PRO2 - SOLUCIÓ FINAL JUNY 2015

PROBLEMA 1

El problema es podia resoldre amb un nombre diferent de bucles (1, 2 o tres), el punt clau al recorre la llista de nodes
es el mantenir correctament els apuntadors a l'inici (amb acces des del anterior) i el final de la subcua que es demana
i la seva longitud.

Una vegada recorreguda la llista s'han de tenir els apundadors correctes, un al primer element de la subcua des de l'anterior element i un altre a l'ultim. Per construir la subcua nomes cal donar valor a primer (amb seg del primer apuntador) i
ultim i no oblidar posar a NULL l'apuntador seg de ultim.

Fer el desencadenament correcte del p.i., es pot fer facilment ja que es te acces al primer element de la subcua
des del anterior, nomes cal fer que el seguent apunti al seguent de l'apuntador al final de la subcua. Cal anar amb compte per
fer les actualitzacions en l'ordre correcte per a no perdre la infomacio, concretament, el seguent de l'ultim de la subcua
no es pot posar a NULL fins que no hem reencadenat el p.i.

Per finalitzar, nomes queda actualizar la longitud del p.i. per deixar tot com cal.

Aquesta es una posible solucio:

```
void extreure_max_subseq(const T elem, Cua& sub) {
 /* Pre: el p.i. conte una cua C, sub es buida
    Post: sub conte la subcua mes llarga entre dues aparicions d'elem
          Al p.i. se l'ha eliminat la subcua mes llarga entre dues
aparicions d'elem
 */
  node_cua* p1 = primer_node;

  while (p1 != NULL and p1->info != elem){
     p1 = p1 -> seguent;
  }
  if (p1 != NULL) {
    isub = p1;
    p1 = p1 -> seguent;
    node_cua* isub_max = NULL;
    node_cua* fsub_max = NULL;
    node_cua* isub;
    node_cua* fsub;
    int maxlen = 0;
    int len = 0;
    /* Inv:
          Entre isub_max i fsub_max hi ha la subcua mes
          llarga entre dues aparicions d'elem que son abans de p1
          maxlen es la longitud de la subcua
          isub apunta a la darrera aparicio d'elem i
          p1 a l'element seguent a isub
    */
    while (p1 != NULL) {
      while (p1 != NULL and p1->info != elem){
        fsub = p1;
        p1 = p1 -> seguent;
        ++len;
```

```
      }
      if (p1 != NULL) {
         if (len >= maxlen) {
            isub_max = isub;
            fsub_max = fsub;
            maxlen = len;
         }
         isub = p1;
         p1 = p1->seguent;
         len = 0;
      }
    }
    if (maxlen != 0) {/* Hi ha alguna sequencia */
      sub.primer_node = isub_max->seguent; /* Inici la cua sub  */
      sub.ultim_node = fsub_max;             /* Final de la cua sub */
      isub_max -> seguent = fsub_max->seguent; /* treiem els elements
del p.i.*/
      fsub_max -> seguent = NULL; /* NULL de fi de llista a la subcua
*/
      sub.longitud = maxlen; /* Longitud de la cua sub */
      longitud -= maxlen;    /* Longitud de la nova cua */
    }
  }
}
```

PROBLEMA 2.1

Aquest problema almenys diferent solucions. Aquí en presentem dues:

Variant 1:

```
bool es_complet() const {
     int h;
     return i_es_complet(primer,h);
}

// Pre: cert
// Post: el resultat indica si la jerarquia apuntada per p representa
// un arbre complet. Si és cert, h conté l'altura d'aquesta jerarquia
static bool i_es_complet(node_arbreNari p, int& h) {
     if (p == NULL) {
         h = 0;
         return true;
     } else {
          bool ok = i_es_complet(p->seg[0],h);
          i = 1;
          while (i < seg.size() and ok) {
               int haux;
               ok = i_es_complet(p->seg[i],haux);
               ok = ok and (h == haux);
               ++i;
          }
          h = h+1;
          return ok;
     }
}
```

Aquesta solució té la pega que recorre el segon fill sencer encara
que de seguida es podria veure que l'arbre sencer no és equilibrat,

perquè al segon fill es troba un node a altura superior a la del primer fill,
o un arbre buit a una altura inferior a la del primer fill.
Això s'evitaria amb la variant següent:

Variant 2:

```
// Pre: p apunta a una jerarquia P de nodes, h= H
// Post: si init és fals, el resultat diu si la jerarquia P és una arbre equilibrat
// i en h hi ha l'altura de la jerarquia. Si init és true, el resultat
// diu si la jerarquia P és un arbre equilibrat d'altura H
static bool i_es_complet(node* p, int& h, bool &init);
```

amb crida inicial

```
bool es_complet() const {
    int h = -1;
          bool init = false;
    return i_es_complet(primer,h,init);
}
```

i implementació:

```
static bool i_es_complet(node_arbreNari* p, int& h, bool& init) {
 if (p == NULL) {
   if (init) return (h == 0);
   else {
     h = 0;
     init = true;
     return true;
    }
  } else {
    if (init) {
      if (h==0) return false;
      --h;
    }
    if (not i_es_complet(p->seg[0],h, init)) return false;
    int i = 1;
    while (i < p->seg.size()) {
       if (not i_es_complet(p->seg[i],h, init)) return false;
       ++i;
    }
    ++h;
    return true;
  }
}
```

Una altra solució completament diferent es basa en fer un recorregut per nivells,
amb una Cua<node*>. Serà iterativa i no recursiva.

Errors freqüents són:

- aplicar definicions necessàries però no suficients d'"arbre complet", com ara:
  - comprovar que tot node o té N fills no buit o té N fills buits (necessària però no suficient))
  - comprovar que tot fill és complet, però no que tots els fills tenen la mateix altura
  - comprovar que té N^h+...+N+1 nodes, on h és la distància a l'arrel del fill de més a l'esquerra

- no aturar el recorregut de l'arbre quan es troba un
  fill no complet, o d'altura incorrecta.

PROBLEMA 2.2

```
// Pre: h >= altura(p), n es l'aritat de l'arbre N-ari al que pertany
// la jerarquia apuntada per p
// Post: la jerarquia apuntada per p ha estat completada a un arbre
complet
// d'altura h afegint nodes amb valor V.
static void i_completa(node_arbreNari*& p, int n, int h, const T& V) {
     if (h == 0) {
         // si h == 0, p ha de ser null i la post ja es compleix
     } else {
         // h > 0
         if (p == NULL) {
             p = new node_arbreNari;
             p -> info = V;
             p -> seg = vector<node_arbreNari*>(n,NULL);
         }
         for (int i = 0; i < p->n; ++i) i_completa(p->seg[i],n,h-1,V);
     }
}
```

Alternativament, es podria fer una funció i_crea_complet(int h)
que retorna una jerarquia completa d'altura h i plena de V's,
i amb un bucle similar a l'anterior que la crida, però acaba
sent més complicat.

Errors freqüents són:
- no gestionar bé els casos base. per exemple,
  no fer res quan p == NULL i h > 0 (cal crear un node)
- no crear el vector p->seg després de crear un nou node,
  o no assegurar que conté valors NULL (tant si es deixa
  com a fulla com si es fa una crida recursiva amb p->seg[i]).