Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 4 de septiembre 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 (4.0 puntos)

El diseño de la función genérica **`sumar`**`(A)` se apoya en la función genérica auxiliar **`sumar`**`(A,f)`.

Se pide demostrar formalmente la corrección del código de la función **`sumar`**`(A)` aportando el conjunto de pruebas que permitan concluir que su diseño es correcto: a) corrección del código que precede al bucle, b) corrección del código a iterar en el bucle, c) corrección del código que sigue al bucle y d) terminación del bucle. Se valorará la claridad en la presentación de cada una de las pruebas y, en su caso, la adecuada justificación de las relaciones entre predicados que en ellas se satisfagan.

```
// Esta constante se debe definir de forma que N >= 1
const int N = ...;

/*
 * Pre: N > 0 AND f >= 0 AND f < N
 * Post: sumar(A,f) = (SIGMA beta EN [0,N-1]. A[f][beta]))
 */
template <typename T>
T sumar (const T A[N][N], const int f);

/*
 * Pre: N > 0
 * Post: sumar(A) = (SIGMA alfa EN [0,N-1]. (SIGMA beta EN [0,N-1]. A[alfa][beta]))
 */
template <typename T>
T sumar (const T A[N][N]) {
    T s = sumar(A, 0);
    int f = 0;
    while (f != N - 1) {
        f = f + 1;
        s = s + sumar(A, f);
    }
    return s;
}
```

Problema 2º (2.5 puntos)

El diseño de la función genérica **`ordenar`**`(v)` se presenta a continuación. Se han etiquetado las líneas de su código con las primeras letras del alfabeto (**`a`**, **`b`**, ..., **`i`** y **`j`**). Se asume que el coste de ejecutar el código de cada una de dichas líneas es constante, es decir $\boldsymbol{O}(1)$, y va a ser denominado t_a, t_b, ..., t_i y t_j, respectivamente.

```
// La constante DIM debe definirse con un valor mayor que 0
const int DIM = ...;

// esPermutación(v1,v2,i,j) es cierto si y solo si los elementos de v1[i..j] son
//                          una permutación de los elementos v2[i..j]
// ordenados(v,i,j) es cierto si y solo si los elementos de v[i..j] están ordenados
//                  de menor a mayor valor

/*
 * Pre: v = Vo AND DIM > 0
 * Post: esPermutación(v,Vo,0,DIM-1) AND ordenados(v,0,DIM-1)
 */
template <typename Tipo>
void ordenar (Tipo v[DIM]) {
    for (int i = 1; i <= DIM - 1; ++i) {                                    // a
        // DIM > 0 AND esPermutación(v,Vo,0,DIM-1) AND ordenados(v,0,i-1)
        Tipo dato = v[i];                                                   // b
        int j = i;                                                          // c
        int iLimite = 0;                                                    // d
        while (j != iLimite) {                                              // e
            if (v[j-1] <= dato) {                                           // f
                iLimite = j;                                                // g
            }
            else {
                v[j] = v[j-1];                                              // h
                j = j - 1;                                                  // i
            }
        }
        v[j] = dato;                                                        // j
        // DIM > 0 AND esPermutación(v,Vo,0,DIM-1) AND ordenados(v,0,i)
    }
}
```

Se pide analizar el coste en tiempo de ejecutar **`ordenar`**`(v)` en función del valor de la constante **`DIM`**, según el siguiente guión:

1. Explicar y justificar cuáles son las disposiciones iniciales de datos en el vector **`v`** que proporcionan los casos asintóticamente peores y mejores en cuanto al coste en tiempo de **`ordenar`**`(v)` o justificar si no cabe distinguir casos diferenciados ya que hay un único caso general para caracterizar asintóticamente el coste en tiempo.

2. Determinar la función de coste en tiempo, $t(DIM)$, en los casos extremos identificados en el apartado anterior. 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. 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 cada una de las funciones de coste en tiempo, $\boldsymbol{O}(t(DIM))$, determinadas en el apartado anterior.

Problema 3º (3.5 puntos)

Diseñar sin bucles la función **`mayor`**`(sec)`. En el código a desarrollar no se puede hacer uso de ninguna función de las bibliotecas C++ predefinidas. En caso de ser precisa una inmersión especificar las funciones auxiliares (de modo formal o no formal, pero siempre con rigor). Conviene tener presente que la calificación del problema estará en función de la corrección del diseño de la función o funciones que integren la solución.

```
/*
 * <secuencia> ::= <numero> { <separadores> <numero> } <nulo>
 * <numero> ::= <digito> { <digito> }
 * <digito> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
 * <nulo> ::= '\0'
 * <separadores> ::= " " { " " }
 */

// El carácter nulo que denota el final de una cadena de caracteres
const char NULO = '\0';

/*
 * Pre: <sec> almacena una secuencia de caracteres que responde a la sintaxis de <secuencia>
 *      y que contiene uno o más números
 * Post: Devuelve el valor del mayor de los números almacenados en <sec>. Ejemplos:
 *       1. La invocación mayor("405") devuelve 405
 *       2. La invocación mayor("405 1098 745") devuelve 1098
 *       3. La invocación mayor("405  1098  745") devuelve 1098
 *       4. La invocación mayor("00 0 000 0000") devuelve 0
 *       5. La invocación mayor("102 00098 0222 37") devuelve 222
 */
int mayor (const char sec []);
```

Una solución del problema 1º

Las pruebas que demuestran formalmente la corrección del diseño de la función **sumar**(A) se han anotado intercaladas sobre el propio código de la función.

```
/*
 * Pre: N > 0
 * Post: sumar(A) = (SIGMA alfa EN [0,N−1]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
 */
template <typename T>
T sumar (const T A[N][N]) {
    // N > 0
    // => [1]
    // N > 0 AND 0 >= 0 AND 0 < N

    // N > 0 AND f = 0 AND sumar(A,0) = (SIGMA beta EN [0,N−1]. A[0][beta]) )
    // => [2]
    // N > 0 AND 0 >= 0 AND 0 < N AND
    // sumar(A,0) = (SIGMA alfa EN [0,0]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
    T s = sumar(A, 0);
    // N > 0 AND 0 >= 0 AND 0 < N AND
    // s = (SIGMA alfa EN [0,0]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
    int f = 0;
    // INV: N > 0 AND f >= 0 AND f < N AND
    //        s = (SIGMA alfa EN [0,f]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
    while (f != N − 1) {
        // N > 0 AND f >= 0 AND f < N AND f != N − 1 AND
        // s = (SIGMA alfa EN [0,f]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
        f = f + 1;
        // N > 0 AND f >= 1 AND f <= N − 1 AND
        // s = (SIGMA alfa EN [0,f−1]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
        // => [3]
        // N > 0 AND f >= 0 AND f < N

        // N > 0 AND f >= 1 AND f < N AND
        // s = (SIGMA alfa EN [0,f−1]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) ) AND
        // sumar(A,f) = (SIGMA beta EN [0,N−1]. A[f][beta]) )
        // => [4]
        // N > 0 AND f >= 0 AND f < N AND
        // s + sumar(A,f) = (SIGMA alfa EN [0,f]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
        s = s + sumar(A, f);
        // N > 0 AND f >= 0 AND f < N AND
        // s = (SIGMA alfa EN [0,f]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
    }
    // N > 0 AND f = N − 1 AND
    // s = (SIGMA alfa EN [0,f]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
    // => [5]
    // s= (SIGMA alfa EN [0,N−1]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
    return s;
    // sumar(A) = (SIGMA alfa EN [0,N−1]. (SIGMA beta EN [0,N−1]. A[alfa][beta]) )
}
```

Estas son las pruebas anotadas sobre el algoritmo anterior:

- **[1]** Su satisfacción prueba que la precondición de la invocación **sumar**(A,0) que sigue se satisface.
- **[2]** La satisfacción de la relación es inmediata y prueba la corrección del código programado antes del bucle.

- **[3]** Su satisfacción prueba que la precondición de la invocación **`sumar`**`(A,f)` que sigue se satisface.
- **[4]** Su satisfacción es inmediata y prueba la corrección del código a iterar en el bucle.
- **[5]** La satisfacción de la relación es inmediata ya que el primero de los predicados establece que $f = N - 1$ y prueba la corrección del código programado después del bucle.

La terminación del bucle se ha probado haciendo uso de la función de cota $f_{cota}(f) = N - f$ ya que esta función toma valores enteros decrecientes en cada bucle (condición [CT1]) y los valores enteros que toma están acotados inferiormente por el valor 1 (condición [CT2]).

```
/*
 * Pre: N > 0
 * Post: sumar(A) = (SIGMA alfa EN [0,N-1]. (SIGMA beta EN [0,N-1]. A[alfa][beta]) )
 */
template <typename T>
T sumar (const T A[N][N]) {
    T s = sumar(A, 0);
    int f = 0;
    while (f != N - 1) {
        // INV: N > 0 AND f >= 0 AND f < N AND
        //      s = (SIGMA alfa EN [0,f]. (SIGMA beta EN [0,N-1]. A[alfa][beta]) )

        // La prueba de la terminación se sustenta en la función de cota: f_cota(f) = N - f

        // f = A AND f_cota(antes) = f_cota(f) = N - f = N - A

        f = f + 1;
        s = s + sumar(A, f);

        // f = A + 1 AND f_cota(después) = f_cota(f) = N - f = N - A - 1

        // Condiciones que prueban la terminación del bucle:
        // [CT1] Los valores que toma la función de cota propuesta decrecen en cada iteración:
        //       f_cota(antes) > f_cota(después) ya que N - A > N - A - 1
        // [CT2] Los valores enteros que toma la función de cota tienen como límite inferior 1:
        //       INV => f < N => f_cota(f) = N - f > 0 => f_cota(f) = N - f >= 1
    }
    return s;
}
```

Una solución del problema 2º

El coste en tiempo de ejecutar **`ordenar`**`(v)` depende del número **`DIM`** de datos a ordenar pero, fijado un valor de **`DIM`** ***suficientemente grande*** (estamos haciendo un análisis asintótico del coste), también dependerá de la disposición de los **`DIM`** elementos de **`v`**.

- **Caracterización de los casos mejores**. Se presentan cuando los datos inicialmente almacenados en **`v`** están ya ordenados de menor a mayor valor. En tales casos el bloque interior del bucle **`while`** solo se ejecuta una vez, en la que se satisface la condición de la línea **`f`** que provoca la consiguiente terminación del bucle al satisfacerse $j = iLimite$. El coste de ejecutar el bucle interior se limita a la siguiente suma de costes $t_e + t_f + t_g + t_e$.

  Función de coste en tiempo en los casos mejores. Se va a determinar la función de coste como suma de costes de las diferentes líneas a ejecutar:

  $$t_{caso\ mejor}(DIM) = t_a + (\Sigma\alpha \in [1, DIM-1].\ t_b + t_c + t_d + t_e + t_f + t_g + t_e + t_j + t_a)$$
  $$t_{caso\ mejor}(DIM) = t_a + (t_b + t_c + t_d + 2 \times t_e + t_f + t_g + t_j + t_a) \times (DIM - 1)$$

  Caracterización asintótica de la función de coste en tiempo en los casos mejores.

  $$\boldsymbol{O}(t_{caso\ mejor}(DIM)) = \boldsymbol{O}(t_a + (t_b + t_c + t_d + 2 \times t_e + t_f + t_g + t_j + t_a) \times (DIM - 1))$$
  $$\boldsymbol{O}(t_{caso\ mejor}(DIM)) = \boldsymbol{O}((t_b + t_c + t_d + 2 \times t_e + t_f + t_g + t_j + t_a) \times (DIM - 1))$$
  $$\boldsymbol{O}(t_{caso\ mejor}(DIM)) = \boldsymbol{O}(DIM - 1) = \boldsymbol{O}(DIM)$$

- **Caracterización de los casos peores**. Se presentan cuando los datos almacenados en **`v`** están ordenados de mayor a menor valor. En tales casos en cada iteración del bucle **`for`** hay que insertar el dato **`v`**`[i]` en la ubicación `v[0]` (línea **`j`**), tras haber desplazado cada uno de los elementos de **`v`**`[0..i-1]` a la posición siguiente (línea **`h`**). El bloque interior del bucle **`while`** se ejecuta **`i`** veces y el coste de cada una de estas iteraciones es $t_f + t_h + t_i + t_e$. La ejecución del bucle interior while concluye cuando se satisface que $j = 0$.

  Función de coste en tiempo en los casos peores. Se va a determinar la función de coste como suma de costes de las diferentes líneas a ejecutar:

  $$\begin{aligned} t_{caso\ peor}(DIM) = {} & t_a + (\Sigma\alpha \in [1, DIM-1].\ t_b + t_c + t_d + t_e + \\ & (\Sigma\beta \in [1, i].\ t_f + t_h + t_i + t_e) + t_j + t_a) \end{aligned}$$
  $$\begin{aligned} t_{caso\ peor}(DIM) = {} & t_a + (t_b + t_c + t_d + t_e + t_j + t_a) \times (DIM - 1) + \\ & (t_f + t_h + t_i + t_e) \times (\Sigma\alpha \in [1, DIM-1].\ i) \end{aligned}$$
  $$\begin{aligned} t_{caso\ peor}(DIM) = {} & t_a + (t_b + t_c + t_d + t_e + t_j + t_a) \times (DIM - 1) + \\ & (t_f + t_h + t_i + t_e) \times (\Sigma\alpha \in [1, DIM-1].\ \alpha) \end{aligned}$$
  $$\begin{aligned} t_{caso\ peor}(DIM) = {} & t_a + (t_b + t_c + t_d + t_e + t_j + t_a) \times (DIM - 1) \\ & + (t_f + t_h + t_i + t_e)\tfrac{1}{2} DIM(DIM - 1) \end{aligned}$$
  $$\begin{aligned} t_{caso\ peor}(DIM) = {} & (t_f + t_h + t_i + t_e)\tfrac{1}{2} DIM(DIM - 1) + \\ & (t_b + t_c + t_d + t_e + t_j + t_a) \times (DIM - 1) + t_a \end{aligned}$$

  Caracterización asintótica de la función de coste en tiempo en los casos peores.

  $$\begin{aligned} \boldsymbol{O}(t_{caso\ peor}(DIM)) = \boldsymbol{O}(& (t_f + t_h + t_i + t_e)\tfrac{1}{2} DIM(DIM - 1) \\ & + (t_b + t_c + t_d + t_e + t_j + t_a) \times (DIM - 1) + t_a) \end{aligned}$$
  $$\boldsymbol{O}(t_{caso\ peor}(DIM)) = \boldsymbol{O}((t_f + t_h + t_i + t_e)\tfrac{1}{2} DIM(DIM - 1))$$
  $$\boldsymbol{O}(t_{caso\ peor}(DIM)) = \boldsymbol{O}(DIM(DIM - 1)) = \boldsymbol{O}(DIM^2)$$

Una solución del problema 3º

A continuación se presenta un diseño sin bucles de la función **mayor**(sec) sin haber hecho uso de ninguna función de las bibliotecas C++ predefinidas. El diseño se apoya en las funciones auxiliares **mayor**(sec, i, maximo) y **nuevo**(sec, i, numero, valor). La función **mayor**(sec, i, maximo) puede verse como el resultado de una inmersión o generalización de la función **mayor**(sec) planteada mediante refuerzo de la precondición de ésta.

```
/*
 * <secuencia> ::= <numero> { <separadores> <numero> } <nulo>
 * <numero> ::= <digito> { <digito> }
 * <digito> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
 * <nulo> ::= '\0'
 * <separadores> ::= " " { " " }
 */

// El carácter nulo que denota el final de una cadena de caracteres
const char NULO = '\0';

/*
 * Pre: <sec> almacena una secuencia de caracteres que responde a la sintaxis de <secuencia>,
 *      <valor> es el valor del número integrado por los dígitos que preceden en <sec>
 *      al carácter <sec[i]> y sea i = Io
 * Post: el carácter sec[i] no es un dígito y lo precede una secuencia de dígitos que incluye
 *       el carácter sec[Io] que definen un número almacenado en <sec>, cuyo valor entero ha
 *       sido asignado a <numero>
 */
void nuevo (const char sec [], int& i, int& numero, const int valor) {
    if (sec[i] >= '0' && sec[i] <= '9') {
        // sec[i] es un dígito
        i = i + 1;
        nuevo(sec, i, numero, 10 * valor + sec[i-1] -'0');
    }
    else {
        // sec[i] no es un dígito
        numero = valor;
    }
}
```

```
/*
 * Pre: <sec> almacena una secuencia de caracteres que responde a la sintaxis de <secuencia>,
 *      <maximo> es el valor del mayor de los números almacenados en la subsecuencia
 *      <sec[0..i-1]>, i > 0 y sec[i-1] no almacena un dígito
 * Post: Devuelve el valor del mayor de los números almacenados en <sec>
 */
int mayor (const char sec [], const int i, const int maximo) {
    if (sec[i] != NULO) {
        if (sec[i] < '0' || sec[i] > '9') {
            // sec[i] no es el carácter NULO ni es un dígito
            return mayor(sec, i + 1, maximo);
        }
        else {
            // sec[i] es el primer dígito de un número
            int numero;
            int desde = i;
            // reconoce el número que comienzo en sec[desde] y asigna su valor a <numero>
            nuevo(sec, desde, numero, 0);
            // actualiza, en su caso, el valor de <maximo> y prosigue la exploración
            // de <sec> a partir del carácter sec[desde] que es el que sigue al
            // dígito menos significativo del número reconocido
            if (maximo >= numero) {
                // no procede modificar el máximo valor reconocido hasta el momento
                // es el de <numero>
                return mayor(sec, desde, maximo);
            }
            else {
                // el nuevo máximo valor reconocido hasta el momento es el de <numero>
                return mayor(sec, desde, numero);
            }
        }
    }
    else {
        // sec[i] es el carácter NULO; la exploración de la secuencia concluye
        // y devuelve el máximo valor reconocido en <sec>
        return maximo;
    }
}

/*
 * Pre: <sec> almacena una secuencia de caracteres que responde a la sintaxis de <secuencia>
 *      y que puede contener uno o más números
 * Post: Devuelve el valor del mayor de los números almacenados en <sec>. Ejemplos:
 *       1. La invocación mayor("405") devuelve 405
 *       2. La invocación mayor("405 1098 745") devuelve 1098
 *       3. La invocación mayor("405   1098   745") devuelve 1098
 *       4. La invocación mayor("0 00 000 0000") devuelve 0
 *       5. La invocación mayor("102 00098 0222 37") devuelve 222
 */
int mayor (const char sec []) {
    // Asigna a <primero> el valor del primer número de <sec>
    int i = 0;
    int primero;
    nuevo(sec, i, primero, 0);
    // Lanza la exploración de nuevos números a partir de <sec[i]> y devuelve
    // el valor del mayor de todos los números almacenados en <sec>
    return mayor(sec, i, primero);
}
```