Disseny recursiu

Ricard Gavaldà

Programació 2
Facultat d'Informàtica de Barcelona, UPC
Primavera 2019

Aquesta presentació no substitueix els apunts

Resum en 4 punts

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

Recursió, definicions recursives i inducció

Principis de disseny recursiu

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

Material addicional: Immersions per enfortiment de la Pre

Material addicional: Recursivitat lineal final i algorismes iteratius

Recursió, definicions recursives i inducció

Alçària d'una pila

Vam veure versions iterativa i recursiva:

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

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

I vam afirmar que feien el mateix

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(P.pop())

Advertiment: P.pop() retorna void. Quan fem aquests raonaments suposem que també retorna la pila resultat, però això no serà vàlid en el codi.

Exemple: factorial

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

```
/* Pre: n >= 0 */
/* Post: el resultat es n! */
int fact(int n) {
    int f = 1;
    while (n > 0) { f *= n; --n; }
    return f;
}
```

Exemple: factorial

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

Lema:

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

```
/* Pre: n >= 0 */
/* Post: el resultat es n! */
int fact(int n) {
    if (n == 0) return 1;
    else return n*fact(n-1);
}
```

En general

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 en P.top() i P.pop()
Exemple: producte X*Y es pot fer Y més petit com a (Y-1) o com a (Y/2)

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

Suma dels elements d'una pila

Si $P = e_1, \ldots, e_n$, $\text{suma}(P) = e_1 + \cdots + e_n$

Inductivament:

$$\mathit{suma}(P) = \begin{cases} 0 & \text{si } P \text{ és buida} \\ P.\mathit{top}() + \text{suma}(P.\mathit{pop}()) & \mathit{altrament} \end{cases}$$

Cerca en una pila

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

Versió iterativa: exercici
Ull: que sigui una *cerca*, no un *recorregut*

Versió recursiva:

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

$$\begin{array}{c}\text{(x apareix a P)}\\ \text{si i només si}\\ \text{(x == P.top()) o (x apareix a P.pop())}\end{array}$$

I *ara* pensem com codificar la definició eficientment en C++

Igualtat de piles

Donades dues piles, dir si són iguals
= per a cada posició i, els elements i-èssims coincideixen

Versió recursiva:

- si p1 i p2 són buides, són iguals
- si p1 és buida i p2 no, o a l'inrevés, llavors són diferents
- si ni p1 ni p2 són buides, ...
  si p1.top() != p2.top(), les piles són diferents
  i si p1.top() == p2.top() cal que a més
  p1.pop() i p2.pop() siguin iguals

Igualtat de piles, versió recursiva

```
/* Pre: p1 = P1, p2 = P2 */
/* Post: El resultat ens indica si P1 i P2 són iguals,
         i p1 i p2 poden haver canviat */
bool piles_iguals(stack<int>& p1, stack<int>& p2) {
    if (p1.empty() and p2.empty()) return true;
    else if (p1.empty() or p2.empty()) return false;
    else if (p1.top() != p2.top()) return false;
    else {
        p1.pop(); p2.pop();
        return piles_iguals(p1,p2);
    }
}
```

Igualtat de piles, versió recursiva (2)

Una variant:

```
/* Pre: p1 = P1, p2 = P2 */
/* Post: El resultat ens indica si P1 i P2 són iguals,
         i p1 i p2 poden haver canviat */
bool piles_iguals(stack<int>& p1, stack<int>& p2) {
    if (p1.empty() or p2.empty())
        return (p1.empty() and p2.empty());
    else if (p1.top() != p2.top()) return false;
    else {
        p1.pop(); p2.pop();
        return piles_iguals(p1,p2);
    }
}
```

Igualtat de piles, versió iterativa

Queda com a exercici. Ull: cerca i no recorregut!
Observació: és temptador comprovar abans que res si `alcaria(p1) != alcaria(p2)`

Seria eficient si tenim una operació `p.size()` que no recorre la pila.

No si l'haguessim de calcular recorrent la pila

Principis de disseny recursiu

Principis de disseny recursiu

Volem implementar recursivament una funció

```
// Pre: Pre(x)
// Post: el valor retornat és una funció F(x)
tipus_sortida F(tipus_entrada x);
```

o una acció

```
// Pre: Pre(x) i x = X
// Post: Post(X,r)
void F(T1 x, T2 r);
```

(`x` i `r` poden ser més d'un paràmetre)

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" dels paràmetres `x` tal que
   - si la mida de `x` és 0 (o negativa), som en un cas base
   - les crides recursives es fan amb paràmetres `x` de mida menor que `x`
   - ha de ser sempre un enter

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 $Pre(x)$,

- 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

Formalització:

- Triem una funció mida dels paràmetres, valor enter
- Demostrem: Tot paràmetre de mida ≤ 0 és un cas base
- Demostrem: Cada crida recursiva fa decrèixer la mida dels paràmetres

Correctesa d'un algorisme recursiu

- A demostrar: Per a tots paràmetres x tal que $\text{Pre}(x)$ és cert, l'algorisme acaba satisfent $\text{Post}(x)$
- Quan x és un cas base, sense recursió: Demostrem-ho directament
- Hipòtesi d'inducció: Per a tots paràmetres x' tal que $\text{Pre}(x')$ és cert, i tals que x' és "més petit" que x l'algorisme acaba satisfent $\text{Post}(x')$
- Quan x entra als casos casos recursius: Comprovar que totes les crides recursives amb paràmetres x' de mida més petita que la de x i que satisfan $\text{Pre}(x')$
- Apliquem la H.I. per veure que després de la crida amb x' se satisfà $\text{Post}(x')$
- Argumentem que els càlculs addicionals ens porten a $\text{Post}(x)$
- Finalment, comprovem que totes les accions, funcions i mètodes cridats satisfan les precondicions respectives

Exemples

Producte
Factorial
Nombres binomials
Potència ràpida
Mergesort en un vector
Revessar una llista
Cerca un element en una cua
Sumar k als elements d'un arbre

Producte

```
// Pre: x = X, y = Y >= 0
// Post: el resultat és X*Y
int producte(int x, int y);
```

Definició 1: $X * Y = \underbrace{X + X + \cdots + X}_{Y}$

Definició 2: Per a $Y \geq 0$,

$$X * Y = \begin{cases} 0 & \text{si } Y = 0 \\ X + X * (Y - 1) & \text{si } Y > 0 \end{cases}$$

(mida = Y)

Factorial

```
// Pre: n >= 0
// Post: el resultat és n!
int factorial(int n;
```

Definició 1: $n! = n \cdot (n-1) \cdots 2 \cdot 1$
Definició 2: Per a $n \geq 0$,

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

(mida = n)

Nombres binomials

```
// Pre: n >= m >= 0
// Post: el resultat es el binomial (n sobre m)
int binomial (int n, int m);
```

Recordem:

$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$ = nombre de subconjunts de $\{1, \ldots, n\}$ de mida m

Nombres binomials, II

Exemple: poden haver-hi diverses definicions recursives equivalents, que poden portar a solucions d'eficiència i elegància diferents

Suposem que triem "mida" $= n$

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

Nombres binomials, III

I si triem "mida" $= m$?

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

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 solució més ineficient)

Potència ràpida

```
// Pre: x > 0, y >= 0
// Post: el resultat es x elevat a y
int potencia(int x, int y);
```

Observem que

$$X^Y = \begin{cases} 1 & \text{si } Y = 0 \\ X * X^{Y-1} & \text{si } Y > 0 \text{ (sempre)} \\ (X^2)^{Y/2} & \text{si } Y > 0 \text{ i a més } Y \text{ parell} \end{cases}$$

Potència ràpida

En la implementació, prioritzem la segona regla sobre la primera perquè és més ràpida:

```
// Pre: x > 0, y >= 0
// Post: el resultat es x elevat a y
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x,y-1);
    else return potencia(x*x,y/2);
}
```

Mida: Y va bé, però també $2\log_2(Y)$, ja que $\log_2(Y/2) = \log_2(Y) - 1$

Mergesort

```
// Pre: 0 <= e <= d < v.size()
// Post: els elements de v[e..d] son els inicials,
//       pero ordenats creixentment
void mergesort(vector<double>& 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() i v[e..m] i v[m+1..d], per separat,
//     son ordenats creixentment
// Post: els elements de v[e..d] son els inicials,
//     pero ordenats creixentment, i la resta de v no ha canviat
void fusiona(vector<double>& v, int e, int m, int d);
```

Revessar una llista

```
// Pre: l es una llista amb elements e1...en, n>=0
// Post: l conte la llista amb elements en...e1
void revessar(list<double>& l);
```

Definició recursiva, amb mida = `l.size()`:

- revessada(llista buida) = llista buida
- revessada(e_1 seguit de llista L') = revessada(L') seguit d'e_1

Revessar una llista

```
// Pre: l es una llista amb elements e1...en, n>=0
// Post: l conte la llista amb elements en...e1
void revessar(list<double>& l) {
    if (not l.empty()) {
        double x = *(l.begin());
        l.erase(l.begin());
        revessar(l);
        l.insert(l.end(),x);
    }
}
```

Cerca d'un element en una cua

```
// Pre:  c = C
// Post: el resultat indica si x apareix en C
bool cerca(queue<double> &c, double x) {
}
```

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

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

- cerca(cua buida, x) = false
- cerca(e_1 seguit de cua C', x) = (($e_1 = x$) o cerca(C',x))

Cerca d'un element en una cua

```
// Pre:  c = C
// Post: el resultat indica si x apareix en C
bool cerca(queue<double> &c, double x) {
    if (c.empty()) return false;
    else if (c.front() == x) return true;
    else {
        c.pop();
        return cerca(c, x);
    }
}
```

Sumar k als elements d'un arbre

```
/* Pre: a = A */
/* Post: retorna un arbre amb la mateixa forma que A, on
      el valor de cada node es la suma del valor del node
      corresponent d'A mes k */
BinTree<int> suma_k(const BinTree<int> &a, int k);
```

Definició no recursiva: la donada ("tots els nodes de l'arbre")

Definició recursiva:

- suma_k(arbre buit, k) = arbre buit
- suma_k(BinTree(x,$a1$,$a2$),k) = BinTree($x+k$,suma_k($a1$,k),suma_k($a2$,k))

Sumar k als elements d'un arbre

```
/* Pre: a = A */
/* Post: retorna un arbre amb la mateixa forma que A, on
      el valor de cada node es la suma del valor del node
      corresponent d'A mes k */
BinTree<int> suma_k(const BinTree<int> &a, int k) {
    if (a.isempty()) return BinTree();
    else {
        int n = a.value() + k;
        BinTree<int> a1 = suma_k(a.left(),k);
        BinTree<int> a2 = suma_k(a.right(),k);
        return BinTree<int>(n,a1,a2);
    }
}
```

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

Cerca d'un Estudiant en un vector d'Estudiants

```
/* Pre: x es un dni valid */
bool cerca(const vector<Estudiant> &v, int x);
/* Post: El resultat diu si hi ha algun estudiant
         amb dni x a v */
```

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

Creem una còpia del vector de mida v.size()-1?? Ineficient!

Cerca d'un Estudiant en un vector d'Estudiants

Plantegem:

```
/* Pre: x es un dni valid i 0 <= j < v.size() */
bool i_cerca(const vector<Estudiant> &v, int x, int j);
/* Post: El resultat diu si hi ha algun estudiant
         amb dni x a v[0..j] */
```

Buscar en tot el vector és

```
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 es un dni valid i 0 <= j < v.size() */
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);
}
/* Post: El resultat ens diu si hi ha algun estudiant
         amb dni x a v[0..j] */
```

Correctesa: Inducció sobre j:
Si cerca(v,x,j-1) retorna ...(j-1) llavors aquest a funció retorna ...(j)

Cerca d'un Estudiant en un vector d'Estudiants

Alternativa: eliminar "0 <= j" en la Pre

```
/* Pre: x es un dni valid i j < v.size() */
```

i canviar el case base de

```
if (j == 0) return (v[0].consultar_DNI() == x);
```

a

```
if (j < 0) return false;
```

Motiu: codi més compacte, i x no és a v[0..negatiu]

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:
  - Una crida recursiva només fa una part de la feina
- Enfortiment de la pre:
  - Una 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: -- */
/* Post: retorna la suma de v */
int suma(const vector<int> &v);
```

Funció d'immersió:

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

Crida inicial - implementació de suma:

```
int suma(const vector<int> &v) {
    return i_suma(v, v.size()-1);
}
```

Implementació de la funció d'immersió

```
/* Pre: 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;  // v[0..i] vector buit
    else return i_suma(v,i-1) + v[i];
}
```

Immersió alternativa

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

Funció d'immersió:

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

Crida inicial - implementació de suma:

```
int suma(const vector<int> &v) {
    return i_suma(v,0);
}
```

Implementació de la funció d'immersió

```
/* Pre: i >= 0 */
/* 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;   // v[i..v.size()-1] vector buit
    else return v[i] + i_suma(v,i+1);
}
```

Cerca en un vector ordenat

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

La Post equival a:

```
/* Post: El valor retornat es la posicio on es troba l'element x dins
   el vector v[0..v.size()-1]. Si x no es troba a v[0..v.size()-1],
   llavors el valor retornat es un nombre negatiu. */
```

Afebliment de la post

Post:

```
/* Post: El valor retornat es la posicio on es troba l'element x dins
   el vector v[0..v.size()-1]. Si x no es troba a v[0..v.size()-1],
   llavors el valor retornat es un nombre negatiu. */
```

Afebliments de la Post possibles:

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

```
/* Post: El valor retornat es la posicio on es troba l'element x dins
   el vector v[i..j]. Si x no es troba a v[i..j],
   llavors el valor retornat es un nombre negatiu. */
```

Si només fem un canvi, cerca seqüencial
Si fem els dos, arribem a la cerca recursiva

Important

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

Material addicional:
Immersions per enfortiment de la Pre

Suma dels elements d'un vector

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

Enfortiment de la Pre:

```
/* Pre: v.size() > 0, 0 <= i < v.size(), sum es la suma
   dels elements de v des de 0 fins a i */
/* Post: el valor retornat es la suma de tots els 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(), sum es la suma
   dels elements de v des de 0 fins a i */
/* Post: el valor retornat es la suma de tots els 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 */
int suma(const vector<int> &v) {
    return i_suma(v,0,v[0]);
}
/* Post: el valor retornat es la suma dels elements de v */
```

Implementació de la funció d'immersió

```
/* Pre: v.size() > 0, 0 <= i < v.size(), sum es la suma
   dels elements de v des de 0 fins a i */
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+1]);
    }
}
/* Post: el valor retornat es la suma de tots els elements de v */
```

Material addicional: Recursivitat lineal final i algorismes iteratius

Recursivitat lineal final

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

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

Recursivitat no final ("*" després de crida)

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 final
Crida inicial: fact(n) = i_fact(n,0,1);

Exemple: factorial

Transformem paràmetres en variables locals, Pre en Invariant:

```
// Pre: n > 0
int i_fact(int n) {
   int i = 0; int p = 1;  // de la crida inicial
   while (i != n) {   // del cas base
       // Inv: n >= i >= 0, p = i!
       p = p*(i+1);   // de la crida recursiva
       i = i+1;
   }
   return p;          // del cas base
}
```

Recursivitat lineal no final i final I

Recursivitat lineal no final:

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

Funció amb postcondició constant: la postcondició de la crida recursiva és la mateixa que la postcondició de la funció

Recursivitat lineal no final i final II

Recursivitat lineal final:

```
/* Pre: 0 <= i <= v.size(); suma_parcial = suma dels primers
   i elements de v */
int i_suma(const vector<int> &v, int i, int suma_parcial) {
    int suma;
    if (i == v.size()) suma = suma_parcial;
    else suma = i_suma(v, i + 1, suma_parcial + v[i]);
    return suma;
}
/* Post: el valor retornat es la suma de tots els elements
   del vector v */
```

Transformació recursivitat lineal final a iteració, I

Esquema de funció recursiva lineal final:

```
/* Pre: Q(x) */
T2 f(T1 x, T3 y) {
  T2 s;

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

Esquema original:

```
/* Pre: Q(x) */
T2 f(T1 x, T3 y) {
  Tipus2 r,s;

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

Transformació recursivitat lineal final a iteració, II

Funció recursiva lineal final:

```
/* Pre: Q(x) */
T2 f(T1 x, T3 y) {
  T2 s;

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

Iteració equivalent:

```
/* Pre: Q(x) */
T2 f(T1 x, T3 y) {
  T2 s;

  while (not c(x)) {
     y = gy(x,y);
     x = g(x);
  }
  s = d(x,y);
  return s;
}
/* Post: R(x,y,s) */
```

Suma d'un vector d'enters

Implementació recursiva lineal:

```
/* Pre: 0 <= i <= v.size(); suma_parcial = suma dels primers
   i elements de v */
int i_suma(const vector<int> &v, int i, int suma_parcial) {
    int suma;

    if (i == v.size()) suma = suma_parcial;
    else suma = i_suma(v, i + 1, suma_parcial + v[i]);

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

Transformació a iteratiu

Clau: adaptar l'esquema general d'una funció recursiva amb un sol paràmetre a tres paràmetres.

```
/* Pre: 0 <= i <= v.size(); suma_parcial=suma dels primers
   i elements de v */
int i_suma(const vector<int> &v, int i, int suma_parcial) {
    int suma;

    while (i != v.size()) {
        suma_parcial = suma_parcial + v[i];
        ++i;
    }
    suma = suma_parcial;

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