Ingeniería Informática - Depto. de Informática e Ingeniería de Sistemas
Examen de Programación 2 - 13 de septiembre de 2013

- 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 4 del Curso de Programación 2: Introducción a la especificación formal de algoritmos*.

Problema 1º (1.5 puntos)

Especificar formalmente, mediante un par de predicados pre y postcondición, el comportamiento de los métodos ***esTriangular*** (0.5 puntos) y ***esCuriosa*** (1.0 punto) cuyas especificaciones no formales se muestran a continuación.

```
/**
 * n es un número natural distinto de 0.
 * El método devuelve cierto si n es un número triangular y falso en caso contrario.
 * Un número natural es triangular si es igual a la suma de un cierto número de naturales
 * consecutivos comenzando por el 1
 */
private static boolean esTriangular (int n)

/**
 * T es una referencia a una tabla de enteros no vacía.
 * El método devuelve cierto si T es curiosa y falso en caso contrario.
 * Diremos que una tabla de enteros es curiosa si alguno de sus elementos es un totalizador.
 * Diremos que un elemento de una tabla de enteros es un totalizador si su valor coincide con
 * la suma de todos elementos de la tabla, exceptuado él mismo.
 */
public static boolean esCuriosa (int [] T)
```

Problema 2º (2.5 puntos)

Caracterizar asintóticamente el coste de ejecución de una invocación del método público Java *ordenar(T)* en función del número *DIM* de referencias a objetos almacenadas en la tabla a ordenar ($DIM = T.length$) asumiendo que la invocación del método de comparación de objetos [*Dato*] de nombre *compareTo* presenta un coste cuyo orden es constante, es decir, $\boldsymbol{O}(1)$. Presentar los cálculos detallados que conducen al resultado. Distinguir, en caso necesario, el coste asintótico en el caso mejor y en el caso peor, explicando muy claramente cuándo se presentan esos casos extremos.

```
/**
 * Pre: T = To
 * Post: esPermutación(T,To,0,T.length−1) AND ordenada(T,0,T.length−1)
 */
public static <Dato extends Comparable <Dato>> void ordenar (Dato[] T) {
    /*
     * Ordenación de la tabla T aplicando el método de inserción
     */
    for (int i=1; i<=T.length−1; ++i) {
        Dato dato = T[i];
        int j = i, iLimite = 0;
        while (j!=iLimite) {
            if (T[j−1].compareTo(dato)<=0) {
                iLimite = j;
            }
            else {
                T[j] = T[j−1]; −−j;
            }
        }
        T[j] = dato;
    }
}
```

Problema 3º (2.5 puntos)

Se pide diseñar dos veces el método *binarizar(double[], int[])* de forma que su código no tenga bucles (ni tampoco en los métodos privados auxiliares que hubiera que desarrollar). En el primer diseño se aplicará la técnica de inmersión mediante refuerzo de la precondición y en el segundo diseño se aplicará la técnica de inmersión mediante debilitamiento de la postcondición. Se valorará fundamentalmente que los dos métodos *binarizar(double[], int[])* y sus métodos auxiliares presenten todos ellos un diseño correcto.

```
/* Pre: T.length > 0 ∧ T.length = B.length ∧ T = To   */
/* Post: (∀α ∈ [0,T.length − 1].(T[α] ≥ 0.0 → B[α] = 1) ∧ (T[α] < 0.0 → B[α] = 0))
         ∧ T = To ∧ binarizar(T,B) = (Σα ∈ [0,B.length − 1].B[α])   */
public static int binarizar (double[] T, int[] B) {
  | ...    Código del método    ...
}
```

Problema 4º (3.5 puntos)

El código iterativo que se presenta a continuación realiza una búsqueda binaria con garantía de éxito en una tabla ordenada.

```
/* T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1])
   ∧ (∃α ∈ [0, T.length − 1].T[α] = dato) ∧ izdo = 0 ∧ dcho = T.length − 1   */
while ( izdo! = dcho ) {
    int medio = (izdo + dcho)/2;
    if ( dato <= T[medio] ) {
        dcho = medio;
    }
    else {
        izdo = medio + 1;
    }
}
/* T[izdo] = dato   */
```

Se pide:

1. Escribir un invariante del bucle **while** que permita demostrar la corrección del diseño. [1 punto]
2. A partir del mismo invariante, escribir las pruebas formales que permiten demostrar la corrección del código del bloque a iterar en el bucle **while**. [1.5 puntos]
3. Escribir las pruebas formales que permiten demostrar la terminación del bucle **while**. [0.5 puntos]
4. Escribir las pruebas formales que faltarían para completar la demostración de la corrección de la anterior instrucción iterativa **while** (satisfacción inicial del invariante del bucle y satisfacción de la postcondición del bucle). [0.5 puntos]

Una solución del problema 1º

```
/* Pre: n > 0   */
/* Post: esTriangular(n) = (∃α ∈ ℕ. n = (Σβ ∈ [1, α].β))   */
public static boolean esTriangular (int n)
```

```
/* Pre: T! = null ∧ T.length > 0   */
/* Post: esCuriosa(T) = (∃α ∈ [0, T.lenth].esTotalizador(T, α))
        Donde : esTotalizador(V, i) = (V[i] = (Σα ∈ [0, i − 1].V[α])
                                              + (Σα ∈ [i + 1, T.length − 1].V[α]))   */
private static boolean esCuriosa (int[] T)
```

Una solución del problema 2º

Etiquetamos las líneas del método *ordenar(Dato[])* que contienen código ejecutable.

```
/**
 * Pre: T = To
 * Post: esPermutación(T,To,0,T.length−1) AND ordenada(T,0,T.length−1)
 */
public static <Dato extends Comparable <Dato>> void ordenar (Dato[] T) {
    /*
     * Ordenación de la tabla T aplicando el método de inserción
     */
    for (int i=1; i<=T.length−1; ++i) {                    /* a */
        Dato dato = T[i];                                  /* b */
        int j = i, iLimite = 0;                            /* c */
        while (j!=iLimite) {                               /* d */
            if (T[j−1].compareTo(dato)<=0) {               /* e */
                iLimite = j;                               /* f */
            }
            else {
                T[j] = T[j−1]; −−j;                        /* g */
            }
        }
        T[j] = dato;                                       /* h */
    }
}
```

El caso mejor se presenta cuando en la primera iteración del bucle **while** se satisface la condición $T[j-1].compareTo(dato) <= 0$ de su instrucción condicional. En tal caso el bucle deja de iterar de forma inmediata. El caso mejor se presenta cuando los datos de la tabla están ya ordenados de menor a mayor valor al ser invocado el método *ordenar*.

El caso peor se presenta cuando en la ninguna de las iteraciones del bucle **while** se satisface la condición $T[j-1].compareTo(dato) <= 0$ de su instrucción condicional. En tal caso el bloque de instrucciones asociado al bucle se ejecuta i veces. El caso peor se presenta cuando los datos de la tabla están ya ordenados al ser invocado el método *ordenar*, pero en orden inverso, es decir de mayor a menor valor.

Para simplificar la notación denominaremos DIM al número de datos $T.length$ de la tabla a ordenar.

- **Caracterización asintótica del coste en tiempo en el caso mejor**

  Es sencillo calcular la suma de costes al ejecutar el código ya que el bloque del bucle *while* sólo se ejecuta una vez.

  $t(DIM) = t_a + (\Sigma\alpha \in [1, DIM-1].t_b + t_c + t_d + (t_e + t_f + t_d) + t_h + t_a)$

  $t(DIM) = t_a + (t_b + t_c + 2 \times t_d + t_e + t_f + t_h + t_a)(DIM - 1)$

  Y la caracterización asintótica del coste en tiempo será:

  $\boldsymbol{O}(t(DIM)) = \boldsymbol{O}(t_a + (t_b + t_c + 2 \times t_d + t_e + t_f + t_h + t_a)(DIM - 1) = \boldsymbol{O}(DIM)$

  El coste en tiempo, en el caso mejor, es lineal en el número de datos *T.length* de la tabla a ordenar.

- **Caracterización asintótica del coste en tiempo en el caso peor**

  Para calcular la suma de costes hay que tener en cuenta que el bloque a iterar en el bucle *while* se ejecuta i veces y, en cada una de ellas, se ejecuta el bloque *else* de la instrucción condicional.

  $t(DIM) = t_a + (\Sigma\alpha \in [1, DIM-1].t_b + t_c + t_d + (\Sigma\beta \in [1, \alpha].t_e + t_g + t_d) + t_h + t_a)$

  $t(DIM) = t_a + (\Sigma\alpha \in [1, DIM-1].t_b + t_c + t_d + (t_e + t_g + t_d).\alpha + t_h + t_a)$

  $t(DIM) = t_a + (t_b + t_c + t_d + t_h + t_a)(DIM - 1) + (t_e + t_g + t_d)(\Sigma\alpha \in [1, DIM-1].\alpha)$

  $t(DIM) = t_a + (t_b + t_c + t_d + t_h + t_a)(DIM - 1) + \frac{1}{2}(t_e + t_g + t_d)DIM(DIM - 1)$

  Y la caracterización asintótica del coste en tiempo será:

  $\boldsymbol{O}(t(DIM)) = \boldsymbol{O}(t_a + (t_b + t_c + t_d + t_h + t_a)(DIM - 1) + \frac{1}{2}(t_e + t_g + t_d)DIM(DIM - 1))$

  $\boldsymbol{O}(t(DIM)) = \boldsymbol{O}(\frac{1}{2}(t_e + t_g + t_d)DIM(DIM - 1)) = \boldsymbol{O}(DIM(DIM - 1)) = \boldsymbol{O}(DIM^2)$

  El coste en tiempo, en el caso peor, es cuadrático en el número de datos *T.length* de la tabla a ordenar.

Una solución del problema 3º

Código del método *binarizar(double[], int[]nt)* diseñado sin bucles aplicando la técnica de inmersión mediante refuerzo de la precondición.

```
/**
 * Pre: T.length>0 AND T.length = B.length  AND T = To
 * Post: (PT alfa EN [0.T.length-1].(T[alfa]>=0.0 -> B[alfa]=1)
 *                                    AND (T[alfa]<0.0 -> B[alfa]=0))
 *       AND T = To  AND binarizar(T,B) = (SIGMA alfa EN [0.B.length-1].B[alfa])
 */
public static int binarizar (double[] T, int [] B) {
    return binarizar (T, B, 0, 0);
}

/**
 * Pre: T.length>0 AND T.length = B.length AND numDatos>=0
 *      AND numDatos<=T.length AND T = To
 *      AND (PT alfa EN [0,numDatos-1].(T[alfa]>=0.0 -> B[alfa]=1)
 *                                    AND (T[alfa]<0.0 -> B[alfa]=0))
 *      AND cuenta = (SIGMA alfa EN [0.numDatos-1].B[alfa])
 * Post: (PT alfa EN [0,T.length-1].(T[alfa]>=0.0 -> B[alfa]=1)
 *                                    AND (T[alfa]<0.0 -> B[alfa]=0))
 *       AND T = To
 *       AND binarizar(T,B,numDatos,cuenta) = (SIGMA alfa EN [0,B.length-1].B[alfa])
 */
public static int binarizar (double[] T, int [] B, int numDatos, int cuenta) {
    if (numDatos<T.length) {
        if (T[numDatos]>=0.0) {
            cuenta = cuenta + 1; B[numDatos] = 1;
        }
        else {
            B[numDatos] = 0;
        }
        return binarizar (T, B, numDatos + 1, cuenta );
    }
    else {
        return cuenta;
    }
}
```

Código del método *binarizar(double[], int[]nt)* diseñado sin bucles aplicando la técnica de inmersión mediante debilitamiento de la postcondición.

```
/**
 * Pre: T.length>0 AND T.length = B.length   AND T = To
 * Post: (PT alfa EN [0.T.length-1].(T[alfa]>=0.0 -> B[alfa]=1) AND (T[alfa]<0.0 -> B[alfa]=0))
 *       AND T = To AND binarizar(T,B) = (SIGMA alfa EN [0.B.length-1].B[alfa])
 */
public static int binarizar (double[] T, int [] B) {
    return binarizar (T, B, T.length );
}

/**
 * Pre: T.length>0 AND T.length = B.length AND numDatos>=0 AND numDatos<=T.length
 *      AND T = To
 * Post: (PT alfa EN [0.numDatos-1].(T[alfa]>=0.0 -> B[alfa]=1)
 *                                    AND (T[alfa]<0.0 -> B[alfa]=0))
 *       AND T = To
 *       AND binarizar(T,B,numDatos) = (SIGMA alfa EN [0,numDatos-1].B[alfa])
 */
public static int binarizar (double[] T, int [] B, int numDatos) {
    if (numDatos>0) {
        if (T[numDatos-1]>=0.0) {
            B[numDatos-1] = 1;
            return 1 + binarizar (T,B,numDatos-1);
        }
        else {
            B[numDatos-1] = 0;
            return binarizar (T,B,numDatos-1);
        }
    }
    else {
        return 0;
    }
}
```

Una solución del problema 4º

```
/* T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1])
   ∧ (∃α ∈ [0, T.length − 1].T[α] = dato) ∧ izdo = 0 ∧ dcho = T.length − 1   */
while ( izdo! = dcho ) {
    int medio = (izdo + dcho)/2;
    if ( dato <= T[medio] ) {
        dcho = medio;
    }
    else {
        izdo = medio + 1;
    }
}
/* T[izdo] = dato   */
```

1. Invariante del bucle **while** que permite demostrar la corrección del código:

   Inv: $T.length > 0 \ \wedge \ (\forall \alpha \in [0, T.length - 2].T[\alpha] \leq T[\alpha + 1]) \ \wedge \ izdo \leq dcho$
   $\wedge \ (\exists \alpha \in [izdo, dcho].T[\alpha] = dato)$

2. Pruebas que demuestran la corrección del código del bloque a iterar asociado al bucle **while**, a partir del invariante **Inv**:

```
/*  T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo < dcho
    ∧ (∃α ∈ [izdo, dcho].T[α] = dato)   */
int medio = (izdo + dcho)/2;
/*  T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo < dcho
    ∧ (∃α ∈ [izdo, dcho].T[α] = dato) ∧ medio = (izdo + dcho)/2   */
if ( dato <= T[medio] ) {
    /*  T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo < dcho
        ∧ (∃α ∈ [izdo, dcho].T[α] = dato) ∧ medio = (izdo + dcho)/2 ∧ dato ≤ T[medio]
        ⇒
        T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo ≤ dcho
        ∧ (∃α ∈ [izdo, medio].T[α] = dato)   */
    dcho = medio;
}
else {
    /*  T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo < dcho
        ∧ (∃α ∈ [izdo, dcho].T[α] = dato) ∧ medio = (izdo + dcho)/2 ∧ dato > T[medio]
        ⇒
        T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo ≤ dcho
        ∧ (∃α ∈ [medio + 1, dcho].T[α] = dato)   */
    izdo = medio+1;
}
/*  Inv: T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo ≤ dcho
        ∧ (∃α ∈ [izdo, dcho].T[α] = dato)   */
```

3. Demostración de la terminación del bucle **while**. Para ello se va a considerar la función de cota $f_{cota} = dcho - izdo$.

```
while ( izdo! = dcho ) {
    /* izdo < dcho ∧ izdo = X ∧ dcho = Y ∧ f_cota(inicio) = Y − X */
    int medio = (izdo + dcho)/2;
    /* izdo < dcho ∧ izdo = X ∧ dcho = Y ∧ medio = (X + Y)/2 */
    if ( dato <= T[medio] ) {
        dcho = medio;
        /* izdo ≤ dcho ∧ izdo = X ∧ dcho = (X + Y)/2
           ∧ f_cota(fin_1) = (X + Y)/2 − X */
    }
    else {
        izdo = medio+1;
        /* izdo ≤ dcho ∧ izdo = X ∧ izdo = (X + Y)/2 + 1
           ∧ f_cota(fin_2) = Y − (X + Y)/2 − 1 */
    }
    /* Inv: T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo ≤ dcho
            ∧ (∃α ∈ [izdo, dcho].T[α] = dato) */
}
```

La demostración de la terminación del bucle se sustenta en dos pruebas:

- El valor de la función de cota, $f_{cota} = dcho - izdo$, **disminuye en cada iteración**. En efecto, por una parte se satisface que $Y - X > (X + Y)/2 - X$ por otra parte, $Y - X > Y - (X + Y)/2 - 1$, ya que se satisface que $X < Y$.
- **La secuencia de valores decrecientes que toma la función de cota es finita**. En efecto, basta observar la condición $izdo \leq dcho$ que impone el invariante **Inv** del bucle, la cual determina que se satisfaga que $f_{cota} = dcho - izdo \geq 0$. Es decir, la función de cota está acotada inferiormente por el valor 0.

4. Pruebas finales que completan la demostración de la corrección de la instrucción iterativa **while**.

```
/* T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1])
   ∧ (∃α ∈ [0, T.length − 1].T[α] = dato) ∧ izdo = 0 ∧ dcho = T.length − 1
   ⇒
   T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo ≤ dcho
   ∧ (∃α ∈ [izdo, dcho].T[α] = dato) */
while ( izdo! = dcho ) {
    int medio = (izdo + dcho)/2; if ( dato <= T[medio] ) {
        dcho = medio;
    }
    else {
        izdo = medio+1;
    }
    /* Inv: T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo ≤ dcho
            ∧ (∃α ∈ [izdo, dcho].T[α] = dato) */
}
/* T.length > 0 ∧ (∀α ∈ [0, T.length − 2].T[α] ≤ T[α + 1]) ∧ izdo = dcho
     ∧ (∃α ∈ [izdo, dcho].T[α] = dato)
   ⇒
   T[izdo] = dato */
```