Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 20 de junio 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)

La que sigue es una especificación no formal de la función **`digitoUnidades`**`(v,n)`.

```
/*
 * Devuelve el dígito que más veces aparece como cifra de las unidades entre los naturales
 * almacenados en los <n> primeros elementos del vector <v>, con n <= #v, cuando estos elementos
 * se escriben en base 10.
 */
int digitoUnidades(const int v[], const int n);
```

Así, por ejemplo, si el contenido del vector **`vec`** es el siguiente:

```
vec = [ 18, 24, 10044, 16, 135, 28, 8, 4, 2217, 10 ]
```

La invocación **`digitoUnidades`**`(vec, 10)` puede devolver los valores 4 o 8, cualquiera de ellos, ya que son los dígitos de unidades más repetidos entre los 10 primeros elementos del vector **`vec`** (ambos están repetidos tres veces).

Por su parte, la invocación **`digitoUnidades`**`(vec, 7)` devolverá el valor 8 ya que es el dígito de unidades más repetido entre los 7 primeros elementos del vector **`vec`** (es el único que está repetido tres veces; las repeticiones de los demás dígitos no alcanza este número).

En este problema se pide lo siguiente:

- Sustituir la especificación no formal de la función **`digitoUnidades`**`(v,n)` por una especificación formal.
- Diseñar sin bucles el código de la función **`digitoUnidades`**`(v,n)`, apoyándose, en su caso, en las funciones auxiliares que se estime conveniente. Estas funciones, que no podrán ser funciones predefinidas, deberán contar con su correspondiente especificación formal y su código también estará libre de bucles.
- En este problema se valorará, esencialmente, que sean correctos los diseños (especifícación + código) de cada una de las funciones que integran la solución.

Problema 2º (3.0 puntos)

Demostrar formalmente la corrección de la función genérica que se presenta a continuación, aportando el conjunto de pruebas que permitan concluir que su diseño es correcto, sin olvidar incluir la prueba de su terminación.

Se valorará la claridad de la presentación de las pruebas y, en su caso, la adecuada justificación de la satisfacción de las relaciones entre predicados que se presenten en cada prueba.

```
// Pre: i ≥ 0 ∧ j < #orig ∧ j < #copia
// Post: (∀α ∈ [i, j].copia[α] = orig[α])
template <typename Dato>
void copiar (const Dato orig[], Dato copia[], const int i, const int j) {
    if ( i <= j ) {
        copiar(orig, copia, i, j - 1);
        copia[j] = orig[j];
    }
}
```

Problema 3º (3.0 puntos)

En este problema se pide:

1. Escribir los predicados **"*más fuertes*"** **`P1`**, **`P2`**, **`P3`**, **`P4`**, **`P5`** y **`P6`** que se satisfarán al alcanzar la ejecución del código de la función **`permutar`**`(v,n)` los puntos de su código en los que han sido escritos, asumiendo que la función **`permutar`**`(v,n)` se ha invocado satisfaciendo su precondición. **[1.5 puntos]**

2. A partir de los predicados **`P4`** y **`P5`**, probar formalmente la correccción del código a iterar en el bucle **while**, es decir, la corrección de la secuencia de cuatro instrucciones comprendida entre los predicados **`P4`** y **`P5`**. **[1.5 puntos]**

```
// Pre: n > 0 ∧ n <= #v ∧ (∀α ∈ [0, n − 1].v[α] = Vo[α])
// Post: (∀α ∈ [0, n − 1].v[α] = Vo[n − 1 − α])
template <typename T>
void permutar (T v[], const int n) {
    // P1
    int i = 0;
    // P2
    int medio = n / 2;
    // P3
    while ( i < medio ) {
        // P4
        T aux = v[i];
        v[i] = v[n-1-i];
        v[n-1-i] = aux;
        i = i + 1;
        // P5
    }
    // P6
}
```

Problema 4º (2.0 puntos)

```
/*
 * Pre: n >= 0
 * Post: elevar(x, n) = (PROD alfa EN [1,n]. x)
 */
double elevar (const double x, const int n) {
    int i = n;                                          // a
    double resultado = 1.0;                             // b
    double potencias = x;                               // c
    while (i != 0) {                                    // d
        if (i % 2 != 0) {                               // e
            resultado = potencias * resultado ;         // f
        }
        i = i / 2;                                      // g
        potencias = potencias * potencias ;             // h
     }
     return resultado ;                                 // i
}
```

Se pide un análisis del coste en tiempo de ejecutar una invocación **`elevar`**`(x,n)` a la función anterior. El análisis ha de constar de cuatro apartados en los que se responda las siguientes cuestiones:

1. Explicar de qué parámetro o combinación de parámetros depende su coste en tiempo.

2. Explicar si su coste asintótico en tiempo responde a una función de coste única o hay diversas funciones de coste según sea el valor del parámetro o combinación de parámetros del que depende el coste. En el segundo caso explicar claramente para qué valores de dichos parámetros presenta un coste asintótico en tiempo mayor (casos peores) y menor (casos mejores).

3. Deducir el valor de las funciones que caracterizan asintóticamente su coste en los casos peor y mejor, si es que cabe una diversidad de casos, o el valor de su funcion de coste si solo cabe considerar un único caso general.

   Se deberá explicar con claridad cada uno de los pasos dados para deducir la función o funciones de costes anteriores.

   Para deducir esta función o funciones de coste se asumirá que el coste de ejecutar el código de cada una de las líneas de la función es constante e igual a t_a, t_b, t_c, t_d, t_e, t_f, t_g, t_h e t_i, de acuerdo a la letra con la que ha sido etiquedada cada línea.

4. Si hubiera una diversidad de casos en cuanto al coste, explicar la diferencia existente entre las funciones que caracterizan asintóticamente el coste en tiempo en los casos mejor y peor.

Una solución del problema 1º

Especificación formal y diseño sin bucles de la función **`digitoUnidades`**`(v,n)`.

```
/*
 * Pre: n >= 0 AND n <= #v
 * Post: (PT alfa EN [0,n-1], v[alfa] = 0)
 */
void anular (int v[], const int n) {
    if (n > 0) { v[n - 1] = 0; anular(v, n-1); }
}

/*
 * Pre: n > 0 AND n <= #v AND iMayor >= 0 AND iMayor < hasta AND
 *      hasta < n AND (PT alfa EN [0,hasta]. v[alfa] <= v[iMayor])
 * Post: indiceMayor(v,iMayor,hasta,n) = IND AND IND >= 0 AND IND < n AND
 *       (PT alfa EN [0,n-1]. v[alfa] <= v[IND])
 */
int indiceMayor (const int v[], const int iMayor, const int hasta, const int n) {
    if (hasta != n - 1) {
        if (v[hasta + 1] > v[iMayor]) {

            return indiceMayor(v, hasta + 1, hasta + 1, n);
        }
        else {
            return indiceMayor(v, iMayor, hasta + 1, n);
        }
    }
    else {
        return iMayor;
    }
}

/*
 * Pre: desde >= 0 AND AND desde <= n AND n <= #v AND
 *      (PT alfa EN [0,n-1]. v[alfa} >= 0) AND
 *      (PT alfa EN [0,9]. repeticiones[alfa] = (NUM beta [0, desde-1]. v[beta] = alfa) )
 * Post: (PT alfa EN [0,9]. repeticiones[alfa] = (NUM beta [0, n-1]. v[beta] = alfa) )
 */
void contarRepeticiones (const int v[], const int desde, const int n, int repeticiones []) {
    if (desde < n) {
        int digito = v[desde] % 10;
        repeticiones [ digito ] = repeticiones [ digito ] + 1;
        contarRepeticiones (v, desde + 1, n, repeticiones );
    }
}

/*
 * Pre: n >= 1 AND n <= #v AND (PT alfa EN [0,n-1]. v[alfa} >= 0)
 * Post: digitoUnidades(v,n) = D AND D >= 0 AND <= 9 AND
 *       (PT alfa EN [0,9]. (NUM beta EN[0,n-1]. v[beta] = D) >=
 *                          (NUM beta EN[0,n-1]. v[beta] = v[alfa]) )
 */
int digitoUnidades (const int v[], const int n) {
    const int DIEZ = 10;
    int repeticiones [DIEZ];
    anular( repeticiones , DIEZ);
    contarRepeticiones (v, 0, n, repeticiones );
    return indiceMayor( repeticiones , 0, 0, DIEZ);
}
```

Una solución del problema 2º

Tres de las cuatro pruebas que demuestran formalmente la corrección de la función **`copiar`**`(orig, copia, i, j)` se presentan a continuación (falta la prueba de la terminación de la secuencia de invocaciones recursivas). Las tres pruebas se han anotado intercaladas en su código.

```
/*
 * Pre: i >= 0 AND j < #orig AND j < #copia
 * Post: (PT alfa EN [i,j]. copia[alfa] = orig[alfa])
 */
template <typename Dato>
void copiar (const Dato orig [], Dato copia [], const int i, const int j) {
    // i >= 0 AND j < #orig AND j < #copia
    if (i <= j) {
        // i >= 0 AND i <= j AND j < #orig AND j < #copia
        // => [1]
        // i >= 0 AND j - 1 < #orig AND j - 1 < #copia
        copiar(orig, copia, i, j - 1);
        // i >= 0 AND i <= j AND j < #orig AND j < #copia AND
        // (PT alfa EN [i,j-1]. copia[alfa] = orig[alfa])
        // => [2]
        // j >= 0 AND j < #orig AND j < #copia AND
        // (PT alfa EN [i,j-1]. copia[alfa] = orig[alfa]) AND orig[j] = orig[j]
        copia[j] = orig[j];
        // (PT alfa EN [i,j]. copia[alfa] = orig[alfa])
    }
    else {
        // i >= 0 AND i > j AND j < #orig AND j < #copia
        // => [3]
        // (PT alfa EN [i,j]. copia[alfa] = orig[alfa])
    }
}
```

Justificación de las pruebas presentadas:

- Se ha omitido la prueba de la satisfacción de las condiciciones de dominio de la guarda de la instrucción condicional ($i \leq j$) dado que Dom($i \leq j$) = $true$
- Al calcular la precondición más debil que debe satisfacerse inmediatamente antes de la instrucción de asignación $copia[j] = orig[j]$; se han incorporado sus condiciciones de dominio, es decir, Dom($copia[j] = orig[j]$) = $j \geq 0 \ \wedge \ j < \#orig \ \wedge \ j < \#copia$.
- La relación entre predicados [1] es evidente y prueba la satisfacción de la precondición de la invocación **`copiar`**`(orig,copia,i,j-1)`.
- La satisfacción de la relación entre predicados [2] es inmediata y prueba la corrección del código programado en la primera de las cláusulas de la instrucción condicional.
- La relación entre predicados [3] es también evidente dado que `[i,j]` define un conjunto vacío de índices y prueba la corrección del código programado en la segunda de las cláusulas de la instrucción condicional, su cláusula implícita **`else`**.

La demostración de la terminación de las sucesivas invocaciones recursivas se ha anotado sobre el código de la función. Se ha realizado a partir de la función de cota $f_{cota} = j - i$ y consiste en probar que el valor de la función de cota decrece al producirse una nueva invocación recursiva (condición 1) y que el valor que toma la función de cota al producirse una nueva invocación recursiva está acotado inferiormente en $\mathbb{Z}$ (condición 2).

```
/*
 * Pre: i >= 0 AND j < #orig AND j < #copia
 * Post: (PT alfa EN [i,j]. copia[alfa] = orig[alfa])
 */
template <typename Dato>
void copiar (const Dato orig [], Dato copia [], const int i, const int j) {
    // Proponemos esta función de cota: f_cota = j - i
    // Al invocar a esta función: f_cota( invocación_inicial ) = j - i

    if (i <= j) {
        // i >= 0 AND i <= j AND j < #orig AND j < #copia
        copiar(orig, copia, i, j - 1);
        copia[j] = orig[j];

        // Al invocar copiar(orig,copia,i,j-1):
        //          f_cota( invocación_recursiva ) = j - 1 - i
        // Se satisfacen las dos condiciones que garantizan la terminación de la recursión:
        // 1) El valor de la función de cota decrece:
        //          f_cota( inicio ) > f_cota( invocación_recursiva )
        //    ya que:
        //          j - i > j - 1 - i
        // 2) En las invocaciones recursivas, la función de cota toma valores en Z que
        //    están acotados inferiormente:
        //          i >= 0 AND i <= j AND j < #orig AND j < #copia AND copia[j] = original[j]
        //          =>
        //          f_cota = j - i >= 0
    }
}
```

Una solución del problema 3º

1. Los seis predicados pedidos se han anotado sobre el propio código de la función para facilitar su comprensión.

```
// Pre: n > 0 ∧ n <= #v ∧ (∀α ∈ [0, n − 1].v[α] = Vo[α])
// Post: (∀α ∈ [0, n − 1].v[α] = Vo[n − 1 − α])
template <typename T>
void permutar (T v[], const int n) {
    // P1: n > 0 ∧ n <= #v ∧ (∀α ∈ [0, n − 1].v[α] = Vo[α])
    int i = 0;
    // P2: n > 0 ∧ n <= #v ∧ (∀α ∈ [0, n − 1].v[α] = Vo[α]) ∧ i = 0
    int medio = n / 2;
    // P3: n > 0 ∧ n <= #v ∧ (∀α ∈ [0, n − 1].v[α] = Vo[α]) ∧ i = 0 ∧ medio = n/2
    while ( i < medio ) {
        // P4: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i < medio ∧ medio = n/2 ∧
        //     (∀α ∈ [0, i − 1].v[α] = Vo[n − 1 − α]) ∧
        //     (∀α ∈ [0, i − 1].v[n − 1 − α] = Vo[α]) ∧
        //     (∀α ∈ [i, n − 1 − i].v[α] = Vo[α])
        T aux = v[i];
        v[i] = v[n-1-i];
        v[n-1-i] = aux;
        i = i + 1;
        // P5: n > 0 ∧ n <= #v ∧ i ≥ 1 ∧ i ≤ medio ∧ medio = n/2 ∧
        //     (∀α ∈ [0, i − 1].v[α] = Vo[n − 1 − α]) ∧
        //     (∀α ∈ [0, i − 1].v[n − 1 − α] = Vo[α]) ∧ aux = Vo[i − 1] ∧
        //     (∀α ∈ [i, n − 1 − i].v[α] = Vo[α])
    }
    // P6: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i = medio ∧ medio = n/2 ∧
    //     (∀α ∈ [0, n − 1].v[α] = Vo[n − 1 − α])
}
```

2. Los cálculos de precondiciones más débiles, **pmd01**, **pmd02**, **pmd03** y **pmd04** (numeradas en orden de cálculo), que hacen correcto el diseño a partir de cada una de las instrucciones de la secuencia y la prueba final de la corrección de la secuencia completa (la satisfacción de la relación [1] entre los predicados **P4** y **pmd01**), se presentan a continuación.

   La satisfacción de la relación [1] es evidente si se tiene en cuenta que:

   $(\forall \alpha \in [i, n-1-i].v[\alpha] = Vo[\alpha]) \equiv$
   $v[i] = Vo[i] \ \wedge \ (\forall \alpha \in [i+1, n-2-i].v[\alpha] = Vo[\alpha]) \ \wedge \ v[n-1-i] = Vo[n-1-i]$

```
// P4: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i < medio ∧ medio = n/2 ∧
//     (∀α ∈ [0, i−1].v[α] = Vo[n−1−α]) ∧
//     (∀α ∈ [0, i−1].v[n−1−α] = Vo[α]) ∧
//     (∀α ∈ [i, n−1−i].v[α] = Vo[α])
// ⇒ [1]
// pmd04 : n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ n−1−i ≥ 0 ∧ n−1−i < #v ∧
//         i < medio ∧ medio = n/2 ∧
//         (∀α ∈ [0, i−1].v[α] = Vo[n−1−α]) ∧ v[n−1−i] = Vo[n−1−i] ∧
//         (∀α ∈ [0, i−1].v[n−1−α] = Vo[α]) ∧ v[i] = Vo[i] ∧
//         (∀α ∈ [i+1, n−2−i].v[α] = Vo[α])
T aux = v[i];
//pmd03 : n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ n−1−i ≥ 0 ∧ n−1−i < #v ∧
//         i < medio ∧ medio = n/2 ∧
//         (∀α ∈ [0, i−1].v[α] = Vo[n−1−α]) ∧ v[n−1−i] = Vo[n−1−i] ∧
//         (∀α ∈ [0, i−1].v[n−1−α] = Vo[α]) ∧ aux = Vo[i] ∧
//         (∀α ∈ [i+1, n−2−i].v[α] = Vo[α])
v[i] = v[n-1-i];
// pmd02 : n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ n−1−i ≥ 0 ∧ n−1−i < #v ∧
//         i < medio ∧ medio = n/2 ∧
//         (∀α ∈ [0, i].v[α] = Vo[n−1−α]) ∧
//         (∀α ∈ [0, i−1].v[n−1−α] = Vo[α]) ∧ aux = Vo[i] ∧
//         (∀α ∈ [i+1, n−2−i].v[α] = Vo[α])
v[n-1-i] = aux;
//pmd01 : n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i < medio ∧ medio = n/2 ∧
//         (∀α ∈ [0, i].v[α] = Vo[n−1−α]) ∧
//         (∀α ∈ [0, i].v[n−1−α] = Vo[α]) ∧ aux = Vo[i] ∧
//         (∀α ∈ [i+1, n−2−i].v[α] = Vo[α])
i = i + 1;
// P5: n > 0 ∧ n <= #v ∧ i ≥ 1 ∧ i ≤ medio ∧ medio = n/2 ∧
//     (∀α ∈ [0, i−1].v[α] = Vo[n−1−α]) ∧
//     (∀α ∈ [0, i−1].v[n−1−α] = Vo[α]) ∧ aux = Vo[i−1] ∧
//     (∀α ∈ [i, n−1−i].v[α] = Vo[α])
```

Una solución del problema 4º

1. El coste en tiempo de ejecutar una invocación de la función **`elevar`**`(x,n)` depende exclusivamente del valor de su segundo parámetro, del valor de **`n`**.

2. Hay una diversidad de casos en cuanto a la caracterización de su función de coste.

   El **caso mejor** se presenta cuando el valor del parámetro **`n`** es una potencia de 2 ya que en tal caso el código de la línea **`f`** solo se ejecuta una vez (en la última iteración del búcle).

   El **caso peor** se presenta cuando el valor del parámetro **`n`** es igual a una unidad menos que una potencia de 2, es decir, $n = 2^k - 1$, ya que en tal caso el código de la línea **`f`** se ejecuta **`k`** veces, es decir, $log_2\ (n+1)$ veces.

3. **Función de coste en tiempo en el caso mejor**. En el caso mejor el valor de **`n`** es igual a una potencia de 2, es decir, $n = 2^k$. El bucle iterará $1 + k$ veces, es decir, $1 + log_2\ n$ veces y la línea **`f`** solo será ejecutada una vez.

   $$t_{mejor}(n) = t_a + t_b + t_c + t_d + (\Sigma\alpha \in [1, 1 + log_2\ n].\ t_e + t_g + t_h + t_d) + t_f + t_i$$
   $$t_{mejor}(n) = (t_e + t_g + t_h + t_d) \times (1 + log_2\ n) + t_a + t_b + t_c + t_d + t_f + t_i$$
   $$t_{mejor}(n) = (t_e + t_g + t_h + t_d) \times log_2\ n + t_a + t_b + t_c + t_d + t_e + t_f + t_g + t_h + t_i$$

   Función de coste en tiempo en el caso peor. En el caso peor el valor de **`n`** es una unidad inferior a una potencia de 2, es decir, $n = 2^k - 1$. El bucle iterará $k = log_2\ (n+1)$ veces, es decir, $log_2\ (n+1)$ veces y la línea **`f`** será ejecutada en todas esas iteraciones.

   $$t_{peor}(n) = t_a + t_b + t_c + t_d + (\Sigma\alpha \in [1, log_2\ (n+1)].\ t_e + t_f + t_g + t_h + t_d) + t_i$$
   $$t_{peor}(n) = (t_e + t_f + t_g + t_h + t_d) \times log_2\ (n+1) + t_a + t_b + t_c + t_d + t_i$$

4. Las funciones de coste en tiempo en el caso mejor y peor son ambas logarítmicas en **`n`**. Si tenemos en cuenta que, para valores de **`n`** suficientemente grandes, $log_2\ n \approx log_2\ (n+1)$ su única diferencia radica en el coeficiente del término logarítmico que es mayor en el caso peor $(t_e + t_f + t_g + t_h + t_d)$ que en el caso mejor $(t_e + t_g + t_h + t_d)$ a causa del diferente número de veces que se ejecuta la línea **`f`** del código.