Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 26 de Junio de 2013

- Disponer sobre la mesa en lugar visible un ***documento de identificación*** provisto de fotografía. Escribir ***nombre y dos apellidos*** en cada una de las hojas de papel que haya sobre la mesa.
- Comenzar a resolver cada una de las partes del examen ***en una hoja diferente*** para facilitar su corrección por profesores diferentes.
- El tiempo total previsto para realizar el examen es de ***dos horas y media***. No está permitido consultar libros ni apuntes, excepto los siguientes documentos: *Breve resumen del lenguaje Java* y *Capítulo 1 de las Notas del curso*.
- Está prohibido disponer en el aula de examen de teléfonos móviles o de otros dispositivos de comunicación, ni apagados ni encendidos.

Problema 1º (4.5 puntos)

Dado el método *binarizar(double[], int[])* cuyo código se muestra más adelante se pide:

1. Escribir un invariante del bucle **while** que permita demostrar la corrección del código que precede al bucle, del código posterior al bucle y del código a iterar. [1 punto]
2. A partir del invariante anterior, escribir las pruebas que permiten demostrar la corrección de la secuencia de instrucciones que precede al bucle. [0.5 puntos]
3. A partir del mismo invariante, escribir las pruebas que permiten demostrar la corrección de la instrucción posterior al bucle. [0.5 puntos]
4. A partir del mismo invariante, escribir las pruebas que permiten demostrar la corrección del código del bloque a iterar en el bucle **while**. [2 puntos]
5. Escribir las pruebas que permiten demostrar la terminación del bucle **while**. [0.5 puntos]

/* **Pre**: $T.length > 0 \wedge T.length = B.length$ */
/* **Post**: $(\forall \alpha \in [0, T.length-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \wedge (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge\ binarizar(T, B) = (\Sigma\alpha \in [0, B.length-1].B[\alpha])$ */

```
public static int binarizar (double[] T, int[] B) {
    int i = 0;
    int cuenta = 0;
    while ( i != T.length ) {
        if ( T[i] >= 0.0 ) {
            B[i] = 1; cuenta = cuenta + 1;
        }
        else {
            B[i] = 0;
        }
        i = i + 1;
    }
    return cuenta;
}
```

Problema 2º (2.5 puntos)

Dado el siguiente método:

```
/* Pre: n ≥ 1 */
/* Post: (n < 10 → num(n) = n/10.0)
        ∧ (n ≥ 10 → num(n) = num(n − 2) + 2.5 × num(n − 1)) */
public static double num (int n) {
|   if ( n < 10 ) {
|   |   return (double) n/10.0;
|   }
|   else {
|   |   return num(n-2) + 2.5*num(n-1);
|   }
}
```

Se pide caracterizar asintóticamente el coste, en tiempo y en memoria, de la invocación *num(p)*, en función del valor del argumento *p*. Se valorará el planteamiento de las ecuaciones que determinan el coste de la invocación, la claridad de su resolución, los resultados obtenidos y la propiedad de las explicaciones complementarias aportadas.

Problema 3º (3.0 puntos)

Dada la especificación del siguiente método:

```
/* Pre: T.length > 0 ∧ T.length = S.length */
/* Post: prodEscalar(T, S) = (Σα ∈ [0.T.length − 1].T[α] ∗ S[α]) */
public static double prodEscalar (double[] T, double[] S) {
|   ...    Código del método    ...
}
```

Se pide realizar los diseños que se indican a continuación indicando, para cada uno de ellos, la técnica de diseño aplicada:

1. Un primer diseño sin bucles del método *prodEscalar(**double[]**, (**double[]**)* mediante un diseño recursivo por inmersión por refuerzo de la precondición. Se valorará esencialmente la adecuada especificación del método o métodos auxiliares que asegure la corrección del código. [**1.5 puntos**]

2. Un nuevo diseño sin bucles del método *prodEscalar(**double[]**, (**double[]**)* mediante un diseño recursivo por inmersión por debilitamiento de la postcondición. Se valorará esencialmente la adecuada especificación del método o métodos auxiliares que asegure la corrección del código. [**1.5 puntos**]

Una solución del problema 1º

1. Invariante del bucle **while** que permite demostrar la corrección del código del método:

   Inv: $i \geq 0 \ \land \ i \leq T.length \ \land \ T.length = B.length$
   $\land \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \land \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
   $\land \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$

2. Prueba de la corrección de la secuencia de dos instrucciones que preceden al bucle **while**, a partir del invariante **Inv** del bucle:

   /* $T.length > 0 \ \land \ T.length = B.length$
   $\Rightarrow$
   $0 \geq 0 \ \land \ 0 \leq T.length \ \land \ T.length = B.length$
   $(\forall \alpha \in [0, 0-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \land \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
   $\land \ 0 = (\Sigma \alpha \in [0, 0-1].B[\alpha])$ */
   int i = 0;
   int cuenta = 0;
   /* **Inv**: $i \geq 0 \ \land \ i \leq T.length \ \land \ T.length = B.length$
   $\land \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \land \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
   $\land \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$ */

3. Prueba de la corrección de la última instrucción ***return*** *cuenta* a partir del invariante **Inv** del bucle **while**:

   /* $i = T.length \ \land \ T.length = B.length$
   $\land \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \land \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
   $\land \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$
   $\Rightarrow$
   $(\forall \alpha \in [0, T.length-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \land \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
   $\land \ cuenta = (\Sigma \alpha \in [0, B.length-1].B[\alpha])$ */
   return *cuenta;*
   /* $(\forall \alpha \in [0, T.length-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \land \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
   $\land \ binarizar(T, B) = (\Sigma \alpha \in [0, B.length-1].B[\alpha])$ */

4. Demostración de la terminación del bucle **while**. Para ello se va a considerar la función de cota $f_{cota} = T.length - i$.

   while ($i! = T.length$) {
   | /* $i \geq 0 \ \land \ i < T.length \ \land \ i = X \ \land \ f_{cota} = T.length - X$ */
   | **if** ($T[i] >= 0.0$) {
   | | B[i] = 1; cuenta = cuenta + 1;
   | }
   | **else** {
   | | B[i] = 0;
   | }
   | i = i + 1;
   | /* $i \geq 0 \ \land \ i \leq T.length \ \land \ i = X + 1 \ \land \ f_{cota} = T.length - X - 1$ */
   }

   La demostración de la terminación del bucle se sustenta en dos pruebas:

- El valor de la función de cota, $f_{cota} = T.length - i$, **disminuye en cada iteración**. En efecto: $T.length - X > T.length - X - 1$ ya que el valor de la variable i crece en una unidad al ejecutar el bloque a iterar.
- **La secuencia de valores decrecientes que toma la función de cota es finita**. En efecto, basta observar la condición $i \leq T.length$ que impone el invariante **Inv** del bucle, la cual determina que se satisfaga que $f_{cota} = T.length - i \leq 0$. Es decir, la función de cota está acotada inferiormente por el valor 0.

5. Prueba de la corrección del código del bloque a iterar asociado al bucle **while**, a partir del invariante **Inv**:

/* $i \geq 0 \ \wedge \ i < T.length \ \wedge \ T.length = B.length$
$\wedge \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$ */
if ($T[i] >= 0.0$ **)** {
/* $i \geq 0 \ \wedge \ i < T.length \ \wedge \ T.length = B.length \ \wedge \ T[i] \geq 0.0$
$\wedge \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$
$\Rightarrow$
$i + 1 \geq 0 \ \wedge \ i < T.length \ \wedge \ T.length = B.length$
$\wedge \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ (T[i] \geq 0.0 \rightarrow 1 = 1) \ \wedge \ (T[i] < 0.0 \rightarrow 1 = 0)$
$\wedge \ cuenta + 1 = (\Sigma \alpha \in [0, i-1].B[\alpha]) + 1$ */
B[i] = 1; cuenta = cuenta + 1;
}
else {
/* $i \geq 0 \ \wedge \ i < T.length \ \wedge \ T.length = B.length \ \wedge \ T[i] < 0.0$
$\wedge \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$
$\Rightarrow$
$i + 1 \geq 0 \ \wedge \ i < T.length \ \wedge \ T.length = B.length$
$\wedge \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ (T[i] \geq 0.0 \rightarrow 0 = 1) \ \wedge \ (T[i] < 0.0 \rightarrow 0 = 0)$
$\wedge \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha]) + 0$ */
B[i] = 0;
}
/* $i + 1 \geq 0 \ \wedge \ i + 1 \leq T.length \ \wedge \ T.length = B.length$
$\wedge \ (\forall \alpha \in [0, i+1-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ cuenta = (\Sigma \alpha \in [0, i+1-1].B[\alpha])$ */
i = i + 1;
/* **Inv**: $i \geq 0 \ \wedge \ i \leq T.length \ \wedge \ T.length = B.length$
$\wedge \ (\forall \alpha \in [0, i-1].(T[\alpha] \geq 0.0 \rightarrow B[\alpha] = 1) \ \wedge \ (T[\alpha] < 0.0 \rightarrow B[\alpha] = 0))$
$\wedge \ cuenta = (\Sigma \alpha \in [0, i-1].B[\alpha])$ */

Una solución del problema 2º

- **Caracterización asintótica del coste en tiempo**

  La ecuación recurrente que rige el coste en tiempo de una invocación *num(p)* para un valor de *p* suficientemente grande es:

  $$t(p) = k_1 + t(p-2) + t(p-1)$$

  Donde k_1 representa la suma de costes constantes de una invocación con $p \geq 10$.

  Reordenamos la ecuación recurrente:

  $$t(p) - t(p-1) - t(p-2) = k_1 \times 1^p$$

  Escribimos la ecuación característica:

  $$(x^2 - x - 1)(x-1) = 0$$

  Cuyas raíces son:

  $$x = (1+\sqrt{5})/2, \quad x = (1-\sqrt{5})/2 \quad \text{y} \quad x = 1$$

  La solución general de la recurrencia es por lo tanto:

  $$\begin{aligned} t(p) &= c_1 \times ((1+\sqrt{5})/2)^p + c_2 \times ((1-\sqrt{5})/2)^p + c_3 \times 1^p \\ &= c_1 \times ((1+\sqrt{5})/2)^p + c_2 \times ((1-\sqrt{5})/2)^p + c_3 \end{aligned}$$

  Y la caracterización asintótica del coste en tiempo de una invocación *num(p)* será:

  $$\boldsymbol{O}(t(p)) = \boldsymbol{O}(c_1 \times ((1+\sqrt{5})/2)^p + c_2 \times ((1-\sqrt{5})/2)^p + c_3) = \boldsymbol{O}((1+\sqrt{5})/2)^p)$$

  El coste en tiempo es exponencial en el valor *p* del parámetro del método.

- **Caracterización asintótica del coste en memoria**

  La ecuación recurrente que rige el coste en memoria de una invocación *num(p)* para un valor de *p* suficientemente grande es:

  $$mem(p) = mem_{inv} + mem_{int} + mem_{double} + mem(p-1)$$

  Donde mem_{int} y mem_{double} denotan la cantidad de memoria necesaria para almacenar un dato de tipo **int** y de tipo **double**, respectivamente. Y mem_{inv} la cantidad de memoria precisa para gestionar una invocación (dirección de rotorno, enlace estático y enlace dinámico).

  Reordenamos la ecuación recurrente:

  $$mem(p) - mem(p-1) = (mem_{inv} + mem_{int} + mem_{double}) \times 1^p$$

  Escribimos la ecuación característica:

  $$(x-1)(x-1) = 0$$

  Cuyas raíces son:

  $$x = 1 \ \text{(raíz doble)}$$

  La solución general de la recurrencia es por lo tanto:

  $$mem(p) = c_1 \times p \times 1^p + c_2 \times 1^p = c_1 \times p + c_2$$

  Y la caracterización asintótica del coste en memoria de una invocación *num(p)* será:

  $$\boldsymbol{O}(mem(p)) = \boldsymbol{O}(c_1 \times p + c_2) = \boldsymbol{O}(p)$$

  El coste en memoria es lineal en el valor *p* del parámetro del método.

Una solución del problema 3º

Código del método *prodEscalar(double[], double[])* diseñado por inmersión mediante refuerzo de la precondición.

```
/**
 * Pre:  T.length>0 AND T.length=S.length
 * Post:  prodEscalar(T,S)  = (SIGMA alfa EN [0,T.length−1].T[alfa]*S[alfa])
 */
public static double prodEscalar(double[] T, double[] S) {
   return prodEscalar (T, S, 0, 0.0);
}

/**
 * Pre:  T.length>0 AND T.length=S.length AND hasta>=0 AND hasta<=T.length AND
 *        prod = (SIGMA alfa EN [0,hasta−1].T[alfa]*S[alfa])
 * Post:  prodEscalar(T,S, hasta, prod)  = (SIGMA alfa EN [0,T.length−1].T[alfa]*S[alfa])
 */
private static double prodEscalar(double[] T, double[] S, int hasta, double prod) {
    if (hasta<T.length) {
        return prodEscalar(T, S, hasta+1, prod + T[hasta]*S[hasta]);
    }
    else {
        return prod;
    }
}
```

Código del método *prodEscalar(double[], double[])* diseñado por inmersión mediante debilitamiento de la postcondición.

```
/**
 * Pre:  T.length>0 AND T.length=S.length
 * Post:  prodEscalar(T,S)  = (SIGMA alfa EN [0,T.length−1].T[alfa]*S[alfa])
 */
public static double prodEscalar(double[] T, double[] S) {
   return prodEscalar(T, S, T.length−1);
}

/**
 * Pre:  T.length>0 AND T.length=S.length AND hasta>=0 AND hasta<T.length
 * Post:  prodEscalar(T,S,hasta)  = (SIGMA alfa EN [0,hasta].T[alfa]* S[alfa])
 */
private static double prodEscalar(double[] T, double[] S, int hasta) {
    if (hasta == 0) {
        return T[0]*S[0];
    }
    else {
        return T[hasta]*S[hasta]+ prodEscalar(T, S, hasta−1);
    }
}
```