# Soluciones del primer parcial de PRO2 (7/5/2018)

Profesores de PRO2

9 de Mayo de 2018

## 1 Problema 1

### 1.1

En primer lugar veremos una posible solución que recorre "en paralelo" las dos listas a fusionar. Cuando el elemento en curso $x$ (apuntado por $it2$) en $t2$ es menor que el elemento en curso $y$ (apuntado por $it1$) entonces se elimina $x$ de $t2$ y se inserta en $t1$. Si quedan elementos en $t2$ al acabar el primer bucle, eso quiere decir que todos ellos son mayores que cualquier elemento de $t1$, y el segundo bucle los va eliminando de $t2$ e insertando en $t1$.

```
void fusio_ord(list<int>& t1, list<int>& t2) {
  if (not t2.empty()) {
     if (t1.empty()) t1 = t2;
     else {
        It i1 = t1.begin();
        It i2 = t2.begin();
        // Invariante (ver apartado 1.2)
        while (i1 != t1.end() and i2 != t2.end()) {
           if (*i1 <= *i2) ++i1;
           else {
              t1.insert(i1, *i2);
              i2 = t2.erase(i2);
           }
        }
        while (i2 != t2.end()) {
           t1.insert(t1.end(), *i2);
           i2 = t2.erase(i2);
        }
     }
  }
}
```

En vez de eliminar elementos de $t2$ e insertarlos en $t1$ podemos utilizar `splice` bien para "transferir" un solo elemento de $t2$ a $t1$, bien para "transferir" en bloque a $t1$ todos los elementos que resten en $t2$ después del primer bucle.

```
void fusio_ord(list<int>& t1, list<int>& t2) {
  if (not t2.empty()) {
     if (t1.empty()) t1.splice(t1.end(), t2);
     else {
        It i1 = t1.begin();
        It i2 = t2.begin();
        // Invariante (ver apartado 1.2)
        while (i1 != t1.end() and i2 != t2.end()) {
           if (*i1 <= *i2) ++i1;
           else {
              ++i2;
              t1.splice(i1, t2, t2.begin());
           }
        }
        if (i2 != t2.end()) t1.splice(t1.end(), t2);
     }
  }
}
```

## 1.2

Para describir el invariante del bucle principal de nuestra solución del apartado anterior (sirve para las dos soluciones que hemos presentado) empezaremos con algunas definiciones previas:

1. Dada una lista $T$ y un iterador $it$ sobre listas diremos que $it$ *itera en* $T$ si $it$ apunta a un elemento de $T$ o $it = T.\texttt{end}()$.

2. Dada una lista $T$ y un iterador $it$ que itera sobre $T$, denotamos por $\texttt{prefix}(T, it)$ la sublista de $T$ que comienza en $T.\texttt{begin}()$ y termina en el elemento anterior al que apunta $it$, si $it \neq T.\texttt{end}()$; en otro caso, si $it = T.\texttt{end}()$, entonces $\texttt{prefix}(T, it) = T$.

3. Dada una lista $T$ y un iterador $it$ que itera sobre $T$, denotamos por $\texttt{suffix}(T, it)$ la sublista de $T$ que comienza en el elemento al que apunta $it$ (inclusive) y termina en $T.\texttt{end}()$.

El invariante del bucle principal es entonces la conjunción de las siguientes propiedades

1. $i1$ itera en $t1$ e $i2$ itera en $t2$.

2. $t1$ es la fusión ordenada de las listas $T1$ y $\texttt{prefix}(T2, i2)$.

3. $t2 = \texttt{suffix}(T2, i2)$.

4. Para todo par de elementos $x$ y $z$ tales que $x$ es un elemento de `prefix`$(t1, i1)$ y $z$ es un elemento de $t2$ se tiene que $x \leq z$.

5. Para todo par de elementos $x$ y $z$ tales que $x$ es un elemento de `prefix`$(t1, i1)$ y $z$ es un elemento de `suffix`$(T1, i1)$ se tiene que $x \leq z$.

   Esta parte del invariante se deduce de la propiedad #2 ($t1$ es la fusión ordenada …) y podría omitirse; aquí hemos optado por explicitarla al ser la homóloga de la propiedad #4 anterior,

### 1.3

Cuando el bucle termina la propiedad #2 del invariante nos dice que $t1$ es la fusión ordenada de las listas $T1$ y `prefix`$(T2, i2)$ y la propiedad #3 que $t2 =$ `suffix`$(T2, i2)$. Ambas propiedades implicarán la postcondición de `fusion_ord` si y sólo `prefix`$(T2, i2) = T2$ y $t2$ es vacía. Esto se cumple si y sólo si $i2 = t2$.`end`(). En efecto, para cualquier $T$, `prefix`$(T, T.end()) = T$ y `suffix`$(T, T.end())$ es la lista vacía.

### 1.4

Recordemos que el método de ordenación por fusión ordena recursivamente una lista $T$ como sigue: si la lista contiene 0 ó 1 elementos, no hay que hacer nada, ya que la lista está ordenada; en caso contrario, se subdivide la lista $T$ en dos sublistas $T1$ y $T2$ (de manera que $T$ sería la concatenación de $T1$ y $T2$), se ordenan recursivamente $T1$ y $T2$, y finalmente se fusionan las dos sublistas ordenadas mediante la función `fusio_ord` del apartado 1.1.

Aunque la partición de la lista $T$ en dos sublistas puede hacerse de cualquier modo que se desee con tal de que cada una de ellas tenga al menos un elemento ($T$ tiene al menos 2) y garantizar así que la recursividad termina, lo más eficiente es dividir $T$ en dos sublistas de igual o aproximadamente igual longitud. En la solución que presentamos a continuación usamos un iterador $p$ sobre $t$, lo hacemos avanzar sobre la lista $\lfloor n/2 \rfloor - 1$ veces y transferimos—en bloque—los $n - \lfloor n/2 \rfloor$ elementos finales de $t$ a otra lista $t2$ mediante `splice`. Se hacen las llamadas recursivas sobre $t$ y $t2$ y por último se hace la fusión ordenada de $t$ y $t2$, que deja el resultado final sobre la lista $t$ (y deja vacía la lista $t2$):

```cpp
typedef list<int>::iterator It;

void ordena(list<int>& t) {
  if (t.size() > 1) {
     int m = int(t.size())/2;
     It p = t.begin();
     for (int i = 0; i < m; ++i) ++p;
     list<int> t2;
     t2.splice(t2.end(), t, p, t.end());
     ordena(t); // ordena la primera mitad de T
```

```
    ordena(t2); // ordena la segunda mitad de T
    fusio_ord(t, t2);
  }
}
```

# 2 Problema 2

## 2.1

```
int nivell(const BinTree<int>& t, int k) {
    if (t.empty()) return 0;
    else if (k == 0) return 1;
    else return nivell(t.left(),k-1) + nivell(t.right(), k-1);
}
```

## 2.2

Utilizaremos la siguiente función de inmersión:

// Pre: $prof = P \wedge 0 \leq k < \mathtt{altura}(T) \wedge prof.\mathtt{size}() = k + \mathtt{altura}(T)$

```
  void profile_i(const BinTree<int>& T, int k, vector<int>& prof);
```

// Post: Para todo $i$, si $0 \leq i < k$ entonces $prof[i] = P[i]$ y si $k \leq i < prof.\mathtt{size}()$ entonces $prof[i] = P[i] + N_{i-k}(T)$

Entonces la función `profile` que se nos pide se implementa así:

```
vector<int> profile(const BinTree<int>& T, int h) {
    vector<int> prof(h, 0);
    profile_i(T, 0, prof);
    return prof;
}
```

En efecto, con $k = 0$ y el vector $prof$ de tamaño igual a la altura de $T$ e inicializado a 0, la postcondición de `profile_i` implica que $prof[i] = N_i(T)$ para toda $i$, $0 \leq i < h = \mathtt{altura}(T)$. La función de inmersión hace un recorrido en preorden del árbol; en cada llamada recursiva $T$ es el subárbol que estamos visitando y $k$ es el nivel de la raíz de $T$ dentro del árbol que nos dan en la llamada inicial a `profile_i`. Por lo tanto los nodos del nivel $i - k$ de $T$ están en el nivel $i$ del árbol inicial.

```
  void profile_i(const BinTree<int>& T, int k,
                 vector<int>& prof) {
    if (not T.empty()) {
      ++prof[k];
      profile_i(T.left(), k+1, prof);
      profile_i(T.right(), k+1, prof);
```

```
    }
  }
```

### 2.3

Si $T$ es un árbol vacío, obviamente $\texttt{nsat}(T) = 0$; ningún nivel está saturado. Por otro lado si $T$ no es vacío, supongamos que su subárbol izquierdo tiene $n_1$ niveles saturados y que el subárbol derecho tiene $n_2$ niveles saturados. Entonces $T$ tiene un nivel saturado (el nivel 0) más el mínimo entre $n_1$ y $n_2$ pues para que un cierto nivel $j$ esté saturado en $T$ es necesario que el nivel $j-1$ tanto en el subárbol izquierdo como en el derecho estén **ambos** saturados. Con esta observación, la implementación recursiva de `nsat`, satisfaciendo todos los requisitos del enunciado, resulta simple:

```
int nsat(const BinTree<int>& T) {
  if (T.empty()) return 0;
  // not T.empty()
   int n1 = nsat(T.left());
   // si n1 = 0 entonces min(n1, n2) = 0
   if (n1 == 0) return 1;
   else {
      int n2 = nsat(T.right());
      if (n1 < n2) return 1 + n1;
      else return 1 + n2;
   }
}
```

La solución que os proponemos arriba emplea una pequeña "optimización": si en $T$.`left`() no hay ningún nivel saturado entonces no hace falta hacer la llamada recursiva a `nsat` en el subárbol derecho. Una solución alternativa dada más abajo, sin la optimización anterior, y usando la función "auxiliar" `min` también se considerará válida:

```
int nsat(const BinTree<int>& T) {
  if (T.empty())
     return 0;
  else
     return 1 + min(nsat(T.left()), nsat(T.right()));
}
```