Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas

Examen de Programación 2 - 4 de septiembre de 2018

- 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 (2.0 puntos)

Asumiremos que la función que sigue se ejecuta mediante una invocación que satisface su precondición. Se pide escribir las aserciones **`P1`** y **`P2`** más fuertes que satisfagan los datos de la función al alcanzar la ejecución de su código los puntos en los que han sido escritas. Las aserciones **`P1`** y **`P2`** han de ser formales, es decir, han de ser escritas como predicados matemáticos.

```
/*
 * Pre: n > 0 AND n <= #v
 * Post: <encontrados> es cierto si y sólo si hay un par de elementos v[izdo]
 *       y v[dcho] del vector <v> cuyos valores coinciden, verificando que
 *       izdo >= 0, izdo < dcho y dcho <= n - 1
 */
template <typename T>
void coinciden (const T v[], const int n, bool& encontrados, int& izdo, int& dcho) {
    izdo = 0;
    dcho = n - 1;
    encontrados = false;
    while (!encontrados && izdo != n - 1) {
        if (v[izdo] != v[dcho]) {
            if (izdo < dcho - 1) {
                dcho = dcho - 1;
            }
            else {
                izdo = izdo + 1;
                dcho = n - 1;
            }
        }
        else {
            encontrados = true;
        }
        // P1
    }
    // P2
}
```

Problema 2º (2.0 puntos)

```
/*
 * Pre: n > 0 AND n <= #v
 * Post: <encontrados> es cierto si y sólo si hay un par de elementos v[izdo]
 *       y v[dcho] del vector <v> cuyos valores coinciden, verificando que
 *       izdo >= 0, izdo < dcho y dcho <= n - 1
 */
template <typename T>
void coinciden (const T v[], const int n, bool& encontrados, int& izdo, int& dcho) {
    izdo = 0;                                                    // a
    dcho = n - 1;                                                // b
    encontrados = false;                                         // c
    while (!encontrados && izdo != n - 1) {                      // d
        if (v[izdo] != v[dcho]) {                                // e
            if (izdo < dcho - 1) {                               // f
                dcho = dcho - 1;                                 // g
            }
            else {
                izdo = izdo + 1;                                 // h
                dcho = n - 1;                                    // i
            }
        }
        else {
            encontrados = true;                                  // j
        }
    }
}
```

Se pide analizar el coste en tiempo de ejecutar la instrucción de invocación **`coinciden`**`(v,n,encontrados,izdo,dcho)` en función del valor que tome el parámetro **`n`**, según el siguiente guión:

1. Explicar y justificar cuáles son las disposiciones iniciales de los **`n`** primeros elementos del vector **`v`** que proporcionan los casos asintóticamente peores y mejores en cuanto al coste en tiempo de la instrucción de invocación **`coinciden`**`(v,n,encontrados,izdo,dcho)` o, en su caso, justificar si no cabe distinguir casos diferenciados ya que hay un único caso general para caracterizar asintóticamente el coste en tiempo.

2. Determinar, en su caso, la función de coste en tiempo, $t_{mejor}(n)$ y $t_{peor}(n)$, en los casos extremos identificados en el apartado anterior o bien en el caso único general, $t_{general}(n)$. Dicha función o funciones de coste se expresarán en función de los costes en tiempo t_a, t_b, ..., t_i y t_j de las diferentes líneas del algoritmo en las que hay condiciones que evaluar o instrucciones que ejecutar. Se deberán detallar y, en su caso, justificar los diferentes pasos dados para calcular la función o funciones de coste pedidas.

3. Caracterizar asintóticamente el orden de la función o funciones de coste en tiempo, $\boldsymbol{O}(t_{mejor\ y\ peor\ o\ general}(n))$, determinadas en el apartado anterior.

Problema 3º (2.0 puntos)

A continuación se presenta la especificación de la función **`estaOrdenado`**`(nombre)`. Los datos que se correspondan con el tipo **`<T>`** han de soportar las seis operaciones de relación: **`==`**, **`!=`**, **`<`**, **`<=`**, **`>`** y **`>=`**.

```
/*
 * Pre: <nombre> es el nombre de un fichero binario que almacena una secuencia de datos
 *      de tipo <T>
 * Post: Ha devuelto <true> si y sólo si la secuencia de datos almacenada en el fichero
 *       <nombre> está ordenada de menor a mayor valor (cada dato es menor o igual que
 *       el que le sigue)
 */
template <typename T>
bool estaOrdenado (const char nombre[]);
```

En este problema se pide diseñar sin bucles el código de la función **`estaOrdenado`**`(nombre)`, apoyándolo, en caso necesario, en una o más funciones auxiliares que deberán estar convenientemente especificadas (no formalmente, pero sí de forma clara, precisa y rigurosa) y deberán ser programadas también sin bucles. En este problema se valorará la corrección de la función o funciones que integren el diseño.

Observación: No es necesario escribir las cláusulas `#include` que den visibilidad a las funciones de biblioteca invocadas en el diseño pedido.

Problema 4º (4.0 puntos)

Un diseño sin bucles de la función **`sumarCuad`**`(v,i,j,suma)` se presenta a continuación.

```
/*
 * Pre:  i >= 0 AND j >= 0 AND i < #v AND j < #v
 * Post: suma = (SIGMA alfa EN [i, j]. v[alfa]^2)
 */
void sumarCuad (double v[], const int i, const int j, double & suma) {
    if (i == j) {
        suma = v[i] * v[j];
    }
    else if (i < j) {
        sumarCuad(v, i + 1, j - 1, suma);
        suma = suma + v[i] * v[i];
        suma = suma + v[j] * v[j];
    }
    else {
        suma = 0.0;
    }
}
```

Se pide aportar el conjunto de pruebas formales que permiten demostrar la corrección del diseño anterior. Se valorarán las pruebas aportadas y la claridad de las explicaciones que justifican cada una de las pruebas.

Una solución del problema 1º

Las aserciones **P1** y **P2** pedidas se ha escrito insertas en el código para facilitar su comprensión.

```
/*
 * Pre: n > 0 AND n <= #v
 * Post: <encontrados> es cierto si y sólo si hay un par de elementos v[izdo]
 *       y v[dcho] del vector <v> cuyos valores coinciden, verificando que
 *       izdo >= 0, izdo < dcho y dcho <= n - 1
 */
template <typename T>
void coinciden (const T v[], const int n, bool& encontrados, int& izdo, int& dcho) {
    izdo = 0;
    dcho = n - 1;
    encontrados = false;
    while (!encontrados && izdo != n - 1) {
        if (v[izdo] != v[dcho]) {
            if (izdo < dcho - 1) {
                dcho = dcho - 1;
            }
            else {
                izdo = izdo + 1;
                dcho = n - 1;
            }
        }
        else {
            encontrados = true;
        }
        // P1: n > 0 AND n <= #v AND izdo >= 0 AND izdo <= dcho AND dcho <= n - 1 AND
        //     (PT alfa EN [0, izdo-1]. (PT beta EN [alfa+1, n-1]. v[alfa] != v[beta]) ) AND
        //     (PT beta EN [dcho+1, n-1]. v[izdo] != v[beta]) AND
        //     (encontrados -> v[izdo] = v[dcho]) AND
        //     (NOT encontrados -> v[izdo] != v[dcho+1] OR dcho = n - 1)
    }
    // P2: n > 0 AND n <= #v AND izdo >= 0 AND izdo <= dcho AND dcho <= n - 1 AND
    //     (PT alfa EN [0, izdo-1]. (PT beta EN [alfa+1, n-1]. v[alfa] != v[beta]) ) AND
    //     (PT beta EN [dcho+1, n-1]. v[izdo] != v[beta]) AND
    //     (encontrados -> v[izdo] = v[dcho]) AND
    //     (NOT encontrados -> izdo = n-1 AND dcho = n-1)
}
```

Una solución del problema 2º

1. El **caso mejor** en coste en tiempo se presenta cuando `v[0] = v[n-1]`. En tal caso, el código programado dentro del bucle sólo se ejecuta una vez, para un valor de la variable $izdo$ igual a `0`, es decir, para $izdo = 0$ y para un valor de la variable $dcho$ igual a `n-1`, es decir, para $dcho = n - 1$.

   El **caso peor** en coste en tiempo se presenta cuando no existen en `v[0,n-1]` dos elementos iguales. En tal caso el bucle se ejecuta para valores de la variable comprendidos desde $izdo = 0$ hasta $izdo = n - 2$ ya que acaba cuando $izdo = n - 1$. Para un valor determinado de la variable $izdo$, la variable $dcho$ toma en cada iteración valores decrecientes partiendo de $dcho = n - 1$ y acabando con $dcho = izdo + 1$, es decir, un total de $n - 1 - (izdo + 1)0n - izdo - 2$ iteraciones con el mismo valor de $izdo$.

2. **Función de coste en tiempo**, $t_{mejor}(n)$ **en el caso mejor** se presenta cuando `v[0] = v[n-1]`. La suma de costes de las instrucciones ejecutadas y las condiciones evaluadas en tal caso es la siguiente:

   $$t_{mejor}(n) = t_{inv} + t_a + t_b + t_c + t_d + t_e + t_j + t_d$$

   El coste t_d está repetido dos veces. La función de coste en tiempo, en el caso mejor, queda como sigue:

   $$t_{mejor}(n) = t_{inv} + t_a + t_b + t_c + 2 \times t_d + t_e + t_j$$

   Función de coste en tiempo, $t_{peor}(n)$ **en el caso peor** se presenta cuando no hay en `v[0,n-1]` dos elementos iguales. El interior del bucle **while** se ejecuta para valores de $izdo$ comprendidos en $[0, n - 2]$. Para cada valor de $izdo$ el bucle itera para los valores de $dcho$ comprendidos en $[izdo + 1, n - 1]$, en sentido decreciente. Teiendo en cuenta el análisis anterior, podemos plantear la suma de costes de las instrucciones ejecutadas y las condiciones evaluadas en el caso peor:

   $$\begin{aligned} t_{peor}(n) = {} & t_{inv} + t_a + t_b + t_c + t_d + \\ & (\Sigma\alpha \in [0, n-2].(n - \alpha - 2)(t_e + t_f + t_g + t_d) + t_e + t_f + t_h + t_i + t_d) \end{aligned}$$

   Sumamos los términos $t_e + t_f + t_h + t_i + t_d$ que son constantes en las $n - 1$ iteración y sacamos factor común el factor constante $(t_e + t_f + t_g + t_d)$.

   $$\begin{aligned} t_{peor}(n) = {} & t_{inv} + t_a + t_b + t_c + t_d + (t_e + t_f + t_h + t_i + t_d)(n - 1) + \\ & (t_e + t_f + t_g + t_d)(\Sigma\alpha \in [0, n-2].(n - \alpha - 2)) \end{aligned}$$

   El último término de la suma es una progresión aritmética que va a ser sumada a continuación.

   $$t_{peor}(n) = t_{inv} + t_a + t_b + t_c + t_d + (t_e + t_f + t_h + t_i + t_d)(n-1) + (t_e + t_f + t_g + t_d)\frac{(n-1)(n-2)}{2}$$

   Ordenamos finalmente los términos según grado decreciente en n.

   $$t_{peor}(n) = \frac{t_e + t_f + t_g + t_d}{2}(n-1)(n-2) + (t_e + t_f + t_h + t_i + t_d)(n-1) + t_{inv} + t_a + t_b + t_c + t_d$$

3. **Caracterización asíntótica de la función de coste**, $t_{mejor}(n)$, **en el caso mejor**. En la simplificación se aplica la regla de la sumaa y la regla de las constantes multiplicativas:

   $$\boldsymbol{O}(t_{mejor}(n)) = \boldsymbol{O}(t_{inv} + t_a + t_b + t_c + 2 \times t_d + t_e + t_j)$$
   $$\boldsymbol{O}(t_{mejor}(n)) = \boldsymbol{O}(Máx(t_{inv}, t_a, t_b, t_c, 2 \times t_d, t_e, t_j)) = \boldsymbol{O}(1)$$

   Caracterización asíntótica de la función de coste, $t_{peor}(n)$, **en el caso peor**. En la simplificación se aplica la regla de la suma (dos veces, en el primer paso y en el último) y la regla de las constantes multiplicativas (en el segundo paso):

   $$\begin{aligned} \boldsymbol{O}(t_{peor}(n)) = \boldsymbol{O}(& \tfrac{t_e + t_f + t_g + t_d}{2}(n-1)(n-2) + (t_e + t_f + t_h + t_i + t_d)(n-1) + \\ & t_{inv} + t_a + t_b + t_c + t_d) \end{aligned}$$
   $$\boldsymbol{O}(t_{peor}(n)) = \boldsymbol{O}(\tfrac{t_e + t_f + t_g + t_d}{2}(n-1)(n-2))$$
   $$\boldsymbol{O}(t_{peor}(n)) = \boldsymbol{O}((n-1)(n-2)) = \boldsymbol{O}(n^2 - 3n + 2) = \boldsymbol{O}(n^2)$$

Una solución del problema 3º

```
/*
 * Pre: El flujo <f> está asociado a un fichero binario que almacena una secuencia de
 *      datos de tipo <T>, el valor de <ultimo> es el del último dato leído de dicho
 *      fichero y los datos leídos del fichero están ordenados de menor a mayor valor
 * Post: Ha devuelto <true> si y sólo si la secuencia de datos almacenada en el fichero
 *       asociado a <f> está ordenada de menor a mayor valor (cada dato es menor o igual
 *       que el que le sigue)
 */
template <typename T>
bool estaOrdenado (ifstream & f, const T ultimo) {
    T siguiente;
    f.read( reinterpret_cast <char *>(&siguiente), sizeof( siguiente ));
    if (!f.eof()) {
        if (ultimo <= siguiente) {
            // <ultimo> y <siguiente> están ordenados
            return estaOrdenado(f, siguiente );
        }
        else {
            // El fichero no está ordenado ya que <ultimo> y <siguiente> no lo están
            return false;
        }
    }
    else {
        // Todos los datos del fichero han sido leídos, luego está ordenado
        return true;
    }
}

/*
 * Pre: <nombre> es el nombre de un fichero binario que almacena una secuencia de datos
 *      de tipo <T>
 * Post: Ha devuelto <true> si y sólo si la secuencia de datos almacenada en el fichero
 *       <nombre> está ordenada de menor a mayor valor (cada dato es menor o igual que
 *       el que le sigue)
 */

template <typename T>
bool estaOrdenado (const char nombre[]) {
    ifstream f;
    f.open(nombre, ios :: binary );
    T primero;
    f.read( reinterpret_cast <char *>(&primero), sizeof(primero ));
    if (!f.eof()) {
        // Almacena uno o más datos
        bool loEsta = estaOrdenado(f, primero);
        f.close ();
        return loEsta;
    }
    else {
        // No almacena ningún dato, luego 'sus datos' están ordenados
        f.close ();
        return true;
    }
}
```

Una solución del problema 4º

El conjunto de pruebas que demuestran formalmente la corrección del código de la función **sumarCuad**(v,i,j,suma) se presentan anotadas sobre el propio código de la función, a excepción de la prueba de la terminación de su ejecución, que se presenta posteriormente.

```
/*
 * Pre:   i >= 0 AND j >= 0 AND i < #v AND j < #v
 * Post: suma = (SIGMA alfa EN [i, j]. v[alfa]^2)
 */
void sumarCuad (double v[], const int i, const int j, double & suma) {
    if (i == j) {
        // i >= 0 AND j >= 0 AND i = j AND i < #v AND j < #v
        // => [1]
        // v[i] * v[j] = (SIGMA alfa EN [i, j]. v[alfa]^2)
        suma = v[i] * v[j];
    }
    else if (i < j) {
        // i >= 0 AND j >= 0 AND i < j AND i < #v AND j < #v
        // => [2]
        // i + 1 >= 0 AND i + 1 < #v AND j - 1 < #v
        sumarCuad(v, i + 1, j - 1, suma);
        // i >= 0 AND j >= 0 AND i < j AND i < #v AND j < #v AND
        // suma = (PT alfa EN [i+1, j-1]. v[alfa]^2)
        // => [3]
        // suma + v[i] * v[i] + v[j] * v[j] = (PT alfa EN [i, j]. v[alfa]^2)
        suma = suma + v[i] * v[i];
        // suma + v[j] * v[j] = (SIGMA alfa EN [i, j]. v[alfa]^2)
        suma = suma + v[j] * v[j];
        // suma = (SIGMA alfa EN [i, j]. v[alfa]^2)
    }
    else {
        // i >= 0 AND j >= 0 AND i > j AND i < #v AND j < #v
        // => [4]
        // 0.0 = (SIGMA alfa EN [i, j]. v[alfa]^2)
        suma = 0.0;
    }
}
```

Observaciones y justificaciones:

- La satisfacción de la relación [1] es inmediata si se tiene en cuenta que $i = j$ ya que, en tal caso, $v[i] * v[j] = (\Sigma\alpha \in [i,j].\ v[\alpha]^2)$. La satisfacción de la relación [1] prueba la corrección de la instrucción $suma = v[i] * v[j]$ programada en el primero de los bloques de la instrucción condicional.

- La satisfacción de la relación [2] es inmediata. Con ella se prueba la satisfacción de la precondición $i + 1 >= 0 \ \wedge \ i + 1 < \#v \ \wedge \ j - 1 < \#v$ de la invocación recursiva **sumarCuad**(v,i+1,j-1,suma).

- Tras la instrucción de invocación **sumarCuad**(v,i+1,j-1,suma) se satisface su postcondición $suma = (\Sigma\alpha \in [i+1, j-1].\ v[\alpha]^2)$ y también se satisfacen las condiciones que limitan los valores de los parámetros i y j, es decir, $i >= 0 \ \wedge \ i < j \ \wedge \ i < \#v \ \wedge \ j < \#v$ dado que el valor de dichos parámetros no se ha modificado al ser ejecutada la invocación **sumarCuad**(v,i+1,j-1,suma).

- La verificación de la satisfacción de la relación [3] es inmediata si se tiene en cuenta que la diferencia entre los sumatorios $(\Sigma\alpha \in [i+1, j-1].\ v[\alpha]^2)$ (en el primer de los predicados

relacionados en [3]) y $(\Sigma\alpha \in [i,j].\ v[\alpha]^2)$ (en el segundo de los predicados relacionados en [3]) la constituyen los términos $v[i]^2 + v[j]^2$. La satisfacción de la relación [3] prueba la corrección de la secuencia de tres instrucciones programadas en el bloque asociado a la cláusula **`else if`**.

- Es inmediata la satisfacción de la relación [4] si se tiene en cuenta que $i > j$ ya que, en tal caso, se satisface que $(\Sigma\alpha \in [i,j].\ v[\alpha]^2) = 0.0$. La satisfacción de la relación [4] prueba la corrección de la instrucción $suma = 0.0$ programada en el bloque asociado a la cláusula **`else`**.

La prueba de la terminación de la función **`sumarCuad`**`(v,i,j,suma)` se construye a continuación a partir de la función de cota $f_{cota}(i,j) = j - i$.

Se han anotado sobre el propio código las pruebas de la satisfacción de las dos condiciones exigidas a la función de cota al producirse la invocación recursiva (decrecimiento en Z y acotación inferior del valor que toma al producirse la invocación recusiva).

```
/*
 * Pre:   i >= 0 AND j >= 0 AND i < #v AND j < #v
 * Post: suma = (SIGMA alfa EN [i, j]. v[ alfa ]^2)
 */
void sumarCuad (double v[], const int i, const int j, double & suma) {
    // f_cota (i,j) = j - i
    // f_cota (invocación inicial ) = j - i
    if (i == j) {
        suma = v[i] * v[j];
    }
    else if (i < j) {
        //  i >= 0 AND j >= 0 AND i < j AND i < #v AND j < #v
        sumarCuad(v, i + 1, j - 1, suma);
        // i >= 0 AND j >= 0 AND i < j AND i < #v AND j < #v AND
        // suma = (PT alfa EN [i+1, j-1]. v[ alfa ]^2)
        // f_cota ( invocación recursiva ) = j - 1 - i - 1 = j - i - 2
        // La secuencia de invocaciones recursiva termina ya que se satisfacen las dos
        // condiciones exigidas :
        // C1. El valor de la función de cota decrece en cada invocación :
        //                 j - i > j - i - 2
        // C2. El valor de la función de cota está limitado inferiormente en Z por -1:
        //                 i < j => f_cota( invocación recursiva ) = j - i - 2 >= - 1
        suma = suma + v[i] * v[i];
        suma = suma + v[j] * v[j];
    }
    else {
        suma = 0.0;
    }
}
```