variable para almacenar el valor de la casilla anterior de la secuencia y hacer la búsqueda entre la segunda casilla de la secuencia y la última.

Examen de Programación I
(Ingeniería Informática)

Septiembre 2009

Segunda parte (5 puntos, 50% nota del examen)

Se dispone de una clase llamada `Fecha` que permite almacenar una fecha compuesta por el día, mes y año. Dispone de una operación, `numDias`, para calcular el número de días entre dos fechas. La interfaz de esta clase es:

```
public class Fecha {
    /**
     * Constructor al que se le pasa el día, mes y año
     */
    public Fecha(int dia, int mes, int año)  {...}

    /**
     * Retorna el número de días entre la fecha del objeto actual y la fecha
     * del objeto f (positivo si f es posterior a actual)
     */
    public int numDias(Fecha f) {...}
}
```

Se desea implementar otra clase llamada `HistoriaConsumos` para almacenar el historial de consumos eléctricos de un cliente. Este historial consiste en guardar lecturas del contador eléctrico, cada una junto a la fecha en que se realizó. Para ello la clase guarda en un array, `kwh`, las lecturas del contador en kilovatios hora, y en otro array, `fecha`, la fecha de cada lectura. Estos arrays tienen una estructura "paralela", de modo que la lectura número `i` se guarda en `kwh[i]` (la lectura del contador) y en `fecha[i]` (la fecha de esa lectura). Se suponen las fechas ordenadas de modo que la fecha `i` es anterior a la `i+1`.

Ambas tablas son de tamaño variable y se van rellenando desde las casillas de índices más bajos. El atributo entero `num` indica cuántas lecturas válidas hay, de modo que los arrays tienen lecturas válidas entre las casillas `0` a `num-1`. El resto de las casillas no tienen valores válidos.

La interfaz de esta clase es la siguiente:

```
public class HistoriaConsumos {

    // Dos arrays paralelos: kwh[i] se leyó en la fecha fecha[i]
    private double[] kwh; // lectura del contador en kilovatios-hora
    private Fecha[] fecha; // fecha de lectura del contador
    private int num; // numero de lecturas del contador válidas

    /**
     * Constructor que crea el historial de consumos vacío,
     * pero con espacio para guardar un número máximo de lecturas
     * del contador igual a max
     */
    public HistoriaConsumos(int max) {...}
```

```
    /**
     * Añade una lectura del contador si se puede
     */
    public void añadeLectura(double lectura, int dia, int mes, int año) {...}

    /**
     * Calcula y retorna el número de días con consumo medio superior
     * a c kilovatios-hora
     */
    public int numDiasConsumoSuperiorA(double c) {...}

    /**
     * Calcula y retorna el consumo medio diario en torno a la fecha f
     */
    public double consumoMedioEnFecha(Fecha f) {...}

}
```

Se pide implementar en Java los métodos de la clase según lo descrito en sus comentarios de documentación, y teniendo en cuenta lo siguiente:

1) Constructor (0.5 puntos): Crea los dos arrays con el tamaño indicado por `max` y pone `num` a cero.

2) `añadeLectura()` (1 punto): Si `num` es igual al tamaño de uno de los arrays la nueva lectura no cabe, por lo que se pone un mensaje de error en pantalla. En caso contrario se incrementa `num` y se mete la nueva lectura y su fecha en las casillas `num-1` de ambos arrays.

3) `numDiasConsumoSuperiorA()` (1.5 puntos): Si `num` es menor que 2 no se puede calcular el número de días, y se retorna el valor 0. En caso contrario se calculará y retornará el número total de días en los que el consumo medio es superior a `c`. Para ello se recorre (parcialmente) la tabla de consumos entre la primera y penúltima casillas válidas. Para cada casilla `i` se calcula el consumo como la diferencia de la lectura del contador de la casilla `i+1` menos la de la `i`. Asimismo se calcula el número de días entre la fecha de la casilla `i` y la de la `i+1`. Si el consumo dividido entre el número de días es superior a `c`, el número de días se añade a una variable que guarda el número total de días de consumo superior a `c`. Finalizado el recorrido, se retorna el valor de esa variable.

4) `consumoMedioEnFecha()` (2 puntos): Si el `num` es menor que 2 o si `f` es anterior a la primera fecha almacenada o posterior a la última, se retorna `-1.0` para indicar que no se puede realizar el cálculo. En caso contrario, hay que calcular el consumo medio entre las dos fechas consecutivas de la tabla que tienen a `f` entre ambas. Para ello se usa el esquema de búsqueda en tablas para buscar la casilla `i` que cumple que `fecha[i]` es anterior o igual a `f` y `fecha[i+1]` es posterior a `f`. Una vez encontrada esa casilla, se retornará la diferencia del consumo entre las lecturas `i+1` e `i` dividida entre el número de días transcurridos entre ambas lecturas.

Nota: para saber si una fecha es anterior o posterior a otra se puede usar el método `numDias()` y ver si el número de días transcurridos entre una fecha y otra es positivo o negativo.