Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 15 de septiembre de 2016

- 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 dos documentos señalados en la convocatoria del examen (Guía sintáxis C++ y Apartado 1.2 de los apuntes del curso).

Problema 1º (2 puntos)

A continuación se presenta un diseño recursivo de la función **`multiplicar`**`(a,b)`.

```
/*
 * Pre: b >= 0
 * Post: multiplicar (a,b) = a * b
 */
int multiplicar (const int a, const int b) {
    if (b == 0) {                                // linea a
        return 0;                                // linea b
    }
    else {
        if (b % 2 == 1) {                        // linea c
            return a + multiplicar (2*a, b/2);   // linea d
        }
        else {
            return multiplicar (2*a, b/2);       // linea e
        }
    }
}
```

En este problema se pide analizar el coste de ejecutar una invocación **`multiplicar`**`(a,b)` de la función anterior:

- Indicar de qué parámetro o combinación de parámetros depende su coste en tiempo.
- Explicar en que circunstancias y para qué valores de los parámetros se presentan los casos asintóticamente peores y los casos asintóticamente mejores, en cuanto a su coste en tiempo.
- Deducir, paso a paso, la forma de las funciones de coste en los casos peores y mejores.
- Caracterizar asintóticamente cada una de las funciones de coste determinadas en el apartado anterior.

Problema 2º (2.5 puntos)

En este problema se pide demostrar formalmente la corrección del diseño recursivo de la función **`multiplicar`**`(a,b)` mostrado en el problema anterior. Se presentarán, anotadas sobre el propio código o escritas aparte, las pruebas que permiten concluir la corrección de la totalidad de su código, así como su terminación.

Problema 3º (2.5 puntos)

El comportamiento de la función **`repetido`**`(v,i)` se especifica a continuación. El dato **`DIM`** es una constante entera y positiva definida previamente.

```
// Número de componentes de los vectores con los que se va a trabajar (DIM > 0)
const int DIM = ...;

/*
 * Pre: (EX alfa EN [0,DIM−1]. (Núm beta EN [0,DIM−1]. v[beta] = v[alfa]) >= i)
 * Post: repetido (v, i ) = IND AND IND >= 0 AND IND < DIM
 *       AND ((NUM alfa EN [0,DIM−1]. v[alfa] = v[IND]) >= i)
 */
template <typename T>
int repetido (const T v [], const int i );
```

En este problema se pide hacer un diseño sin bucles de la función **`repetido`**`(v,i)`. En caso de ser precisa una inmersión, explicar la técnica de inmersión que ha sido aplicada y tener presente que la calificación del problema estará en función de la correccíon del diseño de la función o funciones que constituyan la solución.

Problema 4º (3 puntos)

El listado que se presenta a continuación presenta un diseño iterativo de la función **`tresElegidos`**`(v,n,min,primero,segundo,tercero)`. Se pide escribir los predicados **`P1`**, **`P2`** y **`P3`** más fuertes (los más restrictivos) que sean invariantes de cada uno de los tres bucles programados en su código.

```
/*
 * Pre:  (NUM alfa EN [0,n−1]. v[alfa]  >= min) > 2
 * Post:  primero  >= min  AND segundo >= min AND tercero >= min AND
 *          (EX alfa  EN [0,n−1]. v[alfa]  = primero  AND
 *                (EX beta  EN [alfa+1,n−1]. v[beta]  = segundo)  AND
 *                (EX gamma EN [beta+1,n−1]. v[gamma] = tercero) )
 */
void tresElegidos  (const int v[], const int n, const int min,
                    int& primero, int& segundo, int& tercero) {
    int i = 0;
    // P1
    while (v[i] < min) {
        // P1
        i = i + 1;
        // P1
    }
    primero = v[i];
    i = i + 1;
    // P2
    while (v[i] < min) {
        // P2
        i = i + 1;
        // P2
    }
    segundo = v[i];
    i = i + 1;
    // P3
    while (v[i] < min) {
        // P3
        i = i + 1;
        // P3
    }
    tercero = v[i];
}
```

Una solución del problema 1º

Analisis del coste de ejecutar una invocación **`multiplicar`**`(a,b)`:

El coste depende del valor del parámetro **`b`**.

Los casos mejores se presentan cuando el valor de del parámetro **`b`** es una potencia de 2 ya que, en tales caso, la condición de la línea **`c`** solo se satisface en la penúltima invocación recursiva.

Los casos peores se presentan cuando el valor de del parámetro **`b`** es una unidad inferior a una potencia de 2 ya que, en tales caso, la condición de la línea **`c`** se satisface en todas las invocaciones a la función **`multiplicar`**`(a,b)`.

- Función de coste en los casos mejores:

  Recurrencia:

  $t(n) = t_a + t_c + t_e + t(b/2)$

  Cambio de variable: $m = log_2\, b$

  $t(m) - t(m-1) = t_a + t_c + t_e$

  Ecuación característica:

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

  Raíces:

  $x = 1$ (doble)

  Solución en función de la variable **`m`**:

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

  Solución en función de la variable **`b`**:

  $t(b) = c_1.\, log_2\, b + c_2$

  Caracterización asintótica de la función de coste en los casos mejores:

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

- Función de coste en los casos peores:

  La recurrencia es similar a la del cado anterior:

  $t(n) = t_a + t_c + t_d + t(b/2)$

  Cambio de variable: $m = log_2\, b$

  $t(m) - t(m-1) = t_a + t_c + t_d$

  Ecuación característica:

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

  Raíces:

  $x = 1$ (doble)

  Solución en función de la variable **`m`**:

  $t(m) = c_3.\, m \times 1^m + c_4 \times 1^m = c_3.\, m + c_4$

  Solución en función de la variable **`b`**:

  $t(b) = c_3.\, log_2\, b + c_4$

  Caracterización asintótica de la función de coste en los casos peores:

  $\boldsymbol{O}(t(b)) = \boldsymbol{O}(c_3.\, log_2\, b + c_4) = \boldsymbol{O}(log_2\, b) = \boldsymbol{O}(log\, b)$

Una solución del problema 2º

Demostración formal la corrección de la función recursiva **`multiplicar`**`(a,b)`. Las pruebas que la integran se han anotadas intercaladas en el propio código de la función.

```
/*
 * Pre: b >= 0
 * Post: multiplicar (a,b) = a * b
 */
int multiplicar (const int a, const int b) {
    // f_cota = b
    if (b == 0) {
        // b = 0
        // => [1]
        // 0 = a * b
        return 0;
    }
    else {
        if (b % 2 == 1) {
            // b > 0 AND b % 2 = 1
            // => [2]
            // b/2 >= 0
            // b > 0 AND b % 2 = 1 AND multiplicar(2*a, b/2) = 2*a * b/2
            // => [3]
            // a + multiplicar (2*a, b/2) = a * b
            return a + multiplicar (2*a, b/2);
            // f_cota = b / 2
            // 1) La función de cota ha decrecido: b > 0 => b > b/2 y
            // 2) La función de cota está acotada inferiormente : b >= 0 => f_cota >= 0
        }
        else {
            // b > 0 AND b % 2 = 0
            // => [4]
            // b/2 >= 0
            // b > 0 AND b % 2 = 0 AND multiplicar(2*a, b/2) = 2*a * b/2
            // => [5]
            // multiplicar (2*a, b/2) = a * b
            return multiplicar (2*a, b/2);
            // f_cota = b / 2
            // 1) La función de cota ha decrecido: b > 0 => b > b/2 y
            // 2) La función de cota está acotada inferiormente : b >= 0 => f_cota >= 0
        }
    }
}
```

Justificación de las cinco pruebas anotadas para demostrar la corrección de diferentes elementos del código anterior:

1 Resulta obvio que el primer predicado, $b = 0$, garantiza la satisfacción del segundo, $0 = a * b$.

2 Si el valor de b es positivo, al dividirlo por 2, el resultado será nulo o positivo, $b/2 >= 0$.

3 El primer predicado establece que el resultado de la invocación $multiplicar(2 * a, b/2)$ es igual a $2*a*b/2$ y que el valor de b es impar y positivo. Esta última condición tiene como consecuencia que $2*a*b/2 = a*b-a$. Sustituyendo en el segundo predicado el resultado de la invocación recursiva se obtiene: $a+multiplicar(2*a, b/2) = a*b \equiv a+2*a*b/2 = a*b \equiv a+(a*b-a) = a*b$, cuya satisfacción es evidente.

4 Si el valor de b es positivo, al dividirlo por 2, el resultado será nulo o positivo, $b/2 >= 0$.

5 El primer predicado establece que el resultado de la invocación $multiplicar(2 * a, b/2)$ es igual a $2 * a * b/2$ y que el valor de b es par y positivo. Esta última condición tiene como consecuencia que $2 * a * b/2 = a * b$. Sustituyendo en el segundo predicado el resultado de la invocación recursiva se obtiene: $multiplicar(2 * a, b/2) = a * b \equiv 2 * a * b/2 = a * b \equiv a * b = a * b$, cuya satisfacción es obvia.

Una solución del problema 3º

Solución resultante de un diseño recursivo por inmersión mediante refuerzo de la precondición de la función **repetido**(v,i).

```
/*
 * Pre:  (EX alfa  EN [indice,DIM−1]. (Núm beta EN [0,DIM−1]. v[beta] = v[alfa])  >= i))
 *         AND desde >= 1 AND desde < DIM AND indice >= 0 AND indice < DIM AND cuenta <= i
 *         AND cuenta = (NUM alfa EN [0,desde−1]. v[alfa]  = v[ indice ])
 * Post:   repetido (v, i, indice, desde, cuenta)  = INDEX AND INDEX >= indice AND INDEX < DIM
 *           AND ((NUM alfa EN [0,DIM−1]. v[alfa] = v[INDEX]) >= i)
 */
template <typename T>
int repetido (const T v[], const int i, const int indice, const int desde, const int cuenta) {
    if (cuenta == i) {
        return indice;
    }
    else if (v[desde] == v[indice]) {
        return repetido (v, i, indice, desde + 1, cuenta + 1);
    }
    else if (desde < DIM − 1) {
        return repetido (v, i, indice, desde + 1, cuenta);
    }
    else {
        return repetido (v, i, indice + 1, indice + 2, 1);
    }
}

/*
 * Pre:  (EX alfa  EN [0,DIM−1]. (Núm beta EN [0,DIM−1]. v[beta] = v[alfa])  >= i)
 * Post:   repetido (v, i)  = IND  AND IND >= 0 AND IND < DIM
 *           AND ((NUM alfa EN [0,DIM−1]. v[alfa] = v[IND]) >= i)
 */
template <typename T>
int repetido (const T v[], const int i) {
    return repetido (v, i, 0, 1, 1);
}
```

Una solución del problema 4º

En el código de la función **`tresElegidos`**`(v,n,min,primero,segundo,tercero)` se han anotado los predicados invariantes **I1**, **I2** y **I3** más fuertes asociados a los tres bucles programados en su código.

```
/*
 * Pre:  (NUM alfa EN [0,n−1]. v[alfa]  >= min) > 2
 * Post:  primero >= min AND segundo >= min AND tercero >= min AND
 *          (EX alfa  EN [0,n−1]. v[alfa]  = primero AND
 *               (EX beta  EN [alfa+1,n−1]. v[beta]  = segundo) AND
 *               (EX gamma EN [beta+1,n−1]. v[gamma] = tercero) )
 */
void tresElegidos (const int v[], const int n, const int min,
                   int& primero, int& segundo, int& tercero) {
    int i = 0;
    while (v[i] < min) {
        // P1: (NUM alfa EN [0,n−1]. v[alfa]  >= min) > 2 AND
        //       i >= 0 AND (NUM alfa EN [0,i−1]. v[alfa] >= min) = 0
        i = i + 1;
    }
    primero = v[i];
    i = i + 1;
    while (v[i] < min) {
        // P2: (NUM alfa EN [0,n−1]. v[alfa]  >= min) > 2 AND
        //       i >= 0 AND (NUM alfa EN [0,i−1]. v[alfa] > min) = 1 AND
        //       (EX alfa  EN [0,i−1]. primero = v[alfa])  AND primero >= min
        i = i + 1;
    }
    segundo = v[i];
    i = i + 1;
    while (v[i] < min) {
        // P3: (NUM alfa EN [0,n−1]. v[alfa]  >= min) > 2 AND
        //       i >= 0 AND (NUM alfa EN [0,i−1]. v[alfa] >= min) = 2 AND
        //       (EX alfa  EN [0,i−1]. segundo = v[alfa]  AND
        //                              (EX beta  [0, alfa−1]. primero  = v[beta])  )
        //       AND primero >= min AND segundo >= min
        i = i + 1;
    }
    tercero = v[i];
}
```