# Programació recursiva

R. Ferrer i Cancho

Universitat Politècnica de Catalunya

PRO2 (curs 2010-2011)
Versió 0.5

Avís: aquesta presentació no pretén ser un substitut dels apunts oficials de l'assignatura.

# On som?

- Tema 5: Programació recursiva
- 7a sessió

Avui

- Dissenyar i verificar programes recursius.

# Exemple de funció recursiva

```
int factorial(int n)
/* Pre: n >= 0 */
{
  int f, f_aux;
  if (n < 2) f = 1;
  else {
     f_aux = factorial(n - 1);
     f = n*f_aux;
  }
  return f;
}
/* Post: el resultat retornat es n!
```

Pre implicita: `n` no es massa gran com per provocar vessaments

# Esquema de programa recursiu simple

- 1 cas base
- 1 cas recursiu

```
Tipus2 f(Tipus1 x)
/* Pre: Q(x) */
{
  Tipus2 r,s;

  if (c(x)) s = d(x);
  else {
    r = f(g(x));
    s = h(x,r);
  }
  return s;
}
/* Post: R(x,s) */
```

# Aplicació de l'esquema a la funció `factorial`

```
int factorial(int n)
/* Pre: n >= 0 */
{
  int f, f_aux;
  if (n < 2) f = 1;
  else {
     f_aux = factorial(n - 1);
     f = n*f_aux;
  }
  return f;
}
/* Post: el resultat retornat es n!
```

```
c(n) = n < 2
d(n) = 1
g(n) = n-1
h(n, f_aux) = n*f_aux;
```

# Programa recursiu general

- Més d'un cas directe
  - condicions d'entrada $c_i$
  - instruccions directes $d_i$
- Més d'un cas recursiu
  - condicions d'entrada $c_j$,
  - funcions $g_j$ i $h_j$.

# Correctesa d'un algorisme recursiu (o iteratiu)

- Si inicialment les variables compleixen la Pre, al final de l'execució compleixen la Post (**per inducció**).
- Demostrar que el programa acaba (funció fita).

# Demostració de correctesa per inducció

Inducció sobre el nombre de crides recursives que resten fins arribar al cas directe.

- Cas directe (correctesa cas directe):
  $Q(x) \wedge c(x) \implies R(x, d(x))$
- Hipòtesi d'inducció: $R(g(x), r) \quad (Q(g(x)) \implies R(g(x), r))$
- Pas d'inducció (correctesa cas recursiu):
  - $\neg c(x) \wedge Q(x) \implies Q(g(x))$ (correctesa crida recursiva / poder aplicar la HI)
  - $\neg c(x) \wedge Q(x) \wedge R(g(x), r) \implies R(x, h(x, r))$ (pas d'inducció)

Potser necessari verificar que x satisfà la pre de $g(x)$.

# Acabament del programa recursiu

- Demostrar que el nombre de crides recursives és finit.
- Cercar funció fita: disminueix a cada crida recursiva (funció dels paràmetres).
- A l'esquema: $t(x)$.

Cal demostrar

- $Q(x) \implies t(x) \in \mathbb{N}$ (no pot passar per sota de zero)
- $Q(x) \wedge \neg c(x) \implies t(g(x)) < t(x)$ (decreixement a cada crida recursiva)

# Disseny inductiu

Detectar els casos senzills i resoldre'ls sense crides recursives

- En l'esquema: determinar $c(x)$ i $d(x)$.

Considerar el cas o casos recursius i resoldre'ls

- Cridar recursivament amb un paràmetre d'entrada *"menor"* que l'actual.
- Afegir instruccions ($h(x, r)$) per tal que el codi compleixi la postcondició $R(x, s)$ des de $R(g(x), r)$.

En acabar proces: demostració de correctesa.

# Demostració correctesa factorial recursiu

Exercici pissarra

```
int factorial(int n)
/* Pre: n >= 0 */
{
  int f, f_aux;
  if (n < 2) f = 1;
  else {
     f_aux = factorial(n - 1);
        /* HI: f_aux = (n - 1)! */
     f = n*f_aux;
  }
  return f;
}
/* Post: el resultat retornat es n!
```

- Correctesa cas base.
- Correctesa cas recursiu (via hipòtesi d'inducció).
- Precondicions.
- Acabament.

# Igualtat de piles

```
bool piles_iguals(stack<int> &p1, stack<int> &p2 )
/* Pre: p1 = P1 i p2 = P2 */
{
  bool ret;
  if (p1.empty() or p2.empty()) ret = p1.empty() and p2.empty();
  else if (p1.top() != p2.top()) ret = false;
  else {
    p1.pop();
    p2.pop();
    ret = piles_iguals(p1, p2);

    /* HI: ret = "P1 sense l'últim element afegit i P2 sense l'últim
    element afegit son dues piles iguals" */
  }
  return ret;
}
/* Post: El resultat ens indica si les dues piles inicials P1 i P2
són iguals */
```

# Justificació

Pissarra

- Cas on cal verificar que $x$ satisfà la prec de $g(x)$ (`pop()`)
- Cal demostrar que Pre de `top()` se satisfà a la 2a branca.
- Condició del cas recursiu:

```
not p1.empty() and not p2.empty() and
p1.top() == p2.top()
```

- Possibles fites

# Mida d'un arbre

```
int mida(const BinTree<int> &a)
/* Pre: cert */
{
  int x;
  if (a.es_buit()) x = 0;
  else {
    int y = mida(a.left());
    /* HI1: y="nombre de nodes del subarbre esquerre d'a" */

    int z = mida(a.right());
    /* HI2: z="nombre de nodes del subarbre dret d'a" */

    x = y + z + 1;
  }
  return x;
}
/* Post: El resultat és el nombre de nodes de l'arbre inicial a */
```

# Justificació

Pissarra

# Cerca d'un estudiant en una cua de estudiants

```
bool cerca_rec_cua_Estudiant(queue<Estudiant> &c, int i)
/* Pre: i es un dni valid i c=C */
{
  bool ret;
  if (c.empty()) ret = false;
  else if (c.front().consultar_DNI() == i) ret = true;
  else {
    c.pop();
    ret = cerca_rec_cua_Estudiant(c, i);
        /* HI: ret ens diu si hi ha algun estudiant amb dni i a C
sense el primer element */
  }
  return ret;
}
/* Post: El resultat ens diu si hi ha algun estudiant amb dni i a C */
```

# Justificació

Pissarra

- Condició del cas recursiu:

```
not c.empty() and c.front().consultar_DNI() != i
```

- Comprovació addicional: la segona branca respecta la Pre de c.front().

# Immersió o generalització d'una funció

Definició: generalització d'una funció recursiva on introduïm

- Paràmetres addicionals
- Resultats addicionals
- Tots dos

Motivació:

- Necessitat de generalitzar una funció
- Poder fer un disseny recursiu
- Obtenir solucions recursives alternatives més senzilles o eficients.

# Funció d'immersió: funció auxiliar

La funció original crida la funció d'immersió.

- Fixant els paràmetres addicionals.
- Rebutjant resultats retornats.

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size() > 0 */
{
return i_suma_vect_int(v, v.size()-1);
}
/* Post: el valor retornat es la suma de tots els elements
   del vector */
```

# Canvis en l'especificació

Canvis en els paràmetres impliquen canvis en l'especificació:

- Afebliment de la post.
- Enfortiment de la pre.

Tipus:

- Immersions d'especificacions per *afebliment* de la *post*.
- Immersions d'especificacions per *enfortiment* de la *pre*.

# Exemples d'immmersió per afebliment

- Objectiu: poder fer un algoritme recursiu (immersió d'especificació).
- La funció recursiva té paràmetres addicionals
- La post es més feble que la de la funció original.

# Suma dels elements d'un vector

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size() > 0 */
/* Post: el valor retornat es la suma de tots els elements
del vector */
```

equivalent a

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size() > 0 */
/* Post: el valor retornat es la suma dels elements del vector
fins a la posicio v.size() - 1 */
```

# Afebliment

Fitar la part del vector que hem de sumar amb un paràmetre addicional.

```
int i_suma_vect_int(const vector<int> &v, int i)
/* Pre: v.size()> 0; 0 <= i < v.size() */
/* Post: el valor retornat es la suma de tots els elements
del vector v fins a la posicio i */
```

Compte: a `i_suma_vect_int`, `i_` vol dir "immersió" (no indica el nom del parametre addicional).

# Implementació de la funció d'immersió

```
int i_suma_vect_int(const vector<int> &v, int i)
/* Pre: v.size() > 0; 0 <= i < v.size() */
{
  int suma;
  if (i == 0) suma = v[0];
  else suma = i_suma_vect_int(v, i-1) + v[i];
              /* HI: el valor retornat per i_suma_vect_int(v,i-1) es la
              suma de tots els elements de v fins a la posicio i-1 */

  return suma;
}
/* Post: el valor retornat es la suma de tots els elements
del vector v fins a la posicio i */
```

# Funció original

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size() > 0 */
{
return i_suma_vect_int(v, v.size()-1);
}
/* Post: el valor retornat es la suma de tots els elements
del vector */
```

# Cerca dicotòmica

```
int cerca_vect_int(const vector<int> &v, int n)
/* Pre: v.size() > 0; v esta ordenat de creixentment */
/* Post: El valor retornat es la posicio on es troba l'element n dins
el vector v. Si n no es troba a v, llavors el valor retornat es un
nombre negatiu. */
```

equivalent

```
int cerca_vect_int(const vector<int> &v, int n)
/* Pre: v.size() > 0; v esta ordenat creixentment */
/* Post: El valor retornat es la posicio on es troba l'element n
entre la posició 0 i la v.size() - 1 del vector v.
Si n no s'hi troba, llavors el valor retornat es un
nombre negatiu. */
```

# Afebliment de la post

Immersió: introduir dos paràmetres addicionals amb la finalitat d'indicar l'inici i el final de la zona on buscarem l'element.

```
int i_cerca_vect_int(const vector<int> &v, int n, int esq, int dre)
/* Pre: v.size() > 0; v esta ordenat creixentment;
   0 <= esq <=v.size(); -1 <= dre < v.size(); esq <= dre + 1 */

/* Post: el valor retornat es la posicio de v entre els valors esq i
dre, on es troba el valor n. Si n no es troba a v[esq...dre], es
retorna un nombre negatiu */
```

# Implementació

```
int i_cerca_vect_int(const vector<int> &v, int n, int esq, int dre)
/* Pre: v.size() > 0; v esta ordenat creixentment;
   0 <= esq <=v.size(); -1 <= dre < v.size(); esq <= dre + 1 */
{
  int posicio, mig;

  if (dre < esq) posicio = -1;
  else {
    mig = (esq + dre)/2;
    if (v[mig] == n) posicio = mig;
    else {
      if (v[mig]< n) posicio = i_cerca_vect_int(v, n, mig + 1, dre);
                  /* HI1: el valor retornat es la posicio de
                     v[mig+1...dre] on es troba n. Si n no es troba
                     a v[mig+1...dre] el valor retornat es -1. */
      else posicio = i_cerca_vect_int(v, n, esq, mig - 1);
        /* HI2: el valor retornat es la posicio de v[esq...mig-1]
           on es troba n. Si n no es troba a v[esq...mig-1] el valor
           retornat es -1. */
    }
  }
  return posicio;
}
/* Post: el valor retornat es la posicio de v entre els valors esq i
dre, on es troba el valor n. Si n no es troba a v[esq...dre], es
retorna un nombre negatiu */
```

# Justificació

- Cas directe
- Cas recursiu
- Precondicions: `v[mig]` respecta la Prec de tot acces a vector, pre de les crides recursives
- Fita: dre - esq + 1

Pissarra

# Crida a la funció d'immmersió

```
int cerca_vect_int(const vector<int> &v, int n)
/* Pre: v.size() > 0; v esta ordenat creixentment */
{
return i_cerca_vect_int(v, n, 0, v.size() - 1);
}
/* Post: El valor retornat es la posicio on es troba el element n dins
el vector v. Si n no es troba a v, llavors el valor retornat es un
nombre negatiu. */
```

# Nombre de zeros en una matriu

```
int comptar_zeros_matriu_int(const vector<vector<int> > &m)
/* Pre: la matriu te almenys un rengle y almenys una columna */
/* Post: El valor retornat es el nombre de zeros en la
   matriu m */
```

A la prec cal afegir que la matriu es realment rectangular (cada rengle té el mateix nombre de columnes)

# Afebliment de la post

```
int comptar_zeros_matriu_int(const vector<vector<int> > &m)
/* Pre: la matriu te almenys un rengle y almenys una columna */
/* Post: El valor retornat es el nombre de zeros en la matriu m
desde la posicio m[0][0] fins la posicio
m[m.size()-1][m[0].size()-1] */
```

Crides recursives hauran de comptar els zeros d'una submatriu.
Immersió: afegir paràmetres extra que indiquin quina és la submatriu a considerar.

```
int i_comptar_zeros_matriu_int(const vector<vector<int> > &m,
    int i, int j)
/* Pre: la matriu te almenys un rengle y almenys una columna;
        0 <= i < m.size(); 0 <= j < m[0].size() */
/* Post: El valor retornat es el nombre de zeros en la matriu m
desde la posicio m[i][j] fins la posicio
m[m.size() - 1][m[0].size() - 1] */
```

# Implementació

```
int i_comptar_zeros_matriu_int(const vector<vector<int> > &m, int i, int j)
/* Pre: la matriu te almenys un rengle y almenys una columna;
        0 <= i < m.size(); 0 <= j< m[0].size() */
{
  int suma = 0;
  if (i == m.size()-1 and j == m[0].size()-1) { if (m[i][j] == 0) ++suma; }
  else {
    if (m[i][j] == 0) ++suma;
    if (j == m[0].size() - 1) {
       suma = suma + i_comptar_zeros_matriu_int(m, i + 1, 0);
                     /* HI:el valor retornat es el nombre de zeros de m
                     des de la posició m[i+1][0] fins la darrera posició */
    }
    else {
       suma = suma + i_comptar_zeros_matriu_int(m, i, j + 1);
                   /* HI:el valor retornat es el nombre de zeros de m
                   des de la posició m[i][j+1] fins la darrera posició */
    }
  }
  return suma;
}
/* Post: El valor retornat es el nombre de zeros en la matriu m
desde la posicio m[i][j] fins la posicio m[m.size()-1][m[0].size()-1] */
```

# Crida a la funció d'immersió

```
int comptar_zeros_matriu_int(const vector<vector<int> > &m)
/* Pre: la matriu te almenys un rengle y almenys una columna */
{
  return i_comptar_zeros_matriu_int(m, 0, 0);
}
/* Post: El valor retornat es el nombre de zeros en la matriu m. */
```