# **Estructures de la Informació**. Control parcial

Dept. de Llenguatges i Sistemes Informàtics
E.P.S.E.V.G., 23 de novembre de 2012, 12:00-14:00

**IMPORTANT**: Resol els problemes en fulls separats.

1. (5 punts. 60 minuts) **Estructures Lineals**

   a) Implementa el mètode constructor per còpia de la següent classe llista:

```
template <typename T>
class llista {
   ...
 private:
   // Llista doblement enllaçada, no circular i sense element fantasma.
   struct node {
     T info;
     node *seg;
     node *ant;
   };
   node* _head;
   int _size;
};
```

   b) Fes el mètode `unique` de l'anterior classe llista que modifica la llista eliminant tots els elements repetits. En cas que la llista no tingui cap element repetit la deixa igual.

   Per exemple, donada la llista d'enters {1, 3, 14, 1, 1, 3, 9, 1, 3, 7} la crida al mètode `unique` deixaria la llista com {1, 3, 14, 9, 7}.

   NOTA 1: Cal que implementis els mètodes addicionals que utilitzis explícitament.

   NOTA 2: Pots suposar que la classe T disposa de l'operador d'igualtat.

2. (2 punts. 20 minuts) **Eficiència algorísmica**

Donades les següents funcions:

```
double funcio_1 (double x, int n) {
  if (n==0) {
    return 1;
  } else {
    return x * funcio_1(x,n-1);
} }

double funcio_2 (double x, int n) {
  if (n==0) {
    return 1;
  } else if (n%2==0) {
    double y = funcio_2(x,n/2);
    return y * y;
  } else {
    double y = funcio_2(x,n/2);
    return y * y * x;
} }

double funcio_3 (double x, int n) {
  if (n==0) {
    return 1;
  } else if (n%2==0) {
    return funcio_3(x,n/2) * funcio_3(x,n/2);
  } else {
    return funcio_3(x,n/2) * funcio_3(x,n/2) * x;
} }
```

a) Analitza el cost de les funcions recursives següents. Si és necessari, pots utilitzar els teoremes adjunts.

b) Per què creus que serveix cada una d'aquestes funcions?

IMPORTANT: Raona la teva resposta.

**Decreixement aritmètic**

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n-c)+g(n) & si\ n_0\leqslant n \end{cases}$$

on $n_0$ és una constant, $c\geqslant 1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb *k* constant. Llavors,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<1 \\ \Theta\left(n^{k+1}\right) & si\ a=1 \\ \Theta\left(a^{n/c}\right) & si\ a>1 \end{cases}$$

**Decreixement geomètric**

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n/b)+g(n) & si\ n_0\leqslant n \end{cases}$$

on $n_0$ és una constant, $b>1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb *k* constant. Llavors,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<b^k \\ \Theta\left(n^k\log n\right) & si\ a=b^k \\ \Theta\left(n^{\log_b a}\right) & si\ a>b^k \end{cases}$$

3. (3 punts. 40 minuts) **Arbres**

   a) Implementa la funció:

```
{Pre: Els arbres binaris no tenen elements repetits}
template <typename T>
nat elements_iguals (const Abin<T> &a, const Abin<T> &b) throw()
```

   que, donat dos arbres binaris, retorni el nombre d'elements que tenen els dos arbres en comú; és a dir, que són presents tant en el primer com en el segon arbre.

   Disposes de les classes `Abin<T>` i `Abin<T>::iterador` amb els mètodes públics vistos a classe i llistats a continuació.

   NOTA 1: Cal que implementis les funcions addicionals que utilitzis.

   NOTA 2: Pots suposar que la classe T disposa de l'operador d'igualtat.

```
template <typename  T>
class Abin {
 public:
   Abin() throw(error);

   Abin(const Abin<T> &a) throw(error);
   Abin& operator=(const Abin<T> &a) throw(error);
   ~Abin() throw();

   Abin(Abin<T> &fesq, const T &x, Abin<T> &fdret) throw(error);

   bool es_buit() const throw();

   Iterador sobre arbres binaris.
   friend class iterador {
   public:
     friend class Abin;

     iterador() throw();

     Abin<T> arbre() const throw(error);


     Retorna l'element apuntat per l'iterador.
     const T& operator*() const throw(error);

     iterador fesq() const throw(error);
     iterador fdret() const throw(error);

     bool operator==(const iterator &it) const throw();
     bool operator!=(const iterator &it) const throw();

     static const int IteradorInvalid = 410;
   };

   iterador arrel() const throw();
   iterador final() const throw()
};
```

## Solució problema 1

a)

```
template <typename T>
llista<T>::llista(const llista<T> &l) throw(error) {
  if (l._size == 0) {
    _head = NULL;
    _size = 0;
  }
  else {
    _head = new node;
    _head->info = l._head->info;
    _head->ant = NULL;
    _head->seg = NULL;
    node *ant = _head;
    node *n = l._head->seg;
    while (n != NULL) {
      try {
        node *aux = new node;
        aux->info = n->info;
        aux->ant = ant;
        aux->seg = NULL;
        ant->seg = aux;
        ant = ant->seg;
        n = n->seg;
      }
      catch (error) {
        destrueix(_head);
        throw;
      }
    }
    _size = l._size;
  }
}

// Mètode privat de classe.
// Caldria declarar-lo com a static dins de la representació de la classe.
template <typename T>
void llista<T>::destrueix(node* n) throw() {
  if (n != NULL) {
    destrueix(n->seg);
    delete n;
  }
}
```

b)

```
template <typename T>
void llista<T>::unique() throw() {
  node *n = _head;
  while (n!= NULL) {
    r_unique(n->info, n->seg);
    n = n->seg;
  }
}

// Mètode privat de classe.
// Caldria declarar-lo com a static dins de la representació de la classe.
template <typename T>
void llista<T>::r_unique(const T &e, node* n) throw() {
  if (n != NULL) {
    if (e == n->info) {
      n->ant->seg = n->seg;
      if (n->seg != NULL) {
        n->seg->ant = n->ant;
      }
      r_unique(e, n->seg);
      delete n;
      --_size;
    }
    else {
      r_unique(e, n->seg);
    }
  }
}
```

## Solució problema 2

a)

`funcio_1` utilitzarem les equacions de recurrència pel cas de decreixement aritmètic.

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<1 \\ a\cdot T(n-c)+\Theta(n^k) & si\ 1\leqslant n \end{cases}$$

on *a*=1, *c*=1, *k*=0.

Llavors es compleix que *a=1* i, per tant, el cost temporal de `funcio_1` és
$T_{funcio\_1}(n) = \Theta(n^{k+1})=\Theta(n)$

`funcio_2` utilitzarem les equacions de recurrència pel cas de decreixement geomètric.

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<1 \\ a\cdot T(n/b)+\Theta(n^k) & si\ 1\leqslant n \end{cases}$$

on *a*=1, *b*=2, *k*=0.

Llavors es compleix que $a=b^k$ i, per tant, el cost temporal de `funcio_2` és
$T_{funcio\_2}(n) = \Theta(n^k \log n)=\Theta(\log n)$

funcio_3 utilitzarem les equacions de recurrència pel cas de decreixement geomètric.

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<1 \\ a\cdot T(n/b)+\Theta(n^k) & si\ 1\leqslant n \end{cases}$$

on *a*=2, *b*=2, *k*=0.

Llavors es compleix que $a>b^k$ i, per tant, el cost temporal de `funcio_3` és
$T_{funcio\_3}(n) = \Theta(n^{\log_b a})=\Theta(n^{\log_2 2})=\Theta(n)$

b) Les tres funcions implementen de manera diferent la potència d'un nombre ($x^n$). Com es pot comprovar la `funcio_2` és la més eficient.

## Solució problema 3

```
template <typename T>
nat elements_iguals (const Abin<T> &a, const Abin<T> &b) throw() {
  return relements_iguals(a.arrel(), a.final(), b);
}


template <typename T>
nat relements_iguals (const Abin<T>::iterador &it,
                      const Abin<T>::iterador &end,
                      const Abin<T> &b) throw() {
  nat res = 0;
  if (it != end) {
    if (conte(b.arrel(), b.final(), *it)) {
      res = 1;
    }
    res += relements_iguals(it.fesq(), end, b) +
           relements_iguals(it.fdret(), end, b);
  }
  return res;
}


template <typename T>
bool conte (const Abin<T>::iterador &it, const Abin<T>::iterador &end,
            const T &e) throw() {
  bool res = false;
  if (it != end) {
    res = (*it == e) or conte(it.fesq(), end, e) or
          conte(it.fdret(), end, e);
  }
  return res;
}
```