Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 15 de junio de 2017

- Disponer sobre la mesa, en lugar visible, un ***documento de identificación*** provisto de fotografía y escribir el ***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 sintaxis C++ y Apartado 1.2 de los apuntes del curso).

Problema 1º (2.5 puntos)

Demostrar formalmente la corrección de la función **`sumaSerie`**`(a,r,n)` aportando el conjunto de pruebas que permiten concluir que su diseño es correcto. Se valorará la claridad de la presentación de las pruebas y, en su caso, la adecuada justificación de las relaciones entre predicados que en ellas se satisfagan.

```
/*
 * Pre: n >= 0 AND r > 1.0
 * Post: sumaSerie(a,r,n) = (SIGMA alfa EN [0,n]. a * r^alfa)
 */
double sumaSerie (const double a, const double r, const int n) {
    if (n > 0) {
        return a + sumaSerie(a * r, r, n - 1);
    }
    else {
        return a;
    }
}
```

Problema 2º (2.5 puntos)

Demostrar formalmente la corrección de la función **`cuadrado`**`(n)` aportando el conjunto de pruebas que permiten concluir que su diseño es correcto. Se valorará la claridad de la presentación de las pruebas y, en su caso, la adecuada justificación de las relaciones entre predicados que en ellas se satisfagan.

```
/*
 * Pre: n >= 1
 * Post: cuadrado(n) = n^2
 */
int cuadrado (const int n) {
    int i = 1;
    int r = 1;
    while (i != n) {
        i = i + 1;
        r = r + 2 * i - 1;
    }
    return r;
}
```

Problema 3º (2.5 puntos)

Los tres funciones **A**(n), **B**(n) y **C**(n), que se presentan a continuación, resuelven el mismo problema ya que calculan y devuelven el valor de 2^n.

```
/*
 * Pre: n >= 0
 * Post: A(n) = 2^n
 */
long A (int n) {
    if (n == 0) {                    // a
        return 1;                    // b
    }
    else {
        return 2*A(n-1);             // c
    }
}

/*
 * Pre: n >= 0
 * Post: B(n) = 2^n
 */
long B (int n) {
    if (n == 0) {                    // a
        return 1;                    // b
    }
    else {
        return B(n-1) + B(n-1);      // c
    }
}

/*
 * Pre: n >= 0
 * Post: C(n) = 2^n
 */
long C (int n) {
    if (n == 0) {                    // a
        return 1;                    // b
    }
    else {
        long r = C(n/2);             // c
        if (n %2 == 0) {             // d
            return r * r;            // e
        }
        else {
            return 2 * r * r;        // f
        }
    }
}
```

Se pide un análisis comparativo de los costes de las tres funciones anteriores que incluya:

- **Caracterización asintótica del coste de cada una de ellas en función del valor de su parámetro n**. Deben detallarse, paso a paso, las ecuaciones y cálculos que conduzcan a cada resultado y acompañarse de la explicación de en qué consiste cada paso.
- **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 4º (2.5 puntos)

Diseñar sin bucles la función **hayRepes**(palabra,lasHay,c) del modo más eficiente posible. En caso de ser precisa una inmersión, explicar la técnica de inmersión que ha sido aplicada, especificar formalmente las funciones auxiliares y tener presente que la calificación del problema estará en función de la corrección del diseño de la función o funciones que integren la solución.

```
/*
 * Pre:  LONG = strlen(palabra)
 * Post:  lasHay = (EX alfa  EN [0,  LONG−2].
 *                          (EX beta  EN [alfa+1,  LONG−1]. palabra[alfa] = palabra[beta])  )
 *         AND
 *         (lasHay −> (EX alfa EN [0,  LONG−2].
 *                                  (palabra[ alfa ]  = c)  AND
 *                                  (EX beta  EN [alfa+1,  LONG−1]. palabra[beta] = c) )
 */
void hayRepes (const char palabra [],  bool& lasHay,  char& c);
```

En el código a desarrollar puede hacerse uso de la función **srtlen**(cadena) predefinida en la biblioteca **<cstring>**, pero de ninguna otra función predefinida en dicha biblioteca.

Una solución del problema 1º

Demostración formal de la corrección del diseño recursivo se la función **`sumaSerie`**`(a,r,n)`. Las pruebas que demuestran formalmente la corrección del diseño se han anotado intercaladas sobre el propio código de la función. Las pruebas [2] y [3] demuestran la corrección del código programado en los casos recursivo y base, respectivamente. La satisfacción de las dos condiciones [4] demuestran que la recurrencia es finita.

```
/*
 * Pre: n >= 0 AND r > 1.0
 * Post: sumaSerie(a,r,n) = (SIGMA alfa EN [0,n]. a * r^alfa)
 */
double sumaSerie (const double a, const double r, const int n) {
    // f_cota = n
    if (n > 0) {
        // n > 0 AND r > 1.0
        // => [1]
        // n - 1 >= 0 AND r > 1.0

        // n > 0 AND r > 1.0 AND
        // sumaSerie(a*r,r,n-1) = (SIGMA alfa EN [0,n-1]. a * r * r^alfa)
        // => [2]
        // a + sumaSerie(a * r, r, n - 1) = (SIGMA alfa EN [0,n]. a * r^alfa)
        return a + sumaSerie(a * r, r, n - 1);
        // f_cota = n - 1
        // 1) El valor de la función de cota decrece ya que:
        //          n >= 0 -> n >= n - 1                    [4]
        // 2) El valor de la función de cota es un entero acotado inferiormente por -1:
        //          n >= 0 => f_cota = n - 1 >= -1          [4]

        // sumaSerie(a,r,n) = (SIGMA alfa EN [0,n]. a * r^alfa)
    }
    else {
        // n = 0 AND r > 1.0
        // => [3]
        // a = (SIGMA alfa EN [0,n]. a * r^alfa)
        return a;
        // sumaSerie(a,r,n) = (SIGMA alfa EN [0,n]. a * r^alfa)
    }
}
```

Estas son las pruebas anotadas sobre el algoritmo anterior. Se han omitido las pruebas relativas a condiciones de dominio de expresiones por no plantear problemas la evaluación de ninguna de ellas.

- **[1]** Su satisfacción prueba que la precondición de la invocación recursiva que sigue se satisface.
- **[2]** La satisfacción de la relación es inmediata ya que:
  $a + (\Sigma\alpha \in [0, n-1].(a \times r) \times r^{\alpha}) = a + r \times (\Sigma\alpha \in [0, n-1].a \times r^{\alpha}) = (\Sigma\alpha \in [0, n].a \times r^{\alpha})$
  y ello prueba la corrección del código programado para resolver el caso recursivo.
- **[3]** Su satisfacción es inmediata ya que $a = a \times r^0$ y prueba la corrección del código programado para resolver el caso no recusivo (caso base).
- **[4]** La satisfacción de las dos condiciones presentadas (que el valor entero de la función de cota decrece al producirse una invocación recursiva y que el valor de la función de cota está acotado inferiormente por el valor -1) prueban que la sucesión de invocaciones recursivas no se prolonga indefinidamente, es decir, que la función termina.

Una solución del problema 2º

Las cuatro pruebas se han anotado intercaladas sobre el propio código. Las pruebas se sustentan en el siguiente predicado invariante del bucle $i \geq 1 \ \wedge \ i \leq n \ \wedge \ r = i^2$

```
/*
 * Pre: n >= 1
 * Post: cuadrado(n) = n^2
 */
int cuadrado (const int n) {
    // n >= 1
    // => [1]
    // 1 >= 1 AND 1 <= n AND 1 = 1^2
    int i = 1;
    // i >= 1 AND i <= n AND 1 = i^2
    int r = 1;
    // Inv: i >= 1 AND i <= n AND r = i^2 => Dom(i != n) = cierto
    while (i != n) {    // f_cota (i) = n - i

        // i = A AND f_cota(antes )( i ) = n - A

        // i >= 1 AND i < n AND r = i^2
        // => [2]
        // i >= 0 AND i <= n - 1 AND r + 2 * (i + 1) - 1= (i + 1)^2
        i = i + 1;
        // i >= 1 AND i <= n AND r + 2 * i - 1 = i^2
        r = r + 2 * i - 1;
        // i >= 1 AND i <= n AND r = i^2

        // i = A + 1 AND f_cota(despues )( i ) = n - A -1

        // El bucle termina ya que [4]:
        // 1) el valor de la f_cota decrece en cada iteración :
        //        n - A > n - A - 1
        // 2) el valor de la f_cota es un entero acotado inferiormente por 0:
        //        // Inv => n - i >= 0 => f_cota(i) = n - i >= 0
    }
    // i = n AND r = i^2
    // => [3]
    // r = n^2
    return r;
    // cuadrado(n) = n^2
}
```

Justificación de las pruebas presentadas:

- La relación entre predicados [1] es evidente y prueba la corrección del código que precede al bucle.
- La satisfacción de relación entre predicados [2] prueba la corrección del código a iterar en el bucle y se verifica teniendo en cuenta que $(i+1)^2 - 2 \times (i+1) \times 1 + 1^2 = (((i+1) - 1)^2 = i^2$.
- La relación entre predicados [3] es también evidente y prueba la corrección del código que sigue al bucle,.
- La prueba de la terminación del bucle [4] se deduce de la satisfacción de las dos condiciones para la función de cota $fcota(i) = n - i$: (1) Su valor decrece en cada iteración $n - A > n - A - 1$ y (2) toma valores en el conjunto de los enteros, $\mathbf{Z}$, acotados inferiormente por 0, es decir, $fcota(i) \geq 0$

Una solución del problema 3º

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

  Ecuación recurrente:

  $t_A(n) = t_a + t_c + t_A(n-1)$

  $t_A(n) - t_A(n-1) = (t_a + t_c) \times 1^n$

  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 **`B`**`(n)`.

  Ecuación recurrente:

  $t_B(n) = t_a + t_c + 2 \times t_B(n-1)$

  $t_B(n) - 2 \times t_B(n-1) = (t_a + t_c) \times 1^n$

  Ecuación característica y sus raices:

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

  Solución general de la recurrencia:

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

  Caracterización asintótica de la función de coste:

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

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

  Ecuación recurrente (la ejecución de la línea **`f`** es ligeramente más costosa que la de la línea **`e`**, se va a calcular el coste en el caso peor; el resultado en el caso mejor será similar):

  $t_C(n) = t_a + t_c + t_f + t_C(n/2)$

  $t_C(n) - t_C(n/2) = t_a + t_c + t_f$

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

  $t_C(m) - t_C(m-1) = (t_a + t_c + t_f) \times 1^m$

  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:

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

  En el caso mejor lo único que varía es que la constante c_1 será ligeramente menor que el el caso peor.

  Caracterización asintótica de la función de coste:

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

- Análisis comparativo de la eficiencia de las tres funciones.

  La función asintóticamente más eficiente es **`C`**`(n)` ya que su coste es logarítimico en el valor del parámetro **n**. Esta es la que conviene utilizar. Le sigue en eficiencia la función **`A`**`(n)` cuyo coste depende linealmente del valor del parámetro **n**. La más ineficiente es la función **`B`**`(n)` cuyo coste depende exponencialmente del valor del parámetro **n**.

Una solución del problema 4º

Se presenta una primera solución basada en un diseño recursivo por inmersión mediante refuerzo de la precondición.

```
/*
 * Pre: LONG = strlen(palabra) AND i >= 0 AND i < j AND j < LONG AND
 *      (PT alfa EN [0, i-1].(PT beta EN [alfa+1, LONG-1]. palabra[alfa] != palabra[beta]) ) AND
 *      (PT beta EN [i+1, j-1]. palabra[i] != palabra[beta])
 * Post: lasHay = (EX alfa EN [0, LONG-2].
 *                    (EX beta EN [alfa+1, LONG-1]. palabra[alfa] = palabra[beta]) ) AND
 *       (lasHay -> (EX alfa EN [0, LONG-2].
 *                      palabra[alfa] = c AND (EX beta EN [alfa + 1, LONG-1]. palabra[beta] = c) )
 */
void hayRepesRef (const char palabra [], bool& lasHay, char& c, const int i, const int j) {
    if (j < int( strlen (palabra ))) {
        if (palabra[i] == palabra[j]) {
            // palabra[i] está repetida
            lasHay = true;
            c = palabra[i];
        }
        else {
            // palabra[i] y palabra[j] no coinciden; hay que seguir buscando
            hayRepesRef(palabra, lasHay, c, i , j + 1);
        }
    }
    else {
        if (i < int( strlen (palabra )) - 1) {
            // hay que seguir buscando posibles repeticiones de palabra[i+1]
            hayRepesRef(palabra, lasHay, c, i + 1 , i + 2);
        }
        else {
            // no hay ningún carácter repetido
            lasHay = false ;
        }
    }
}

/*
 * Pre: LONG = strlen(palabra)
 * Post: lasHay = (EX alfa EN [0, LONG-2].
 *                    (EX beta EN [alfa+1, LONG-1]. palabra[alfa] = palabra[beta]) ) AND
 *       (lasHay -> (EX alfa EN [0, LONG-2].
 *                      (palabra[alfa] = c) AND (EX beta EN [alfa+1, LONG-1]. palabra[beta] = c) )
 */
void hayRepes (const char palabra [], bool& lasHay, char& c) {
    hayRepesRef(palabra, lasHay, c, 0, 1);
}
```

También presenta una solución alternativa basada en un diseño recursivo por inmersión mediante debilitamiento de la postcondición.

```
/*
 * Pre: LONG = strlen(palabra) AND i >= 0 AND i < j AND j < LONG
 * Post: lasHay = (EX beta EN i+1, LONG−1]. palabra[i] = palabra[beta]) OR
 *                 (EX alfa EN [i+1, LONG−2].
 *                       (EX beta EN [alfa+1, LONG−1]. palabra[alfa] = palabra[beta]) ) AND
 *       (lasHay −> (EX alfa EN [i, LONG−2].
 *                       (palabra[ alfa ] = c) AND (EX beta EN [alfa+1, LONG−1]. palabra[beta] = c) )
 */
void hayRepesDeb (const char palabra [], bool& lasHay, char& c, const int i, const int j) {
    if (j < int( strlen (palabra ))) {
        if (palabra[i] == palabra[j]) {
            // palabra[i] está repetida
            lasHay = true;
            c = palabra[i];
        }
        else {
            // palabra[i] y palabra[j] no coinciden; hay que seguir buscando
            hayRepesDeb(palabra, lasHay, c, i , j + 1);
        }
    }
    else {
        if (i < int( strlen (palabra )) − 1) {
            // hay que seguir buscando posibles repeticiones de palabra[i+1]
            hayRepesDeb(palabra, lasHay, c, i + 1 , i + 2);
        }
        else {
            // no hay ningún carácter repetido
            lasHay = false ;
        }
    }
}

/*
 * Pre: LONG = strlen(palabra)
 * Post: lasHay = (EX alfa EN [0, LONG−2].
 *                       (EX beta EN [alfa+1, LONG−1]. palabra[alfa] = palabra[beta]) ) AND
 *       (lasHay −> (EX alfa EN [0, LONG−2].
 *                       (palabra[ alfa ] = c) AND (EX beta EN [alfa+1, LONG−1]. palabra[beta] = c) )
 */
void hayRepes (const char palabra [], bool& lasHay, char& c) {
    hayRepesDeb(palabra, lasHay, c, 0, 1);
}
```