Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas

Examen de Programación 2 4-Septiembre-2012

- 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 documentos facilitados por los profesores de la asignatura.

Los tres métodos que se presentan a continuación resuelven el mismo problema; calculan y devuelven el valor de 2^n.

```
/**
 * Pre: n>=0
 * Post: pot2_A(n) = 2^n
 */
private static long pot2_A (int n) {
    if (n==0) { return 1; }
    else { return 2*pot2_A(n-1); }
}

/**
 * Pre: n>=0
 * Post: pot2_B(n) = 2^n
 */
private static long pot2_B (int n) {
    if (n==0) { return 1; }
    else { return pot2_B(n-1) + pot2_B(n-1); }
}

/**
 * Pre: n>=0
 * Post: pot2_C(n) = 2^n
 */
private static long pot2_C (int n) {
    if (n==0) { return 1; }
    else {
        long r = pot2_C(n/2);
        /* A */
        if (n%2==0){ return r*r; }
        else { return 2*r*r; }
    }
}
```

Problema 1º [3 puntos]

Hacer un análisis comparativo de los costes de los tres métodos anteriores que incluya:

- **Caracterización asintótica del coste de cada uno de ellos en función del valor de su parámetro n**. Deben detallarse, paso a paso, las ecuaciones y cálculos que conduzcan a cada resultado.
- **Análisis comparativo de su eficiencia y conclusiones sobre cuál de ellos conviene utilizar**. Se valorará en este apartado la calidad y precisión técnica del informe comparativo y la claridad y corrección de su redacción.

Problema 2º [3 puntos]

1. Probar formalmente la corrección del método **pot2_C** detallando con claridad el conjunto de cálculos y pruebas formales que constituyen la demostración. [2.5 puntos]

   Sugerencia: Como primer paso escribir el axioma **A** que ha de satisfacerse tras calcular el valor de la variable **r**. Este predicado tiene la forma $A \equiv n \geq 0 \wedge r = ?$

2. Una invocación *pot2_C(-6)* devuelve como resultado el valor *64* y una invocación *pot2_C(-7)* devuelve como resultado el valor *128*. ¿Significa ello que el método **pot2_C** no es correcto? ¿Es correcto o no lo es? Aclararlo explicando con una terminología adecuada y una buena redacción las razones que fundamentan la respuesta. [0.5 puntos]

Problema 3º [3 puntos]

El método *comprobar* devuelve el valor de un índice de la tabla **T** que almacena un dato igual al del siguiente elemento de la tabla, salvo que no haya elementos consecutivos iguales en la tabla, en cuyo caso devuelve un valor negativo.

```
/**
 * Pre: T.length>0
 * Post: (comprobar(T)<0 -> (PT alfa EN [0,T.length-2].T[alfa]!=T[alfa+1]))
 *       AND (comprobar(T)>=0 -> T[comprobar(T)]=T[comprobar(T)+1])
 */
private static int comprobar (int [] T)
```

En este problema se pide:

- Hacer un primer diseño sin bucles del método *comprobar* aplicando un diseño recursivo por inmersión mediante ***debilitamiento de la postcondición***.
- Hacer un nuevo diseño sin bucles del método *comprobar* aplicando un diseño recursivo por inmersión mediante ***refuerzo de la precondición***, manteniendo constante la postcondición.

En ambos diseños se valorarán esencialmente los siguientes aspectos: (1) Que, en cada caso, se haya **aplicado el método de diseño pedido**. (2) Que los **métodos auxiliares** que sea preciso desarrollar vengan **especificados formalmente**. (3) Que **el diseño de cada uno de los métodos escritos sea correcto** respecto de su especificación.

Problema 4º [1 punto]

Escribir un predicado invariante **INV** del bucle que sea suficientemente fuerte como para sustentar el conjunto de pruebas que constituyen la demostración formal de la corrección de la siguiente instrucción iterativa.

```
/* !acabar ∧ i = T.length − 1 */
while ( !acabar ) {
    /* Predicado invariante : INV */
    if ( T[i] == x ) {
        acabar = true;
    }
    else {
        acabar = (i==0);  i = i - 1;
    }
}
/* ((∀α ∈ [0, T.length − 1].T[α] ≠ x)  →  i < 0)
   ∧ ((∃α ∈ [0, T.length − 1].T[α] = x)  →  T[i] = x) */
```

Nota: No se pide ninguna de las pruebas que constituyen las demostración de la corrección de la instrucción iterativa anterior, sólo se debe escribir un predicado invariante del bucle con las características mencionadas.

Una solución del problema 1º

- Caracterización asintótica de la función de coste $t_A(n)$ de una invocación ***pot2_A(n)***.

  Ecuación recurrente:

  $t_A(n) = k + t_A(n-1)$

  $t_A(n) - t_A(n-1) = k.\,1^n$

  Ecuación característica y sus raices:

  $(x-1)^2 = 0$ Raíces: $x = 1$ (doble)

  Solución general de la recurrencia:

  $t_A(n) = c_1.n,1^n + c_2.\,1^n = c_1.n + c_2$

  Caracterización asintótica de la función de coste:

  $\boldsymbol{O}(t_A(n)) = \boldsymbol{O}(c_1.n + c_2) = \boldsymbol{O}(n)$

- Caracterización asintótica de la función de coste $t_B(n)$ de una invocación ***pot2_B(n)***.

  Ecuación recurrente:

  $t_B(n) = k + 2.\,t_B(n-1)$

  $t_B(n) - 2.\,t_B(n-1) = k.\,1^n$

  Ecuación característica y sus raices:

  $(x-2)(x-1) = 0$ Raíces: $x = 2$ y $x = 1$

  Solución general de la recurrencia:

  $t_B(n) = c_1.\,2^n + c_2.\,1^n = c_1.\,2^n + c_2$

  Caracterización asintótica de la función de coste:

  $\boldsymbol{O}(t_B(n)) = \boldsymbol{O}(c_1.\,2^n + c_2) = \boldsymbol{O}(2^n)$

- Caracterización asintótica de la función de coste $t_C(n)$ de una invocación ***pot2_C(n)***.

  Ecuación recurrente (idéntica para las dos cláusulas de la instrucción alternativa más interna):

  $t_C(n) = k + t_C(n/2)$

  $t_C(n) - t_C(n/2) = k$

  Cambio de variable $m = log_2\, n$ (es decir, $n = 2^m$):

  $t_C(m) - t_C(m-1) = k.\,1^m$

  Ecuación característica y sus raices:

  $(x-1)^2 = 0$ Raíces: $x = 1$ (doble)

  Solución general de la recurrencia:

  $t_C(m) = c_1.\,m.\,1^m + c_2.\,1^m = c_1.\,m + c_2$

  Deshacemos el cambio de variable:

  $t_C(n) = c_1.\,log_2\, n + c_2$

  Caracterización asintótica de la función de coste:

  $\boldsymbol{O}(t_C(n)) = \boldsymbol{O}(c_1.\,log_2\, n + c_2) = \boldsymbol{O}(log\, n)$

- Análisis comparativo de la eficiencia de los cuatro métodos.

  El método asintóticamente más eficiente es ***pot2_C(n)*** ya que su coste es logarítimico en el valor del parámetro **n**. Le siguen los métodos ***pot2_A(n)*** cuyo coste depende linealmente del valor del parámetro **n**. El más ineficiente es el método ***pot2_B(n)*** cuyo coste depende exponencialmente del valor del parámetro **n**.

Una solución del problema 2º

1. Prueba de la corrección del método *pot2_C*.

```
/* Pre: n ≥ 0      */
/* Post: pot2_C(n) = 2^n      */
private long pot2_C (int n) {
    /*  Función de cota: f_cota(n) = n     Su valor inicial: f_cota(inicial) = n   */
    /*  n ≥ 0     ⇒      Dom(n == 0) = true   */
    if ( n == 0 ) {
        /*  n = 0     ⇒      1 = 2^n   */
        return 1;
        /*  pot2_C(n) = 2^n   */
    else {
        /*  n > 0      ⇒      n/2 ≥ 0   */
        /*  n > 0 ∧ pot2_C(n/2) = 2^(n/2)   */
        /*  n > 0 ∧ pot2_C(n/2) = 2^(n/2)      ⇒      n > 0 ∧ pot2_C(n/2) = 2^(n/2)   */
        long r = pot_C(n/2);     /*  Valor que toma función de cota: f_cota(invocación) = n/2   */
        /*  n > 0 ∧ r = 2^(n/2)   */
        if ( n%2 == 0 ) {
            /*  n > 0 ∧ r = 2^(n/2) ∧ n%2 = 0      ⇒      r * r = 2^n   */
            return r*r;
            /*  pot2_C(n) = 2^n   */
        else {
            /*  n > 0 ∧ r = 2^(n/2) ∧ n%2 ≠ 0      ⇒      2 * r * r = 2^n   */
            return 2*r*r;
            /*  pot2_C(n) = 2^n   */
        }
        /*  pot2_C(n) = 2^n
    }
}
```

Una invocación que satisfaga la precondición ($n \geq 0$) termina ya que:

- Si se satisface la precondición, la función de cota toma inicialmente un valor cero o positivo: $n \geq 0 \Rightarrow f_{cota}(n) = n \geq 0$
- La función de cota decrece desde su valor inicial y el que toma al producirse una invocación recursiva: $n \geq 0 \Rightarrow f_{cota}(inicial) = n > f_{cota}(invocación) = n/2$
- Este decrecimiento no puede ser indefinido ya que la función de cota, cuando la invocación satisface la precondición, toma siempre valores cero o positivos como se ha probado anteriormente.

2. Aclaración y explicación sobre la corrección del método. El método **pot2_C** es correcto como acaba de ser demostrado. La corrección garantiza que el método termina rindiendo los resultados esperados siempre que su invocación satisfaga la precondición. En caso de invocarse el método con un valor negativo del parámetro **n**, es decir, con un valor que hace que la precondición no se satisfaga, el método diseñado no está obligado a rendir resultados que satisfagan la postcondición y, en caso de hacerlo, no por ello deja de estar correctamente diseñado.

Una solución del problema 3º

1. Diseño recursivo mediante debilitamiento de la postcondición.

```
/**
 * Pre: T.length>0
 * Post: (comprobar(T)<0 -> (PT alfa EN [0,T.length-2].T[alfa]!=T[alfa+1]))
 *       AND (comprobar(T)>=0 -> T[comprobar(T)]=T[comprobar(T)+1])
 */
private static int comprobar (int [] T) {
    return comprobar(T, T.length-1);
}

/**
 * Pre: T.length>0 AND hasta>=0 AND hasta<T.length
 * Post: (comprobar(T,hasta)<0 -> (PT alfa EN [0,hasta-1].T[alfa]!=T[alfa+1]))
 *       AND (comprobar(T,hasta)>=0 -> T[comprobar(T)]=T[comprobar(T)+1])
 */
private static int comprobar (int [] T, int hasta) {
    if (hasta==0) { return -1; }
    else if (T[hasta-1]==T[hasta]) { return hasta-1; }
    else { return comprobar(T, hasta-1); }
}
```

2. Diseño recursivo con postcondición constante mediante refuerzo de la precondición.

```
/**
 * Pre: T.length>0
 * Post: (comprobar(T)<0 -> (PT alfa EN [0,T.length-2].T[alfa]!=T[alfa+1]))
 *       AND (comprobar(T)>=0 -> T[comprobar(T)]=T[comprobar(T)+1])
 */
private static int comprobar (int [] T) {
    return comprobar(T, 0);
}

/**
 * Pre: T.length>0 AND (PT alfa EN [0,desde-1].T[alfa]!=T[alfa+1])
 *      AND desde>=0 AND desde<T.length
 * Post: (comprobar(T)<0 -> (PT alfa EN [0,T.length-2].T[alfa]!=T[alfa+1]))
 *       AND (comprobar(T)>=0 -> T[comprobar(T)]=T[comprobar(T)+1])
 */
private static int comprobar (int [] T, int desde) {
    if (desde==T.length-1) { return -1; }
    else if (T[desde]==T[desde+1]) { return desde; }
    else { return comprobar(T, desde+1); }
}
```

Una solución del problema 4º

```
/* !acabar ∧ i = T.length − 1 */
while ( !acabar ) {
    /* INV : i ≥ −1 ∧ i ≤ T.length − 1 ∧ (∀α ∈ [i + 1, T.length − 1].T[α] ≠ x)
                ∧ (acabar → T[i] = x ∨ i = −1)        */
    if ( T[i] == x ) {
      | acabar = true;
    }
    else {
      | acabar = (i==0);  i = i - 1;
    }
}
/* ((∀α ∈ [0, T.length − 1].T[α] ≠ x) → i < 0)
    ∧ ((∃α ∈ [0, T.length − 1].T[α] = x) → T[i] = x)  */
```