Disseny recursiu
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Resum

1. Recursió és inducció. Comenceu amb una definició recursiva
2. Si no podeu fer recursió, proveu d'afegir més paràmetres (funció d'immersió)
3. Si es repeteixen càlculs, afegiu paràmetres per recordar-los (immersió d'eficiència)

Contingut

1. Recursió, definicions recursives i inducció
2. Principis de disseny recursiu
3. Immersió o generalització d'una funció. Immersions per afebliment de la Post
4. Immersions per enfortiment de la Pre ($*$)
5. Recursivitat lineal final i algorismes iteratius ($**$)

Recursió, definicions recursives i inducció

Alçària d'una pila

Vam veure versions iterativa i recursiva:

```
int alcaria_iter(stack<int>& p) {
    int n = 0;
    while (not p.empty()) {
       ++n;
       p.pop();
    }
    return n;
}

int alcaria_rec(stack<int>& p) {
    if (p.empty()) return 0;
    else {
        p.pop();
        return 1 + alcaria_rec(p);
    }
}
```

D'on hem tret la versió recursiva??

L'alçaria de la pila $P = [e_1, e_2, \ldots, e_n]$ és n

D'on hem tret la versió recursiva??

L'alçaria de la pila $P = [e_1, e_2, \ldots, e_n]$ és n

Lema: això és EQUIVALENT A

- Si P és buida, alçària(P) = 0
- altrament, alçària(P) = 1 + alçària(desapilar(P))

D'on hem tret la versió recursiva??

En la definició recursiva, desapilar(P) és la funció abstracta que ens retorna la pila resultant de desapilar el cim de la pila P:

```
// Pre: p = P no és una pila buida
p.pop();
// Post: p = desapilar(P)
```

El mètode `pop` modifica la pila sobre la qual s'aplica i retorna `void`.

Exemple: factorial

$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
  int f = 1;
  // n = N ≥ 0
   while (n > 0) {
     // Inv: f = N!/n! ∧ n > 0
     // Fita: n
     f = f * n;
     --n;
   }
   return f;
}
```

Exemple: factorial

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

Exemple: factorial

$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$

Lema:

- $0! = 1$
- per a tot $n > 0$, $n! = n \cdot (n-1)!$

Exemple: factorial

$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$

Lema:

- $0! = 1$
- per a tot $n > 0$, $n! = n \cdot (n-1)!$

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Definicions recursives

Definició amb “. . . ” suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements “més petits”

Definicions recursives

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Exemple: descomposem una pila no buida $P = [a_1, \ldots, a_n]$ en cim$(P) = a_n$ (que podem obtenir amb `P.top()`) i desapilar$(P) = [a_1, \ldots, a_{n-1}]$ que obtindrem mitjançant `P.pop()`.

Definicions recursives

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Exemple: descomposem una pila no buida $P = [a_1, \ldots, a_n]$ en cim$(P) = a_n$ (que podem obtenir amb `P.top()`) i desapilar$(P) = [a_1, \ldots, a_{n-1}]$ que obtindrem mitjançant `P.pop()`.
Exemple: per el càlcul de x^y es pot fer la descomposició $x^y = x \cdot x^{y-1}$ o $x^y = (x^2)^{y/2}$ si y és par.

Definicions recursives

Definició amb "..." suggereix solucions iteratives

Per aplicar recursivitat, necessitem una definició recursiva

Com trobar-la: considereu operacions que descomposen una dada en elements "més petits"

Exemple: descomposem una pila no buida $P = [a_1, \ldots, a_n]$ en $\text{cim}(P) = a_n$ (que podem obtenir amb `P.top()`) i $\text{desapilar}(P) = [a_1, \ldots, a_{n-1}]$ que obtindrem mitjançant `P.pop()`.
Exemple: per el càlcul de x^y es pot fer la descomposició $x^y = x \cdot x^{y-1}$ o $x^y = (x^2)^{y/2}$ si y és par.

Sovint fem la transformació inconscientment. Si cal formalitzar-la, necessitem inducció

Suma dels elements d'una pila

Si $P = [a_1, \ldots, a_n]$, llavors $\text{suma}(P) = a_1 + \ldots + a_n$

Inductivament:

$$\text{suma}(P) = \begin{cases} 0 & \text{si } P \text{ és buida,} \\ \text{cim}(P) + \text{suma}(\text{desapilar}(P)) & \text{altrament.} \end{cases}$$

Cerca en una pila

Donada una pila p i un element x, dir si x apareix en p

Versió recursiva:

- si p és buida, x no apareix a p
- altrament, si P no és buida ...

$$x \in p \Leftrightarrow x = \text{cim}(P) \vee x \in \text{desapilar}(p)$$

Cerca en una pila

```
template <class T>
bool cerca(stack<T>& p, const T& x) {
  if (p.empty())
    return false;
    else if (x == p.top() return true;
    else {
      p.pop();
      return cerca(p, x);
    }
}
```

Igualtat de piles

Donades dues piles $p = [p_1, \ldots, p_n]$ i $q = [q_1, \ldots, q_m]$, dir si són iguals

$\Longrightarrow$ $m = n$ i per a cada posició i, $p_i = q_i$.

Igualtat de piles

Donades dues piles $p = [p_1, \ldots, p_n]$ i $q = [q_1, \ldots, q_m]$, dir si són iguals

$\Longrightarrow$ $m = n$ i per a cada posició i, $p_i = q_i$.

Versió recursiva:

- si p i q són buides, són iguals
- si p és buida i q no, o a l'inrevés, llavors són diferents
- si p i q no són buides, ...

Igualtat de piles

Donades dues piles $p = [p_1, \ldots, p_n]$ i $q = [q_1, \ldots, q_m]$, dir si són iguals

$\Longrightarrow$ $m = n$ i per a cada posició i, $p_i = q_i$.

Versió recursiva:

- si p i q són buides, són iguals
- si p és buida i q no, o a l'inrevés, llavors són diferents
- si p i q no són buides, ...
  si $\text{cim}(p) \neq \text{cim}(q)$ (`p.top() != q.top()`, les piles són diferents;
  si $\text{cim}(p) = \text{cim}(q)$ llavors $p = q$ si i només si $\text{desapilar}(p) = \text{desapilar}(q)$

Igualtat de piles

```
/* Pre: p = P, q = Q */
/* Post: Retorna cert si i només si P = Q */
bool piles_iguals(stack<int>& p, stack<int>& q) {
    if (p.empty() and q.empty()) return true;
    else if (p.empty() or q.empty()) return false;
    else if (p.top() != q.top()) return false;
    else {
        p.pop(); q.pop();
        return piles_iguals(p,q);
    }
}
```

Igualtat de piles (versió alternativa)

```
/* Pre: p = P, q = Q */
/* Post: Retorna cert si i només si P = Q */
bool piles_iguals(stack<int>& p, stack<int>& q) {
    if (p.empty() or q.empty())
        return p.empty() and q.empty();
    if (p.top() != q.top())
        return false;
    p.pop(); q.pop();
    return piles_iguals(p,q);
}
```

Igualtat de piles

- La versió iterativa queda com a exercici.
- Ull! cerca, NO recorregut! Apliqueu l'esquema de cerca seqüèncial; $P \neq Q$ si trobem $p_i \neq q_i$ o no buidem simultàniament de les dues piles (una és buida i l'altra no)

Igualtat de piles

- La versió iterativa queda com a exercici.
- Ull! cerca, NO recorregut! Apliqueu l'esquema de cerca seqüèncial; $P \neq Q$ si trobem $p_i \neq q_i$ o no buidem simultàniament de les dues piles (una és buida i l'altra no)
- Observació: és temptador comprovar abans que res si `alcaria(p) != alcaria(q)`
- Seria eficient (i convenient), però només si tenim una operació `p.size()` que no recorre la pila! Pero seria ineficient si hem de calcular el nombre d'elements de la pila recorrent (i destruint) la pila

Principis de disseny recursiu

Principis de disseny recursiu

Volem implementar recursivament una funció

```
// Pre: propietat satisfeta per x
// Post: la funció retorna un valor F(x)
tipus_sortida F(tipus_entrada x);
```

o un procediment

```
// Pre: propeitat satisfeta per x = X
// Post: res compleix una certa propietat en termes d'X
void F(T1 x, T2& res);
```

Principis de disseny recursiu

N.B. `x` i `res` poden ser més d'un paràmetre; en el cas dels procediments podem tenir paràmetres de entrada/sortida:

```
// Pre: propeitat satisfeta per x = X
// Post: x = X' compleix una certa propietat en termes
//       del seu valor original X
void F(T1& x)
```

Principis de disseny recursiu

Cal identificar:

- Un o més casos base: Valors de paràmetres en què podem satisfer la Post amb càlculs directes
- Un o més casos recursius: Valors de paràmetres en què podem satisfer la Post si tinguessim el resultat per a alguns paràmetres x' "més petits" que x

Estratègia

1. Triar una funció de "mida" $|x|$ dels paràmetres x tal que
   - $|x| \leq 0 \implies$ som en un cas base
   - les crides recursives es fan amb paràmetres x' amb $|x'| < |x|$
   - ha de ser sempre un enter: per tot x, $|x| \in \mathbb{Z}$

   Fonament: tota seqüència decreixent d'enters no negatius és finita
2. Transformar la definició rebuda del que volem calcular en una definició recursiva (si no ho és d'entrada)

Correctesa d'un algorisme recursiu

A demostrar: Amb tot valor x dels paràmetres que satisfaci Pre,

- l'algorisme acaba - nombre finit de crides recursives
- i acaba satisfent fent $Post(x)$

Acabament: nombre finit de crides recursives

Fonament:

Tota seqüència decreixent de nombres enters no negatius és finita

Acabament: nombre finit de crides recursives

Fonament:

Tota seqüència decreixent de nombres enters no negatius és finita

Formalització:

- Triem una funció de mida $|\cdot|$ dels paràmetres que sempre té valor enter
- Demostrem: Si $|x| \leq 0$ l'algorisme tracta x amb un cas base $\Longrightarrow$ cap crida recursiva
- Demostrem: Cada crida recursiva fa decrèixer la mida dels paràmetres, i.e., si la funció $\mathtt{F}$ amb paràmetre x fa la crida recursiva $\mathtt{F}(x')$ llavors $|x'| < |x|$

N.B. Noteu la similitud entre les propietats de la funció de mida i les de la funció de cota d'una iteració

Correctesa d'un algorisme recursiu

- A demostrar: Si el paràmetre x satisfà la precondició llavors el resultat satisfà la postcondició (una funció d'x)
- Quan x és un cas base ($|x| \leq 0$, no hi ha recursió): es demostra directament aplicant les tècniques de les lliçons anteriors

Correctesa d'un algorisme recursiu

- Si x no és una cas base ($|x| > 0$), apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició ($\texttt{Pre}(x')$ és cert) i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició ($\texttt{Post}(x')$"

Correctesa d'un algorisme recursiu

- Si x no és una cas base ($|x| > 0$), apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició ($\texttt{Pre}(x')$ és cert) i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició ($\texttt{Post}(x')$"

- Hem de demostrar que qualsevol crida recursiva $\texttt{F}(x')$ quan x no és un cas base ($|x| > 0$) compleix: 1) $\texttt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors la H.I.

Correctesa d'un algorisme recursiu

- Si x no és una cas base ($|x| > 0$), apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició ($\texttt{Pre}(x')$ és cert) i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició ($\texttt{Post}(x')$"
- Hem de demostrar que qualsevol crida recursiva $\texttt{F}(x')$ quan x no és un cas base ($|x| > 0$) compleix: 1) $\texttt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors la H.I.
- Aplicant l'H.I. deduim que després d'una crida recursiva $\texttt{Post}(x')$; cal demostrar que l'estat al qual s'arriva just després o fent alguns càlculs addicionals satisfà $\texttt{Post}(x)$

Exemples

- Exponenciació ràpida
- Factorial
- Nombres binomials
- Ordenació per fusió (*Mergesort*) en un vector
- Revessar una llista
- Cercar un element en una cua
- Sumar k als elements d'un arbre

Factorial

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Recordem: $n! = 1$ si $n = 0$; $n! = n \cdot (n-1)!$ si $n > 0$

- Acabament: mida = $|n| = n$. Sempre enter, $|n| = 0$ és cas base, amb $|n| = n > 0$ es fa la crida `fact(n-1)`, $|n-1| = n - 1 < |n| = n$

Factorial

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Recordem: $n! = 1$ si $n = 0$; $n! = n \cdot (n-1)!$ si $n > 0$

- Acabament: mida = $|n| = n$. Sempre enter, $|n| = 0$ és cas base, amb $|n| = n > 0$ es fa la crida `fact(n-1)`, $|n-1| = n - 1 < |n| = n$
- Correcció: si $n = 0$ retornem 1 ($0! = 1$); si $n > 0$ es fa crida amb `fact(n-1)`, llavors $n - 1 \geq 0$, $|n-1| < |n|$ i podem aplicar H.I.

Factorial

```
/* Pre: n ≥ 0 */
/* Post: retorna n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Recordem: $n! = 1$ si $n = 0$; $n! = n \cdot (n-1)!$ si $n > 0$

- Acabament: mida = $|n| = n$. Sempre enter, $|n| = 0$ és cas base, amb $|n| = n > 0$ es fa la crida `fact(n-1)`, $|n-1| = n - 1 < |n| = n$
- Correcció: si $n = 0$ retornem 1 ($0! = 1$); si $n > 0$ es fa crida amb `fact(n-1)`, llavors $n - 1 \geq 0$, $|n-1| < |n|$ i podem aplicar H.I.
- Correcció: H.I. $\Longrightarrow$ `fact(n-1)` $= (n-1)!$, per tant `fact(n)` retorna $n \cdot (n-1)! = n!$

Potència ràpida

```
// Pre: x > 0 ∧ y ≥ 0
// Post: reorna x^y
int potencia(int x, int y);
```

Observem que

$$x^y = \begin{cases} 1 & \text{si } y = 0 \\ x \cdot (x^2)^\lambda & \text{si } y = 2\lambda + 1 > 0 \text{ és senar} \\ (x^2)^\lambda & \text{si } y = 2\lambda \geq 0 \text{ és parell} \end{cases}$$

Potència ràpida

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x*x,y/2);
    else return potencia(x*x,y/2);
}
```

- Acabament: podem agafar $1y| = y$, però també $|y| = \lceil 1 + \log_2(y) \rceil$, ja que $\lceil 1 + \log_2(y/2) \rceil = \lceil \log_2(y) \rceil < \lceil 1 + \log_2(y) \rceil$. Sempre enter (per això fem servir $\lceil \cdot \rceil$). Si $|y| \leq 0$ estem en un cas base ($\log_2 y \leq -1 \implies y \leq 1/2 \implies y \leq 0$). Si $|y| > 0$ llavors no estem en un cas base, $y \geq 1$ i $|y/2| < |y|$.

Potència ràpida

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x*x,y/2);
    else return potencia(x*x,y/2);
}
```

- Correcció: si $y = 0$ llavors retornem $x^0 = 1$. Si $y > 0$, es fa la crida recursiva `potencia(x*x,y/2)`. Com $x > 0$, tenim $x^2 > 0$. I com $y > 0$, llavors $y/2 \geq 0$. A més $|y/2| < |y|$. Es pot aplicar H.I.
- Correcció: si $y = 2\lambda$ és parell, per H.I. `potencia(x*x,y/2)` retorna $(x^2)^\lambda = x^{2\lambda} = x^y$ i la funció retorna el resultat correcte. Si $y = 2\lambda + 1$ és senar, per H.I. `potencia(x*x,y/2)` retorna $(x^2)^\lambda = x^{2\lambda} = x^{y-1}$; llavors la funció retorna $x \cdot x^{y-1} = x^y$, el resultat correcte.

Potència ràpida

Exercici: Demostreu la correcció de la següent implementació alternativa:

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else {
      int p = potencia(x, y/2);
      if (y%2 == 0) return p * p;
      else return x * p * p;
    }
}
```

Nombres binomials

```
// Pre: n ≥ m ≥ 0
// Post: retorna (n m)
int binomial(int n, int m);
```

Recordem:

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$$

és el nombre de subconjunts de $\{1, \ldots, n\}$ de mida m

Nombres binomials: Disseny

Hi ha diverses definicions recursives equivalents, que porten a solucions d'eficiència i elegància diferents
Triant $|(n, m)| = n$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } n = m \\ \frac{n \cdot (n-1)!}{m! \cdot (n-m)(n-1-m)!} = \frac{n}{n-m} \cdot \binom{n-1}{m} & \text{si } n > m \end{cases}$$

Nombres binomials

Triant $|(n, m)| = m$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n!\cdot(n-m+1)}{m\cdot(m-1)!\cdot(n-m+1)\cdot(n-m)!} = \frac{n-m+1}{m}\binom{n}{m-1} & \text{si } m > 0 \end{cases}$$

Nombres binomials

Triant $|(n,m)| = m$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n!\cdot(n-m+1)}{m\cdot(m-1)!\cdot(n-m+1)\cdot(n-m)!} = \frac{n-m+1}{m}\binom{n}{m-1} & \text{si } m > 0 \end{cases}$$

O bé descomposem així:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n\cdot(n-1)!}{m\cdot(m-1)!\cdot((n-1)-(m-1))!} = \frac{n}{m}\binom{n-1}{m-1} & \text{si } m > 0 \end{cases}$$

Nombres binomials

Triant $|(n, m)| = m$:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n!\cdot(n-m+1)}{m\cdot(m-1)!\cdot(n-m+1)\cdot(n-m)!} = \frac{n-m+1}{m}\binom{n}{m-1} & \text{si } m > 0 \end{cases}$$

O bé descomposem així:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \\ \frac{n\cdot(n-1)!}{m\cdot(m-1)!\cdot((n-1)-(m-1))!} = \frac{n}{m}\binom{n-1}{m-1} & \text{si } m > 0 \end{cases}$$

(O bé fem servir el triangle de Tartaglia:

$$\binom{n}{m} = \binom{n}{m-1} + \binom{n-1}{m-1}$$

però la solució és més ineficient)

Nombres binomials

```
// Pre: n ≥ m ≥ 0
// Post: retorna (n m)
int binomial (int n, int m) {
  if (m == 0) return 1;
  else return (binomial(n-1,m-1)/m) * n;
}
```

En aquesta implementació prenem cura de fer primer la divisió ($\binom{n-1}{m-1}$ és divisible entre m) i després fer el producte. Això pot evitar problemes de *overflow*.

Exercici: escriviu funcions recursives i raoneu la seva correctesa basant-se en les altres definicions recursives examinades.

Mergesort

```
// Pre: 0 ≤ e ≤ d < v.size() ∧ v = V
// Post: v[0..e − 1] = V[0..e − 1] ∧ v[d + 1..n − 1] = V[d + 1..n − 1]∧
//       v[e..d] ordenat creixentment i és una permutació de V[e..d]
template <class T>
void mergesort(vector<T>& v, int e, int d) {
    if (e < d) {
        int m = (e + d)/2;
        mergesort(v, e, m);
        mergesort(v, m + 1, d);
        fusiona(v, e, m, d);
    }
}

// Pre: 0 ≤ e ≤ m < d < v.size() ∧ v = V∧
///     v[e..m] i v[m + 1..d] estàn ordenats creixentment
// Post: v[0..e − 1] = V[0..e − 1] ∧ v[d + 1..n − 1] = V[d + 1..n − 1]∧
//       v[e..d] ordenat creixentment i és una permutació de V[e..d]
template <class T>
void fusiona(vector<T>& v, int e, int m, int d);
```


Mida = nor

Revessar una llista

```
// Pre: l = [a_1,...,a_n] ∧ n ≥ 0
// Post:l = [a_n,a_{n-1},...,a_1]
template <class T>
void revessar(list<T>& l);
```

- Per a qualsevol estructura seqüencial $l = [a_1, \ldots, a_n]$ no buida definim head$(l) = [a_1]$, tail$(l) = [a_2, \ldots, a_n]$; donades dues seqüències l_1 i l_2 denotem $l_1 \cdot l_2$ la seqüència resultant de concatenar-les (head, tail i $\cdot$ són operacions abstractes).
- Per a tota llista $l \neq []$, $l =$ head$(l) \cdot$ tail(l).

Llavors podrem donar una definició de la funció abstracta revessar:

- revessar$([]) = []$
- Si $l \neq []$ llavors revessar$(l) =$ revessar(tail) $\cdot$ head(l)

Revessar una llista

```
// Pre: l = [a_1,...,a_n] ∧ n ≥ 0
// Post:l = [a_n,a_{n-1},...,a_1]
template <class T>
void revessar(list<T>& l) {
    if (not l.empty()) {
        T x = *(l.begin());
        l.erase(l.begin()); // = l.pop_front();
        revessar(l);
        l.insert(l.begin(),x); // = l.push_back(x);
    }
}
```

Mida: l.size()

Cerca d'un element en una cua

```
// Pre:  c = C
// Post: retorna cert si i només si x ∈ C
template <class T>
bool cerca(queue<T>& c, const T& x);
```

Definició no recursiva: cerca($e_1 \ldots e_n$,x) = ($\exists i : e_i = x$)

Definició recursiva, amb mida c.`size()`:

- cerca($[], x$) = **false**
- Si $c = [a_1, \ldots, a_n] \neq []$ llavors

$$\text{cerca}(c, x) = (x = \text{front}(c)) \vee \text{cerca}(\text{tail}(c), x)$$

Cerca d'un element en una cua

```
// Pre:  c = C
// Post: retorna cert si i només si x ∈ C
template <class T>
bool cerca(queue<T>& c, const T& x) {
    if (c.empty()) return false;
    else if (c.front() == x) return true;
    else {
        c.pop();
        return cerca(c, x);
    }
}
```

Mida: c.size()

Sumar k als elements d'un arbre

```
/* Pre: a = A */
/* Post: a té la mateixa forma que A, el valor de
         cada node d'a és la suma del valor del node
         corresponent d'A més k */
BinTree<int> suma(const BinTree<int>& a, int k);
```

Definició no recursiva: la donada ("tots els nodes de l'arbre")
Definició recursiva: amb mida $|a|$ = nombre de nodes de l'arbre:

- $\text{suma}(\square, k) = \square$
- $\text{suma}(\text{plantar}(x, a_1, a_2), k) =$
  $\text{plantar}(x + k, \text{suma}(a_1, k), \text{suma}(a_2, k))$

Sumar k als elements d'un arbre

```
/* Pre: a = A */
/* Post: a té la mateixa forma que A, el valor de
         cada node d'a és la suma del valor del node
         corresponent d'A més k */
BinTree<int> suma(const BinTree<int>& a, int k)
  if (a.empty())
    return BinTree<int>();
  else
    return BinTree<int>(a.value()+k, suma(a.left(), k),
                                     suma(a.right(),k));
}
```

- Acabament: si $|a| = 0$ l'arbre és buit i estem en un cas base; amb $|a| > 0$ tenim un cas recursiu, i les crides a `suma` són amb a.`left()` i a.`right()`; els dos subarbres tenen mida inferior a la d'a

Sumar k als elements d'un arbre

```
/* Pre: a = A */
/* Post: a té la mateixa forma que A, el valor de
         cada node d'a és la suma del valor del node
         corresponent d'A més k */
BinTree<int> suma(const BinTree<int>& a, int k)
    if (a.empty()) return BinTree<int>();
    return BinTree<int>(a.value()+k, suma(a.left(), k),
                                     suma(a.right(),k));
}
```

- Correcció: si $|a| = 0$ la solució retorna un arbre buit. Si $|a| > 0$ la precondició de les dos crides recursives es compleix i els paràmetres són de mida inferior: $|a.\text{left}()| < |a|$, $|a.\text{right}()| < |a|$. L'H.I. es pot aplicar.
- Correcció: directament de la definició (i aplicació de la H.I.), la funció retorna el resultat correcte.

Immersió o generalització d'una funció. Immersions per afebliment de la Post

Cerca d'un Estudiant en un vector d'Estudiants

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
//       estudiant amb DNI = x
bool cerca(const vector<Estudiant> &v, int x);
```

Cerca d'un Estudiant en un vector d'Estudiants

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
//       estudiant amb DNI = x
bool cerca(const vector<Estudiant> &v, int x);
```

Problema: Què fem decrèixer? No podem fer més petit el vector!

Creem una còpia del vector de mida $v.\texttt{size}() - 1$? Molt ineficient!

Cerca d'un Estudiant en un vector d'Estudiants

Plantegem:

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j);
```

Cerca d'un Estudiant en un vector d'Estudiants

Plantegem:

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j);
```

Buscar en tot el vector és

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
//       estudiant amb DNI = x i v no és buit
bool cerca(const vector<Estudiant> &v, int x) {
  return i_cerca(v, x, v.size()-1);
}
```

Cerca d'un Estudiant en un vector d'Estudiants

Plantegem:

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j);
```

Buscar en tot el vector és

```
// Pre: x és un DNI vàlid
// Post: retorna cert si i només si v conté al menys un
//       estudiant amb DNI = x i v no és buit
bool cerca(const vector<Estudiant> &v, int x) {
  return i_cerca(v, x, v.size()-1);
}
```

i ara podem fer créixer o decrèixer j!

Cerca d'un Estudiant en un vector d'Estudiants

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
  if (j == 0)
     return v[0].consultar_DNI() == x;
  else if (v[j].consultar_DNI() == x)
     return true;
  else
     return cerca(v,x,j-1);
}
```

Cerca d'un Estudiant en un vector d'Estudiants

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
  if (j == 0)
     return v[0].consultar_DNI() == x;
  else if (v[j].consultar_DNI() == x)
     return true;
  else
     return cerca(v,x,j-1);
}
```

Correctesa: Inducció sobre j:
Si $v[j]$ conté un estudiant amb DNI $= x$ llavors $v[0..j]$ conté un estudiant amb DNI $= x$. En cas contrari, si cerca$(v, x, j-1)$ retorna cert llavor $v[0..j-1]$ i pert tant $v[0..j]$ conté un estudiant amb DNI $= x$.

Cerca d'un Estudiant en un vector d'Estudiants

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
  if (j == 0)
     return v[0].consultar_DNI() == x;
  else if (v[j].consultar_DNI() == x)
     return true;
  else
     return cerca(v,x,j-1);
}
```

Cerca d'un Estudiant en un vector d'Estudiants

```
// Pre: x és un DNI vàlid i 0 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
  if (j == 0)
     return v[0].consultar_DNI() == x;
  else if (v[j].consultar_DNI() == x)
     return true;
  else
     return cerca(v,x,j-1);
}
```

Correctesa: Inducció sobre j:
Però si cerca$(v, x, j-1)$ retorn afals, llavors $v[0..j-1]$ no conté un estudiant amb DNI $= x$ i $v[0..j]$ tampoc.

Cerca d'un Estudiant en un vector d'Estudiants

Alternativa: posar "$-1 \leq j$" en la Pre ($v[0..-1]$ denota un subvector buit)

```
// Pre: x és un DNI vàlid i -1 ≤ j < v.size()
// Post: retorna cert si i només si v[0..j] conté al menys un
//       estudiant amb DNI = x
bool i_cerca(const vector<Estudiant>& v, int x, int j) {
  if (j < 0) return false;
  if (v[j].consultar_DNI() == x) return true;
  return cerca(v,x,j-1);
}
```

Motiu: codi més compacte, funciona inclús si el vector $v.\texttt{size}() = 0$.

Funció d'immersió: funció auxiliar

La funció original crida la funció d'immersió

- Fixant els paràmetres addicionals
- Ignorant alguns dels resultats retornats

Canvis en l'especificació: Immersions

Canvis en els paràmetres impliquen canvis en l'especificació:

- Afebliment de la post: la crida recursiva només fa una part de la feina
- Enfortiment de la pre: la crida recursiva rep feta una part de la feina, ella la completa

La primera sol ser més natural. La segona té l'avantatge que dóna solucions més fàcils de transformar a iteratives (si calgués)

Suma dels elements d'un vector: afebliment de la Post

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v);
```

Suma dels elements d'un vector: afebliment de la Post

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v);
```

Funció d'immersió

```
// Pre: -1 <= i < v.size()
// Post: retorna la suma de v[0..i]
int i_suma(const vector<int>& v, int i);
```

Suma dels elements d'un vector: afebliment de la Post

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v);
```

Funció d'immersió

```
// Pre: -1 ≤ i < v.size()
// Post: retorna la suma de v[0..i]
int i_suma(const vector<int>& v, int i);
```

Crida inicial

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v) {
  return i_suma(v, v.size()-1);
}
```

Implementació de la funció d'immersió

```
// Pre: −1 ≤ i < v.size()
// Post: retorna la suma de v[0..i]
int i_suma(const vector<int>& v, int i) {
  if (i < 0)
     return 0;
  else
     return i_suma(v, i-1) + v[i];
}
```

Immersió alternativa

Funció d'inmersió

```
// Pre: 0 ≤ i ≤ v.size()
// Post: retorna la suma de v[i..v.size() − 1]
int i_suma(const vector<int>& v, int i);
```

Immersió alternativa

Funció d'inmersió

```
// Pre: 0 ≤ i ≤ v.size()
// Post: retorna la suma de v[i..v.size() − 1]
int i_suma(const vector<int>& v, int i);
```

Crida inicial

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v) {
  return i_suma(v, 0);
}
```

Implementació de la funció d'immersió

```
// Pre: 0 ≤ i ≤ v.size()
// Post: retorna la suma de v[i..v.size() − 1]
int i_suma(const vector<int>& v, int i) {
  if (i == v.size())
     return 0;
  else
     return v[i]+i_suma(v, i+1);
}
```

Cerca en un vector ordenat

```
// Pre: v.size() > 0 i v està ordenat creixentment
// Post: Retorna una posicio on és troba l'element x dins
//       el vector v, si x ∈ v. Si x ∉ v, retorna -1.
template <class T>
int cerca(const vector<T>& v, const T& x);
```

Afebliment de la post

Afebliments de la Post possibles:

- canviar v per $v[0..i]$
- o canviar v per $v[j..v.size()-1]$
- ...o les dues coses: canviar v per $v[i..j]$!

```
// Pre: −1 ≤ i, j ≤ v.size(), i ≤ j + 1 i v
//      està ordenat creixentment
// Post: Retorna una posicio on és troba l'element x dins
//       el subvector v[i..j], si x ∈ v[i..j]. Si x ∉ v,
//       retorna -1.
template <class T>
int cerca(const vector<T>& v, const T& x);
```

Afebliment de la post

Afebliments de la Post possibles:

- canviar v per $v[0..i]$
- o canviar v per $v[j..v.size()-1]$
- ...o les dues coses: canviar v per $v[i..j]$!

```
// Pre: −1 ≤ i, j ≤ v.size(), i ≤ j + 1 i v
//      està ordenat creixentment
// Post: Retorna una posicio on és troba l'element x dins
//       el subvector v[i..j], si x ∈ v[i..j]. Si x ∉ v,
//       retorna -1.
template <class T>
int cerca(const vector<T>& v, const T& x);
```

- En les dues primeres alternatives d'afebliment de la Post $\Longrightarrow$ cerca seqüencial
- Amb la tercera podem fer cerca dicotòmica

Immersions

No oblideu de:

- Dir quina immersió fareu, quin paràmetre afegir
- Donar la capçalera de la nova funció d'immersió
- Especificar-la! (paper dels nous paràmetres / resultats)
- Donar la crida inicial des de la funció original

Immersions per enfortiment de la Pre ($*$)

Suma dels elements d'un vector

```
// Pre: v.size() > 0
// Post: retorna la suma dels elements de v
int suma(const vector<int>& v);
```

Suma dels elements d'un vector

```
// Pre: v.size() > 0
// Post: retorna la suma dels elements de v
int suma(const vector<int>& v);
```

Enfortiment de la Pre:

```
// Pre:  v.size() > 0, 0 ≤ i < v.size(), i
//       sum és la suma dels elements de v[0..i]
// Post: retorna la suma dels elements de v
int i_suma(const vector<int>& v, int i, int sum);
```

Per enfortiment de la Pre

```
// Pre:  v.size() > 0, 0 ≤ i < v.size(), i
//       sum és la suma dels elements de v[0..i]
// Post: retorna la suma dels elements de v
int i_suma(const vector<int>& v, int i, int sum);
```

Per enfortiment de la Pre

```
// Pre:  v.size() > 0, 0 ≤ i < v.size(), i
//       sum és la suma dels elements de v[0..i]
// Post: retorna la suma dels elements de v
int i_suma(const vector<int>& v, int i, int sum);
```

Crida inicial per calcular tota la suma del vector:

```
// Pre: v.size() > 0
// Post: retorna la suma dels elements de v
int suma(const vector<int>& v) {
  return i_suma(v, 0, 0);
}
```

Implementació de la funció d'immersió

```
// Pre:  v.size() > 0, 0 ≤ i < v.size(), i
//       sum és la suma dels elements de v[0..i]
// Post: retorna la suma dels elements de v
int i_suma(const vector<int>& v, int i, int sum) {
  if (i == v.size()-1)
     return sum;
  else
     return i_suma(v, i+1, sum+v[i]);
}
```

Recursivitat lineal final i algorismes iteratius (**)

Recursivitat lineal final

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva

Recursivitat lineal final

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (*tail recursion*):

Recursivitat lineal final

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (*tail recursion*):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva

Recursivitat lineal final

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (*tail recursion*):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva
  - El resultat de la funció (cas recursiu) és el resultat que s'ha obtingut de la crida recursiva, sense cap modificació

Recursivitat lineal final

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (*tail recursion*):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva
  - El resultat de la funció (cas recursiu) és el resultat que s'ha obtingut de la crida recursiva, sense cap modificació
- Motivació: mètode simple per transformar un algorisme recursiu lineal final en un algorisme iteratiu

Exemple: factorial

```
int fact(int n) {
    if (n <= 1) return n;
    else return n * fact(n - 1);
}
```

Recursivitat lineal però no final: és fa el producte després de la crida recursiva

Exemple: factorial

Enfortiment de la Pre:

```
// Pre: n ≥ i ≥ 0 ∧ p = i!
// Post: retorna n!
int i_fact(int n, int i, int p) {
  if (i == n)
     return p;
  else
     return i_fact(n, i + 1, p * (i + 1));
}
```

- Recursivitat lineal final
- Crida inicial: `fact(n)` $\equiv$ `i_fact(n,0,1)`

Exemple: factorial

1. Transformem paràmetres en variables locals
2. La Pre és l'invariant del bucle
3. La crida inicial dona com inicialitzar les variables

```
// Pre: n ≥ 0
int i_fact(int n) {
    int i = 0; int p = 1;  // <-- paràmetres crida inicial
    while (i != n) {   // negació del cas base
        // Inv: n ≥ i ≥ 0 ∧ p = i!
        p = p * (i+1);    // <-- paràmetres de
        i = i + 1;        //     la crida recursiva
    }
    return p;           // resultat del cas base
}
```

Transformació recursivitat lineal final a iteració, II

Funció recursiva lineal final:

```
// Pre: P(x,y)
T2 func_rec(T1 x, T3 y) {
  T2 res;

  if (cas_base(x))
    res = sol_directa(x,y);
  else
  res = func_rec(new_x(x),
                 new_y(x,y));
  return res;
}
// Post: Q(x,y,s)
```

Iteració equivalent:

```
// Pre: P(x,y)
T2 func_iter(T1 x, T3 y) {
  T2 res;

  while (not cas_base(x)) {
     y = new_y(x,y);
     x = new_x(x);
  }
  res = sol_directa(x,y);
  return s;
}
// Post: Q(x,y,s)
```

Suma d'un vector d'enters

Implementació recursiva final:

```
/* Pre: 0 ≤ i ≤ v.size() i sum és la suma dels
        elements de v[0..i − 1] */
int i_suma(const vector<int>& v, int i, int sum) {
    if (i == v.size()) return sum;
    else return i_suma(v, i + 1, sum + v[i]);
}
/* Post: retorna la suma de tots els elements
   del vector v */
```

Llamada inicial: $\texttt{suma}(v) \equiv \texttt{i\_suma}(v, 0, 0)$

Transformació a iteratiu

```
/* Pre: cert */
int suma_iter(const vector<int>& v) {
  int i = 0; int sum = 0;

  // Inv: 0 ≤ i ≤ v.size() i sum és
  //         la suma dels elements de v[0..i − 1]
  while (i != v.size()) {
    sum += v[i]; // sum + v[i]
    ++i;         // i+1
  }
  return suma;
}
/* Post: el valor retornat es la suma de tots els elements
   del vector v */
```