Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 11 de Junio de 2012

- 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 siguientes documentos: *Breve resumen del lenguaje Java* y *Capítulo 1 de las Notas del curso*.

Problema 1º (5.5 puntos)

```
/* Pre: T.length > 1 */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1]) */
private static void definirTabla (double[] T, double x, double y) {
  | ... Código del método ...
}
```

En este problema se pide:

1. Hacer un diseño iterativo del método *definirTabla*, empleando exclusivamente bucles **while**. Documentar cada bucle con un predicado invariante que permita sustentar las pruebas que se piden a continuación. [1 punto]
2. Presentar el conjunto de pruebas formales necesarias para demostrar la corrección del código desarrollado en el apartado anterior. [2.5 puntos]
3. Hacer ahora un diseño sin bucles del método *definirTabla* aplicando, en su caso, un diseño recursivo. Especificar formalmente cada uno de los métodos que formen parte del diseño. [2.0 puntos]

Problema 2º (2 puntos)

En este problema se debe comparar la eficiencia de los dos diseños del primer problema. En concreto, se pide:

1. Caracterizar asintóticamente, en función del tamaño de la tabla *T1*, el coste en tiempo de la invocación *definirTabla(T1, 0.8945, -0.4512)* del método desarrollado en el primer apartado del primer problema. Se valorará fundamentalmente la claridad y corrección de los cálculos que permiten deducir los resultados.
2. Caracterizar asintóticamente, en función del tamaño de la tabla *T1*, el coste en tiempo de la invocación *definirTabla(T1, 0.8945, -0.4512)* del método desarrollado en el tercer apartado del primer problema. Se valorará fundamentalmente la claridad y corrección de los cálculos que permiten deducir los resultados.

3. Hacer un análisis comparativo de la eficiencia de los dos diseños propuestos del método *definirTabla* del que se debe concluir una propuesta justificada del método que resulte más eficiente. Se valorará la calidad y claridad de la redacción y la solidez de los argumentos que se expongan.

Problema 3º (2.5 puntos)

Presentar el conjunto de pruebas formales necesarias para demostrar la corrección del código del método *calcular*. El dato *mayor* es un atributo del objeto al que se aplica el método.

```
/* Pre: n ≥ 0      */
/* Post: mayor = (Máx α ∈ [1, ∞].digito(n, α))      */
private void calcular (int n) {
    if ( n < 10 ) {
        mayor = n;
    }
    else {
        calcular(n/10);
        if ( n %10 > mayor ) {
            mayor = n %10;
        }
    }
}
```

Donde: $digito(n, \alpha) = (n/10^{\alpha-1})\,\%10$

Una solución del problema 1º

1. Código iterativo del método *definirTabla*.

```
/* Pre: T.length > 1 */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1]) */
private static void definirTabla (double[] T, double x, double y) {
    T[0] = x;  T[1] = y;
    int i = 1;
    while (i! = T.length − 1) {
        /* Inv: i ≥ 1 ∧ i < T.length
                ∧T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1]) */
        i = i + 1;
        T[i] = T[i-2] + T[i-1];
    }
}
```

2. Conjunto de pruebas que demuestran la corrección del código del método *definirTabla*.

```
/* Pre: T.length > 1 */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1]) */
private static void definirTabla (double[] T, double x, double y) {
    /* T.length > 1  ⇒
       1 ≥ 1 ∧ 1 < T.length ∧ x = x ∧ y = y ∧ (∀α ∈ [2, 1].T[α] = T[α − 2] + T[α − 1]) */
    T[0] = x;  T[1] = y;
    int i = 1;
    /* Inv: i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y
            ∧(∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1])
            ⇒  Dom(i ≠ T.length − 1)  ≡  cierto  */
    while (i! = T.length − 1) {     /* f_cota(i) = T.length − i ∧ f_cota(i) ≥ 0  */
        /* i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1])
           ∧i ≠ T.length − 1
           ⇒  i + 1 ≥ 1 ∧ i + 1 < T.length ∧ T[0] = x ∧ T[1] = y
                ∧T[i − 2] + T[i − 1] = T[i − 2] + T[i − 1]
                ∧(∀α ∈ [2, i + 1 − 1].T[α] = T[α − 2] + T[α − 1])  */
        /* i = A ∧ f_cota(i) = T.length − A  */
        i = i + 1;
        /* i − 1 ≥ 1 ∧ i − 1 < T.length ∧ T[0] = x ∧ T[1] = y
           ∧(∀α ∈ [2, i − 1].T[α] = T[α − 2] + T[α − 1]) ∧ i ≠ T.length)
           ⇒  DOM(T[i] = T[i − 2] + T[i − 1]) ≡ i ≥ 2 ∧ i < T.length  */
        T[i] = T[i-2] + T[i-1];
        /* i = A + 1 ∧ f_cota(i) = T.length − A − 1  */
        /* Inv: i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y
                ∧(∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1])  */
    }
    /* i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1])
       ∧i = T.length − 1
       ⇒  T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1])  */
}
```

El bucle termina ya que: (1) La función de cota decrece tras ejecutar el bloque asociado al bucle ya que $T.length - A > T.length - A - 1$ y (2) la función de cota está acotada inferiormente, $f_{cota}(i) \geq 0$, lo cual garantiza que la sucesión de valores que puede tomar es finita.

3. Diseño recursivo y sin bucles del método *definirTabla*.

```
/* Pre: T.length > 1 */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[alpha − 1]) */
private static void definirTabla (double[] T, double x, double y) {
    T[0] = x;   T[1] = y;
    definirTabla(T, 1);
}

/* Pre: T.length > 1 ∧ i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y
        ∧(∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1]) */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1]) */
private static void definirTabla (double[] T, int i) {
    if (i! = T.length − 1) {
        T[i+1] = T[i-1] + T[i];
        definirTabla(T, i+1);
    }
}
```

Una solución del problema 2º

1. Caracterización asintótica del coste de la invocación *definirTabla(T1, 0.8945, -0.4512)* del método iterativo desarrollado en el primer apartado del primer problema.

```
/* Pre: T.length > 1 */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1]) */
private static void definirTabla (double[] T, double x, double y) {
    T[0] = x;   T[1] = y;                    /* a */
    int i = 1;                               /* b */
    while (i! = T.length − 1) {              /* c */
        /* Inv: i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y
                ∧(∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1]) */
        i = i + 1;                           /* d */
        T[i] = T[i-2] + T[i-1];              /* e */
    }
}
```

La función de coste depende del número de datos $T1.length$ de la tabla **T1**:

$$t(T1.length) = t_a + t_b + t_c + (\Sigma\alpha \in [2, T1.length - 1].t_d + t_e + t_c)$$
$$= (t_d + t_e + t_c)(T1.length - 2) + t_a + t_b + t_c$$

$\boldsymbol{O}(t(T1.length))$ = $\boldsymbol{O}((t_d + t_e + t_c)(T1.length - 2) + t_a + t_b + t_c))$ = $\boldsymbol{O}(T1.length)$

2. Caracterización asintótica del coste de la invocación *definirTabla(T1, 0.8945, -0.4512)* del método sin bucles desarrollado en el tercer apartado del primer problema.

```
/* Pre: T.length > 1 */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[alpha − 1]) */
private static void definirTabla (double[] T, double x, double y) {
    T[0] = x;   T[1] = y;                /* a */
    definirTabla(T, 1);                  /* b */
}

/* Pre: T.length > 1 ∧ i ≥ 1 ∧ i < T.length ∧ T[0] = x ∧ T[1] = y
        ∧(∀α ∈ [2, i].T[α] = T[α − 2] + T[α − 1]) */
/* Post: T[0] = x ∧ T[1] = y ∧ (∀α ∈ [2, T.length − 1].T[α] = T[α − 2] + T[α − 1]) */
private static void definirTabla (double[] T, int i) {
    if (i! = T.length − 1) {             /* c */
        T[i+1] = T[i-1] + T[i];            /* d */
        definirTabla(T, i+1);            /* e */
    }
}
```

La función de coste depende del número de datos $T1.length$ de la tabla **T1**:

$$t(T1.length) = t_a + t_b + t_{definirTabla(T1,1)}$$

Vamos a plantear una recurrencia en función del valor de la variable $n = T1.length - i + 1$:

$$t_{definirTabla(T1,1)}(n) = t_c + t_d + t_e + t_{definirTabla(T1,2)}(n-1)$$

Es decir:

$$t_{definirTabla(T1,1)}(n) - t_{definirTabla(T1,2)}(n-1) = t_c + t_d + t_e$$

Ecuación característica:

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

Raíces:

$x = 1$ (raíz doble)

Solución:

$$t_{definirTabla(T1,1)}(n) = c_1 \times n + c_2 = c_1(T1.length - 1 + 1) + c_2 = c_1(T1.length) + c_2$$

Por lo tanto:

$$t(T1.length) = t_a + t_b + t_{definirTabla(T1,1)} = t_a + t_b + c_1(T1.length) + c_2$$

Y, en consecuencia:

$$\boldsymbol{O}(t(T1.length)) = \boldsymbol{O}(t_a + t_b + c_1(T1.length) + c_2) = \boldsymbol{O}(T1.length)$$

3. Análisis comparativo de la eficiencia de ambos métodos. El coste, en tiempo, de ambas invocaciones es lineal en el número de datos de la tabla **T1**. El primero de los métodos será más eficiente en lo relativo a uso de memoria ya que sólo requiere memoria, además de para los tres parámetros, para la variable local **i**. En cambio la segunda solución necesita, para cada invocacíón recursiva, memoria para los dos parámetros **T** e *i* del método recursivo generalizado *definirTabla*. Por lo tanto la memoria necesaria será lineal en el número de datos de la tabla **T1**.

Por lo tanto, lo aconsejable es programar y utilizar el primero de los métodos, el que responde a un diseño iterativo.

Una solución del problema 3º

Conjunto de pruebas que demuestran la corrección del código del método *calcular*. Se omiten las pruebas relativas a la satisfacción de las condiciones de dominio de las expresiones programadas en el método ($n < 10$, n, $n/10$, $n\,\%10 > mayor$, y $n\,\%10$) ya que todas ellas equivalen al predicado $cierto$

```
/* Pre: n ≥ 0      */
/* Post: mayor = (Máx α ∈ [1,∞].digito(n,α))      */
private void calcular (int n) {
    if ( n < 10 ) {
        /*  n ≥ 0  ∧  n < 10  ⇒  n = (Máx α ∈ [1,∞].digito(n,α))  */
        mayor = n;
    }
    else {
        /*  n ≥ 10  ⇒  n/10 ≥ 0  */
        calcular(n/10);
        /*  n ≥ 10  ∧  mayor = (Máx α ∈ [1,∞].digito(n/10,α))  */
        if ( n %10 > mayor ) {
            /*  n ≥ 10  ∧  mayor = (Máx α ∈ [1,∞].digito(n/10,α))  ∧ n %10 > mayor
                ⇒  n %10 = (Máx α ∈ [1,∞].digito(n,α))  */
            mayor = n %10;
        else {
            /*  n ≥ 10  ∧  mayor = (Máx α ∈ [1,∞].digito(n/10,α))  ∧ n %10 ≤ mayor
                ⇒  mayor = (Máx α ∈ [1,∞].digito(n,α))  */
            ;
        }
    }
}
```

Donde: $digito(n, \alpha) = (n/10^{\alpha-1})\,\%10$

Por claridad, las pruebas de la terminación de la recursión se presentan a continuación (decrecimiento de la función de cota, $f_{cota} = n$ y finitud de la secuencia de valores que toma la función de cota en invocaciones recursivas sucesivas).

```
/* Pre: n ≥ 0      */
/* Post: mayor = (Máx α ∈ [1, ∞].digito(n, α))      */
private void calcular (int n) {
    /*  n ≥ 0  ⇒  f_cota_1 = n  ∧ f_cota_1 ≥ 0  */
    if ( n < 10 ) {
        mayor = n;
    }
    else {
        /*  n ≥ 10  ∧  f_cota_2 = n/10 ≥ 0  */
        /* La función de cota toma valores decrecientes (f_cota_1 > f_cota 2) ya que:
                n ≥ 10  ⇒  n > n/10  */
        /* La función de cota está acotada inferiormente ya que siempre:  f_cota ≥ 0  */
        mayor = calcular(n/10);
        if ( n %10 > mayor ) {
            mayor = n %10;
        }
    }
}
```

Donde: $digito(n, \alpha) = (n/10^{\alpha-1})\,\%10$