Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Cognoms, Nom | D.N.I.

Titulació: Grau en Enginyeria Informàtica
Assignatura: Programació 2 (PRO2)
Duració: 2h

Curs: Q2 2021–2022 (1r Parcial)
Data: 4 de abril de 2022

1. **Reevaluación** [5 puntos]

   Tenemos los resultados de la reevaluación de una asignatura de la FIB en una lista que contiene los DNIs de los estudiantes inscritos en la reevaluación y, para cada uno de ellos, un booleano que indica si ha aprobado o no la reevaluación de la asignatura.

   Más específicamente, hemos definido el tipo `reeval` así

```
struct reeval {
    int dni;
    bool aprueba;
};
```

   y tenemos una lista `list<reeval> l` con los resultados de la reevaluación que está ordenada crecientemente por el campo entero `dni`.

   Con estos resultados queremos actualizar las notas de los estudiantes de la asignatura, que tenemos representadas en un vector de objetos de la clase `Estudiant` ordenado también por DNI, poniendo un 5.0 a los estudiantes que han aprobado la reevaluación. Aplicaremos así la normativa de reevaluación de la FIB, en la que para poder inscribirse en la reevaluación la nota original ha de estar entre 3.5 y 4.9, los estudiantes que aprueban la reevaluación pasan a tener un 5.0, y los que no la aprueban, mantienen su nota original.

   Para ello, deseamos implementar la siguiente acción:

```
void aplica_reevaluacion(const list<reeval> &l, vector<Estudiant> &v)
// Pre: l ordenada crecientemente por el campo dni de sus elementos,
//      v ordenado crecientemente por el DNI de sus estudiantes,
//      v = V,  todos los dni's incluidos en l representan estudiantes
//              incluidos en V que tienen una nota entre 3.5 y 4.9
// Post: v es como V excepto que,
//          para cada elemento de la lista l de la forma <d,true>,
//          la nota del estudiante de v con DNI d es ahora 5.0
```

Por ejemplo, si el contenido inicial de `v` es (con DNI's no realistas)

```
[<111,9.4>, <222,4.3>, <333,3.5>, <444,7.5>, <555,NP>, <666,4.0>, <777,3.7>]
```

y el contenido de `l` es

```
[<222,true>, <333,false>, <777,true>]
```

el contenido final de `v` tras ejecutar `aplica_reevaluación(l,v)` ha de ser

```
[<111,9.4>, <222,5.0>, <333,3.5>, <444,7.5>, <555,NP>, <666,4.0>, <777,5.0>]
```

Os recordamos la especificación Pre/Post de dos métodos de la clase `Estudiant` que es necesario usar en la implementación de `aplica_reevaluacion`:

```
int consultar_DNI() const;
// Pre: cierto
// Post: el resultado es el DNI del parámetro implícito

void modificar_nota(double nota);
// Pre: el parámetro implícito tiene nota, 0 <= nota <= nota_maxima()
// Post: la nota del parámetro implícito pasa a ser nota
```

Se pide:

(a) (2.5 puntos) Implementar la acción `aplica_reevaluacion` **iterativamente**.

(b) (2.5 puntos) Escribir el invariante y la funcion de cota de todos los bucles del apartado anterior y justificar su corrección.

SOLUCIÓ:

(a) (2.5 puntos) Implementar la acción `aplica_reevaluacion` **iterativamente**.

A continuación se presentan dos soluciones correctas y eficientes, la primera con un solo bucle y la segunda con dos bucles (uno dentro del otro).

Solución 1:

```
void aplica_reevaluacion(const list<reeval> &l, vector<Estudiant> &v) {
    list<reeval>::const_iterator it = l.begin();
    int j = 0;
    // Invariante 1 (se escribe luego)
    while (it != l.end()) {
        if (v[j].consultar_DNI() == (*it).dni) {
            if ((*it).aprueba)  v[j].modificar_nota(5.0);
               // no hace falta comprobar que v[j] tenga nota por la Pre
            ++it;
            ++j;
        }
        else {
          // forzosamente ha de ser v[j].consultar_DNI() < (*it).dni,
          // no puede ser nunca v[j].consultar_DNI() > (*it).dni
            ++j;
        }
    }
}
```

Naturalmente, se puede sacar `++j` de las dos ramas del condicional y escribirlo después de éste, eliminando también la rama del `else`.

Solución 2:

```
void aplica_reevaluacion(const list<reeval> &l, vector<Estudiant> &v) {
    list<reeval>::const_iterator it = l.begin();
    int j = 0;
    // Invariante 1 (se escribe luego)
    while (it != l.end()) {
        // Invariante 2 (se escribe luego)
        while (v[j].consultar_DNI() < (*it).dni)  ++j;
        // forzosamente v[j].consultar_DNI() == (*it).dni al salir del bucle
        if ((*it).aprueba) v[j].modificar_nota(5.0);
        ++j;
        ++it;
    }
}
```

Aviso: si en el código anterior se reinicializa `j=0` antes del bucle interno (ya sea éste un `while` o un `for`) la solución resulta ineficiente, porque el coste pasa entonces de lineal a cuadrático.

(b) (2.5 puntos) Escribir el invariante y la funcion de cota de todos los bucles del apartado anterior y justificar su corrección.

- *Invariante 1*: (válido para la Solución 1 y para el bucle externo de la Solución 2)
  i. $0 \leq j \leq v.size()$,
  ii. it referencia a un elemento de l o vale $l.end()$,
  iii. $it != l.end() => j < v.size()$ and $v[j].consultar\_DNI() \leq (*it).dni$,
  iv. v es como V excepto que para cada elemento de $l[: it)$, o $l[l.begin() : it)$, de la forma <d,true>, la nota del estudiante de v con DNI d es ahora 5.0,
  v. l ordenada crecientemente por el campo dni de sus elementos,
  vi. v ordenado crecientemente por el DNI de sus estudiantes,
  vii. todos los dni's incluidos en l representan estudiantes incluidos en V que tienen una nota entre 3.5 y 4.9 en V.

  Las últimas tres condiciones se heredan de la Pre. La condición iv), que es un debilitamiento de la Post, es fundamental para poder justificar la corrección del algoritmo (que se cumpla la Post al acabar). La condición iii) es necesaria para poder justificar el cuerpo del bucle. La condición ii) no se puede expresar como $l.begin() \leq it \leq l.end()$, porque $<$ no está definido para los iteradores.
- *Invariante 2*: (para el bucle interno de la Solución 2)
  i. $0 \leq j < v.size()$,
  ii. it referencia a un elemento de l,
  iii. $v[j].consultar\_DNI() \leq (*it).dni$.
- *Función de cota*: $f = v.size() - j$ sirve para los tres bucles; $f = |l[it :)|$, o $f = |l[it : l.end())|$, sólo sirve para el bucle externo de la Solución 2 (no para el externo de la Solución 1, porque no decrece en cada iteración); es incorrecto expresar esta última función de cota como $f = l.end() - it$, porque $-$ no está definido para los iteradores (en todo caso, se puede decir "el número de elementos de l desde el referenciado por it hasta el final de la lista").
- *Terminación*: La condición i) de los invariantes, junto con $it != l.end()$ si se entra en el bucle externo, implica que $f = v.size() - j \geq 0$ siempre y $f > 0$ si se entra en cualquiera de los tres bucles. La función $f = v.size() - j$ decrece en 1 en cada iteración del bucle externo de la Solución 1 y del interno de la Solución 2 al incrementar la j; en cada iteración del bucle externo de la Solución 2, f decrece en un valor ≥ 1, porque la j crece en un valor ≥ 1.

- *Inicializaciones*:
  - Asignando $j = 0$ y $it = l.begin()$ se cumplen todas las condiciones del Invariante 1, algunas de ellas gracias a las condiciones de la Pre.
  - En el bucle interno de la Solución 2, no es necesario ninguna inicialización, porque el Invariante 1, junto con $it! = l.end()$ si se entra en el bucle externo, implica las tres condiciones del Invariante 2.
- *Condición del bucle*:
  - La negación de la condición del bucle externo es $it == l.end()$. Esta condición conjuntamente con la condición iv) del Invariante 1 implica que, al salir del bucle externo, se cumple la Post.
  - En el caso del bucle interno de la Solución 2, el Invariante 2 y la negación de la condición del bucle implica que $v[j].consultar\_DNI() == (*it).dni$ al salir del bucle interno.
- *Cuerpo del bucle*:
  - En la Solución 1, la condición iii) del Invariante 1 nos dice que hemos de tratar dos casos. En el primer caso, cuando los DNIs de $(*it)$ y $v[j]$ coinciden, hay que comprobar si se ha aprobado la reevaluación y si es así modificar la nota de $v[j]$ con un 5.0, avanzando tanto la j como it para pasar a los siguientes elementos. En el segundo caso, el DNI de $v[j]$ es inferior, hay que avanzar sólo la j para continuar buscando en v el DNI de $(*it)$. En ambos casos, se vuelven a cumplir todas las condiciones del Invariante 1.
  - En el bucle externo de la Solución 2 hay que tratar el elemento de la lista referenciado por it para luego incrementar el iterador y que se vuelva a cumplir el Invariante 1. Para tratar el elemento, primero hay que buscarlo en v mediante el bucle interno y, cuando se encuentra (Post del bucle interno), comprobar si ha aprobado y modificar la nota en v si hace falta.
  - En el bucle interno de la Solución 2, basta con avanzar la j hasta que se encuentre el DNI, cumpliéndose el Invariante 2 en cada iteración.

Cognoms, Nom D.N.I.

2. **Diferencias con el preorden de un árbol binario** [5 puntos]

Para este problema, diremos que un árbol binario a es *completo* (*full*, en inglés) si todos sus niveles están llenos de nodos. Sea `tam`(a) el número de elementos de un árbol a, donde `tam` es una función abstracta que no se puede usar dentro del código. Otras propiedades que cumple un árbol completo a son:

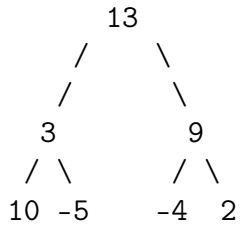
- todos los caminos desde la raíz a las hojas tienen la misma longitud
- si a es vacío entonces es completo
- si a no es vacío, entonces:
  (a) `tam`(a) = 1 + `tam`(a.left()) + `tam`(a.right())
  (b) `tam`(a.left()) = `tam`(a.right()) = (`tam`(a)-1)/2
  (c) todos los subárboles de a son completos

Se desea implementar la siguiente función usando una función de inmersión recursiva.

// Pre: a es un árbol completo, v.size() = `tam`(a)
// Post: el resultado es el número de diferencias (o errores) entre el recorrido en
// preorden de a y el recorrido de v en orden creciente de índice entre 0 y v.size()-1

```
int num_errores(const BinTree<int> &a, const vector<int> &v)
```

Esto es, queremos saber cuántas veces el valor de un nodo de a no coincide con el valor de la posición de v correspondiente a ese nodo en preorden. Por ejemplo, si a es el siguiente árbol binario completo de enteros



y el vector v es [13 4 10 -5 9 -4 1], entonces v.size() = `tam`(a) = 7 y el resultado ha de ser `num_errores`(a,v) = 2, debido a los errores en las posiciones 1 y 6 del vector (v[1]=4≠3 y v[6]=1≠2).

Es obligatorio elegir una de las dos siguientes cabeceras para la función de inmersión:

- Opción 1:

```
int i_num_errores(const BinTree<int> &a, const vector<int> &v,
                  int i, int j)
```

- Opción 2:

```
int ii_num_errores(const BinTree<int> &a, const vector<int> &v,
                   int &i)
```

Para la opción 1 es esencial tener en cuenta que el árbol dado a es completo, mientras que, para la opción 2, la solución desarrollada probablemente será válida (si es correcta) con cualquier árbol binario, no solo con aquellos que son completos.

Ninguna solución que use una inmersión distinta o altere la cabecera elegida se considerará válida. Asimismo, se valorarán muy negativamente las soluciones innecesariamente ineficientes en tiempo (si éste es mayor que proporcional al tamaño del árbol y del vector) o en espacio (por ejemplo, si se usara también algún vector auxiliar).

Se pide:

(a) (1 punto) Escribir la especificación Pre/Post de la función de inmersión elegida.

(b) (2 puntos) Implementar la función de inmersión elegida.

(c) (1.5 puntos) Justificar la corrección de la función de inmersión elegida incluyendo la demostración de que termina siempre.

(d) (0.5 puntos) Implementar la función original (no recursiva) `num_errores`.

SOLUCIÓ:

A continuación se presentan las respuestas correctas correspondientes a las dos opciones entre las que había que elegir.

Opción 1:

Dentro de la primera opción de inmersión hay también dos opciones posibles, dependiendo del significado que se le quiera dar al último parámetro entero de inmersión j, si es el límite derecho del subvector a visitar (Opción 1.1) o es el tamaño de dicho subvector (Opción 1.2); en ambos casos, el primer parámetro de inmersión i es el límite izquierdo del subvector a visitar.

Opción 1.1:

(a) (1 punto) Escribir la especificación Pre/Post de la función de inmersión elegida.

// Pre: a es un árbol completo,
// $0 \leq i \leq v.size()$,
// $-1 \leq j \leq v.size() - 1$,
// $i \leq j + 1$,
// $(j - i + 1) = \texttt{tam}(a)$
// Post: el resultado es el número de diferencias entre el recorrido en preorden
// de a y el recorrido de v en orden creciente de índice entre i y j

```
int i_num_errores(const BinTree<int> &a, const vector<int> &v,
                  int i, int j)
```

(b) (2 puntos) Implementar la función de inmersión elegida.

```
int i_num_errores(const BinTree<int> &a, const vector<int> &v,
                  int i, int j) {
    int num_err = 0;
    if (not a.empty()) {
        if (a.value() != v[i]) ++num_err;
        int pos_media = (i+j)/2;
        num_err += i_num_errores(a.left(),v,i+1,pos_media) +
                   i_num_errores(a.right(),v,pos_media+1,j);
    }
    return num_err;
}
```

(c) (1.5 puntos) Justificar la corrección de la función de inmersión elegida incluyendo la demostración de que termina siempre.

- *Terminación:* Podemos tomar como función de tamaño $f = \texttt{tam}(a)$ o también $f = j - i + 1$ (en la Pre se especifica que es el mismo valor). Claramente, si $f = 0$ estamos en el caso base ($a.empty()$). Por otro lado, en cada llamada recursiva, la función f decrece porque, si a no es vacío, $\texttt{tam}(a.left())<\texttt{tam}(a)$ y $\texttt{tam}(a.right())<\texttt{tam}(a)$.
- *Caso base:* Si $a.empty()$, el resultado num_err ha de ser 0 porque el recorrido en preorden de a es vacío y no hay diferencias con $v[i..j]$, que también será un subvector vacío.
- *Los argumentos en las llamadas recursivas cumplen la Pre:* Primero, los hijos de un árbol completo también son árboles completos. Segundo, si a no es vacío, entonces $i \leq j$ y tanto en la primera llamada recursiva ($i' = i + 1$, $j' = (i + j)/2$) como en la segunda ($i' = (i + j)/2 + 1$, $j' = j$) se cumplen las tres condiciones de rango: $0 \leq i' \leq v.size()$, $-1 \leq j' \leq v.size() - 1$ y $i' \leq j' + 1$; y también se cumple $(j' - i' + 1) = \texttt{tam}(a.left()) = \texttt{tam}(a.right()) = (j - i)/2$.

- *Caso recursivo:* Si a no es vacío, el resultado num_err ha de ser la suma del número de errores del recorrido en preorden de $a.left()$ más el número de errores del recorrido en preorden de $a.right()$ más un 1 o un 0 dependiendo de si la raíz de a, $a.value()$, difiere o no de $v[i]$, el primer elemento del subvector $v[i..j]$. Por hipótesis de inducción, las dos llamadas recursivas devuelven respectivamente los dos primeros valores de dicha suma y el tercer valor (1 o 0) se obtiene antes de las llamadas con la inicialización de num_err y la instrucción `if` interna.

(d) (0.5 puntos) Implementar la función original (no recursiva) `num_errores`.

```
int num_errores(const BinTree<int> &a, const vector<int> &v) {
    return i_num_errores(a,v,0,v.size()-1);
}
```

Opción 1.2:

(a) (1 punto) Escribir la especificación Pre/Post de la función de inmersión elegida.

```
// Pre: a es un árbol completo,
//      0 ≤ i ≤ v.size(),
//      v.size() − i ≥ tam(a),
//      j = tam(a)
// Post: el resultado es el número de diferencias entre el recorrido en preorden
//      de a y el recorrido de v en orden creciente de índice entre i y (i + j − 1)
int i_num_errores(const BinTree<int> &a, const vector<int> &v,
                  int i, int j)
```

(b) (2 puntos) Implementar la función de inmersión elegida.

```
int i_num_errores(const BinTree<int> &a, const vector<int> &v,
                  int i, int j) {
    int num_err = 0;
    if (not a.empty()) {
        if (a.value() != v[i]) ++num_err;
        ++i;
        int j_son = (j-1)/2;
        num_err += i_num_errores(a.left(),v,i,j_son) +
                   i_num_errores(a.right(),v,i+j_son,j_son);
    }
    return num_err;
}
```

(c) (1.5 puntos) Justificar la corrección de la función de inmersión elegida incluyendo la demostración de que termina siempre.

- *Terminación:* Podemos tomar como función de tamaño $f =$`tam`(a) o también $f = j$ (en la Pre se especifica que es el mismo valor). Claramente, si $f = 0$ estamos en el caso base ($a.empty()$). Por otro lado, en cada llamada recursiva, la función f decrece porque, si a no es vacío, `tam`$(a.left())<$`tam`(a) y `tam`$(a.right())<$`tam`(a).
- *Caso base:* Si $a.empty()$, el resultado num_err ha de ser 0 porque el recorrido en preorden de a es vacío y no hay diferencias con $v[i..i+j-1]$, que también será un subvector vacío.
- *Los argumentos en las llamadas recursivas cumplen la Pre:* Primero, los hijos de un árbol completo también son árboles completos. Segundo, si a no es vacío, entonces $i \leq v.size() - 1$ y tanto en la primera llamada recursiva ($i' = i + 1$) como en la segunda ($i' = i + 1 + (j-1)/2$) se cumplen las dos condiciones de rango: $0 \leq i' \leq v.size()$ y $v.size() - i' \geq$`tam`$(a.left()) =$ `tam`$(a.right())$; y también se cumple $json = (j-1)/2 =$ `tam`$(a.left()) =$ `tam`$(a.right())$.
- *Caso recursivo:* Si a no es vacío, el resultado num_err ha de ser la suma del número de errores del recorrido en preorden de $a.left()$ más el número de errores del recorrido en preorden de $a.right()$ más un 1 o un 0 dependiendo de si la raíz de a, $a.value()$, difiere o no de $v[i]$, el primer elemento del subvector $v[i..i+j-1]$. Por hipótesis de inducción, las dos llamadas recursivas devuelven respectivamente los dos primeros valores de dicha suma y el tercer valor (1 o 0) se obtiene antes de las llamadas con la inicialización de num_err y la instrucción `if` interna.

(d) (0.5 puntos) Implementar la función original (no recursiva) `num_errores`.

```
int num_errores(const BinTree<int> &a, const vector<int> &v) {
    return i_num_errores(a,v,0,v.size());
}
```

Opción 2:

(a) (1 punto) Escribir la especificación Pre/Post de la función de inmersión elegida.

// Pre: $0 \leq i \leq v.size()$,
// $v.size() - i \geq \texttt{tam}(a)$,
// $i = I$
// Post: el resultado es el número de diferencias entre el recorrido en preorden
// de a y el recorrido de v en orden creciente de índice entre I y $(I+\texttt{tam}(a)-1)$
// $i = I+ \texttt{tam}(a)$

```
int ii_num_errores(const BinTree<int> &a, const vector<int> &v,
                   int &i)
```

(b) (2 puntos) Implementar la función de inmersión elegida.

```
int ii_num_errores(const BinTree<int> &a, const vector<int> &v,
                   int &i) {
    int num_err = 0;
    if (not a.empty()) {
        if (a.value() != v[i]) ++num_err;
        ++i;
        int num_err_izq = ii_num_errores(a.left(), v, i);
        int num_err_der = ii_num_errores(a.right(), v, i);
        num_err += num_err_izq + num_err_der;
    }
    return num_err;
}
```

(c) (1.5 puntos) Justificar la corrección de la función de inmersión elegida incluyendo la demostración de que termina siempre.

- *Terminación:* Podemos tomar como función de tamaño $f = \texttt{tam}(a)$ o también $f = v.size() - i$ (la segunda es una cota superior de la primera). Claramente, si $f = 0$ estamos en el caso base ($a.empty()$). Por otro lado, en cada llamada recursiva, la función f decrece porque, si a no es vacío, $\texttt{tam}(a.left())<\texttt{tam}(a)$ y $\texttt{tam}(a.right())<\texttt{tam}(a)$; también la segunda función f decrece porque $i > I$ en las llamadas.
- *Caso base:* Si $a.empty()$, el resultado num_err ha de ser 0 porque el recorrido en preorden de a es vacío y no hay diferencias con $v[I..I + tam(a) - 1]$, que también será un subvector vacío. En este caso, se cumple $i = I+\texttt{tam}(a) = I + 0 = I$, porque no se modifica la i.
- *Los argumentos en las llamadas recursivas cumplen la Pre:* Si a no es vacío, entonces $i \leq v.size() - 1$ y tanto en la primera llamada recursiva ($i = I + 1$) como en la segunda ($i = I + 1 + tam(a.left())$) se cumplen las dos condiciones de rango: $0 \leq i \leq v.size()$ y $v.size() - i \geq \texttt{tam}(a.left())$ o $v.size() - i \geq \texttt{tam}(a.right())$, respectivamente.

- *Caso recursivo:* Si a no es vacío, el resultado num_err ha de ser la suma del número de errores del recorrido en preorden de $a.left()$ más el número de errores del recorrido en preorden de $a.right()$ más un 1 o un 0 dependiendo de si la raíz de a, $a.value()$, difiere o no de $v[I]$, el primer elemento del subvector $v[I..I + tam(a) - 1]$. Por hipótesis de inducción, las dos llamadas recursivas devuelven respectivamente los dos primeros valores de dicha suma y el tercer valor (1 o 0) se obtiene antes de las llamadas con la inicialización de num_err y la instrucción `if` interna. Por otro lado, en la Post queremos que $i = I +$ `tam`$(a) = I + 1 +$ `tam`$(a.left()) +$ `tam`$(a.right())$; para ello, primero incrementamos la i ($i = I+1$), después i aumentará en `tam`$(a.left())$ por hipótesis de inducción en la primera llamada recursiva, y finalmente i aumentará en `tam`$(a.right())$ por hipótesis de inducción en la segunda llamada recursiva.

(d) (0.5 puntos) Implementar la función original (no recursiva) `num_errores`.

```
int num_errores(const BinTree<int> &a, const vector<int> &v){
    int i = 0;
    return ii_num_errores(a,v,i);
}
```