Universitat de Lleida. Escola Universitària Politècnica.

Introducció a l'algorísmica

Setembre 2001

Josep Maria Ribó i Balust

Índex

Capítol 1

Especificació d'algorismes

1.1 Definicions inicials

Algorisme Procediment per a resoldre un problema, escrit amb tot rigor en una notació formal i no ambígua que anomenarem *notació algorísmica* o també *pseudocodi*.

Existeixen diversos paradigmes per expressar procediments per resoldre problemes (els paradigmes imperatiu, lògic i funcional). Nosaltres utilitzarem el paradigma imperatiu. Segons aquest, un algorisme és una seqüència de zero o més *instruccions* escrites en un llenguatge algorísmic.

Els tipus d'instruccions que podem utilitzar per construir algorismes són les següents:

- L'assignació ($var := exp$;)
- La composició seqüencial ($a_1; a_2$;)
- La composició alternativa (**si**...**fsi**)
- La composició iterativa (**mentre** ... **fmentre**)
- Crida a acció o funció (f($p_1, ..., p_n$);)

Més endavant estudiarem en detall la semàntica d'aquestes instruccions.

Un algorisme es descriu en llenguatge algorísmic o *pseudocodi* de la manera següent:

algorisme <nom_algorisme> **és**

<instruccions>

falgorisme

Tipus de dades: Anomenarem *tipus* a la *classe de valors* a la que pertany cada valor que pot aparèixer en un algorisme. Són exemples de tipus els naturals, els enters, els caràcters...

Un tipus de dades consta de:

1. Un conjunt de valors i
2. Una llista d'operacions que es poden realitzar sobre aquells valors.

Exemple: El tipus de dades *enter* consta de:

1. El conjunt de valors: -1000, -50, -20, -1, 0, 1, 50, 100, 2000...
2. La llista d'operacions: **+, −, *, div, mod**.

Variable: Una variable és una *caixa* on podem emmagatzemar una informació rellevant pel nostre algorisme de tal manera que aquesta informació pugui canviar si així es desitja. Una variable l'anomenarem amb una cadena de caracters: *x, y, producte, quantitat.* En una variable són importants:

1. El seu valor: Informació concreta emmagatzemada per la variable en un moment determinat.
2. El seu tipus: Classe de valors que pot emmagatzemar la variable i que determinarà, a la seva vegada, les operacions que es poden realitzar sobre ella.

Exemple:

La variable anomenada i és de tipus enter i conté el valor 123. Com i és de tipus enter podrem fer sobre ella les operacions $+$, $-$, $*$, **div** i **mod**.

Declaració de variables: Instrucció del llenguatge que assigna a cada variable que intervé a l'algorisme el tipus d'aquella variable i, opcionalment, un valor inicial. Tota variable que intervé en un algorisme ha d'haver estat declarada abans de ser utilitzada.

Exemple:

x:enter(12);

aa:caràcter('A');

Aquestes dues instruccions declaren les variables x i aa de tipus enter i caràcter respectivament i els donen un valor inicial de 12 i 'A'.

La declaració de variables es fa en una zona de declaració de variables que notem de la manera següent:

var
x: enter(12);
aa:caràcter('A');
fvar

Hi ha dos punts dins de la descripció d'un algorisme en els quals pot aparèixer una zona de declaració de variables:

1. Abans de la descripció de l'algorisme:
   var
   ...
   fvar
   algorisme
   ...
   falgorisme
   Les variables declarades d'aquesta manera s'anomenen *variables globals.*
2. A l'inici de la descripció de l'algorisme.
   algorisme
   var
   ...
   fvar
   ...
   falgorisme
   Les variables declarades d'aquesta manera s'anomenen *variables locals.*

Tornarem a parlar de les variables globals i locals a 2.5.4

Constant: Etiqueta amb la qual ens referim a un valor immutable (això és, que no pot variar al llarg de tota l'execució de l'algorisme).

Exemple:

$A = 12$
$B = "Pepet"$

Insistim en què el valor d'una constant no pot canviar.

La declaració de constants es fa en una zona de declaració de constants que notem de la manera següent:

const
$A = 12$
$B = "Pepet"$
fconst

Aquesta zona declarativa es troba abans de la zona de declaració de les variables globals.

Expressió vàlida: Una expressió vàlida d'un tipus T és un conjunt de constants, variables i símbols de crida a operacions agrupats segons unes certes regles sintàctiques. L'avaluació d'una expressió vàlida de tipus T dóna un valor de tipus T.

Com a exemple, $2 + 3$ és una expressió vàlida de tipus enter i $(x < 2) \wedge cert \wedge t$ és una expressió vàlida de tipus booleà si x és una variable de tipus natural o enter i t és una variable booleana.

Estat: Entenem per estat d'un algorisme en un instant determinat de la seva execució les propietats que compleixen les variables que formen part d'aquell algorisme en aquell instant. Aquestes propietats habitualment fan referència als valors que tenen les variables. L'estat d'un algorisme en un moment determinat el descrivim en forma d'*enunciats* o *assercions* que situem al punt exacte de l'algorisme on es compleixen.

Per exemple, si pensem que immediatament després de l'execució d'un determinat algorisme anomenat A, una variable x que intervé en algorisme té un valor més gran que 0, podem escriure:

A;
$\{x > 0\}$

$\{x > 0\}$ és un enunciat o asserció que expressa una propietat sobre l'estat de l'algorisme A immediatament després de la seva execució.

Amb tot el que hem definit, l'estructura d'un algorisme és la següent:

const

<declaració constants>

fconst

var

<declaració variables globals>

fvar

algorisme

var
<declaració variables locals>
fvar

<instruccions>

falgorisme

Les intruccions les estudiarem al capítol 2.

1.2 Tipus predefinits

Al llenguatge algorísmic considerem els tipus predefinits següents:

- natural
- enter
- booleà
- caràcter
- real
- cadena
- vector

1.2.1 Natural

El tipus *natural* representa els nombres naturals i el 0.

Constants

Les constants de tipus natural es representen de la forma habitual: 0, 1, 2, 346...

Declaració de variables de tipus natural

i: natural;
Declara una variable anomenada i de tipus natural
Opcionalment es poden inicialitzar variables de tipus natural al mateix temps que es declaren:
i: natural(9);
La variable natural i té el valor 9.

Operacions

Les operacions associades al tipus *natural* són les següents:

- $+(i_1$: natural, i_2:natural) $\longrightarrow$ natural
  Retorna la suma entre i_1 i i_2.
- $-(i_1$: natural, i_2:natural) $\longrightarrow$ natural
  Retorna la resta entre i_1 i i_2.
- $*(i_1$: natural, i_2:natural) $\longrightarrow$ natural
  Retorna el producte entre i_1 i i_2.
- **div**(n: natural, d:natural) $\longrightarrow$ natural
  Retorna el quocient de la divisió entera entre i_1 i i_2.

- **mod**(n: natural, d:natural) $\longrightarrow$ natural

  Retorna el residu de la divisió entera entre n i d.

  Si n **div** $d = q$ i n **mod** $d = r$ es compleix que:

  $n = d * q + r$ i $0 \leq r < d$.

  Exemples:

  10 **div** 3 =3
  10 **mod** 3 =1
  3 **div** 10=0
  3 **mod** 10 =3
  3 **div** 0=indefinit (error)
  3 **mod** 0 =indefinit (error)

- =(i_1: natural, i_2: natural) $\longrightarrow$ booleà

  Retorna cert si i_1 és igual a i_2 i fals en cas contrari. També podem comparar naturals usant la resta dels operadors relacionals: $\leq$, $\geq$, $<$, $>$.

1.2.2 Enter

El tipus *enter* representa els nombres enters.

Constants enteres

Les constants de tipus enter es representen de la forma habitual: 0, 1, -4, -1000, 578...

Declaració de variables de tipus enter

i: enter;

Declara una variable anomenada i de tipus enter.

Opcionalment es poden inicialitzar variables de tipus enter al mateix temps que es declaren:

i: enter(-9);

La variable entera i té el valor -9.

Operacions

- +(i_1: enter, i_2:enter) $\longrightarrow$ enter

  Retorna la suma entre i_1 i i_2.

- $-$(i_1: enter, i_2:enter) $\longrightarrow$ enter

  Retorna la resta entre i_1 i i_2.

- *(i_1: enter, i_2:enter) $\longrightarrow$ enter

  Retorna el producte entre i_1 i i_2.

- **div**(n: enter, d: enter) $\longrightarrow$ enter

  Retorna el quocient de la divisió entera entre n i d.

- **mod**(n: enter, d: enter) $\longrightarrow$ enter

  Retorna el residu de la divisió entera entre n i d.

  Nota important: les operacions **div** i **mod** nmomés estan definides si $n \geq 0$ i $d > 0$.

- $=$(i_1: enter, i_2: enter) $\longrightarrow$ booleà

  Retorna cert si i_1 és igual a i_2 i fals en cas contrari. També podem comparar enters usant la resta dels operadors relacionals: $\leq$, $\geq$, $<$, $>$.

1.2.3 Booleà

El tipus booleà representa els valors de veritat $CERT$ i $FALS$.

Constants

Existeixen dues constants de tipus booleà que representarem de la manera següent: $CERT$ i $FALS$.

Declaració de variables de tipus booleà

b: booleà;

Declara una variable anomenada b de tipus booleà.

Opcionalment es poden inicialitzar variables de tipus booleà al mateix temps que es declaren:

b: booleà(CERT);

La variable booleana b té el valor CERT.

Operacions

- $\wedge$(b_1: booleà, b_2:booleà) $\longrightarrow$ booleà

  Retorna la **i** lògica entre b_1 i b_2. El seu comportament està descrit per la taula següent:

b_1	b_2	$b_1 \wedge b_2$
FALS	FALS	FALS
FALS	CERT	FALS
CERT	FALS	FALS
CERT	CERT	CERT

  Notem que $b_1 \wedge b_2$ avalua CERT només quan totes dues ho són.

- $\vee$(b_1: booleà, b_2:booleà) $\longrightarrow$ booleà

  Retorna la **o** lògica entre b_1 i b_2. El seu comportament està descrit per la taula següent:

b_1	b_2	$b_1 \vee b_2$
FALS	FALS	FALS
FALS	CERT	CERT
CERT	FALS	CERT
CERT	CERT	CERT

  Notem que $b_1 \vee b_2$ avalua FALS només quan totes dues ho són.

- $\neg(b$: booleà) $\longrightarrow$ booleà

  Retorna la negació de b. El seu comportament està descrit per la taula següent:

b	$\neg b$
FALS	CERT
CERT	FALS

- $=(b_1$: booleà, b_2: booleà) $\longrightarrow$ booleà

  Retorna cert si b_1 és igual a b_2 ($FALS = FALS$ i $CERT = CERT$) i fals en cas contrari $CERT \neq FALS$. També podem compararbooleans usant la resta dels operadors relacionals: $\leq$, $\geq$, $<$, $>$ ($FALS < CERT$).

Expressions booleanes

Veiem alguns exemples d'expressions booleanes:

CERT
CERT$\wedge\neg$FALS
$b_1 \wedge (b_2 \vee \neg b_3)$
$b_1 \wedge (n = 0)$
$n > 9$

on b_1, b_2, b_3 : booleà; i n: natural

1.2.4 Caràcter

El tipus *caràcter* representa els caràcters.

Els caràcters es representen internament usant algun tipus de codificació. Les més esteses són: la codificació ASCII (que s'ha utilitzat massivament fins ara) i la UNICODE (que amplia la ASCII. Ara s'està estenent el seu ús).

Qualsevol codificació manté la propietat següent:

> Si c_1 és anterior alfabèticament a c_2, el codi de c_1 és menor que el de c_2.

Constants

Les constants caràcter les visualitzem entre apòstrofs: 'a'.

Notem la diferència entre el caràcter '0' i el natural 0 ('0' és el caràcter que representa el natural 0). Internament el caràcter '0' es representa amb el valor que li pertoca segons la codificació usada (en ASCII, per exemple '0' està codificat com 48). En canvi, el natural 0 es representa usualment en complement a 2.

Declaració de variables de tipus caràcter

c: caràcter;

Declara una variable anomenada c de tipus caràcter.

Opcionalment es poden inicialitzar variables de tipus caràcter al mateix temps que es declaren:

c: caràcter('p');

La variable caràcter c té el valor 'p'.

Operacions

- codi(c:caràcter) $\longrightarrow$ natural

  Retorna el codi intern que s'utilitza per representar el caràcter. Aquest codi sol ser ASCII o UNICODE.

  Exemples:

  codi ('a')=97 (i.e. 97 és el codi ASCII del caràcter 'a')
  codi ('b')=98 (i.e. 98 és el codi ASCII del caràcter 'b')
  codi ('A')=65 (i.e. 65 és el codi ASCII del caràcter 'A')
  codi ('B')=66 (i.e. 66 és el codi ASCII del caràcter 'B')
  codi ('0')=48 (i.e. 48 és el codi ASCII del caràcter '0')
  codi ('1')=49 (i.e. 49 és el codi ASCII del caràcter '1')
  codi ('9')=57 (i.e. 57 és el codi ASCII del caràcter '9')

- caract(i:natural) $\longrightarrow$ caràcter

  Retorna el caràcter que té codi i. Si no hi ha cap caràcter amb codi i, el resultat d'aquesta operació és indefinit.

  Exemples:

  caract(97)='a'
  caract(99999)=indefinit

- $+(c_1$:caràcter, i: natural) $\longrightarrow$ natural

  En realitat és un abús de notació. La forma correcta és codi(c_1)+i

  Exemple: 'a'+1=98

- $-(c_1$:caràcter, i: natural) $\longrightarrow$ enter

  En realitat és un abús de notació. La forma correcta és codi(c_1)$-i$

  Exemple: 'b'$-$1=97

- $-(c_1$:caràcter, c_2: caràcter) $\longrightarrow$ enter

  Retorna la diferència entre el codi del primer caràcter i el del segon.

  En realitat és un abús de notació. La forma correcta és codi(c_1)$-$codi(c_2).

  Exemples:

  'b'$-$'a'=1
  'a'$-$'b'=-1
  '4'$-$'0'=4

  Noteu al tercer exemple que el resultat de l'operació és l'enter 4, mentre que els operadors són els caràcters '4' i '0'.

- $=(c_1$: caràcter, c_2: caràcter) $\longrightarrow$ booleà

  Retorna cert si c_1 és igual a c_2 i fals en cas contrari. També podem comparar caràcters usant la resta dels operadors relacionals: $\leq$, $\geq$, $<$, $>$. La comparació en aquests casos, es fa segons el valor del seu codi.

1.2.5 Vector

El tipus *vector* representa una seqüència d'elements d'un tipus determinat als que s'accedeix a través de la posició que ocupen dins de la seqüència. La longitud màxima del vector es prefixa en el moment de la seva declaració.

Cal notar que, mentre que els tipus que hem vist fins ara eren atòmics (un valor de tipus natural, per exemple, no es pot descompondre en unitats menors), els vectors són *compostos* o *estructurats*: Un valor de tipus vector es pot veure com una col.lecció de valors que s'estructuren d'una manera determinada.

Constants

Les constants de tipus vector les representarem de la manera següent:

1	2	3	4	5	6	7	8
6	5	2	9	10	78	4	2

Aquesta figura representa una constant vector de naturals i el llegim de la manera següent:

L'índex 1 del vector està ocupat pel valor 6.
L'índex 2 del vector està ocupat pel valor 5.
L'índex 3 del vector està ocupat pel valor 2.
...
L'índex 8 del vector està ocupat pel valor 2.

També es diu que 6 és el valor associat a l'índex 1.

Declaració de variables de tipus vector

v: vector<natural>(N);

Declara la variable v com a vector de naturals, de manera que podrà contenir fins a N naturals (N ha de ser una constant, per exemple, N=20). N és la *capacitat* del vector.

Aquesta declaració crea el vector següent:

v:

1	2	3	4	...	N
				...	

Aquesta declaració *és absolutament equivalent* a la següent:

v: **vector**[1..N] **de** natural;

En aquests apunts utilitzarem ambdues indistintament.

Com sempre, podem aprofitar la declaració d'una variable per tal d'inicialitzar-la:

v: vector<natural>(N,5);

o bé, de manera equivalent:

v: **vector**[1..N] **de** natural (5);

El vector que es crea en aquest cas és el següent:

v:

1	2	3	4	...	N
5	5	5	5	...	5

Finalment, podem inicialitzar un vector amb el valor d'un altre de la manera següent:

v_2: vector<natural>(v);

Després d'aquesta declaració v_2 i v són dos vectors iguals.

Operacions

Les operacions que es poden fer sobre un vector són les següents:

- Accés als elements individuals d'un vector.

  Donada la declaració:

  v: vector<natural>(N,5);

  Podem accedir a l'element situat a l'ındex i ($i = 1..N$), fent $v[i]$ ($v[1], ..., v[5]$) .

  En aquest cas, $v[i] = 5$ pe r a tot $i : 1 \leq i \leq N$.

  L'accés $v[k]$ per $k > N$ o bé $k \leq 0$ dóna un valor indefinit i és del tot incorrecte.

  És interessant notar que els vectors ens permeten accedir de manera directa a un dels seus components a través de l'índex on està situat aquell component. Per aquest motiu es diu que els vectors proporcionen *accés directe per índex*. A l'assignatura d'EDALG estudiarem estructures de dades que permeten fer *accés directe per contingut o clau*.

- =(v_1: vector, v_2: vector) $\longrightarrow$ booleà

  Retorna CERT si v_1 és igual a v_2 (això és: si v_1 i v_2 tenen la mateixa capacitat i els valors associats a cadascun dels índexos de v_1 coincideix amb el valor associat al mateix índex de v_2) i FALS en cas contrari.

1.2.6 Cadena

El tipus *cadena* representa les cadenes de caràcters.

Constants

Les constants cadena les representem entre doble cometa: "pepet".

la cadena buida la representem "".

Declaració de variables de tipus cadena

cad: cadena;

Declara una variable anomenada *cad* de tipus cadena.

Opcionalment es poden inicialitzar variables de tipus cadena al mateix temps que es declaren:

cad: cadena("anna");

La variable cadena *cad* té el valor "anna".

Representació interna de les cadenes

Habitualment, els llenguatges de programació representen les cadenes com a vectors amb una determinada marca de final.

Per exemple, la cadena "anna" es representaria de la manera següent:

1	2	3	4	5
a	n	n	a	$\mathcal{M}$

'$\mathcal{M}$' representa el caràcter *marca de final*.

Notem la diferència entre el caràcter 'a' i la cadena "a".

La representació del caràcter 'a' es far en termes d'un sol caràcter:

a

Mentre que la cadena "a" es representa mitjançant un vector amb dos caràcters ('a' i la marca de final $\mathcal{M}$):

1	2
a	$\mathcal{M}$

Igualment la cadena buida ("") es representa amb un vector tal que el seu primer índex coincideix amb la marca de final '$\mathcal{M}$'.

Operacions

- +(c_1: cadena, c_2:cadena) $\longrightarrow$ cadena

  Retorna una cadena formada per al cadena c_1 en primer lloc i la cadena c_2 tot seguit.

  "pepet"+"anna"="pepetanna"

- Accés a un element de la cadena.

  L'accés a un element de la cadena es fa de la mateixa manera que a un vector (com hem dit, les cadenes se solen representar com a vectors).

  Així doncs:

  s: cadena("anna");

  es té que s[1]='a', s[2]='n'..., s[5]='$\mathcal{M}$'.

  Per altra banda, s[6]= indefinit.

  Cal recordar que no s'ha d'accedir de cap manera a posicions de la cadena més enllà de la marca de final. Un algorisme que faci això serà considerat incorrecte.

- longitud(s:cadena) $\longrightarrow$ natural

  Retorna el nombre de caràcters de què consta la cadena s (sense comptar la marca de final).

  longitud("anna")=4
  longitud("")=0

- inserir(s:cadena, pos:natural, c: caràcter) $\longrightarrow$ cadena

  Retorna la cadena que consisteix en afegir el caràcter c a la posició pos de la cadena s. Si pos no és una posició de la cadena, el resultat és indefinit.

  inserir("anna",3,'x')="anxna"
  inserir("anna",6,'x')=indefinit

  Novament recordem que aquesta darrera situació és una situació errònia.

- =(s_1: cadena, s_2: cadena) $\longrightarrow$ booleà

  Retorna cert si s_1 és igual a s_2 i fals en cas contrari. També podem comparar cadenes usant la resta dels operadors relacionals: $\leq$, $\geq$, $<$, $>$.

1.3 Especificació d'algorismes

Definició 1 (Especificació d'un algorisme)

Especificar un algorisme vol dir fer explícit amb tot rigor i detall:

1. *Quines variables i de quin tipus intervenen a l'algorisme (fer la declaració de les variables de l'algorisme).*
2. *Estat inicial (Enunciats que expressen propietats que se suposen certes abans de l'execució de l'algorisme i que són necessàries per a que l'algorisme arribi al seu resultat). A l'estat inicial se l'anomena precondició (P)*
3. *Nom de l'algorisme.*
4. *Estat final (enunciats que s'hauran de complir després de l'execució de l'algorisme). A l'estat final se l'anomena postcondició (Q).*

Definició 2 (Correctesa d'un algorisme)

Direm que un algorisme A és correcte si qualsevol execució que comença complint la precondició de A, acaba complint la postcondició de A.

Notem que d'aquesta definició es dedueix que no fem cap afirmació sobre aquelles execucions que comencen en un estat que no compleix la precondició de A.

1.4 Exemples

En aquest apartat proposem alguns exemples d'especificació d'alguns algorismes senzills. Al capítol 2 ens dedicarem a veure *com* podem dissenyar algorismes que assoleixin aquestes especificacions.

1.4.1 Exemples senzills

1. Acció nul.la.

   x : enter
   $\{x = X\}$
   nul.la
   $\{x = X\}$

   Representem per X al valor que té la variable x inicialment.

   Es tracta de dissenyar una acció tal que arribi a un estat final en el qual el valor de la variable x sigui X, o sigui, exactament el mateix que tenia a l'inici.

   Queda clar que, en aquest cas, l'algorisme que hauríem de dissenyar de tal manera que complís l'especificació fóra molt senzill: Es tractaria de l'algorisme nul. No caldria fer res ja que la postcondició ja es compleix abans de l'inici de l'algorisme.

   Aquesta reflexió ens porta al primer algorisme correctament verificat: l'algorisme nul (que notarem per $\emptyset$ o *no_op*):

   $\{P\}\emptyset\{Q\}$ correcte sii $P \Longrightarrow Q$.

2. Intercanvi.

   x, y:enter
   $\{x = X \wedge y = Y\}$

intercanvi
$\{x = Y \wedge y = X\}$

Volem dissenyar un algorisme que intercanviï els valors de les variables x i y.

3. Còpia

   x, y:enter
   $\{x = X\}$
   còpia
   $\{x = X \wedge y = X\}$

   En aquest cas, volem un algorisme que posi a la variable y una còpia del valor de la variable x. Notem que no ens importa el valor inicial que pogués tenir la variable y.

4. Arrel.

   x, y:enter
   $\{x > 0\}$
   arrel
   $\{y = \sqrt{x}\}$

   Aquesta especificació presenta, però, algun problema: els x que no són quadrats perfectes no la poden satisfer ja que $\sqrt{}(x)$ no és cap valor enter. De fet, no hi ha cap algorisme que satisfaci l'especificació. Seria més apropiada la següent:

   x, y:enter
   $\{x = X^2\}$
   arrel
   $\{y = X\}$

Fins ara hem escrit les especificacions dels algorismes però no hem entrat en com s'escriurien els algorismes que complissin aquelles especificacions. Tot seguit ens encarregarem d'això. Per a escriure un algorisme necessitem un llenguatge. Aquest llenguatge estarà format bàsicament per quatre maons molt determinats, la combinació dels quals donarà lloc als algorismes: *l'assignació, la composició seqüencial, els condicionals i les iteracions.*

1.4.2 Altres exemples

- Igualtat entre vectors

  Especificar un algorisme tal que, donats dos vectors de naturals v_1 i v_2 de capacitat N i inicialitzats entre els seus índexos 1..N, determini si són iguals (i.e. si contenen exactament els mateixos valors).

  v_1, v_2: vector<natural>(N)
  ig: booleà

  $\{v_1[1..N] = V_1[1..N] \wedge v_2[1..N] = V_2[1..N]\}$
  iguals
  $\{ig = (\forall i : 1 \leq i \leq N \bullet v_1[i] = v_2[i])\}$

Capítol 2

Disseny d'algorismes iteratius

Al capítol anterior hem presentat alguns elements del llenguatge algorísmic (constants, variables, tipus...) i hem descrit què vol dir especificar un algorisme. En essència, ara sabem que especificar un algorisme vol dir indicar amb tot rigor i detall **què** esperem que faci un algorisme (postcondició) partint d'un determinat estat inicial (precondició). Ens hem deixat, però, un aspecte fonamental: **com** s'ho pot fer un algorisme per partir d'un estat incial que satisfaci la precondició i arribar a un estat final que compleixi la postcondició.

Aquest és l'objectiu d'aquest capítol: dissenyar algorismes que satisfacin una postcondició. Per això introduirem cinc instruccions diferents que formen part del llenguatge algorísmic i que ens serviran per desenvolupar algorismes. Aquestes cinc instruccions són *l'assignació, la composició seqüencial, la composició alternativa*, *la composició iterativa* i *la crida a accions i funcions.* Descriurem amb detall el seu funcionament i presentarem exemples de disseny d'algorismes a partir d'una especificació.

2.1 L'assignació

2.1.1 Idea intuitiva

L'assignació és una instrucció de la notació algorísmica que consisteix en donar un valor a una variable:

$$x := E$$

on x és una variable, E és una expressió vàlida del tipus de la variable i ':=' és un símbol que es llegeix *pren per valor.*

Exemple

Dissenyar un algorisme S que respecti la següent especificació:

x, y: enter
$\{cert\}$
S
$\{x = 4\}$

Intuïtivament sembla clar el què cal fer. Hem de donar a la variable x el valor 4. Però això és precisament el que fa la instrucció de l'assignació. Per tant, la solució al problema plantejat és $x := 4$.

Exemple 2

Fer un algorisme que compleixi la següent especificació:

x, y: enter

$\{x = X\}$
còpia
$\{y = X\}$

És clar que volem copiar a la variable y el valor emmagatzemat a la variable x, sigui quin sigui aquest valor. De nou, intuïtivament, això ho identifiquem ràpidament amb una assignació de la forma: $y := x$.

Exemple 3

Fer un programa S d'acord amb la següent especificació:

x, y, z: enter
$P = \{x > 0 \wedge y > 0 \wedge senar(x) \wedge senar(y)\}$
S
$Q = \{parell(z) \wedge z > x \wedge z > y\}$

Per resoldre aquest problema podem considerar la propietat ben coneguda que la suma de dos nombres senars dóna un nombre parell. Com, a més a més, x i y són dos nombres positius, queda clar que $x + y$ *és un enter positiu més gran que* x *i que* y *i parell*.

Essent així sembla clar que assignar $x + y$ a z resoldrà el nostre problema.

$S : z := x + y$

Notem altre cop que a la precondició no fem cap hipòtesi sobre el valor de z

2.1.2 Definició formal de l'assignació correcta

Analitzant els exemples anteriors, podem observar que si tenim una assignació de la forma:

$$\{P\}$$
$$z := E$$
$$\{Q\}$$

Qualsevol propietat que volem que z compleixi a la postcondició $\{Q\}$, l'ha de complir E a la precondició $\{P\}$ (1)

Podríem expressar això mateix, d'una altra manera, dient que P *ha de garantir que* E *compleixi totes aquelles propietats que* Q *exigeix que compleixi* z.

Aquesta idea la veiem molt clara en el darrer exemple: Volem que z compleixi les propietats de ser parell i més gran que x i que y. Ens decidim per l'assignació $z := x + y$ perquè la precondició P ens garanteix que $x + y$ serà parell i més gran que x i que y. Aquesta és doncs la definició d'una assignació correcta:

Definició 3

Direm que el següent algorisme especificat

$$\{P\}$$
$$x := E$$
$$\{Q\}$$

on P *és la precondicicó,* Q*, la postcondició,* x *una variable i* E *una expressió vàlida del tipus de la variable és una assignació correcta si i només si* P *garanteix que* E *compleix totes les propietats que* Q *demana a* x.

2.2 La composició seqüencial

2.2.1 Idea intuïtiva

Si només poguéssim utilitzar una assignació per construir algorismes, queda clar que no podríem resoldre la major part dels problemes que ens plantegem. Afegim, doncs, més potència al nostre llenguatge. Una idea bastant natural per a resoldre problemes és *anar per passes*. És a dir: Fer primer una petita acció que ens acosti mínimament a la postcondició i, tot seguit, aprofitar el resultat que hem obtingut amb la primera acció per a fer-ne una altra que ens deixi una mica més aprop de la solució. I així successivament fins a resoldre completament el problema.

És a dir:

$$\{P\}$$
$$S_1;$$
$$S_2;$$
$$(...)$$
$$S_n;$$
$$\{Q\}$$

on cada S_i representa una instrucció. Fins ara, l'única instrucció que coneixem és l'assignació.

A la tècnica que aprofita aquesta idea se l'anomena *composició seqüencial d'instruccions.*

Exemple

Imaginem que un cert llenguatge de programació no disposa de l'operador *producte* ni d'operacions aritmètiques amb més de dos operands. Es tracta de fer un algorisme que es pugui traduir ràpidament a aquest llenguatge de programació i que satisfaci la següent especificació:

x, y:enter
$\{P\} = \{y = Y\}$
S
$\{Q\} = \{x = 3 * Y\}$

Sembla clar que haurem de substituir l'operador producte, del qual no disposem, per un altre del repertori. Aquest operador serà, evidentment, l'operador suma. Intuïtivament el que caldrà fer serà:

$x := y + y + y$

Com només disposem d'operacions binàries ho haurem de fer en dues tongades:

Si fem en primer lloc:

$x := y + y$

aconseguim avançar una mica vers la solució final ja que acumulem a x el valor $2 * Y$. Però això encara no és suficient. A x hi hem d'acumular encara una altra vegada el valor de y. Per tant, podem fer:

$x := x + y$

Intuïtivament en aquest punt $x = 3 * Y$.

Per tant, la solució que proposem és la següent:

$\{P\}$
$x := y + y;$ (1)
$\{x = 2 * Y\}$
$x := x + y;$ (2)

$\{x = 3 * Y\}$
$\{Q\}$

Una manera equivalent d'expressar el que acabem de dir és la següent:

L'estat al que s'arriba després d'executar (1) ($\{x = 2 * Y\}$) haurà de ser precondició suficient per a que, un cop executada (2), s'assoleixi la postcondició Q ($\{x = 3 * Y\}$).

Exemple 2

Especificar i resoldre el problema de l'intercanvi:

x, y:enter
$\{P\} = \{x = X \wedge y = Y\}$
S
$\{Q\} = \{x = Y \wedge y = X\}$

Una primera temptativa per a resoldre el problema podria ser la següent:

x:=y;
y:=x;
(1)

Però aquesta aproximació té el problema que es perd el valor de x quan se li assigna el valor de y. Efectivament, la postcondició que es compleix a (1) és la següent:

(1): $\{x = Y \wedge y = Y\}$

La idea bona serà guardar el valor de x en una variable temporal:

$x, y, temp$:enter
$\{P\} = \{x = X \wedge y = Y\}$
$\{Q_3\}$
temp:=x; /*guardem X per poder-lo assignar després a y*/
$\{Q_2\}$
x:=y; /* Assignem a x el valor Y */
$\{Q_1\}$
y:=temp; /*Assignem a y el valor X*/
$\{Q\} = \{x = Y \wedge y = X\}$

Intuïtivament $Q_2 = \{y = Y \wedge x = X \wedge temp = X\}$ i $Q_1 = \{y = Y \wedge x = Y \wedge temp = X\}$.

2.2.2 Definició formal de la composició seqüencial correcta

Definició 4

Direm que
$\{P\}$
S_1;
$\{Q_1\}$
S_2
$\{Q\}$
on P és la precondició,
Q és la postcondició i
S_1 i S_2 són composicions seqüencials,

és una composició seqüencial correcta sii
$\exists$ *un enunciat Q_1 tal que $\{P\}$ $S_1\{Q_1\}$ és un algorisme correcte i $\{Q_1\}$ S_2 $\{Q\}$ també ho és.*

2.3 La composició alternativa

2.3.1 Idea intuïtiva

Amb les eines que hem donat fins ara, la seqüència d'instruccions executada era sempre la mateixa. La composició aternativa ens permetrà executar una seqüència d'instruccions o una altra depenent d'unes certes condicions B_i. La notació que emprarem és la següent:

{P}
si
$B_1 \longrightarrow S_1$
$B_2 \longrightarrow S_2$
(...)
$B_n \longrightarrow S_n$
fsi
{Q}

On B_i $(i = 1..n)$ són expressions booleanes i S_i $(i = 1..n)$ seqüències d'instruccions.

La seqüència S_i es podrà executar només en el cas en que B_i avaluï cert. A cada execució d'una composició alternativa, només s'executarà una seqüència d'instruccions S_i. Les lleis que guien l'execució d'una composició alternativa com l'anterior són les següents:

1. Quan s'entra a la composició alternativa, almenys alguna de les guardes B_i $(i = 1..n)$ avalua cert $(P \Longrightarrow B_i$, per alguna $i = 1..n)$. Si hi ha dues (o més) guardes certes (B_i i B_j) no està definit quina de les seqüències d'instruccions s'executarà: S_i o S_j.

2. La seqüència d'instruccions S_i que s'executi ha de permetre arribar a la postcondició Q. Això és, l'algorisme

   $\{P \wedge B_i\}$
   S_i
   $\{Q\}$

   ha de ser correcte.

Exemple: El màxim

Presentem un algorisme per a obtenir el màxim dels valors emmagatzemats a dues variables enteres x, y:

$\{P\} = \{x = X \wedge y = Y\}$
si
$x \leq y \longrightarrow max := y;$
$x > y \longrightarrow max := x;$
fsi
$\{Q\} = \{max = maxim(X, Y)\}$

A l'apartat següent formalitzem la idea de la composició alternativa.

2.3.2 Definició formal d'una composició alternativa correcta

Definició 5

Sigui S un algorisme amb la forma següent:
$\{P\}$
si
$B_1 \longrightarrow S_1$

$B_2 \longrightarrow S_2$
(...)
$B_n \longrightarrow S_n$
fsi
$\{Q\}$

on $B_1, ..., B_n$ són expressions booleanes que anomenem condicions o guardes i $S_1, .., S_n$ són seqüències d'instruccions.

A aquest algorisme l'anomenarem composició alternativa correcta si es compleixen les següents condicions:

1. $P \Longrightarrow B_1 \vee B_2 \vee ... \vee B_n$
2. $\forall i : 1 \leq i \leq n \bullet \{P \wedge B_i\} S_i \{Q\}$

La primera propietat estableix que abans de l'execució d'una composició alternativa, almenys una de les guardes B_i ha de resultar certa. La segona indica que la precondició de la composició alternativa (P) conjuntament amb una de les guardes que són certes (B_i) han de ser suficients per a garantir la postcondició després de l'execució de l'acció S_i.

Exemple:

Per verificar l'algorisme del màxim cal comprovar tres propietats:

1. $P \Longrightarrow x \leq y \vee x > y$
2. $\{P \wedge x \leq y\} max := y \{Q\}$ és un algorisme correcte.
3. $\{P \wedge x > y\} max := x \{Q\}$ és un algorisme correcte.

Les tres comprovacions són trivials i les deixem com a exercici.

Exemple 2:

Dissenyar i verificar un algorisme S que satisfaci la següent especificació:

x, y, z, p, m, g: enter
$P = \{x = X \wedge y = Y \wedge z = Z\}$
S
$Q = \{(p, m, g) = permutacio(X, Y, Z) \wedge p \leq m \leq g\}$

O sigui, la idea és col.locar a p el menor entre els valors X, Y i Z, a m el mitjà i a g el gran.

Per tal de resoldre el problema seguirem les següents passes:

1. Trobar p i m tals que $p \leq m$ comparant els valors de dues variables qualsevols (exemple: x i y). Fent això obtindrem el resultat intuïtiu següent:
   $R_1 = \{P \wedge (p, m) = permutacio(X, Y) \wedge p \leq m\}$
2. Trobar el p definitiu, o sigui, aquell que compleix: $p \leq m \wedge p \leq g$.
   $R_2 = \{P \wedge (p, m, g) = permutacio(X, Y, Z) \wedge p \leq m \wedge p \leq g\}$
3. Ordenar, finalment, els valors relatius de m i g.
   $Q = \{(p, m, g) = permutacio(X, Y, Z) \wedge p \leq m \leq g\}$

Amb aquestes consideracions serà més senzill fer el disseny de l'algorisme ja que només haurem de pensar com satisfer les tres especificacions parcials. Fem-ho:

$\{P\}$

si

$y < x \longrightarrow p := y; m := x;$
$y \geq x \longrightarrow p := x; m := y;$

fsi

$\{R_1\}$ [1]

si

$p > z \longrightarrow g := m; m := p; p := z;$
$p \leq z \longrightarrow g := z;$

fsi

$\{R_2\}$ [2]

si

$m > g \longrightarrow temp := m; m := g; g := temp;$
$m \leq g \longrightarrow$ no_op;

fsi

$\{Q\}$ [3]

Per tal de verificar l'algorisme, comprovarem la correctesa dels tres subalgorismes per separat, o sigui veurem si són certes les següents propietats:

1. Que l'acompliment de R_2 a [2] ens garanteix l'acompliment de Q a [3].
2. Que l'acompliment de R_1 a [1] ens garanteix l'acompliment de R_2 a [2].
3. Que l'acompliment de P a l'inici és suficient per a garantir R_1 a [1].

Si 1, 2 i 3 són certs, aleshores és clar que la precondició P és suficient per a garantir el resultat Q.

Fem doncs les tres verificacions parcials per separat (òbviament, l'ordre en què es facin les tres comprovacions és del tot irrellevant):

1. $\{R_2\}S_3\{Q\}$
   $R_2 = \{P \wedge (p, m, g) = permutacio(X, Y, Z) \wedge p \leq m \wedge p \leq g\}$
   si
   $m > g \longrightarrow temp := m; m := g; g := temp;$
   $m \leq g \longrightarrow$ no_op;
   fsi
   $Q = \{(p, m, g) = permutacio(X, Y, Z) \wedge p \leq m \leq g\}$
   Per verificar aquest condicional, cal comprovar:
   (a) $R_2 \Longrightarrow m > g \vee m \leq g$
   La qual cosa és evident.
   (b) $\{R_2 \wedge m > g\}temp := m; m := g; g := temp; \{Q\}$
   Cal veure que les tres assignacions ens permeten obtenir els dos conjuntants de Q ($(p, m, g) = permutacio(X, Y, Z) \wedge p \leq m \leq g$).
   R_2 ens garanteix que $p \leq m$ i $p \leq g$. Com, a més, $g < m$, si intercanviem els valors de g i m, obtenim directament la postcondició Q.
   (c) $\{R_2 \wedge m \leq g\}no\_op\{Q\}$
   Trivial.

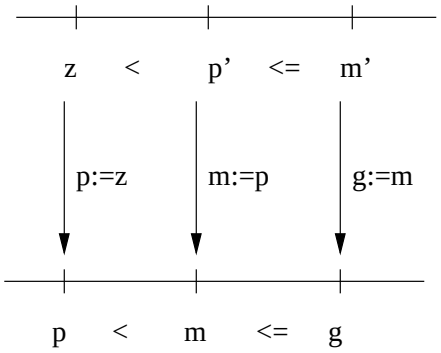


Figura 2.1: Anàlisi d'un dels casos de l'algorisme d'ordenació de tres valors

2. $\{R_1\}S_2\{R_2\}$

   $R_1 = \{P \wedge (p,m) = permutacio(X,Y) \wedge p \leq m\}$

   si

   $p > z \longrightarrow g := m; m := p; p := z;$
   $p \leq z \longrightarrow g := z;$

   fsi

   $R_2 = \{P \wedge (p,m,g) = permutacio(X,Y,Z) \wedge p \leq m \wedge p \leq g\}$

   Com en el cas anterior, caldrà veure:

   (a) $R_1 \Longrightarrow p > z \vee p \leq z$
   Com abans és evident.

   (b) $\{R_1 \wedge p > z\}g := m; m := p; p := z\{R_2\}$
   Per a verificar això, caldrà assegurar que, partint de $\{R_1 \wedge p > z\}$ i després de les assignacions, obtenim R_2.

   - Certament $(p,m,g) = permutacio(X,Y,Z)$ ja que els valors (m, p) que hem col.locat amb les assignacions a g i m eren una permutació de X i Y (això ho garanteix R_1) i, d'altra banda, $p = Z$.
   - Per veure que $p \leq m \wedge p \leq g$, anomenem p' i m' a les variables p i m abans de les assignacions. Com estem en el cas $\{R_1 \wedge p > z\}$ estarem en la situació indicada a la part superior de la figura 2.1.
     Per tant, és cert que després de les assignacions tenim $p \leq m$ i $p \leq g$.

   (c) $\{R_1 \wedge p \leq z\}\ g := z\ \{R_2\}$

   - Novament $(p,m,g) = permutacio(X,Y,Z)$ perquè $(p,m) = PERM(X,Y)$ i $g = Z$ després de l'assignació $g := z$;
   - Finalment es té $p \leq m$ i $p \leq g$ perquè $p \leq m$ ja es complia a R_1 i, d'altra banda sabem que $p \leq z$ i $g = z$ després de l'assignació $g := z$.

   Procedint d'igual manera, verificarem el darrer dels condicionals:

3. $\{P\}S_1\{R_1\}$

   $P = \{x = X \wedge y = Y \wedge z = Z\}$

   si
   $y < x \longrightarrow p := y; m := x;$
   $y \geq x \longrightarrow p := x; m := y;$

fsi

$R_1 = \{P \wedge (p, m) = permutacio(X, Y) \wedge p \leq m\}$

En aquest cas caldria veure:

- $P \Longrightarrow y < x \vee y \geq x$
- $\{P \wedge y < x\} p := y; m := x \{R_1\}$
- $\{P \wedge y \geq x\} p := x; m := y \{R_1\}$

La verificació d'aquestes tres condicions no planteja cap problema i la deixem com a exercici.

2.4 La composició iterativa

2.4.1 Idea intuïtiva

Amb la composició seqüencial, podem repetir una instrucció un nombre fix de vegades. Malauradament, això no és suficient. Per la major part dels problemes necessitem repetir una instrucció (o un bloc d'instruccions) un nombre indeterminat de vegades. Aquest nombre pot canviar a cada nova execució. Per exemple, si considerem l'algorisme d'obtenir el quadrat dels n primers nombres naturals, haurem de repetir l'acció de *trobar el quadrat d'un nombre* n vegades. Aquesta n pot ser diferent en cada nova execució de l'algorisme.

La composició iterativa (que, habitualment anomenarem bucle) és una nova instrucció del llenguatge que permetrà repetir l'execució d'una seqüència d'instruccions un nombre variable de vegades.

Una composició iterativa la notarem de la següent manera:

mentre B **fer**
  S
fmentre

A B se l'anomena *condició de continuació* i a S, *cos de la composició iterativa.*

El significat intuitiu d'un bucle com l'anterior és el següent:

Executar el bloc d'instruccions S mentre la condició B sigui certa.

Més específicament, el seu funcionament el podem precisar de la següent manera:

1. Avaluar la condició B.
2. Si B resulta certa,
   (a) Executar el bloc S
   (b) Recomençar el procés (tornar a 1).
3. Si B resulta falsa, acabar l'execució de la composició iterativa.

Tot seguit, presentem un exemple que utilitza una composició iterativa.

Exemple

Volem fer un algorisme que satisfaci aquesta especificació:

$P = \{x > 0 \wedge y > 0\}$
producte
$Q = \{prod = x * y\}$

en una màquina que no ens permet utilitzar l'operador producte.

Aquesta especificació és un cas clar en el qual caldrà utilitzar composició iterativa atès que considerarem el producte de dos naturals com un nombre variable de sumes (sumar y vegades el valor x). Òbviament, y i x seran diferents a cada nova execució.

Intuïtivament, podem traduir la idea de sumar y vegades el valor de x en termes d'acumular en una certa variable *prod* x unitats més cada volta i això fer-ho durant y voltes.

Un algorisme S que compleixi l'especificació demanada amb les idees presentades és el següent:

$\{P\}$
$prod := 0;\ \ j := 0;$
mentre $j < y$ **fer**
  $prod := prod + x;$
  $j := j + 1;$
fmentre
$\{Q\}$

Si estudiem el cos del bucle de l'exemple ens adonarem de, com a mínim, tres coses:

1. A cada volta s'avança vers la postcondició. (A cada volta, *prod* està més aprop de contenir el valor final de $x * y$). La instrucció $prod := prod + x;$ és l'encarregada de que això sigui possible.
2. A cada volta s'avança una mica més vers la condició de final del bucle. (A cada volta j està més aprop de y). La instrucció $j := j + 1$ s'encarrega d'això.
3. Quan s'arriba al final de la darrera volta, $y = j$ i $prod = x * y$ (les dues coses ens asseguren la postcondició Q).

A partir d'aquestes observacions, podem introduir dos elements necessaris per tal de garantir la correctesa d'un bucle: *l'invariant de la composició iterativa* i la *fita de la mateixa*.

L'invariant de la composició iterativa

prod no passa des del seu valor inicial (0) fins el seu valor final ($x*y$) de cop i volta, com per art de màgia. Ben al contari, el camí que porta des de 0 fins a $x * y$ segueix una estratègia perfectament regulada pel bucle que a la fi li permet d'arribar a la seva postcondició.

L'invariant intenta desentranyar aquesta estratègia, fer-la explícita.

Com a exemple, cerquem l'invariant de l'algorisme del producte. Proposem, per això el cas particular del producte entre $x = 5$ i $y = 4$.

La següent taula representa els valors de les variables j i *prod a l'inici de cada volta*. La filera i, per tant, es referirà als valors de les variables just abans de començar la volta i.

núm volta	j	*prod*
1	0	0
2	1	5
3	2	10
4	3	15
5	4	20

La volta 5 mai no s'arribarà a fer perquè $j = y$ i B esdevindrà falsa en aquest punt.

Cadascuna de les fileres correspon a una fotografia de l'estat del bucle en un moment donat (a l'inici d'una volta). Per poc observadors que siguem, ens haurem d'adonar que totes quatre fotografies segueixen un mateix patró que determina l'estratègia que segueix la composició iterativa per a arribar a la seva postcondició. Aquesta estratègia ve donada per les següents dues propietats:

- A l'inici i a la fi de cada volta, $prod = x * j$ ($prod$ val el nombre de voltes que el bucle ha completat multiplicat per x).
- A l'inici i a la fi de cada volta, $j \leq y$ (el número de voltes que hem completat és, en tot moment menor o igual que y). Queda clar que quan haguem completat y voltes, d'aquestes dues propietats es desprén que $prod = x * y$ (justament el resultat al que volíem arribar. Aquesta és, doncs, l'essència del bucle.

A l'enunciat que recull aquelles propietats relatives a les variables que intervenen a la composició iterativa que són certes a l'inici i a la fi *de cada volta* i que expressen l'estratègia que segueix el bucle per tal d'*avançar* vers la seva postcondició se l'anomena *invariant de la composició iterativa*.

A l'exemple que ens ocupa, l'invariant del bucle fóra:

$I = \{prod = x * j \wedge 0 \leq j \leq y\}$.

El nom d'invariant li ve del fet que es tracta d'un enunciat cert a l'inici i a la fi de cada volta. Per tant, és invariant respecte les instruccions que composen el cos del bucle (aquestes instruccions no modifiquen la seva certesa). A la secció ??? utilitzarem aquesta idea per a veure que típicament els cossos dels bucles es composen de dos tipus d'instruccions íntimament relacionats amb el concepte d'invariant:

- Les instruccions que fan avançar el bucle vers el seu final ($j := j + 1;$) i
- Les instruccions que *restauren* la certesa de l'invariant trencada per les instruccions d'avenç ($prod := prod + x;$).

La fita del bucle

Aquesta és una funció a valors enters que té les propietats de fer-se petita a cada volta del bucle i de ser sempre més gran que 0 mentre es compleixi B. En conseqüència, serà una fita superior del nombre de voltes que ens queden per fer abans de sortir del bucle. Així doncs, si trobem una fita per un bucle, haurem garantit que aquell acabarà en algun moment.

A l'exemple que hem presentat, la fita pot ser: $y - j$ (notem que $B \Longrightarrow y - j > 0$).

2.4.2 Definició formal

A l'apartat anterior hem presentat intuitivament els dos conceptes que serveixen per a caracteritzar un bucle: l'invariant i la fita. També hem raonat quines propietats han de complir aquests dos conceptes. El que cal ara és sintetitzar-ho tot plegat. O sigui, escriure una llista formal de propietats que garanteixin que una composició iterativa sigui correcta. Aquesta llista serà de gran ajut, doncs quan vulguem saber si un bucle és correct, només caldrà obtenir el seu invariant, la seva fita i comprovar si compleixen aquelles propietats.

Escrivim-ho tot plegat:

Definició 6

Direm que

$\{P\}$
mentre B **fer** S **fmentre**
$\{Q\}$.

és una composició iterativa correcta amb condició de continuació B, cos S i postcondició Q sii existeix un enunciat I que anomenarem invariant de la composició iterativa i una funció (o fita) t a valors enters tal que acompleixin les següents condicions:

1. $P \Longrightarrow I$
2. $\{B \wedge I\}S\{I\}$
3. $I \wedge \neg B \Longrightarrow Q$
4. $I \wedge B \Longrightarrow t > 0$
5. $\{I \wedge B \wedge t = T\}S\{t < T\}$

Tot seguit comentem el significat d'aquestes propietats.

1. $P \Longrightarrow I$

   L'invariant s'ha de complir en el moment d'entrar a la primera volta del bucle. Dit d'una altra manera, la precondició del bucle (allò que sabem que es compleix abans de començar la seva execució) ha de garantir l'invariant.

   Recordem que hem justificat que l'invariant s'ha de complir a l'inici de cada volta del bucle. El que diem ara és que, en particular, s'ha de complir a l'inici de la primera.

2. $\{B \wedge I\}S\{I\}$

   Les instruccions del cos del bucle no alteren la certesa de I.

   A l'inici de cada volta es compleix $I \wedge B$. Quan s'acabi l'execució de S, I es continuarà complint.

3. $I \wedge \neg B \Longrightarrow Q$

   Si el bucle acaba, ho farà correctament (complint la postcondició Q del bucle).

   Quan acaba el bucle sabem que es compleix I i també $\neg B$ (ja que $\neg B$ és precisament la seva condició de sortida). Aquests dos enunciats han de garantir la postcondició Q.

   Notem que tot el que hem fet fins ara no justifica l'acabament del bucle (encara no hem parlat de la fita). L'únic que estem dient amb aquestes tres primeres propietats és que si el bucle acaba, ho farà correctament.

   Una composició iterativa que compleixi aquestes tres propietats direm que és *parcialment correcta*.

   Per a garantir l'acabament del bucle, necessitarem encara les dues propietats següents:

4. $I \wedge B \Longrightarrow t > 0$

   Sempre que estem dins d'una iteració , la seva fita és més gran que 0.

5. $\{I \wedge B \wedge t = T\}S\{t < T\}$

   La fita es decrementa estrictament a cada volta.

   Si a l'inici d'una volta la fita val T, quan acaba la mateixa, la fita s'ha d'haver decrementat.

Exemple: Verificació de l'algorisme del producte

En aquest exemple demostrarem que, efectivament, l'algorisme del producte que hem proposat més amunt és correcte. Per a fer això, veurem que I i t compleixen les 5 propietats enuncides.

1. $P_1 \Longrightarrow I$.

   On P_1 és l'enunciat que es compleix immediatament abans d'entrar al bucle. O sigui:

   $P_1 = \{x > 0 \wedge y > 0 \wedge prod = 0 \wedge j = 0\}$

   Si substituim a l'invariant *prod* per 0 i j per 0 i, a més, considerem que $y > 0$, obtindrem el següent enunciat que serà cert:

   $\{0 = x * 0 \wedge 0 \leq 0 \leq y\}$.

2. $\{B \wedge I\}S\{I\}$

   La comprovació d'aquesta propietat ens demana de verificar el tros d'algorisme següent:

   $I \wedge B = \{prod = x * j \wedge 0 \leq j < y\}$.
   (1)
   $prod := prod + x$;
   $j := j + 1$;
   $I = \{prod = x * j \wedge 0 \leq j \leq y\}$.

   Per tal d'assegurar la correctesa d'aquest tros d'algorisme hem de comprovar que a la fi de cada volta:

   - $prod = x * j$.
   - $0 \leq j \leq y$

   Per fer-ho, anomenarem $prod'$ i j' als valors de les variables $prod$ i j a l'inici de la volta (1).

   Al final de la volta: $prod = prod' + x$, això és:
   $prod = x * j' + x$ (ja que I ens garanteix que $prod' = x * j'$).
   $prod = x * (j' + 1)$. Però com $j = j' + 1$, tenim que:
   $prod = x * j$.

   D'altra banda, com $B \Longrightarrow j' < y$, podrem assegurar que $j' + 1 \leq y$, o, equivalentment, que $j \leq y$.

3. $I \wedge \neg B \Longrightarrow Q$

   $I \wedge \neg B = \{prod = x * j \wedge 0 \leq j \leq y \wedge j \geq y\} =$
   $\{prod = x * j \wedge 0 \leq j \wedge j = y\} \Longrightarrow$.
   $\{prod = x * y\} = Q$

4. $I \wedge B \Longrightarrow t > 0$

   La pròpia B implica que $y - j > 0$.

5. $\{I \wedge B \wedge t = T\}S\{t < T\}$

   Immediat del fet que j s'incrementa a cada volta i y no varia.

2.4.3 Idees per dissenyar bucles

Com ens ho fem per tal de dissenyar un bucle? Una primera resposata és fer-ho intuïtivament, sense cap mètode, sense encomanar-nos ni a Déu ni al Diable: tal com ho hauríem fet abans de llegir aquestes notes. El problema de fer-ho així ja us el podeu imaginar: com podem raonar que un bucle és correcte? L'experiència demostra que treballar sense mètode porta a algorismes fràgils, sovint incorrectes. A bucles que en algun moment peten o que, en tot cas, mai no estem prou segurs que funcionin bé en tots els casos.

A la secciá anterior hem presentat dues eines molt poderoses per raonar respecte la correctesa d'un bucle: *l'invariant (I) i la fita (t)*. De fet, tots dos ens donen tanta informació respecte el bucle al qual es refereixen que ens permeten caracteritzar-lo. Recordem per què:

- L'invariant ens diu de quina manera cada volta del bucle serveix per a acostar-nos a Q (A cada volta $prod$ conté el producte de x per j, amb una j que s'anirà apropant a y (a mesura que decrementa la fita t). Quan $j = y$ haurem arribat a Q. Recordem que j comptava en cada moment el nombre de voltes que havíem completat).
- L'invariant ens marca clarament què és allò que es compleix a l'entrada d'una volta i què cal que es compleixi a la sortida. Per tant, ens indica com haurem de construir el cos del bucle: aquest haurà de tenir una part amb l'objectiu d'avançar vers el final, decrementant la fita t ($j := j + 1$) i una altra part que restableixi l'invariant que es destrueix en avançar ($prod := prod + x$) amb la finalitat que aquest es compleixi novament al final de la volta.

- L'invariant ens suggereix amb quins valors hem d'inicialitzar les variables que formen part del bucle per a que l'invariant es compleixi a l'entrada de la primera volta ($prod := 0; j := 0$).
- L'invariant permet trobar fàcilment la guarda del bucle (intuïtivament, podríem dir que la negació de la guarda del bucle és allò que li manca a l'invariant per a assolir la postcondició (recordem la propietat $I \wedge \neg B \Longrightarrow Q$). En el cas de l'acció *producte*, $B = \{j \neq y\}$.

D'aquestes reflexions es dedueix que si som capaços de determinar l'invariant i, més tard, la fita del bucle, estarem en molt bones condicions per tal de poder dissenyar la totalitat d'un bucle de manera correcta. De fet, no exagerem ni una mica si diem que l'invariant és la clau que ens permet obtenir la resta dels elements del bucle i, a més, de forma correcta.

Per això us proposo que prengueu com a màxima de programació la següent: *Cada cop que hagueu de dissenyar un algorisme iteratiu, penseu amb l'invariant des de bon començament. Si podeu, determineu l'invariant abans de començar el disseny del bucle. Si no us en sortiu, dissenyeu el bucle al mateix temps que l'invariant.*

Un cop hem justificat la importància de que la cerca de l'invariant sigui una de les primeres coses que cal plantejar-se en el disseny d'un algorisme iteratiu, apareixen dues qüestions més:

1. Com ho fem per trobar l'invariant?
2. I quan tenim l'invariant, com obtenim la resta dels elements del bucle?

De bon cap i principi ja hem d'advertir que no hi ha respostes universals per cap de les dues (cal fer molta pràctica). De tota manera, en les properes seccions intentem donar algunes idees per contestar-les.

Obtenció de l'invariant

Per a obtenir l'invariant ens basarem en el fet que aquest ha de ser relativament semblant a la postcondició (recordem que si a I li afegim $\neg B$ ja obtenim Q). En particular, ha de ser més feble que Q (o, si voleu, Q ha de ser més estricta que I). Efectivament, si I fos tan estricte com Q, aquesta ja estaria establerta des de l'inici i, per tant, no caldria fer cap mena d'iteració ni res semblant.

Com a exemple podem tornar a considerar l'acció *producte*. Q estableix que $prod = x * y$ però I no és, ni de bon tros, tan exigent. I manté la forma de Q ($prod = x * j$) però, seguidament, permet que j variï entre 0 i y.

Per tant, obtindrem l'invariant d'un bucle *afeblint la seva postcondició*.

Utilitzarem dos mètodes per a afeblir la postcondició:

1. Eliminar un conjuntant de la postcondició.

   Posem com a exemple l'acció que troba el màxim comú divisor de dos naturals utilitzant l'algorisme d'Euclides.

   Siguin a i b dues variables de tipus natural i A i B dos valors naturals més grans que 0.

   $\{P\} = \{a = A \wedge b = B \wedge A > 0 \wedge B > 0\}$
   maxcomdiv(a,b);
   $\{Q\} = \{a = mcd(A, B)\}$

   Q estableix que la variable a guarda, al final de l'acció *maxcomdiv*, el màxim comú divisor (mcd) dels valors A i B.

   L'Algorisme d'Euclides es basa en la següent propietat:

   $$mcd(a,b) = \begin{cases} mcd(a-b,b) & si \text{ a>b} \\ mcd(a,b-a) & si \text{ b>a} \\ a & si \text{ a=b} \end{cases}$$

Si l'apliquem per a calcular el màxim comú divisor de 6 i 10, obtindrem el següent:

$mcd(10,6) = mcd(4,6) = mcd(4,2) = mcd(2,2) = 2$

Intuitivament, aquest algorisme manté en tot moment *invariant* el màxim comú divisor dels naturals amb els que tracta (les parelles $< 10,6 >$, $< 4,6 >$, $< 4,2 >$, $< 2,2 >$ tenen totes el mateix màxim comú divisor), al mateix temps que tracta, a cada pas, de fer-los *acostar* l'un a l'altre. Aquesta idea ens suggereix de forma natural l'invariant del bucle: Si anomenem a i b les variables de les quals mantindrem intacte el seu màxim comú divisor, podem proposar com a invariant:

$I = \{mcd(a,b) = mcd(A,B)\}$

Aquest invariant, que hem obtingut intuitivament, es pot obtenir també (encara que d'una forma un xic més artificial, ho he de reconèixer) eliminant un conjuntant de la postcondició.

Efectivament, podem reescriure Q de la següent manera:

$\{Q\} = \{mcd(a,b) = mcd(A,B) \wedge a = b\}$

Això és cert ja que el mcd de a i b val a si $a = b$.

Ara eliminem el conjuntant $a = b$ de Q i obtenim una asserció més feble que és precisament l'invariant:

$\{I\} = \{mcd(a,b) = mcd(A,B)\}$

Com s'ha de complir que $I \wedge \neg B \Longrightarrow Q$, el conjuntant que hem eliminat és un bon candidat a fer de $\neg B$.

Com $mcd(A,B)$ és un valor fix perquè A i B són valors fixs, I ens diu que el valor del $mcd(a,b)$ no variarà en el transcurs de les successives iteracions. Per tant, el propòsit de cada volta serà el d'apropar la a a la b (per tal d'apropar-nos al final del bucle), mantenint fix el seu *mcd*. L'algorisme d'Euclides presentat més amunt ens diu precisament com podem mantenir fix aquest mcd.

$\{P\}$
mentre $(a \neq b)$ **fer**
  si $a < b \longrightarrow b := b - a;$
    $a > b \longrightarrow a := a - b;$
  fsi
$\{Q\}$

Notem finalment que l'acabament està assegurat amb una fita tal com $t = a + b$ que assegura el seu decreixement a cada volta.

2. Substituir una expressió de la postcondició per una variable *nova* amb un rang de valors concret i especificat.

   Substituint una expressió que té un valor fix i determinat per una variable amb un rang de valors associat, relaxem la postcondició, la fem menys exigent.

   Aquesta és la forma més habitual d'afeblir la postcondició per tal de trobar l'invariant i és la forma que hem utilitzat per a obtenir l'invariant de l'acció *producte*. En aquesta acció, la postcondició ens demanava que $prod = x * y$ on x i y eren valors fixos. La idea ha estat canviar l'expressió y per j on j és una variable que es pot moure dins del rang $0..y$ (sempre que introduïm una nova variable, cal afitar-la).

   A partir de la secció 2.4.4 proposem m'es exemples on s'utilitza aquesta forma d'afebliment de la postcondició.

Obtenció de la resta d'elements

Un cop hem proposat un invariant, ja tenim les eines necessàries per obtenir raonadament la resta dels elements del bucle. Recordem que el bucle que volem dissenyar tindrà la forma següent:

{P}
inicialització
mentre B **fer**
  S;
fmentre
{Q}

Els raonaments que ens permetran anar obtenint tots aquests elements a partir de l'invariant són els següents:

1. Deducció de $\neg B$. Ho farem a partir de la propietat $I \wedge \neg B \Longrightarrow Q$.

   Intuitivament $\neg B$ és allò que li manca a I per a tenir Q.

2. Obtenció de les *inicialitzacions*, raonant que seran la seqüència d'instruccions que permetran que I es compleixi immediatament abans de fer la primera volta.

3. Obtenció de la fita del bucle. A partir de I i B. $\neg B$ ens dóna la condició de sortida del bucle i I ens indica el rang de les variables que intervenen a B. A més a més, les inicialitzacions ens donaran el valor de partida. Amb aquesta informació, podrem deduir la fita superior del nombre de voltes que queden per acabar la composició iterativa. Una bona manera per a deduir la fita és preguntar-se *què és allò que cal que es faci petit en cada volta*.En el cas de l'algorisme del producte era la diferència entre y i j.

4. Obtenció del cos del bucle S. El cos del bucle tindrà la següent forma:

   mentre B **fer**
     $\{I \wedge B\}$
   reestablir;
   avançar;
     $\{I\}$
   fmentre

   on *avançar* significa decrementar la fita del bucle (avançar vers la fi del mateix) i *reestablir* vol dir fer les accions necessàries que ens garanteixin que al final de la volta obtindrem novament I.

   A l'acció *producte*, $j := j + 1$ feia el paper *d'avançar*, mentre que $prod := prod + x$ era *reestablir*.

   Finalment, és interessant fer notar que, a voltes, no tindrem dues accions separades, una per avançar i una altra per restablir, ni sempre convindrà fer-les en aquest ordre. Però sempre caldrà fer ambdues coses.

2.4.4 Exemples de desenvolupament d'algorismes iteratius

L'algorisme de la divisió entera

Us proposo de desenvolupar un dels algorismes que vam especificar al final del capítol 1: Es tractava d'obtenir el quocient i el residu de la divisió de dos naturals. L'especificació era la següent:

n, d, q, r: natural

P:$\{n \geq 0 \wedge d > 0\}$

divisió_entera;

Q:$\{n = d * q + r \wedge 0 \leq r < d\}$

Així doncs, l'algorisme que us proposo haurà de calcular sobre la variable q el quocient de la divisió entera entre n i d ($q = n$ **div** d) i sobre la variable r el residu de la mateixa divisió entera ($r = n$ **mod** d).

Plantegem-nos, en primer lloc, quin procediment seguirem per tal de calcular q i r: Sembla que una idea prou assenyada és anar provant els successius quocients començant per 0 (0, 1, 2,...) fins a trobar-ne un que generi un residu menor que d.

Exemple: volem dividir $n = 10$ per $d = 3$. Fem les proves successives:

$q = 0 \longrightarrow r = 10$ ($r > 3 \Longrightarrow$ Continuem)
$q = 1 \longrightarrow r = 7$ ($r > 3 \Longrightarrow$ Continuem)
$q = 2 \longrightarrow r = 4$ ($r > 3 \Longrightarrow$ Continuem)
$q = 3 \longrightarrow r = 1$ ($r > 3 \Longrightarrow$ Final: $q = 3$, $r = 1$)

Notem que aquesta estratègia manté invariant en tot moment el primer dels conjuntants de Q (en tot moment $n = d * q + r$) mentre que el segon (específicament $r < d$) és precisament la condició d'acabament del bucle que hem de fer.

Estem doncs, en un típic cas en el qual l'invariant s'obté eliminant un conjuntant de la postcondició (el qual, per cert, passa a ser la condició d'acabament del bucle):

Q:$\{n = d * q + r \wedge 0 \leq r \wedge r < d\}$

Obtenim I eliminant el darrer conjuntant:

I:$\{n = d * q + r \wedge 0 \leq r\ \}$

El darrer conjuntant passa a ser la condició d'acabament del bucle:

$\neg B = (r < d)$ i, per tant: $B = (r \geq d$.

Amb la qual cosa es garanteix que $I \wedge \neg B \Longrightarrow Q$.

Si ens adonem que r haurà d'anar decreixent a cada itearació (al final haurà de ser menor que d) i, a més, haurà de ser sempre més gran que 0 (ho diu Q), sembla una bona candidata a fita del bucle:

$t = r$.

De moment, tenim:

$P = \{n \geq 0 \wedge d > 0\}$

inicialitzacions;
mentre $r \geq d$ **fer**
  S;
fmentre

Q:$\{n = d * q + r \wedge 0 \leq r < d\}$

- Les inicialitzacions hauran d'anar orientades a que I es compleixi a l'inici de la primera volta: Això passarà si $q := 0$ i $r := d$.
- El cos del bucle (S) haurà de seguir l'estratègia que hem anunciat: incrementar q i actualitzar convenientment r:

  $q := q + 1$;
  $r := r - d$;

  La primera instrucció incrementa q, però això trenca l'invariant. Per tal de restablir-lo, i alhora, d'avançar vers la condició d'acabament ($\neg B$), ens cal actualitzar convenientment r. Intuïtivament, si incrementem el quocient en una unitat obtindrem un residu de la divisió d unitats menors, per això fem $r := r - d$. De tota manera, ens cal assegurar que efectivament aquest parell d'instruccions no trenquen I:

  Si anomenem respectivament q' i r' als valors de les dues variables a l'entrada d'una volta del bucle (q i r seran els valors al final de la volta), tindrem que:

  $I \wedge B \Longrightarrow n = q' * d + r' \wedge 0 \leq r' \wedge r' \geq d$

i haurem de demostrar que, al final d'aquella volta es compleix novament I, és a dir:

I:$\{n = d * q + r \wedge 0 \leq r\ \}$

- com $q = q' + 1$ i $r = r' - d$, a l'inici de la volta es complia:
  $n = d * (q - 1) + r + d$. És a dir:
  $n = d * q + r$
- Com $B \Longrightarrow r' \geq d$, tindrem que $r = r' - d \geq 0$

que són justament les dues coses que calia demostrar.

L'algorisme sencer queda:

$P = \{n \geq 0 \wedge d > 0\}$

$q := 0; r := n;$
mentre $r \geq d$ **fer**
  $q := q + 1;$
  $r := r - d;$
fmentre

Q:$\{n = d * q + r \wedge 0 \leq r < d\}$
I:$\{n = d * q + r \wedge 0 \leq r\ \}$
$t = r$

Càlcul del n-èsim número de Fibonacci

El següent exemple proposa desenvolupar un algorisme que permeti calcular el n-èsim terme de la successió de Fibonacci.

La successió de Fibonacci està definida de la manera següent:

$f_0 = 0$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ per a tot $n > 1$.

Seguint aquesta definició recurrent, els primers termes d'aquesta successió són els següents:

$f_0 = 0$, $f_1 = 1$, $f_2 = 1$, $f_3 = 2$, $f_4 = 3$, $f_5 = 5$, $f_6 = 8$...

L'algorisme hauria de complir la següent especificació:

n, a: natural

$P = \{n > 0\}$

fibonacci

$Q = \{a = f_n\}$

Al final a contindrà el n-èsim terme de la successió.

Quina estratègia seguirem en aquest cas? Com la informació que necessitem per tal de calcular f_{i+1} són els dos anteriors (f_i i f_{i-1}), us proposo que, en tot moment mantinguem en dues variables (per exemple, a i b) aquests dos elements. És a dir, que en tot moment: $a = f_i \wedge b = f_{i-1}$. Aquest és el nucli de l'invariant. El completem indicant que la variable que hem introduït per saber quan hem d'acabar (i) no pot superar n (de fet, quan $i = n$ tindrem que $a = f_n$ i, per tant, l'algorisme haurà acabat)[1]:

I=$\{a = f_i \wedge b = f_{i-1} \wedge 1 \leq i \leq n\}$

[1]Notem que l'essència de I l'obtenim canviant una expressió de Q (en aquest cas n) per una variable nova que introduim (en aquest cas i), obtenint $a = f_i$. Però en aquest cas, si només fem això, l'invariant que ens queda no és prou informatiu per tal de desenvolupar l'algorisme i hi hem hagut d'afegir més coses.

Immediatament deduim també que la fita serà $t = n - i$.

Si continuem el desenvolupament de l'algorisme obtenim el següent:

- La propietat $I \wedge \neg B \Longrightarrow Q$ ens suggereix que $\neg B = (i = n)$ I, per tant $B = (i \neq n)$ o, equivalentment $B = (i < n)$.
- Les inicialitzacions que fan que I sigui cert a la primera volta del bucle són:

  $i := 1; a := 1; b := 0;$

  Amb tot això obtenim:

  $i := 1; a := 1; b := 0;$
  mentre $i < n$ **fer**
  S;
  fmentre
- Per tal de desenvolupar el cos del bucle S haurem de:
  - Avançar vers la condició de finalització del bucle $\neg B$ (Decrementar la fita t).
    En aquest cas això voldrà dir clarament fer $i := i + 1$
  - Reestablir l'invariant.
    Això és, cal aconseguir que $a = f_i$ i $b = f_{i+1}$ després d'haver incrementat i (i.e. cal desplaçar a i b un terme en la successió). Això es pot aconseguir de la manera següent:
    $aux := a;$
    $a := a + b;$
    $b := aux;$

  El cos del bucle que s'obté finalment és doncs:

  mentre $i < n$ **fer**
  $aux := a;$
  $a := a + b;$
  $b := aux;$
  $i := i + 1;$
  fmentre

  Veiem que, efectivament, aquesta seqüència d'instruccions mantenen I:

  Anomenem i', a' i b' als valors de les variables i, a i b respectivament a l'inici de la volta.

  A l'inici es compleix

  $I \wedge B = \{a' = f_{i'} \wedge b' = f_{i'-1} \wedge 1 \leq i' < n\}$

  És a dir $a' = f_{i-1}$ i $b' = f_{i-2}$ (ja que $i' = i - 1$).

  Els valors de a i b (després dels càlculs que fa el cos del bucle) són els següents:

  $a = a' + b' = f_{i-1} + fi - 2 = f_i$
  $b = a' = f_{i-1}$

  Finalment, $1 \leq i' < n \Longrightarrow 1 \leq i \leq n$, ja que $i = i' + 1$.

La solució, completa és la següent:

$P = \{n > 0\}$

$i := 1; a := 1; b := 0;$
mentre $i < n$ **fer**

$aux := a$;
$a := a + b$;
$b := aux$;
$i := i + 1$;
fmentre

$Q = \{a = f_n\}$

$I = \{a = f_i \wedge b = f_{i-1} \wedge 1 \leq i \leq n\}$

$t = n - i$

Obtenció del màxim d'un vector

Sigui t un vector d'enters definit i inicialitzat entre els índexos 1 i N i inf i sup dos naturals:

t :vector<natural>(N); inf, sup: natural;

Volem derivar un algorisme que satisfaci la següent especificació:

$P = \{1 \leq inf \leq sup \leq N \wedge t[1..N] = T[1..N]\}$
màxim;
$Q = \{max = maxim(t, inf, sup) \wedge P\}$

$max = maxim(t, r, k)$ abreuja la següent asserció:
$\{\ \forall i : r \leq i \leq k \bullet t[i] \leq max \wedge \exists i : r \leq i \leq k \bullet t[i] = max\}$

Així doncs, l'algorisme tracta de situar a la variable max el valor més gran emmagatzemat en algun índex del vector t del rang $inf..sup$.

Com l'algorisme que derivarem no modificarà els valors inf, sup o t, és senzill d'entendre que P es complirà sempre en qualsevol punt del desenvolupament de l'algorisme. Per aquest motiu, a partir d'ara donarem aquest fet per suposat i no inclourem P en les diferents assercions (en particular en I i Q).

- Estratègia de disseny. Obtenció de I:

  L'estratègia més raonable per tal de dissenyar aquest algorisme és la que manté en tot moment a max el més gran de tots els valors del vector t des de l'índex inf fins a un cert índex k. Queda clar que quan $k = sup$, max contindrà el valor que indica Q.

  Aquesta estratègia permet d'obtenir ràpidament l'invariant adequat:

  $I = \{max = maxim(t, inf, k) \wedge inf \leq k \leq sup\}$

  Aquest invariant l'hem de llegir de la següent manera:

  Cada cop que iniciem (o acabem) una volta del bucle, max contindrà el més gran de tots els valors associats als índexos llegits fins el moment (tots els compresos entre inf i k).

  Notem que I s'obté introduint la variable nova k que substitueix l'expressió sup. El rang de k és el que tenia sup: $inf \leq k \leq N$, que encara podem afitar més $inf \leq k \leq sup$.

- Obtenció de B.

  Per obtenir Q a partir de I, necessitem que $k = sup$, per tant:

  $\neg B = (k = sup)$ i $B = (k \neq sup)$.

- Inicialització.

  Per tal d'aconseguir que I es compleixi des de l'inici, proposem la següent seqüència d'instruccions:

  $k := inf; max := t[inf]$;

  Notem que $t[inf]$ correspon sempre a un valor del vector t dins del seu rang de definició.

- La fita del bucle vindrà donada per la distància entre k i sup. $t = sup - k$. Aquesta distància s'haurà de fer petita a cada nova volta.

  Aquesta fita es dedueix de les següents propietats:

  - I garanteix que $inf \leq k \leq sup$.
  - B assegura que $k \neq sup$.
  - La inicialització $k := inf$.

- Derivació del cos del bucle.

  - Avançar (decrementar t) serà clarament $k := k + 1$;
    $k := inf; max := t[inf]$;
    mentre k $\neq$ sup **fer**
    $\{I \wedge B\}$
    reestablir;
    [1]
    $k := k + 1$;
    $\{I\}$
    fmentre
  - Reestablir I
    Per a que després de fer $k := k + 1$ es compleixi novament I, podem observar que a [1] només necessitem que $max \geq t[k + 1]$ (ja que a l'inici de la volta $I \Longrightarrow max \geq t[i]$ per a tot i, $inf \leq i \leq k$) i això ho garantirem amb el següent *reestablir*:
    si $t[k + 1] > max \longrightarrow max := t[k + 1]$ **fsi**
    Notem, finalment, que $t[k + 1]$ està dins del rang de t perquè $I \wedge B \Longrightarrow k < sup$.

2.5 Crida a accions i funcions

Amb les instruccions que hem vist fins ara tenim les eines per tal d'escriure qualsevol algorisme, però en necessitem encara algunes altres per tal d'*estructurar* els algorismes, això és: evitar repeticions de codi, fer-los més legibles, entenedors, elegants i abstractes... Una eina per aconseguir això i de la qual està dotat el nostre llenguatge algorísmic i la immensa majoria de llenguatges de programació són les accions i les funcions. La idea global de les accions i funcions és la de substituir un tros d'algorisme que conté una colla d'instruccions i que al final arriba a un determinat estat Q per una única instrucció especial (l'anomenem crida a acció). L'execució d'aquesta instrucció de crida a una acció en realitat inicia l'execució d'una seqüència d'instruccions (anomenada *acció*) que s'encarrega d'arribar a Q.

Les accions i funcions introdueixen, en realitat, un mecanisme d'abstracció funcional. En les properes seccions parlarem només d'accions. Al final presentarem les funcions.

2.5.1 Definició i notació

Definició 7 (acció)

Una acció és un algorisme amb dues caràcterístiques particulars:

1. *La seva execució pot ser iniciada des d'un altre algorisme. A la instrucció d'un algorisme que activa l'execució d'una acció se l'anomena* crida a acció.
2. *Es comunica amb l'algorisme que la crida amb una llista de dades que anomenem* paràmetres.

Notem que, com una acció és un algorisme, pot estar formada per totes les menes d'instruccions que poden aparèixer als algorismes (en particular altres crides a accions).

La notació que fem anar per tal de definir accions és la següent:

acció <nom> (<paràmetres>) **és**
<instruccions>
facció

I la crida es fa simplement amb el seu nom i els paràmetres:

<nom> (<paràmetres>);

Exemple:

$P = \{n \geq 0 \wedge d > 0\}$

acció divisió (n: natural, d: natural, q: natural, r: natural) **és**
  $q := 0; r := n;;$
  mentre $r \geq d$ **fer**
   $q := q + 1;$
   $r := r - d;$
  fmentre
facció

Q:$\{n = d * q + r \wedge 0 \leq r < d\}$

algorisme prova_acció **és**
  var
   x, y, quo, res: natural;
  fvar
  $x := 10;$
  $y := 3;$
  divisio(x, y, quo, res); //instrucció de crida a acció
falgorisme

Convé fer algunes anotacions sobre aquest exemple:

- Les dades de l'algorisme que l'acció necessita conèixer i les de l'acció que necessita conèixer l'algorisme són els paràmetres. En aquest cas: n, d, q i r (que es lliguen amb x, y, *quo* i res). A l'apartat següent discutirem més aspectes dels paràmetres.
- Imaginem ara un algorisme llarguíssim i molt complex que necessités fer tot un seguit de divisions enteres entre algunes de les seves dades. Cada cop que n'hagués de fer una, podria cridar l'acció *divisió* i no s'hauria de preocupar del codi d'aquesta acció. D'això se'n diu *abstracció funcional* (l'algorisme crida a una acció que fa alguna cosa, sense preocupar-se de **com** ho farà). L'abstracció funcional facilita el disseny d'algorismes. Els fa més abstractes.
- El codi dels algorismes que criden accions és molt més net i entenedor. Us imagineu un algorisme en què totes les crides a accions fossin substituïdes pel seu codi? Acabaria per ser incomprensible.

2.5.2 Paràmetres

Definició 8

Els paràmetres són dades que posen en contacte l'acció amb l'algorisme que la crida. Són de tres tipus:

1. Paràmetres d'entrada*: Dades que l'algorisme subministra a l'acció. L'acció necessita llegir-les per tal de dur a terme el seu objectiu. Es noten* **ent**

2. Paràmtres de sortida*: Dades que l'acció subministra a l'algorisme que la crida. Són els resultats de l'acció. Es noten* **sort**.
3. Paràmtres d'entrada/sortida*: Dades que l'acció necessita llegir per tal d'efectuar els seus càlculs i que després modifica i torna a subministrar a l'algorisme que la crida. Es noten* **e/s**

Exemple 1

$P = \{n \geq 0 \wedge d > 0\}$

acció divisió (n:**ent** natural, d:**ent** natural, q: **sort** natural, r: **sort** natural) **és**
  $q := 0; r := n;;$
  mentre $r \geq d$ **fer**
    $q := q + 1;$
    $r := r - d;$
  fmentre
facció

Q:$\{n = d * q + r \wedge 0 \leq r < d\}$

En aquest exemple, el dividend (n) i el divisor (d) són les dades que l'algorisme ha de subministrar a l'acció per tal que aquesta pugui fer la divisió. Són, doncs, els paràmetres d'entrada. El quocient (q) i el residu (r) són els resultats que calcula l'acció divisió i passa a l'algorisme. Són, doncs, paràmetres de sortida.

Exemple 2

P:$\{a = A \wedge b = B\}$

acció intercanvi (a:**e/s** natural, b:**e/s** natural) **és**
  var aux: natural; **fvar**
  $aux := a;$
  $a := b;$
  $b := aux;$
facció

Q:$\{a = B \wedge b = A\}$

En aquest segon exemple, tots dos paràmetres han de ser d'entrada-sortida ja que l'acció llegeix el seu valor i el modifica.

A vegades pot resultar complicat saber, entre totes les variables que té una acció i un algorisme, quines hem de convertir en paràmetres. Un criteri pràctic que ens permet determinar quines dades seran els paràmetres d'una acció és el següent:

> Triarem com a paràmertres d'una acció aquelles dades que necessitem per tal d'especificar l'acció (i.e. per tal de descriure **què** ha de fer aquella acció i no **com** ho ha de fer).

Al primer exemple, les dades que necessitàvem per especificar l'acció de la divisió entera eren el dividend (n), el divisor (d), el quocient (q) i el residu (r). Al segon exemple, les dades necessàries per tal d'especificar una acció eren les dues variables que es volien intercanviar (a i b), la variable que es necessita per fer l'intercanvi (aux) no ens cal per fer l'especificació de l'acció i , per tant, no la incorporem com a paràmetre.

Paràmetres formals i reals

Els paràmetres formals són els que apareixen a la definició de l'acció (n, d, q i r a l'acció *divisió* i a, b a l'acció *intercanvi*).

Els paràmetres reals són els que apareixen a la crida de les accions. Per exemple:

$x := 10; y := 3;$
divisió(x, y, quo, res); //paràmetres reals

O bé:

$x := 10; y := 3;$
intercanvi(x, y); //paràmetres reals

Els paràmetres reals es lliguen un a un amb els paràmetres formals amb l'ordre que apareixen a la definició de l'acció. Per això:

- Hi ha d'haver el mateix nombre de paràmetres formals que de reals i
- Han de coincidir dos a dos en tipus.

Per exemple, no podria passar que a la crida a l'acció *intercanvi* hi hagués 3 paràmetres (o 1) o bé que algun d'ells fos de tipus caràcter.

Quan comença l'acció *divisió* del darrer exemple, $n = 10$ i $d = 3$. A la fi de l'acció, $q = 3$ i $r = 1$. Aquests valors estan lligats a *quo* i res, per la qual cosa, quan prossegueix l'algorisme després de la crida a *divisió* $quo = 3$ i $res = 1$. x i y (que eren paràmetres d'entrada) no s'han modificat ($x = 10$ i $y = 3$).

Quan comença l'acció *intercanvi* del darrer exemple, $a = 10$ i $b = 3$. A la fi de l'acció, $a = 3$ i $b = 10$. Aquests valors estan lligats a x i y, per la qual cosa, quan prossegueix l'algorisme després de la crida a *divisió* es compleix que $x = 3$ i $y = 10$.

Notem que no hi ha cap problema d'utiltizar els mateixos noms de variable pels paràmetres reals i pels formals: intercanvi (a, b);

Pas de paràmetres

A la secció anterior hem indicat que hi havia un lligam un a un entre els paràmetres formals i els reals. Ara bé, com es fa aquest lligam? Això és, quin mecanisme de baix nivell s'utilitza per tal de poder manejar els paràmetres d'entrada, de sortida i d'entrada/sortida de manera que tinguin el comportament que hem presentat? (A aquest mecanisme se l'anomena *pas de paràmetres*).

- Paràmetres d'entrada: El valor del paràmetre real es copia sobre el paràmetre formal en el moment de la crida (abans, per tant, de l'inici de l'execució de l'acció). D'aquest pas de paràmetres se'n diu *pas de paràmetres per valor*.
- Paràmetres de sortida i d'entrada/sortida: En aquest cas el mecanisme anterior no funciona. Els paràmetres formals, en aquest cas, no contenen el valor dels reals sinó que *són una referència als reals*. D'aquest mecanisme se'n diu *pas de paràmetres per referència*.

2.5.3 Exemples

Presentem un algorisme que calcula els quocients de la divisió entera entre el màxim de tots els elements d'un vector i cadascun d'aquests elements. Per fer aquest algorisme ens beneficiarem de l'ús d'accions.

v, w :vector<natural>(N);
max : natural;
P:$\{v[1..N] = V[1..N] \wedge 1 \leq N \wedge \forall i : 1 \leq i \leq N \bullet v[i] \neq 0\}$

algorisme càlcul **és**
  var i, r :natural; **fvar**
  màxim_vector$(v, 1, N, max)$;
  $i := 1$;
  mentre $i \leq N$ **fer**
    divisió$(max, v[i], w[i], r)$;
    $i := i + 1$;
  fmentre
falgorisme

Q:$\{max = maxim(v, 1, N) \wedge \forall i : 1 \leq i \leq N \bullet w[i] = max$ **div** $v[i]\}$

Aquesty algorisme conté dues crides a accions: *divisió* i *màxim_vector*. La primera ja la coneixem. Pel que fa a la segona, convertirem en acció l'algorisme que vam fer per tal d'obtenir el màxim d'un vector:

$P = \{1 \leq inf \leq sup \leq N \wedge t[1..N] = T[1..N]\}$

acció màxim_vector (t: **ent** vector<natural>(N), inf: **ent** natural, sup: **ent** natural, max: **sort** natural)
és
  var k : natural; **fvar**
  $k := inf; max := t[inf]$;
  mentre k $\neq$ sup **fer**
    si $t[k + 1] > max \longrightarrow max := t[k + 1]$ **fsi**
    $k := k + 1$;
  fmentre
facció

$Q = \{max = maxim(t, inf, sup) \wedge P\}$

Exercici: Prova de fer aquest mateix exemple sense usar accions. Quin desl dos dissenys et sembla més clar?

2.5.4 Variables globals i variables locals

- **Variables locals**: Són les variables que es defineixen dins d'un algorisme o acció. Tenen una visibilitat que abarca *únicament* l'algorisme o acció A allà on estan definides (en particular, no són visibles en accions cridades per A). Deixen d'existir en el moment en què aquell algorisme o acció acaba.

- **Variables globals**: Són les variables que es defineixen abans de la descripció d'un algorisme. Són visibles des de qualsevol lloc de l'algorisme i de les accions cridades per aquest[2]. S'utiltizen per dades utilitzades molt (però molt) per l'algorisme.

  L'ús de variables globals compromet la reutilització de les accions i fa el codi menys elegant i abstracte. Per tant, *s'ha de reservar el seu ús per a casos molt concrets.*

2.5.5 Les funcions

Hi ha una varietat d'accions amb unes característiques lleugerament diferents que s'anomenen *funcions*.

Es diferencien de les accions en dos aspectes:

1. Tots els seus paràmetres són d'entrada.
2. Retornen explícitament un valor (anomenat el *resultat de la funció*).

[2]Més pròpiament podríem dir que són visibles des de qualsevol acció o algorisme definits al fitxer on es declaren les variables globals.

Aquestes diferències porten a una sintaxi de definició i de crida lleugerament diferents a les de les accions. Veiem-ne un exemple:

$P = \{n > 0\}$

funció fibonacci (n : natural) **retorna** a : natural **és**
  var i, b, aux : natural; **fvar**
  $i := 1; a := 1; b := 0;$
  mentre $i < n$ **fer**
   $aux := a;$
   $a := a + b;$
   $b := aux;$
   $i := i + 1;$
  fmentre
  retorna a;
  ffunció

$Q = \{a = f_n\}$

algorisme prova_fibo **és**
  var x, y: natural; **fvar**
  $x := 5;$
  y:=fibonacci(x);
falgorisme

$P = \{1 \leq inf \leq sup \leq N \wedge t[1..N] = T[1..N]\}$

funció màxim_vector (t: **ent** vector<natural>(N), inf: **ent** natural, sup: **ent** natural)**retorna** max: natural **és**
  var k : natural; **fvar**
  $k := inf; max := t[inf];$
  mentre k $\neq$ sup **fer**
   si $t[k+1] > max \longrightarrow max := t[k+1]$ **fsi**
   $k := k + 1;$
  fmentre
  retorna max;
facció

$Q = \{max = maxim(t, inf, sup) \wedge P\}$

algorisme prova_maxim **és**
  var x: natural;
  v : vector<natural>(N);
  fvar
  ...
  x :=màxim_vector($v, 1, N$);
falgorisme

2.5.6 Resum: beneficis de les accions i funcions

Les accions i les funcions aporten una sèrie de beneficis importants al disseny d'algorismes. Bàsicament són els següents:

- El dissenyador o dissenyadora de l'algorisme A pot deixar per més endavant la preocupació per alguns aspectes del seu disseny (pot *fer abstracció* d'aquests aspectes) i concentrar-se en tot moment

amb els que són més primordials. D'això se'n diu abstracció funcional.

- El codi que conté accions i funcions és molt més legible.
- Les accions i funcions es poden reutilitzar: més d'un algorisme pot cridar la mateixa acció o funció per tal de resoldre el mateix problema: no cal tornar-la a dissenyar.

2.6 Problemes

Problema 1

La composició seqüèncial d'accions, és commutativa?

Si ho és, demostrar-ho. Si no ho és, donar un contraexemple especificat i verificat.

Problema 2

Sigui S l'acció:

si
$x = 1 \longrightarrow x := -1$
$x = -1 \longrightarrow x := 1$
fsi

Satisfà alguna de les següents especificacions?

$\{enter(x) \wedge x = -X\} \quad S \quad \{x = X \wedge abs(x) = 1\}$

$\{enter(x) \wedge x = -X \wedge abs(x) = 1\} \quad S \quad \{x = X\}$

Problema 3

Demostrar que les dues següents especificacions són equivalents:

$\{enters(x, y, z) \wedge P\}$
si
$x \geq y \longrightarrow S_2; S_0$
$y > x \longrightarrow S_2; S_1$
fsi
$\{Q\}$

$\{enters(x, y, z) \bigwedge P\}$
S_2;
si
$x \geq y \longrightarrow S_0$
$y > x \longrightarrow S_1$
fsi
$\{Q\}$

(i) Per S_2: $x := x + 1;\ y := y + 1$

(ii) Per S_2: $x := y;\ y := x$

(*iii*) Si ho generalitzem per a qualsevol acció S_2, és certa l'equivalència?

Problema 4

Demostrar:

$\{enters(x, y) \wedge (y = 2x \vee y = -2x + 1)\}$
si
    y **mod** $2 = 0 \longrightarrow x := x - y$
    y **mod** $2 = 1 \longrightarrow x := x + y$
fsi
$y := y + 1;$
$\{y = 2x \vee y = -2x + 1\}$

Problema 5

Determinar expressions lògiques B_0 i B_1 tals que el següent algorisme sigui correcte.

$\{enters(x, y, z) \wedge x^y z = C \wedge x \geq 1 \wedge y \geq 0\}$
si
    $B_0 \longrightarrow x := x * x; y := y$ **div** 2
    $B_1 \longrightarrow z := z * x; x := x * x; y := (y - 1)$ **div** $2;$
fsi
$\{x^y z = C \wedge x \geq 1 \wedge y \geq 0\}$

Problema 6

És cert el següent?

$\{enters(x, y)\}$
mentre $(x < y) \vee (y < x)$ **fer**
    si
        $x < y \longrightarrow x := y; y := x$
        y **mod** $2 = 1 \longrightarrow y := x; x := y$
    fsi
fmentre
$\{x = y\}$

Problema 7

Demostrar que les següents especificacions són equivalents:

$\{P\}$
mentre $B_1 \vee B_2$ **fer**
    si
        $B_1 \longrightarrow S$
        $B_2 \longrightarrow S$
    fsi
fmentre

$\{Q\}$

$\{P\}$
mentre $B_1 \vee B_2$ **fer**
  S
fmentre
$\{Q\}$

Problema 8 (Examen Febrer-90 Diplomatura d'informàtica de Barcelona)

Donat el següent algorisme on N és un valor fix:

$\{enters(c,t,i) \wedge N > 0\}$
$c := 1;\ t := 0;\ i := 1;$
mentre $c \leq N$ **fer**
  $t := t + 6 * i;$
  $i := i + 1;$
  $c := c + t + 1;$
fmentre
$\{Q\}$

$I = \{c = i^3 \wedge t = 3i(i-1) \wedge (i-1)^3 \leq N\}$

(a) Especificar la postcondició Q i explicar-la.

(b) Demostrar que la proposició I és l'invariant de la iteració.

(c) Justificar que el bucle acabarà.

(d) Determinar la relació de recurréncia que satisfà la variable t.

Problema 9 (Examen Set-90 de la Diplomatura d'Informàtica de Barcelona)

$\{enter(n) \wedge n \geq 0\}$
$p := 1;\ s := 1;\ i := 0;$
mentre $i < n$ **fer**
  $i := i + 1;$
  $p := 2 * p;$
  $s := s + p;$
fmentre
$\{Q\}$

$I = \{p = 2^i \wedge s := \sum_{k=0}^{k=i} 2^k \wedge i \leq n\}$

Es demana:

(a) Determinar la postcondició Q i justificar que ho és.

(b) Demostrar que I és efectivament l'invariant del bucle.

(c) Notant que s representa la suma d'una progressió geomètrica, indica, explicant-ho molt clarament, si el següent algorisme

$\{enters(i,p)\}$
$p := 1;\ i := 0;$
mentre $i < n+1$ **fer**
    $i := i+1;$
    $p := 2*p;$
fmentre
$s := p-1;$
$\{Q\}$

on el bucle té l'invariant $I' = \{p = 2^i \wedge i \leq n+1\}$, calcula el mateix valor de s.

Problema 10

Considerem el següent algorisme:

$P = \{n > 0\}$

$i := 1; a := 1; b := 0;$
mentre $i < n$ **fer**

$i := i+1;$

$aux := a;$

$a := a+b;$

$b := aux;$

fmentre

$Q = \{a = f_n\}$

1. Què fa aquest algorisme? (i.e. Quin és el significat de f_n?)
2. Quin és l'invariant de la iteració?
3. Dedueix una fita per la mateixa.
4. Verifica formalment l'algorisme.

Ajuda: Per trobar l'invariant del bucle s'ha d'especular amb el paper que juguen la a i la b en cada volta del bucle.

Problema 11

Considerem el següent algorisme:

$P = \{0 < n\}$

$i := 1;$
mentre $2*i < n$ **fer**
$i := 2*i;$
fmentre

$\{Q\}$

1. Què fa aquest algorisme? (Quina és la seva postcondició?)
2. Quin és l'invariant de la iteració?

3. Dedueix una fita per la mateixa.
4. Verifica'l.

Problema 12

Considerem el següent algorisme:

$P = \{x \geq 0 \wedge y > 0\}$

$q := 0; r := x;$
mentre $r \geq y$ **fer**
  $r := r - y;$
  $q := q + 1;$
fmentre

1. Què fa aquest algorisme? (Quina és la seva postcondició?)
2. Quin és l'invariant de la iteració?
3. Dedueix una fita per la mateixa.
4. Verifica'l.

Problema 13

Considerem el següent algorisme:

b:**vector** $[0..n-1]$ **de** enter;

$P = \{n > 0\}$

$i := 1; k := 0;$
mentre $i < n$ **fer**
  si $b[i] \leq b[k] \longrightarrow$ no_op;
    $b[i] \geq b[k] \longrightarrow k := i;$
  fsi
  $i := i + 1;$
fmentre

$\{Q\}$

1. Què fa aquest algorisme? (Quina és la seva postcondició?)
2. Quin és l'invariant de la iteració?
3. Dedueix una fita per la mateixa.
4. Verifica'l.

De tots els problemes que es proposen tot seguit, caldrà fer l'especificació, obtenir l'invariant, la guarda del bucle, les inicialitzacions, la fita del bucle i el seu cos.

Problema 14

Construir i verificar un algorisme que trobi la part entera de l'arrel quadrada d'un número natural. Concretament l'algorisme haurà de satisfer la següent especificació:

$P = \{n \geq 0\}$

aproximacio_arrel;

$Q = \{a^2 \leq n \leq (a+1)^2\}$

Problema 15

Especificar i derivar un algorisme que passi de base 10 a base B.

Problema 16

Donat un vector definit entre 1 i n, d'enters, es tracta de derivar una acció que calculi el nombre d'elements que conté abans del primer 0. Pot ser que al vector no hi hagi cap 0. Com canviaria el disseny si s'assegurés que el vector conté almenys un 0 entre 1 i n?

Problema 17

Un ordinador és capaç de multiplicar i de fer divisions enteres de naturals per 2 molt ràpidament(ja que en la representació binària dels naturals a l'ordinador, aquestes operacions consisteixen simplement en fer un desplaçament a l'esquerra o a la dreta).

Es pot aprofitar aquesta propietat per a desenvolupar un algorisme que permeti multiplicar dos naturals qualssevol usant només multiplicacions i divisions enteres per 2 (i tal vegada alguna suma).

La idea és la següent: Si a i b són dos naturals, aleshores

$$a * b = \begin{cases} (a*2)*(b \text{ div } 2) & \text{si } b \text{ parell} \\ (a*2)*(b \text{ div } 2) + a & \text{si } b \text{ senar} \\ 0 & \text{si } b{=}0 \end{cases}$$

Utilitza aquesta idea per a derivar un algorisme que calculi el producte de dos naturals.

Problema 18

Deriva un algorisme per a calcular l'arrel quadrada per defecte d'un cert natural n. L'especificació de l'algorisme és la següent:

enters(n,a)
$P = \{n \geq 0\}$
$Q = \{a^2 \leq n < (a+1)^2\}$

Problema 19

Dissenyar una acció que, donat un vector no decreixent d'enters i definit entre 1 i n amb $n \geq 1$, calculi la longitud del replà més llarg. Un replà és una seqüència de valors iguals situats en índexos consecutius.

Problema 20

Derivar una acció que trobi la *moda* d'un vector [1..n] de enters. La moda és el valor que apareix repetit més vegades. Considera dues possibilitats: que el vector estigui ordenat i que no ho estigui.

Capítol 3

Exemples d'algorismes iteratius

3.1 Introducció

En aquest capítol presentem exemples d'algorismes iteratius desenvolupats seguint les idees presentades al capítol antyerior. En particular, utilitzarem els conceptes d'invariant i de fita per tal de guiar el desenvolupament dels bucles que continguin.

Les primeres dues seccions presenten l'algorisme de recorregut d'un vector i el de la cerca seqüencial d'un element en un vector). Ambdós algorismes són seqüencials, la qual cosa vol dir que les úniques operacions que fan sobre el vector són:

- Accedir al primer element del mateix i
- Un cop estem situats damunt d'un element determinat, obtenir el següent.

És a dir, el vector es tracta com una seqüència[1]

Recordem que els vectors admeten també un accés *directe per índex*, això és, permeten accedir a $v[9]$ sense haver accedit prèviament a $v[1], v[2], ..., v[8]$. Aquest accés s'utilitza per primer cop en l'algorisme de la cerca dicotòmica (secció 3.4). Veurem que el recurs de l'accés directe ens permetrà augmentar l'eficiència de l'algorisme de cerca.

El capítol presenta altres algorismes molt útils, com ara el de la bipartició d'un vector (que serà la llavor de l'algorisme d'ordenació *quicksort*).

3.2 Recorregut d'un vector

Us proposo, en primer lloc, el problema general de resseguir seqüencialment tots els elements d'un vector per tal d'aplicar un cert tractament a cadascun d'ells. Un exemple d'aquest tipus de problema pot ser *escriure per la sortida estàndar tots els elements d'un vector*. En el cas d'aquest exemple, el tractament que cal fer a cada element és (lògicament) escriure'l per la sortida estàndar.

Per tal de fer un desenvolupament que faci abstracció del tractament particular que s'haurà de fer en cada situació específica, definirem una acció *tractament* de la següent manera:

acció tractament(x:T)

on T és el tipus dels elements del vector que estem recorrent (e.g. T és el tipus *natural* si estem recorrent un vector de naturals).

[1]Una seqüència és una col.lecció d'elements als quals es pot accedir únicament amb aquestes dues operacions.

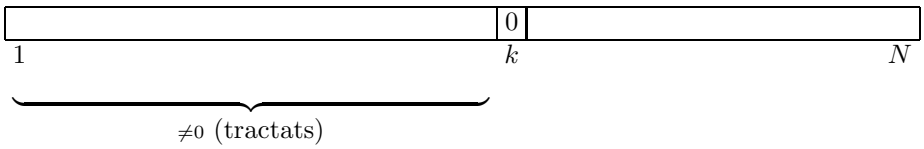


Figura 3.1: Postcondició de l'acció *recorregut*

Aquesta acció realitza un cert tractament sobre l'element x.

Igualment definirem un predicat *tractat(x)* que pren el valor cert si l'element x ha estat tractat i fals en cas contrari.

Un altre element fonamental per tal de fer un desenvolupament de l'algorisme de recorregut és la caracterització del vector que volem recórrer: hem de saber exactament on comença i on acaba aquest vector. Usualment s'utilitzen dues maneres diferents per indicar el final dels elements que cal tractar dins d'un vector:

- Una marca de final (que ja no cal tractar). Exemple: Cal sumar tots els nombres del vector v fins trobar un zero (el qual, òbviament) no cal sumar. Un altre exemple és la representació que vam proposar al capìtol 1 per les cadenes de caràcters (les quals s'emmagatzemaven en un vector i acabaven amb una marca de final).
- Un rang d'índexs que representen la seqüència d'elements que cal recórrer (i tractar). Exemple: Cal escriure tots els elements del vector v entre els seus índexos inf i sup (on inf i sup solen ser dos paràmetres de l'acció que desenvolupa el recorregut).

El que farem en els propers apartats és suggerir un esquema de resolució del problema del recorregut d'un vector per les dues caracteritzacions que acabem de proposar pels vectors.

3.2.1 Recorregut d'un vector provist d'una marca de final

L'especificació

L'especificació d'aquest problema és la següent:

P:$\{\exists j : 1 \leq j \leq N \bullet (v[j] = 0 \wedge \forall i : 1 \leq i < j \bullet v[i] \neq 0) \wedge 1 \leq N\}$

acció recorregut(v: **ent** vector<natural>(N),k: **sort** natural);

Q:$\{v[k] = 0 \wedge \forall i : 1 \leq i < k \bullet (v[i] \neq 0 \wedge tractat(v[i])) \wedge 1 \leq k \leq N\}$

La precondició indica que el vector té zero com a marca de final (el zero apareix sempre al vector i és el primer element que no cal tractar).

La postcondició estableix que a la fi de l'execució de l'acció *recorregut* haurem tractat tots els elements del vector que ocupen índexos anteriors a k (des del que ocupa l'índex 1 fins el que ocupa l'índex $k-1$). Essent l'índex k el que conté la marca de final (zero). Vegeu la figura 3.1 per una interpretació gràfica d'això.

Comencem, tot seguit, el disseny de l'acció.

Invariant

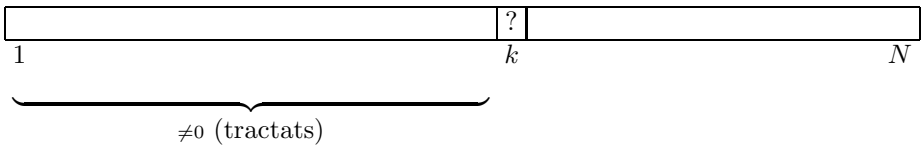


Figura 3.2: Invariant de l'acció *recorregut*

Com ja hem dit, la postcondició estableix que, al final, haurem tractat tots els elements anteriors a líndex k i aquest índex és el que conté la marca de final (zero). De fet, aquesta postcondició ens dóna directament una estratègia que pot seguir el bucle de l'acció per tal de completar-la: podem dissenyar el bucle de tal manera que, en una volta qualsevol, *s'hagin tractat tots els elements de v corresponents a índexos anteriors estrictament a k*. Aquest serà precisament l'invariant del bucle que us proposo:

I:$\{\forall i : 1 \leq i < k \bullet (v[i] \neq 0 \wedge tractat(v[i])) \wedge 1 \leq k \leq N\}$

Notem, doncs, que hem afeblit la postcondició eliminant el conjuntant que indicava que $v[k] = 0$. La figura 3.2 mostra una interpretació gràfica del què diu I.

Condició de continuació (B)

Seguint la propietat que estableix $I \wedge \neg B \Longrightarrow Q$ (i.e. $\neg B$ és el que li falta a I per tal d'obtenir Q) deduïm que $\neg B = (v[k] = 0)$ i, per tant, $B = (v[k] \neq 0)$.

Inicialitzacions

Per tal d'aconseguir que I es compleixi a l'inici de la primera volta, n'hi ha prou amb fer $k := 1$;.

P garanteix que $v[1]$ existirà (en el pitjor dels casos contindrà la marca de final) i serà cert que:

$\forall i : 1 \leq i < 1 \bullet v[i] \neq 0 \wedge tractat(v[i])$

perquè no hi ha cap i que sigui més gran o igual que 1 i, alhora, més petit que 1, l'enunciat esdevé cert per domini nul d'aplicació.

Fita

La fita del bucle pot ser $t = N - k$. k haurà d'anar avançant en cada volta. D'altra banda, si B és certa, t es mantindrà sempre positiva (ja que P ens garanteix que es trobarà la marca de final per algun índex $j \leq N$).

Cos del bucle

Amb tot el que hem dit hem elaborat l'acció següent:

acció recorregut(v: **ent** vector<natural>(N),k: **sort** natural) **és**
  $k := 1$;
  mentre $v[k] \neq 0$ **fer**
    S;
  fmentre
facció

I:$\{\forall i : 1 \leq i < k \bullet (v[i] \neq 0 \wedge tractat(v[i])) \wedge 1 \leq k \leq N\}$
$t = (N - k)$

L'objectiu del cos del bucle S ha de ser avançar (decrementar la fita) tot i mantenint I. Això s'aconsegueix amb el parell d'instruccions següent:

tractament($v[k]$);
$k := k + 1$;

Efectivament, si a l'inici d'una volta qualsevol es compleix $I \wedge B$, totes les propietats que estableix I es compliran a la fi d'aquella volta:

I:$\{\underbrace{\forall i : 1 \leq i < k \bullet (v[i] \neq 0 \wedge tractat(v[i]))}_{(1)} \wedge \underbrace{1 \leq k \leq N}_{(2)}\}$

- (1.a): Tots els elements anteriors a k han sigut tractats perquè:
  - a l'inici de la volta ja s'havien tractat tots els anteriors a k (I ens ho assegura),
  - a la primera instrucció de la volta tractem $v[k]$ i,
  - tot seguit, incrementem k, de forma que, a la fi de la volta, segueix essent cert que hem tractat tots els elements fins l'índex $k - 1$.
- (1.b): Tots els elements d'índex anterior a k tenen un valor diferent de la marca.

  Això és cert perquè a l'inici de la volta es compleix $I \wedge B$.
- (2): $1 \leq k \leq N$

  També és cert, ja que $v[k-1] \neq 0$, segons acabem de veure, i P garanteix que hi ha d'haver algun índex $j \leq N$, pel qual $v[j] = 0$. Evidentment, es complirà que $j \geq k$.

Notem, finalment, que $k := k + 1$ fa decrementar la fita.

L'acció recorregut queda, acabada, de la següent manera:

P:$\{\exists j : 1 \leq j \leq N \bullet (v[j] = 0 \wedge \forall i : 1 \leq i < j \bullet v[i] \neq 0) \wedge 1 \leq N\}$

acció recorregut(v: **ent** vector<natural>(N),k: **sort** natural) **és**
  $k := 1$;
  mentre $v[k] \neq 0$ **fer**
    tractament($v[k]$);
    $k := k + 1$;
  fmentre
facció

Q:$\{v[k] = 0 \wedge \forall i : 1 \leq i < k \bullet (v[i] \neq 0 \wedge tractat(v[i])) \wedge 1 \leq k \leq N\}$

I:$\{\forall i : 1 \leq i < k \bullet (v[i] \neq 0 \wedge tractat(v[i])) \wedge 1 \leq k \leq N\}$
$t = (N - k)$

Aquesta acció l'hem desenvolupat amb vectors però es pot extrapolar a làmbit més general de qualsevol tipus de seqüència. En general, podem dir que l'esquema de recorregut d'una seqüència amb marca de final que no cal tractar és el següent:

esquema recorregut(s: **ent** Seqüència) **és**
  x:=obtenir_primer(s);
  mentre $x \neq$ *Marca* **fer**
    tractament(x);
    x:=obtenir_següent(s);
  fmentre
fesquema

Un exemple

L'acció que hem obtingut la podem particularitzar per tractaments determinats. Per exemple, podem escriure una acció que obtingui el màxim d'un vector de naturals marcat pel zero:

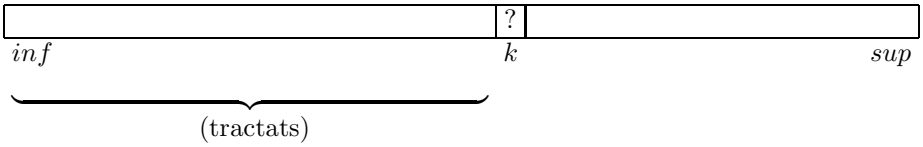


Figura 3.3: Invariant de l'acció *recorregut2*

acció màxim(v: **ent** vector<natural>(N),k: **sort** natural, max: **sort** natural) **és**
  $k := 1$;
  mentre $v[k] \neq 0$ **fer**
    si $v[k] > max \longrightarrow max := v[k]$;
    $k := k + 1$;
  fmentre
facció

En la qual hem introduït un nou paràmetre max i hem particularitzat el tractament.

3.2.2 Recorregut d'un vector en un rang donat per dos índexos

L'especificació

L'especificació d'aquest problema és la següent:

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup + 1 \leq N + 1\}$

acció recorregut2(v: **ent** vector<natural>(N),inf: **ent** natural,sup: **ent** natural);

Q:$\{\forall i : inf \leq i \leq sup \bullet tractat(v[i])\}$

La precondició indica que el vector està inicialitzat entre els índexos inf i sup. Cal notar que és possible que el vector estigui buit ($inf = sup + 1$). En aquell cas, evidentment, no caldrà tractar cap element.

La postcondició estableix que a la fi de l'execució de l'acció *recorregut2* haurem tractat tots els elements del vector que ocupen els índexos entre inf i sup.

Desenvolupament de l'acció

Tot i que el vector estigui caracteritzat de manera diferent, podem aprofitar una estratègia gairebé calcada de la que hem seguit a l'apartat anterior. Aquesta estratègia es pot materialitzar amb el següent invariant:

A l'inici i a la fi de cada volta, haurem tractat tots els elements de v corresponents a índexos entre inf *i* $k - 1$.

Així doncs, introduirem una variable local k que serà el proper índex que ens caldrà tractar.

L'invariant queda de la següent manera:

I:$\{\forall i : inf \leq i < k \bullet tractat(v[i]) \wedge inf \leq k \leq sup + 1\}$

Notem que ara $\neg B = (k = sup + 1)$, ja que només quan $k = sup + 1$ podrem garantir que hem tractat tots els elements del vector, com ens indica Q (Recordem que B és allò que li manca a I per tal d'obtenir Q: $I \wedge \neg B \Longrightarrow Q$).

Notem també que l'invariant i la condició de continuació ens garanteixen el bon funcionament si el vector és buit ($inf = sup + 1$).

Modificant lleugerament els raonaments que hem fet abans, podem obtenir el desenvolupament següent per l'acció *recorregut2*:

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup + 1 \leq N + 1\}$

acció recorregut2(v: **ent** vector<natural>(N),inf: **ent** natural,sup: **ent** natural) **és**
  var k: natural; **fvar**
  $k := inf$;
  mentre $k \leq sup$ **fer**
    tractament($v[k]$);
    $k := k + 1$;
  fmentre
facció

Q:$\{\forall i : inf \leq i \leq sup \bullet tractat(v[i])\}$

I:$\{\forall i : inf \leq i < k \bullet tractat(v[i]) \wedge inf \leq k \leq sup + 1\}$

$t = sup - k$

Un exemple

El problema d'obtenir el màxim d'un vector es pot aplicar a un vector entre un rang d'índexos donat, de la manera següent:

acció màxim2(v: **ent** vector<natural>(N),inf: **ent** natural,sup: **ent** natural) **és**
  var k: natural; **fvar**
  $k := inf$;
  $max := 0$;
  mentre $k \leq sup$ **fer**
    si $v[k] > max \longrightarrow max := v[k]$;
    $k := k + 1$;
  fmentre
facció

3.3 Cerca seqüencial en un vector

En aquesta secció us proposo el problema de cercar en un vector el primer element que compleixi una propietat determinada (e.g el primer element més gran que 100, el primer element primer, el primer parell, el primer amb valor 24...). La més gran diferència entre els algorismes que veurem en aquesta secció i els de recorregut (vegeu la secció 3.2) és que, mentre que els de recorregut havien de recórrer tot el vector (fins la marca de final o tot el rang d'elements entre els índexos inf i sup), els de cerca es podran aturar en el moment en què trobin l'element cercat. Encara que, en el cas pitjor (si l'element cercat no es troba al vector) també l'hauran de recórrer tot. Per aquest motiu ja podem avançar el canvi en la condició de continuació del **mentre**. En el cas del recorregut, aquesta condició era
mentre $\neg final$ **fer**
I la sortida es produia quan $final$ era cert.

En el cas de la cerca, la condició de continuació del **mentre** tindrà usualment la forma:
mentre $\neg trobat \wedge \neg final$ **fer**
Notem que la condició de sortida del bucle, que s'obtindrà per negació de la de continuació serà:
$trobat \vee final$
Que indica que sortirem del bucle, bé quan haguem trobat l'element cercat o bé quan arribem al final del vector.

En els propers apartats ho veurem més detingudament i detallada. Com abans, també farem la distinció entre els vectors que acaben amb marca de final i els que queden determinats per un rang d'índexos entre

els quals es farà la cerca.

Utilitzarem una funció anomenada *trobat* per tal de determinar si un element determinat compleix la propietat desitjada:

funció trobat(x:T) **retorna** booleà.

Retorna cert si x compleix la propietat que regula la cerca i fals en cas contrari.

3.3.1 Cerca seqüencial en un vector acabat amb una marca de final

L'especificació

P:$\{\exists j : 1 \leq j \leq N \bullet (v[j] = 0 \wedge \forall i : 1 \leq i < j \bullet v[i] \neq 0) \wedge 1 \leq N\}$

acció cerca(v: **ent** vector<natural>(N),k: **sort** natural);

Q:$\{\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca) \wedge (trobat(v[k]) \vee v[k] = 0) \wedge 1 \leq k \leq N\}$

La postcondició estableix que a la fi de l'acció, $v[k]$ contindrà el primer element del vector que compleix la propietat desitjada o bé la marca de final si aquell element no existeix.

L'Invariant i la condició de continuació

El conjuntant fort de l'anterior potcondició, el que determina el final del bucle i de l'acció és:
$(trobat(v[k]) \vee v[k] = Marca)$ [1]
Que indica que l'element actual ($v[k]$) és la marca de final o compleix la propietat cercada.

Els altres dos conjuntants, en realitat, ens marquen una possible estratègia a seguir per tal de trobar l'element: es tracta de visitar un a un els elements del vector i avançar només si l'element que acabem de visitar no satisfà la propietat i no és la marca de final. Dit, d'una altra manera, en una volta qualsevol es complirà:

Tots els elements anteriors a l'index k no compleixen la propietat desitjada i no són la marca de final.
O, expressat formalment:

I:$\{\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca) \wedge 1 \leq k \leq N\}$

Notem que hem obtingut I eliminant el conjuntant [1] de Q. Notem també que l'essència dels invariants en el cas del recorregut i de la cerca és la mateixa: *el proper element que cal tractar (recorregut) o comprovar si compleix la propietat (cerca) és* $v[k]$.

A partir de la propietat $I \wedge \neg B \Longrightarrow Q$ s'obté immediatament B:
$\neg B = (trobat(v[k]) \vee v[k] = Marca)$
$B = (\neg trobat(v[k]) \wedge v[k] \neq Marca)$

Inicialitzacions

És clar que, per tal d'obtenir I a l'inici de la primera volta, cal fer: $k := 1$

Fita

Com a fita podem usar $t = N - k$.

$I \wedge B$ ens asseguren que t serà positiva dins del bucle i, a partir de l'estratégia que hem dissenyat (i que marca I), serà natural incrementar k dins del bucle, per la qual cosa t es decrementarà estrictament a cada volta.

Disseny del cos del bucle

Fins ara hem obtingut l'acció següent:

P:$\{\exists j : 1 \leq j \leq N \bullet (v[j] = 0 \wedge \forall i : 1 \leq i < j \bullet v[i] \neq 0) \wedge 1 \leq N\}$

acció cerca(v: **ent** vector<natural>(N),k: **sort** natural) **és**

$k := 1$;
mentre $\neg trobat(v[k]) \wedge v[k] \neq Marca$ **fer**
S;
fmentre
facció

Q:$\{\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca) \wedge (trobat(v[k]) \vee v[k] = Marca) \wedge 1 \leq k \leq N\}$

I:$\{\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca) \wedge 1 \leq k \leq N\}$

$t = N - k$

Les instruccions del cos del bucle S s'hauran d'ocupar de decrementar la fita i de mantenir I. En aquest cas, la manera natural de decrementar la fita ($k := k + 1$) manté I, ja que només incrementem k si $\neg trobat(v[k]) \wedge v[k] \neq Marca$. Per aquest motiu és clar que, *després de fer* $k := k + 1$ es complirà:

- $\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca)$ i també
- $1 \leq k \leq N$ (ja que $v[k-1] \neq Marca$).

L'acció següent és la solució final proposada al problema de la cerca en un vector marcat.

P:$\{\exists j : 1 \leq j \leq N \bullet (v[j] = 0 \wedge \forall i : 1 \leq i < j \bullet v[i] \neq 0) \wedge 1 \leq N\}$

acció cerca(v: **ent** vector<natural>(N),k: **sort** natural) **és**

$k := 1$;
mentre $\neg trobat(v[k]) \wedge v[k] \neq Marca$ **fer**
$k := k + 1$;
fmentre
facció

Q:$\{\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca) \wedge (trobat(v[k]) \vee v[k] = Marca) \wedge 1 \leq k \leq N\}$

I:$\{\forall i : 1 \leq i < k \bullet (\neg trobat(v[i]) \wedge v[i] \neq Marca) \wedge 1 \leq k \leq N\}$

$t = N - k$

Igual com hem fet en la secció anterior, també podem extreure de l'acció que acabem de dissenyar l'esquema general de cerca en una seqüència amb marca de final que no cal tractar. És el següent:

esquema cerca(s: **ent** seqüència) **és**

x:=obtenir_primer(s);
mentre $\neg trobat(x) \wedge \neg final(s)$ **fer**
x:=obtenir_següent(s);
fmentre
facció

Exemple

Obtenir el primer element més gran que 100 en un vector marcat pel 0 (el valor zero actua com a marca de final).

P:$\{\exists j : 1 \leq j \leq N \bullet (v[j] = 0 \wedge \forall i : 1 \leq i < j \bullet v[i] \neq 0) \wedge 1 \leq N\}$

acció cerca_major_100(v: **ent** vector<natural>(N),k: **sort** natural) **és**

$k := 1$;
mentre $v[k] \leq 100 \wedge v[k] \neq 0$ **fer**
$k := k + 1$;
fmentre
facció

Q:$\{\forall i : 1 \leq i < k \bullet (v[i] \leq 100 \wedge v[i] \neq 0) \wedge (v[k] > 100 \vee v[k] = 0) \wedge 1 \leq k \leq N\}$

A l'apartat de problemes es proposen algunes variants de les cerques seqüencials. Una d'elles és aplicar un determinat tractament a tots els elements anteriors al primer que compleix la propietat desitjada (e.g. Escriure tots els elements del vector fins el primer més gran que 100).

3.3.2 Cerca seqüencial d'un vector no nul en un rang donat per dos índexos

Especificació

L'especificació d'aquest problema és la següent:

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup \leq N\}$

acció cerca2(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural);

Q:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge (trobat(v[k]) \vee k = sup) \wedge inf \leq k \leq sup\}$

Fem alguns comentaris respecte aquesta especificació:

- La precondició força a que el vector tingui almenys un element en el qual fer la cerca (i.e. el vector no pot ser nul. Notem que $1 \leq inf \leq sup \leq N$). Aquesta condició era natural i obligatòria pels vectors amb marca de final, ja que almenys havien de contenir la marca. Quan l'extensió de cerca ve donada per dos índexos, pot passar que el rang de cerca sigui nul (i.e. $inf > sup$). Aquest cas el tractarem a l'apartat següent.
- La postcondició estableix
  $(trobat(v[k]) \vee k = sup)$ [2]
  Si l'element cercat es troba a l'índex sup, ambdós disjuntants seran certs en la postcondició (*trobat* i *final*). Això era impossible en el cas en què el vector estés marcat per una marca de final perquè $trobat(Marca) = fals$ sempre.

Desenvolupament

L'invariant es pot obtenir, com en el cas anterior, eliminant el conjuntant [2] de la postcondició. La negació d'aquest conjuntant esdevé aleshores la condició de continuació B. La resta de raonaments són similars als del cas anterior i condueixen a la següent solució:

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup \leq N\}$

acció cerca2(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural) **és**

$k := inf$;
mentre $\neg trobat(v[k]) \wedge k < sup$ **fer**
$k := k + 1$;
fmentre
facció

Q:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge (trobat(v[k]) \vee k = sup) \wedge inf \leq k \leq sup\}$

I:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge inf \leq k \leq sup\}$

$t = sup - k$

D'aquesta acció cal notar almenys dues coses:

- Necessita que el vector contingui almenys un element (i.e. $v[inf]$ ha de ser un element del rang de la cerca sobre v. En cas contrari el valor de $v[inf]$ és indefinit i suposa un accés incorrecte al vector v). Això exclou l'aplicació d'aquest algorisme per vectors amb rang nul de valors (recordem que P establia explicitament que el rang de v sobre el qual fer al cerca no és buit).
- Algun estudiant aventatjat(?) podria pensar que fóra més pràctica una $B = (\neg trobat(v[k]) \wedge k \leq sup)$ que, suposadament, garantiria que no hi ha cap element a $v[inf..sup]$ que compleixi la propietat desitjada si i només si, al final $k = sup + 1$ (amb l'acció *cerca2*, a la sortida del bucle, si $k = sup$ encara no sabem si $v[sup]$ acompleix la propietat desitjada, trobat($v[sup]$), o no). Cal adonar-se que aquesta solució, en general, faria un accés a $v[sup+1]$, fora del rang de cerca i, per tant, incorrecte.

Exemple

Obtenir el primer element més gran que 100 en un vector marcat pel 0 (el valor zero actua com a marca de final).

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup \leq N\}$

acció cerca_sup_100(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural) **és**

```
k := inf;
mentre v[k] ≤ 100 ∧ k < sup fer
  k := k + 1;
fmentre
```

facció

Q:$\{\forall i : inf \leq i < k \bullet v[i] \leq 100 \wedge (v[k] > 100 \vee k = sup) \wedge inf \leq k \leq sup\}$

3.3.3 Cerca seqüencial d'un vector general en un rang donat per dos índexos

Quan no tenim garanties de que existeixi almenys un element dins del rang $inf...sup$ de v, l'especificació del problema de cerca adquireix la seva forma general i és la següent:

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup + 1 \leq N + 1\}$

acció cerca3(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural);

Q:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge (k = sup + 1 \vee trobat(v[k])) \wedge inf \leq k \leq sup + 1\}$

La precondició permet que $inf = sup + 1$, que serà la indicació de que el vector és nul (i.e. no conté cap element).

En aquest cas, la sortida $k = sup+1$ indica que no s'ha trobat cap element dins $v[inf..sup]$ que compleixi la propietat desitjada. Com a cas particular, si el vector fos nul ($inf = sup+1$) hauríem d'obtenir també $k = sup + 1$.

La dificultat en resoldre aquest problema és que k pot arribar a ser $sup + 1$. Però no podem deixar que l'algorisme consulti $v[sup + 1]$. Per aquest motiu, la solució que s'obtindria de la manera habitual (fent $\neg B = (k = sup + 1 \vee trobat(v[k]))$) i que donaria:

acció cerca3_incorrecta(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural) **és**

```
k := inf;
mentre ¬trobat(v[k]) ∧ k ≤ sup fer
  k := k + 1;
fmentre
```

facció

no és correcta perquè si el vector és buit o si cap element no acompleix la propietat desitjada, s'accedeix a $v[sup+1]$. Això es pot resoldre substituint el càlcul de $trobat(v[k])$ per una variable booleana que serà

actualitzada dins del bucle (on ja sabem que $k \leq sup$).

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup + 1 \leq N + 1\}$

acció cerca3(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural) **és**
  var aux_trobat:booleà; **fvar**
  $k := inf$;
  aux_trobat:=fals;
  mentre $\neg aux\_trobat \wedge k \leq sup$ **fer**
    si $trobat(v[k]) \longrightarrow aux\_trobat := cert$
    si no $k := k + 1$;
    fsi
  fmentre
facció

Q:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge (k = sup + 1 \vee trobat(v[k])) \wedge inf \leq k \leq sup + 1\}$

I:$\{(aux\_trobat \wedge trobat(v[k])) \vee (\neg aux\_trobat \wedge \forall i : inf \leq i < k \bullet \neg trobat(v[k])) \wedge inf \leq k \leq sup + 1\}$

$t = sup + 1 - k$

3.3.4 Cerca seqüencial d'un vector en un rang donat per dos índexos amb garantia d'èxit

Hi ha un cas particular de cerca que suposa una simplificació prou important i que, per tant, val la pena tractar-lo separadament. Es tracta del cas en què la precondició ens assegura que el vector conté, entre els índexos corresponents, un valor que acompleix la propietat desitjada (en particular, estem segurs que el vector és no nul entre inf i sup). L'acció que obtenim està basada en *cerca2* amb la ventatja que no li cal preguntar per si hem arribat al final del vector: abans d'arribar-hi haurem trobat l'element cercat.

P:$\{v[inf..sup] = V[inf..sup] \wedge 1 \leq inf \leq sup \leq N \wedge \exists i : inf \leq i \leq sup \bullet trobat(v[i])\}$

acció cerca4(v: **ent** vector<natural>(N),inf: **ent** natural, sup: **ent** natural, k: **sort** natural) **és**

  $k := inf$;
  mentre $\neg trobat(v[k])$ **fer**
    $k := k + 1$;
  fmentre
facció

Q:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge trobat(v[k]) \wedge inf \leq k \leq sup\}$

I:$\{\forall i : inf \leq i < k \bullet \neg trobat(v[i]) \wedge inf \leq k \leq sup\}$

$t = sup - k$

La simplicitat d'aquesta acció porta a la idea següent: per tal d'assegurar-nos que l'element cercat és al vector, l'hi podem afegir a l'índex $sup + 1$. Si a la fi $k = sup + 1$ vol dir que el vector original no contenia l'element cercat; si $k \leq sup$, si el contenia. D'aquesta tècnica se'n diu *tècnica del sentinella* i per a que es pugui aplicar hem d'estar segurs que a l'índex $sup + 1$ de v hi podem posar un element. El problema ??? s'ocupa d'aquesta tècnica.

3.4 La cerca dicotòmica

El mètode de cerca dicotòmica permet fer una cerca d'un element sobre un vector ordenat amb una eficiència molt més gran que la que obteníem amb els algorismes de cerca seqüencial que hem vist en la secció anterior.

El fonament del mètode és prou intuïtiu: Si el vector té tamany n i està ordenat, es comença consultant

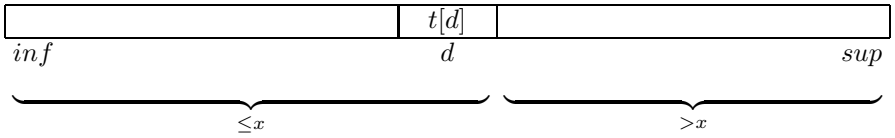


Figura 3.4: Postcondició de l'acció *cerca_dicotòmica*

el valor de l'índex que ocupa aproximadament la posició $n/2$. Si el valor associat a aquest índex és més gran que el que estem buscant, podem descartar tota la meitat de la dreta del vector i continuar la cerca en aquest nou vector més petit (el que va des de l'ındex 1 fins a $n/2-1$). Si el valor resultés ser més petit que el que cerquem, aleshores és la meitat esquerra la que podem descartar. A cada pas de l'algorisme, per tant, descartem la meitat dels elements del vector que considerem en aquell pas. Al final només haurem visitat un nombre d'elements de l'ordre del logaritme en base 2 del nombre total d'elements que conté el vector. En el cas de la cerca seqüencial visitàvem *tots* els elements del vector. Aquest és el motiu de la gran eficiència de l'algorisme de la cerca dicotòmica (e.g. si un vector té uns 1000 elements, si usem un algorisme de cerca seqüencial sobre ell, en el cas pitjor recorrerem 1000 elements. Si usem l'algorisme de la cerca dicotòmica, en recorrerem $log_2(1000) \approx 10$. Una bona diferència, oi?).

Notem que per desenvolupar aquest algorisme necessitem fer un accés directe a l'element d'ındex $n/2$ del vector (on n és el nombre d'elements sobre els quals es fa la cerca). Recordem que en el cas de la cerca seqüencial mai no feiem accessos directes a un índex qualsevol del vector. Els accessos eren seqüencials (sempre accedíem al *següent element* (després de visitar $v[k]$ visitàvem $v[k+1]$). Aquestes reflexions ens donen les dues claus per poder fer cerques dicotòmiques:

- Hem de tenir una estructura de dades amb els elements ordenats.
- Hem de tenir un accés directe a l'element que ocupa una posició qualsevol dins de l'estructura.

3.4.1 Especificació

Es tracta de dissenyar una acció *cerca_dicotòmica* amb la següent especificació:

$P = \{1 \leq inf \leq sup + 1 \leq N + 1 \wedge ordenat(v, inf, sup)\}$

cerca_dicotòmica(v: **ent** vector<natural>(N), inf: **ent** natural, sup: **ent** natural,
x: **ent** natural, d. **sort** natural);

$Q = \{\forall i : inf \leq i \leq d \bullet v[i] \leq x \wedge \forall i : d < i \leq sup \bullet v[i] > x \wedge inf - 1 \leq d \leq sup \wedge P\}$

on el predicat $ordenat(v, inf, sup)$ es defineix de la següent manera[2]:

$ordenat(v, i, j) = \forall k : i \leq k < j \bullet v[k] \leq v[k+1]$.

La postcondició Q estableix que la acció *cerca_dicotòmica* obté aquell índex d tal que el seu valor associat i tots els de la seva esquerra (fins a inf) són menors o iguals que x i tots els de la seva dreta (fins a sup) més grans que x. A la figura 3.4 es mostra gràficament aquesta idea. Per cert, noteu que en algun cas patològic, $d = inf - 1$. Quin seria aquell cas?

D'altra banda, P se satisfarà en qualsevol punt de l'execució de l'algorisme que dissenyarem tot seguit. Per aquest motiu considerarem aquesta informació implícita i, a partir d'ara, omitirem P tant a la postcondició com a l'invariant com a qualsevol asserció intermitja.

[2]Notem que la certesa del predicat *ordenat* ja assegura el fet que v estarà inicialment entre inf i sup.

acció cerca_dicotòmica(v, inf, sup, x, d)
inicialitzacions;
mentre B **fer**
S;
fmentre
falgorisme

Figura 3.5: Esquema de la funció *cerca_dicotòmica*

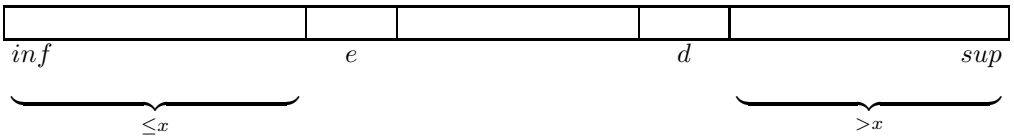


Figura 3.6: Expressió gràfica del nucli de l'invariant I

3.4.2 L'invariant

Intuïtivament, la idea bàsica per desenvolupar l'algorisme és mantenir en tot moment una part esquerra i una dreta del vector *ja tractades* (l'esquerra amb elements menors o iguals que x i la dreta amb elements més grans que x) i una part central per tractar. Ja haureu endevinat que precisament aquesta part central s'anirà fent menor a cada volta fins que a la fi ja no contingui cap element. Aquesta idea (que és, en realitat l'invariant del bucle) es pot expressar gràficament tal com apareix a la figura 3.6. Si formalitzem aquesta figura obtenim el següent invariant:

$I = \{\forall i : inf \leq i < e \bullet v[i] \leq x \wedge \forall i : d < i \leq sup \bullet v[i] > x \wedge$
$\quad inf - 1 \leq d \leq sup \wedge inf \leq e \leq sup + 1 \wedge e \leq d + 1\}$

Si volem obtenir I analíticament, podem reescriure Q de la manera següent:

$Q = \{\forall i : inf \leq i < d + 1 \bullet v[i] \leq x \wedge \forall i : d < i \leq sup \bullet v[i] > x \wedge inf - 1 \leq d \leq sup\}$

i, seguidament, podem substituir $d + 1$ per e. Notem que, si procedim d'aquesta manera, l'afitació d'e a l'invariant s'obté immediatament del rang que pot prendre la variable e que ha substituït $(d + 1)$.

El darrer conjuntant de I $(e \leq d+1)$ és, de fet, redundant ja que es pot deduir de la resta dels conjuntants de I. El fem explícit per motius de claredat.

Evidentment el segment del vector entre els índexs e i d representa la part inexplorada del mateix. Aquella que haurem de fer decréixer a cada nova volta.

L'acció *cerca_dicotòmica* que haurem de desenvolupar guiats per l'invariant que acabem de proposar seguirà l'esquema de la figura 3.5.

3.4.3 La inicialització

Per tal d'aconseguir que I sigui cert abans de la primera volta de la iteració cal fer les següents inicilitzacions:

$e := inf$; $d := sup$;

I s'assoleix per domini nul d'aplicació.

3.4.4 La condició de continuació B

A partir de $I \wedge \neg B \implies Q$, deduïm que allò que li manca a I per tal d'assolir Q és precisament $\neg B = \{e = d + 1\}$[3]. Efectivament, quan $e = d + 1$, I i Q diran exactament el mateix. Notem que aquesta condició es pot formular en llenguatge natural dient que *el bucle s'aturarà en el moment en què tots dos índexos es creuin. No abans.*

Consegüentment: $B = \{e \neq d + 1\}$.

Amb l'ajut de I $(e \leq d + 1)$, aquesta condició B la podem transformar en una altra d'equivalent: $B = \{e \leq d\}$.

3.4.5 La fita del bucle

Es tracta de fer més petit cada vegada el segment de vector inexplorat. Això és, la distància entre d i e. Per tant, proposem la fita $t = d - e + 1$, que coincidexi exactament amb el nombre d'elements inexplorats (els que ocupen la zona central del vector).

Notem que $I \wedge B \Longrightarrow t > 0$

3.4.6 El cos del bucle

Com ja hem dit, la novetat que aporta aquest problema és que podrem aprofitar el fet que el vector v està ordenat per a fer més ràpid el camí vers el final del bucle.

Efectivament, sigui $mig = (e + d)$ **div** 2

Notem que, com dins d'una iteració $e \leq d$, es tindrà que $e \leq mig \leq d$.

D'altra banda, com v està ordenat entre inf i sup, tindrem:

- Si $v[mig] > x \Longrightarrow \forall i : mig \leq i \leq sup : v[i] > x \wedge e \leq mig \leq d.$ [1]

  La intuïció aconsella fer, en aquest cas: $d := mig - 1$ ja que tots els índexos i, amb $i \geq mig$, segur que acompliran que $v[i] > x$, per la qual cosa no té sentit prosseguir la cerca en aquest espai. Notem que amb l'assignació $d := mig - 1$ fem disminuir la mida del nou subvector de cerca (ara es mourà entre e i $mig - 1$).

  Per tal de garantir la correctesa del bucle haurem de provar dues coses:

  1. La instrucció $d := mig - 1$ decrementa *en tots els casos* la fita del bucle $(t = d - e + 1)$.
     Això és cert ja que, com a l'inici de la volta $e \leq mig \leq d \Longrightarrow mig - 1 < d$. Per tant, l'assignació $d := mig - 1$ decrementa la fita i, en conseqüència, fa avançar vers la fi de la iteració.
     En aquest punt no ens podem estar de notar que la instrucció $d := mig$, que aparentment és similar i tan correcta com la que hem proposat, no garanteix el decrement de t (per què?. Posa'n un exemple).
  2. La instrucció $d := mig - 1$ manté l'invariant I.
     Per provar això, ens cal demostrar que

     $$T = \{e \leq mig \leq d \wedge I \wedge \forall i : mig \leq i \leq sup \bullet v[i] > x\}$$

     $$d := mig - 1;$$

     $$I = \{\forall i : inf \leq i < e \bullet v[i] \leq x \wedge \forall i : d < i \leq sup \bullet v[i] > x \wedge$$
     $$inf - 1 \leq d \leq sup \wedge inf \leq e \leq sup + 1 \wedge e \leq d + 1\}$$

[3] Notem que es tracta precisament de la substitució que hem fet per tal d'obtenir I.

és un algorisme correcte (notem que T és tot allò que podem suposar que és cert a l'inici d'una volta si $v[mig] > x$).

$I = \{\underbrace{\forall i : inf \leq i < e \bullet v[i] \leq x}_{(1)} \wedge \underbrace{\forall i : d < i \leq sup \bullet v[i] > x}_{(2)} \wedge$
$\underbrace{inf - 1 \leq d \leq sup}_{(3)} \wedge \underbrace{inf \leq e \leq sup + 1}_{(4)} \wedge \underbrace{e \leq d + 1}_{(5)}\}$

Veiem que, efectivament, cadascun dels 5 conjuntants de I: (1), (2), (3), (4) i (5) s'acompleixen després de l'assignació $d := mig - 1$.

- (1). Cert ja que el conjuntant (1) forma part de I, per tant, ja es complia a l'inici de la volta i l'assignació $d := mig - 1$ no l'ha modificat per a res.
- (2). Cert ja que ara $d = mig - 1$ però v està ordenat i $v[mig] > x$.
- (3). $T \Longrightarrow inf \leq mig \leq sup$ i, per tant, $inf - 1 \leq mig - 1 = d \leq sup$.
- (4). Cert perquè ja formava part de I (i, per tant es complia a l'inici de la volta) i no hem modificat en absolut e.
- (5). Es dedueix dels altres conjuntants de I.

- Si $v[mig] \leq x \Longrightarrow \forall i : inf \leq i \leq mig : v[i] \leq x \wedge e \leq mig \leq d.$ [2]

  Fent un raonament absolutament paral.lel a l'anterior es pot concloure que l'assignació $e := mig + 1$ avança vers la fi de la iteració i manté la certesa de l'invariant I.

Notem un cop més que mig sempre es referirà a un índex dins del rang del vector v (i.e. $inf \leq mig \leq sup$) i, per tant, $v[mig]$ sempre tindrà sentit.

Tot plegat, dóna lloc al següent algorisme de cerca dicotòmica:

acció cerca_dicotòmica(v, inf, sup, x, d)
  $e := inf; d := sup;$
  mentre $e \leq d$ **fer**
    $mig := (e + d)$ div 2;
    si $v[mig] \leq x \longrightarrow e := mig + 1;$
      $v[mig] > x \longrightarrow d := mig - 1;$
    fsi
  fmentre
ffunció

$P = \{1 \leq inf \leq sup + 1 \leq N + 1 \wedge ordenat(v, inf, sup)\}$

$Q = \{\forall i : inf \leq i \leq d \bullet v[i] \leq x \wedge \forall i : d < i \leq sup \bullet v[i] > x \wedge inf - 1 \leq d \leq sup \wedge P\}$

$I = \{\forall i : inf \leq i < e \bullet v[i] \leq x \wedge \forall i : d < i \leq sup \bullet v[i] > x \wedge$
$inf - 1 \leq d \leq sup \wedge inf \leq e \leq sup + 1 \wedge e \leq d + 1\}$

$t = d - e + 1$

3.5 La bipartició d'un vector

3.5.1 L'especificació

$\{P\} = \{1 \leq inf \leq sup + 1 \leq N + 1 \wedge t[inf..sup] = T[inf..sup]\}$

partició(t:**e/s** vector<natural>(N), inf:**ent** natual,sup **ent** natural,x:**ent** natural,k:**sort** natural)

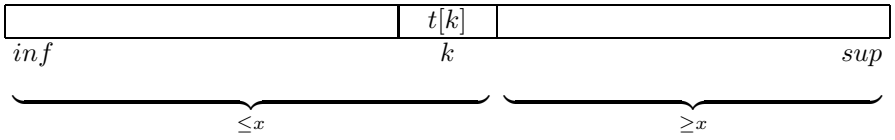


Figura 3.7: Postcondició de l'acció *partició*

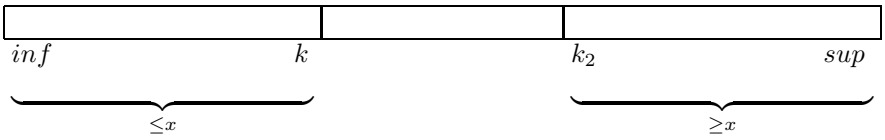


Figura 3.8: Expressió gràfica del nucli de l'invariant I

$$\{Q\} = \{inf - 1 \leq k \leq sup \wedge \\ \forall j : inf \leq j \leq k \bullet t[j] \leq x \wedge \\ \forall i : k + 1 \leq i \leq sup \bullet x \leq t[i]\}$$

3.5.2 L'invariant i la fita

$$\{I\} = \{inf - 1 \leq k \leq sup \wedge inf \leq k_2 \leq sup + 1 \wedge \\ \forall j : inf \leq j \leq k \bullet t[j] \leq x \wedge \\ \forall i : k_2 \leq i \leq sup \bullet x \leq t[i]\}$$

$$t = k_2 - (k + 1)$$

3.5.3 Desenvolupament de l'acció

```
acción partición(t,inf,sup,x,k)
  var  k2 : natural;
  k := inf − 1;
  k2 := sup + 1;
  mentre k2 ≠ k + 1 fer
    si t[k + 1] ≤ x ⟶ k := k + 1;
       t[k2 − 1] ≥ x ⟶ k2 := k2 − 1;
       t[k + 1] > x ∧ t[k2 − 1] < x ⟶
          intercanvi(t[k + 1], t[k2 − 1]);
          k := k + 1;
          k2 := k2 − 1;
    fsi
  fmentre
facció
```

3.6 La bandera republicana

Considerem una filera de n boles ($n \geq 0$). Les boles són de tres colors diferents: vermell, groc i violat[4]. Es tracta de derivar un algorisme de cost lineal que col.loqui primer les boles vermelles, després les grogues

[4]Que codificarem amb els números 1,2 i 3 respectivament.

i, finalment, les violades[5] .

A aquesta acció l'anomenarem *perm_rep* i, en primer lloc, l'especificarem:

t:vector $[1..N]$ de natural. Suposarem que t està inicialitzat entre $1..n$ ($n \leq N$) amb valors naturals entre 1 i 3.

$\{P\} = \{t[1..n] = T[1..n] \wedge 0 \leq n \leq N\}$
perm_rep(t,n)
$\{Q\} = \{\forall i : 1 \leq i \leq k_1 \bullet t[i] = 1 \wedge \forall i : k_1 + 1 \leq i \leq k_2 \bullet t[i] = 2 \wedge \forall i : k_2 + 1 \leq i \leq n \bullet t[i] = 3 \wedge 0 \leq k_1 \leq k_2 \leq n\}$

L'invariant que proposem per a derivar l'acció és el següent:

$\{I\} = \{\forall i : 1 \leq i \leq k_1 \bullet t[i] = 1 \wedge \forall i : k_1 + 1 \leq i \leq j \bullet t[i] = 2 \wedge \forall i : k_2 + 1 \leq i \leq n \bullet t[i] = 3 \wedge 0 \leq k_1 \leq k_2 \leq n \wedge k_1 \leq j \leq n\}$

Aquest invariant afebleix la postcondició perquè fixa la zona del vector ocupada pel color groc entre els índexos $k_1 + 1$ i j (on $k_1 \leq j \leq n$), mentre que la postcondició l'establia entre $k_1 + 1$ i k_2. Així doncs, l'invariant s'obté de la substitució de l'expressió de la postcondició k_2 per la variable j. És clar que la zona del vector compresa entre $j + 1$ i k_2 roman inexplorada. Queda com a exercici el desenvolupament de la composisció iterativa amb aquest invariant.

3.7 La inserció ordenada en un vector

3.8 La fusió de dos vectors ordenats

3.9 Permutacions lexicogràfiques

Problema 21

Variant de l'esquema de cerca: Cercar el primer element que compleixi una determinada propietat i aplicar un tractament concret a tots els anteriors. Per exemple, escriure tots els elements d'un vector fins al primer més gran que 100 (aquest no inclòs).

Fer-ho amb vectors amb marca de final i amb vectors caracteritzats per un rang.

Donar especificació, invariant, fita i codi de les noves accions.

Repetir el problema, però ara també volem escriure el primer més gran que 100.

Problema 22

Modificar l'acció de cerca seqüencial suposant que tenim la informació que el vector conté els elements ordenats. Fer la nova especificació, invariant, fita i codi.

Problema 23

Cerca amb *sentinella*. Hem vist que si sabem que l'element cercat és al vector, l'acció de cerca se simplifica. Una idea pot ser, doncs, col.locar abasns de començar la cerca l'element al final del vector (a

[5]Aquesta és una reformulació del clàssic problema de la bandera holandesa de Dijkstra.

l'índex $sup + 1$). Com quedarien llavors l'especificació, l'invariant, fita i codi de la nova acció de cerca?

Problema 24

Modificar l'algorisme *cerca3* de manera que B tingui només un conjuntant ($k \leq sup$)

Problema 25

Cerca logarítmica. La cerca dicotòmica es pot millorar utilitzant el següent principi: Quan cerquem una paraula al diccionari que comença per v, no obrim el diccionari per la meitat (com faríem aplicant l'algorisme de cerca dicotòmica) sinó per la part final (al contrari passaria si cerquèssim un mot que comenà per la lletra c.

1. Com modificaries l'algorisme de la cerca dicotòmica per a que es comporti d'aquesta manera?
2. Quina és la hipòtesi que s'ha de complir per tal que aquest nou algorisme suposi realment una millora respecte la cerca dicotòmica?

Problema 26

Donats dos vectors ordenats $t_1[1..n_1]$ i $t_2[1..n_2]$ ($n_1, n_2 \geq 0$) tals que no hi ha elements repetits a cap dels dos vectors, derivar un algorisme que compti el nombre d'elements que apareixen repetits a tots dos vectors (nig).

La postcondició d'aquest algorisme és:

$$Q = \{nig = \sum_{i,j \ \ t.q. \ \ t_1[i]=t_2[j]} 1\}$$

Problema 27

Donats dos vectors definits entre 1 i n de enters, ordenats, sense elements repetits i amb una marca de final igual a 999 (el darrer enter de cadascun dels dos vectors), es tracta de derivar una acció que fussioni ambdós vectors obtenint-ne un de nou ordenat, sense elements repetits i provist també de la marca de final.

Problema 28

Els algorismes d'ordenació d'un vector més *immediats* consisteixen bàsicament en tenir el vector dividit en dos parts: una primera ja ordenada i una segona per ordenar amb una sèrie d'elements que s'hauran d'anar afegint progressivament a la primera. A cada nova volta de l'algorisme s'afegeix un nou element de la part no ordenada del vector a la part ordenada (conservant, òbviament, l'ordre. La diferència entre els diferents mètodes *immediats* rau en els diferents criteris per anar construint la part ordenada del vector.

L'algorisme d'ordenació de vectors *de la bombolla* es caracteritza perquè a cada volta es comparen totes les parelles consecutives d'elements des de baix cap a dalt i s'intercanvien les que estan desordenades. D'aquesta manera, a cada volta, l'element més petit dels que encara no estaven ordenats puja a ocupar el seu lloc (com una bombolla deixada anar per un banyista dins de l'aigua) incorporant-se al final de la

part del vector ja ordenada. Es tracta de derivar l'acció *bombolla*. Cal tenir en compte que aquesta acció necessitarà dos bucles. La idea serà derivar primer l'extern i, posteriorment, l'intern.

Problema 29

L'algorisme d'ordenació de vectors anomenat *de selecció directa* consisteix en prendre a cada volta el més petit entre els valors encara no ordenats i col.locar-lo com a darrer dels que ja ho estan. Derivar una acció que resolgui el problema de l'ordenació d'un vector utilitzant aquest mètode. Novament sortiran dos bucles i s'hauran de derivar separadament començant per l'extern.

Problema 30

L'algorisme d'ordenació de vectors anomenat *d'inserció directa* consisteix en prendre el primer element del vector que no forma part de la seqüència ordenada i afegir-lo al lloc adequat de la seqüència que ja ha estat ordenada. De nou es demana la derivació d'aquesta acció i es recorda que, altre cop, sortiran dos bucles.

Problema 31 (Examen Juny-96

L'algorisme d'ordenació *quicksort* necessita una acció per a *bipartir* el vector que cal ordenar en una part que tingui tots els elements més petits o iguals que un determinat valor x i una altra amb tots els seus elements més grans o iguals que x. El propòsit d'aquest problema és fer un disseny per l'acció *partició* però amb algunes diferències respecte el disseny proposat durant el curs.

En particular, aquesta és l'especificació de l'acció que volem construir:

$\{P\} = \{1 \leq inf \leq sup \leq N \wedge \exists i, inf \leq i \leq sup : x = t[i]\}$

partició(t:**e/s** vector[1..N] de enter,inf:**ent** natural, sup **ent** natural,x:**ent** natural,k:**sort** natural)

$\{Q\} = \{inf \leq k \leq sup \wedge$
$\quad \forall j : inf \leq j \leq k : t[j] \leq x \wedge$
$\quad \forall j : k+1 \leq j \leq sup : x \leq t[j]\}$

Notem que la precondició requereix que el *valor de tall* x, estigui present al vector entre els índexos inf i sup, la qual cosa exclou haver de considerar un vector buit.

Seguidament fem una proposta incompleta de l'acció que volem dissenyar:

```
1   acció partició(t,inf,sup,x,k)
2     var k1,k2:natural;
3     k1:=inf; k2:=sup;
4     mentre B fer
5       mentre t[k1]<x fer k1:=k1+1; fmentre
6       {Q1}
7       <codi1>
8       {Q2}
9       <codi2>
10    fmentre
11    k:=<exp>;
12  facció
```

- B representa la condició de continuació de la iteració principal (línia 4).
- <codi1> i <codi2> són dos trossos de codi que hauràs de descobrir.
- <exp> és l'expressió que cal assignar a k a la línia 11 per tal d'assolir la postcondició Q de l'acció.

Es demana:

1. (0,75)Invariant I i fita t de la iteració principal (línia 4).

2. (0,25) Condició de continuació B de la iteració principal.

3. (0,25) Expressió <exp> que cal assignar a k a la línia 11.

4. (0,75) Invariant I_1 i fita t_1 de la iteració de la línia 5 i postcondició Q_1 a la que s'arriba en acabar l'execució d'aquesta iteració (línia 6).

5. (0,5) Contingut de <codi1>

6. (0,25) Enunciat Q_2 que s'assoleix després de l'execució de [codi1].

7. (0,75) Contingut de <codi2> que faci que $\{Q_2\} < codi2 > \{I\}$ sigui correcte.

Suggeriment: Per a fer bé aquest problema cal tenir en compte la següent reflexió: A la línia 5, podem garantir que en algun moment t[k1]≥x. Per què? Qui ho ha de garantir? Pensa també en quin ha de ser el rang de les variables $k1$ i $k2$.

Capítol 4

Disseny recursiu

4.1 Els Fonaments

Calcular el factorial de 6 (6!) resulta molt senzill si es coneix el factorial de 5 (120). Efectivament, n'hi ha prou amb multiplicar 5! (120) per 6 i s'obté ràpidament el valor de 6!=720. Podríem expressar aquest fet d'una forma més general:

$n! = n * (n-1)!$

D'aquesta manera descrivim el factorial de n en termes del factorial de $n-1$, el qual, a la vegada quedarà descrit en termes del factorial de $n-2$, i aquest, en termes del factorial de $n-3$... Si continuem amb aquesta sèrie arribarà un moment en què intentarem calcular el factorial de -10 descrivint-lo en termes del factorial de -11 *i encara no ens aturarem....* Aquesta reflexió ens porta a la conclusió que no podem descriure el factorial en termes de sí mateix eternament. *Ens hem d'aturar en algun moment*:

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

Imaginem que al vector v, entre els seus índexos 1 i 4, hi ha una cadena de caràcters que volem escriure per pantalla de forma invertida (si la cadena és "ROMA", volem escriure "AMOR"). Si ja haguéssim invertit la cadena entre els índexos 2 i 4 ("AMO") fóra molt senzill d'invertir la cadena total: Simplement hauríem d'escriure tot seguit el caràcter del primer índex:"R" i el resultat final fóra "AMOR". Novament hem descrit la resolució d'un problema en termes d'ella mateixa encara que amb uns parámetres menors i, novament, ens podem adonar que si no establim un primer cas en que el problema se solucioni directament no ens aturarem mai.

$$invertir(cadena, index\_ini, index\_fi) = \begin{cases} invertir(cadena, index\_ini + 1, index\_fi); & \\ escriure(cadena[index\_ini]); & \text{si } index\_ini < index\_fi \\ & \\ escriure(cadena[index\_ini]) & \text{si } index\_ini = index\_fi \end{cases}$$

Definició 9

> *Es diu que la solució d'un problema està donada en forma recursiva si està descrita de la següent manera:*
>
> $$solució(problema) = \begin{cases} co(solució(problema\_de\_tamany\_menor)) & \textit{cas recursiu} \\ solució\_no\_recursiva & \textit{cas trivial} \end{cases}$$

A la definició anterior *co* significa *la resta de coses que es fan acompanyant la crida recursiva.* A l'exemple del $factorial(n)$, *co* seria la multiplicació per n del resultat de la crida recursiva $factorial(n-1)$. A

l'exemple de la inversió d'una cadena, *co* seria l'escriptura del primer caràcter de la cadena. D'altrabanda, el *cas recursiu* abarcaria tots aquells valors dels paràmetres que donen lloc a una nova crida recursiva mentre que el *cas trivial* estaria format per tots aquells valors dels paràmetres pels quals es pot calcular directament la solució sense necessitat de noves crides recursives.

Al disseny d'algorismes basat en l'anàlisi feta als dos exemples anteriors se l'anomena *disseny recursiu*. A les accions i funcions desenvolupades utilitzant aquest tipus de disseny se les anomena *accions i funcions recursives*. Tot seguit presentem dues accions recursives que poden resoldre els dos problemes plantejats i que es dedueixen de manera immediata de l'anàlisi feta.

$natural(n)$
$P = \{n \geq 0\}$

funció factorial(n:natural) **retorna** natural **és**
  si $n = 0$ **llavors** retorna 1;
    sinó retorna $n*$factorial$(n-1)$;
  fsi
ffunció

$Q = \{\text{factorial}(n) = n!\}$

par:**vector**[1..n] de caràcter
$natural(i_1, i_2)$
$P = \{1 \leq i_1 \leq i_2 \leq n \wedge par[i_1..i_2] = P[i_1..i_2]\}$

acció invertir (par, i_1, i_2) **és**
  si $i_1 = i_2$ **llavors** escriure($par[i_1]$);
    sinó
      invertir($par, i_1 + 1, i_2$);
      escriure($par[i_1]$);
  fsi
facció

$Q =$\{S'ha escrit per pantalla la cadena *par* entre els índexos i_1 i i_2 en ordre invers\}

El tipus de raonament i anàlisi que cal fer per tal de desenvolupar una acció o funció recursiva és, essencialment, el que s'ha descrit als exemples i pot resumir-se en els següents punts:

1. **Esbrinar com es pot descriure la solució en termes de la solució al propi problema per a paràmetres de tamany menor en algun sentit.**
2. **Trobar un cas base (o trivial) en què la solució no vingui donada de forma recursiva i tal que estigui garantit que en algun moment s'arribarà a aquell cas base.**

És freqüent que els esperits inquiets es preguntin *com és possible que la recursivitat pugui funcionar en un ordinador?* o, dit d'una altra manera, *quina seqüència d'instruccions executa l'ordinador per tal de donar com a resultat el comportament recursiu?* . Aquesta pregunta és més important de cara a fer transformacions de disseny recursiu en iteratiu (veure apartat 4.7) que de cara a fer disseny de funcions recursives. De tota manera, per anar fent boca, a les figures 4.1 i 4.10 es mostra la seqüència d'instruccions que executen les accions recursives $factorial$ i $invertir$.

Observant les figures 4.1 i 4.10 també ens podem adonar d'un detall important: L'execució completa de les accions que contenen una sola crida recursiva genera una seqüència de crides recursives. Aquest fet contrasta amb el cas de les accions que executen més d'una crida recursiva en el seu codi. Aquestes

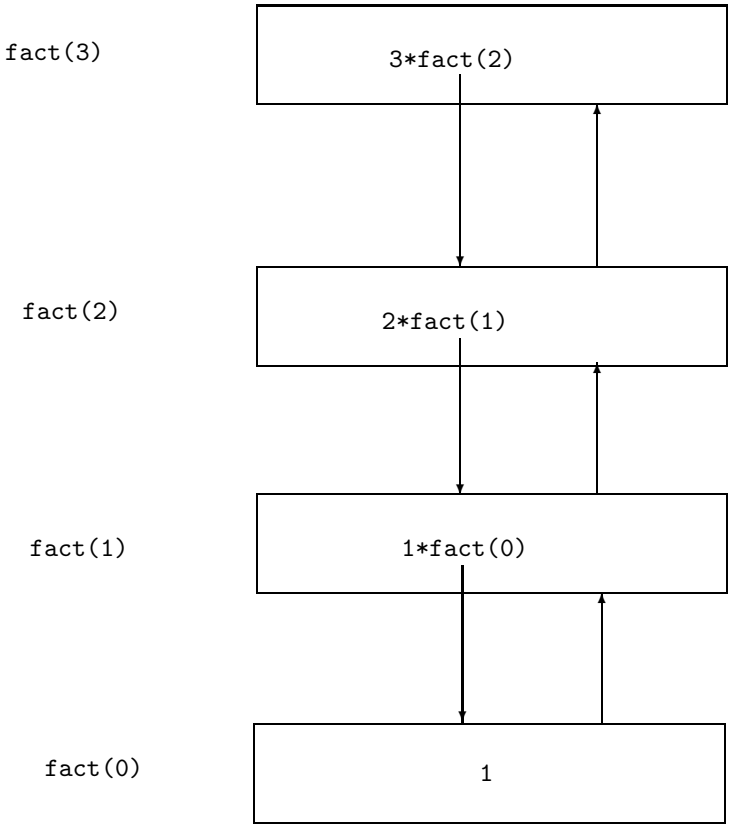


Figura 4.1: Seqüència d'instruccions de la funció factorial

accions generen un arbre de crides recursives. Això tindrà una conseqüència important a l'hora d'avaluar el cost d'una acció recursiva (en general, les accions recursives amb una sola crida són menys costoses que les que contenen més d'una crida recursiva) i a l'hora de transformar una acció recursiva en una altra d'iterativa. Insistirem sobre aquest tema als apartats 4.4, 4.6 i 4.7.

4.2 Raonament sobre la correctesa d'una acció recursiva

4.2.1 Aproximació intuitiva

Considerem la funció $factorial$:

$natural(n)$
$P = \{n \geq 0\}$
funció factorial(n:natural) **retorna** natural **es**
  si $n = 0$ **llavors** retorna 1;
        sinó retorna $n*$factorial$(n-1)$;
  fsi
ffunció
$Q = \{\text{factorial}(n) = n!\}$

Veiem intuitivament què és tot allò que s'ha de complir per a poder garantir que la funció $factorial(n)$ satisfà la seva especificació:

1. *Sempre que comença l'execució de la funció i se satisfà la seva precondició, es compleix un dels dos casos previstos a la instrucció condicional.*

   O sigui, $\{n \geq 0\} \Longrightarrow n > 0 \vee n = 0$. La qual cosa és evident.

2. *Si l'execució de la funció genera una nova crida recursiva, aquesta crida satisfà la seva precondició.*

   O sigui, $\{n \geq 0\} \wedge n > 0 \Longrightarrow \{n - 1 \geq 0\}$

   Aquest enunciat cal llegir-lo de la següent manera:

   Si es compleix la precondició per la crida amb paràmetre n ($n \geq 0$) i estem en un cas recursiu ($n > 0$) aleshores es complirà la precondició de la crida amb paràmetre $n - 1$ ($n - 1 \geq 0$) La qual cosa torna a ser veritat de manera evident.

3. *Si una crida a la funció es fa amb un paràmetre tal que satisfà el cas trivial del condicional, després d'executar-se l'acció no recursiva s'obté la postcondició (resultat) esperada de la funció.*

   O sigui: $n \geq 0 \wedge n = 0 \Longrightarrow 1 = n!$

   La qual cosa és certa ja que $n = 0$.

4. *Si una crida $f(n)$ a la funció es fa amb un paràmetre n tal que satisfà el cas recursiu del condicional i el resultat de la crida recursiva que es genera ($f(n-1)$) satisfà la postcondició d'aquella crida, aleshores el resultat de $f(n)$ també satisfà la seva postcondició.*

   Aquest enunciat, que pot semblar excessivament intrincat, l'escriuríem de la següent manera en el cas de la funció $factorial$:

   $$\underset{(1)}{\{n \geq 0\}} \wedge \underset{(2)}{n > 0} \wedge \underset{(3)}{factorial(n-1) = (n-1)!} \Longrightarrow \underset{(4)}{factorial(n-1) * n = n!}$$

   El conjuntant (1) és la precondició de la crida $f(n)$, el (2) és la condició del cas recursiu i el (3) és la hipòtesi de que la crida recursiva $f(n-1)$ aconsegueix el resultat esperat. Finalment, (4) és la postcondició de la funció substituint $factorial(n)$ directament pel que retorna la funció ($factorial(n-1) * n$).

   Si $factorial(n-1) = (n-1)! \wedge n > 0$ és clar que $factorial(n-1) * n = n!$ i, en conseqüència, la funció $factorial$ assolirà la postcondició també en el cas no recursiu[1].

   Aquesta propietat expressa l'esperit del disseny recursiu. El correcte disseny recursiu es basa en el llençament d'una crida recursiva, la hipòtesi de que el resultat que calcula aquesta crida és correcte i, finalment, la compleció d'aquell resultat per a obtenir el de la crida original.

5. *Cada nova crida recursiva es fa amb uns paràmetres de tamany estrictament menor en algun sentit que els de l'anterior crida. A més, aquest tamany és sempre més gran o igual a zero*

   En el cas del factorial, allò que es fa petit en cada nova crida coincidiria exactament amb el paràmetre. La crida feta amb paràmetre n en generaria una altra amb paràmetre $n - 1$, aquesta una altra amb paràmetre $n - 2$... fins arribar a una crida amb paràmetre 0.

   La successió de crides fóra la següent:

   $n, n-1, n-2, ..., 2, 1, 0$

[1] La propietat 4 ens garanteix que $factorial(n)$ (per $n > 0$) funciona bé si $factorial(n-1)$ funciona bé. Els esperits incrèduls segurament preguntaran ara: però qui ens garanteix que $factorial(n-1)$ funcionarà bé? La resposta és: $factorial(n-2)$ ens ho garanteix. Si continuem amb aquest raonament, obtindrem que $factorial(0)$ assegura el bon funcionament de $factorial(1)$ i (finalment), $factorial(0)$ és correcte per sí mateix.

$P(x)$
funció $f(x : T_1)$ **retorna** T_2 **és**
  si
    $b_t(x)$ $\longrightarrow$retorna $tr(x)$
    $b_r(x)$ $\longrightarrow$retorna $co(f(suc(x)), x)$
  fsi
ffunció
$Q(x, f(x))$

Figura 4.2: Esquema abstracte d'una funció recursiva amb una sola crida

En el cas de l'acció *invertir*, el tamany vindria donat per la diferència entre els dos índexos del vector que es passa com a paràmetre. Com en el cas anterior aquesta diferència es faria a cada nova crida una unitat més petita.

Aprofitem aquest punt per a introduir una reflexió:

Als dos exemples presentats, cada nova crida recursiva es fa amb un tamany una unitat més petit que l'anterior. Així doncs, és evident que si comencem amb tamany n, necessitarem n crides recursives per arribar-ne a una de tamany 1. Si cada nova crida recursiva es fes amb un tamany igual a la meitat de l'anterior, necessitaríem $log(n)$ crides per a arribar-ne a una de tamany 1 i la funció recursiva fóra, en conseqüència, bastant més eficient.

En definitiva, amb les dues primeres condicions establim que les crides es faran sempre amb paràmetres correctes: per un costat satisfaran algun dels casos de la instrucció condicional i, per l'altre, compliran la seva precondició. Les condicions 3 i 4 es deriven de l'anàlisi feta i justifiquen que si la funció acaba, assoleix realment la seva postcondició. Per últim, la condició 5 garanteix que la seqüència de crides es fa cada cop amb uns paràmetres de tamany menor en algun sentit i que aquesta seqüència arriba en algun moment a un valor que es correspon amb un cas trivial pel qual no es fa una nova crida recursiva i la funció acaba la seva execució.

Amb el compliment dels quatre primers punts s'assoleix la *correctesa parcial* de la funció recursiva. Dit d'una altra manera, es garanteix que si la funció acaba, ho fa de forma correcta. El darrer punt assegura que la funció acabarà. Tots cinc punts completen la *correctesa total* de la funció recursiva.

És interessant d'adonar-se que el tipus de raonament que ens porta a dissenyar i verificar una acció recursiva és del tot idèntic al que utilitzem per a fer demostracions per inducció. Recordem que si volem demostrar per inducció que una propietat $P(n)$ és certa per a qualsevol nombre natural n (e.g. La suma de tots els naturals menors o iguals que n val exactament $\frac{n+n^2}{2}$ per a qualsevol $n \geq 1$) és procedeix de la següent manera:

- Demostrar que $P(1)$ és cert.
  En aquest cas, això vol dir que $1 = \frac{1+1}{2}$.
- Suposar cert que $P(n-1)$ és cert.
  O sigui, suposar que $\sum_{i=1}^{n-1} i = \frac{(n-1)+(n-1)^2}{2}$
- Demostrar que $P(n)$ és cert.
  En el nostre cas, $\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n = \frac{(n-1)+(n-1)^2}{2} + n = \frac{n+n^2}{2}$

4.2.2 La formalització

Anem, en primer lloc, a estudiar quina és la forma general d'una funció recursiva i què significa cadascuna de les seves parts. Aquesta forma general apareix a la figura 4.2.

- f és la funció que volem calcular. Es defineix:
  $$\begin{array}{rccc} f: & T_1 & \longrightarrow & T_2 \\ & x & \longrightarrow & f(x) \end{array}$$

- $suc(x)$ és la funció que retorna el paràmetre de la següent crida recursiva a una que té com a paràmetre x. Podríem anomenar-la *funció successor.*

  $$\begin{array}{lccc} suc: & T_1 & \longrightarrow & T_1 \\ & x & \longrightarrow & suc(x) \end{array}$$

  L'aplicació i vegades de la funció suc sobre el paràmetre x es nota $suc^i(x)$ i produeix com a resultat el valor del paràmetre amb el que es fa la i-èssima crida recursiva si no s'ha arribat abans al cas trivial.

  L'existència de la funció successor és obligatòria i inherent a la pròpia recursivitat. Recordem que cada nova crida recursiva cal fer-la amb un paràmetre estrictament menor *en algun sentit* que el de l'anterior.

  Al cas del factorial, $suc(n) = n - 1$.

  Finalment notarem $x = suc^0(x)$.

- b_t És la guarda del condicional al cas trivial.

  $$\begin{array}{lccc} b_t: & T_1 & \longrightarrow & boole\text{à} \\ & \text{x} & \longrightarrow & b_t(x) \end{array}$$

  Al cas del factorial, $b_t(n)$ és $n = 0$.

- b_r És la guarda del condicional al cas recursiu.

  $$\begin{array}{lccc} b_r: & T_1 & \longrightarrow & boolea \\ & \text{x} & \longrightarrow & b_r(x) \end{array}$$

  Al cas del factorial, $b_r(n)$ és $n > 0$.

- $tr(x)$ és la funció que retorna el resultat de f quan es crida amb un paràmetre que forma part dels casos trivials.

  $$\begin{array}{lccc} tr: & T_1 & \longrightarrow & T_2 \\ & x & \longrightarrow & tr(x) \end{array}$$

  L'existència d'aquesta funció també és obligatòria. En cas contrari es produiria una recursivitat infinita.

  En el cas del factorial tr és la funció constant 1: $tr(n) = 1$

- $co(f(suc(x)), x)$ representa tot allò que cal fer després d'obtenir el resultat de la crida recursiva $f(suc(x))$ per tal d'obtenir el resultat de $f(x)$.

  $$\begin{array}{lccc} co: & T_2, T_1 & \longrightarrow & T_2 \\ & f(suc(x)), x & \longrightarrow & co(f(suc(x)), x) \end{array}$$

  Els dos paràmtetres de la funció co representen allò que coneixem en el moment de l'aplicació de la funció co:

  - Per un costat el paràmetre amb el que s'ha fet la crida original a f: x,
  - per un altre, el resultat de la crida recursiva $f(suc(x))$.

  En el cas del factorial: $co(factorial(n-1), n) = factorial(n-1) * n$

Podem descriure el comportament d'aquesta funció abstracta de la forma que es mostra a la figura 4.3.

Aquest esquema l'hem d'interpretar de la següent manera:

Volem obtenir el resultat d'aplicar la funció f al paràmetre x. Per això calculem primer suc(x) i, recursivament, f(suc(x)). A partir d'aquest resultat, obtindrem f(x) fent co(f(suc(x)),x).

Amb tot això, les cinc condicions que hem presentat intuitivament a l'inici de l'apartat, les podem reescriure ara més formalment:

Definició 10

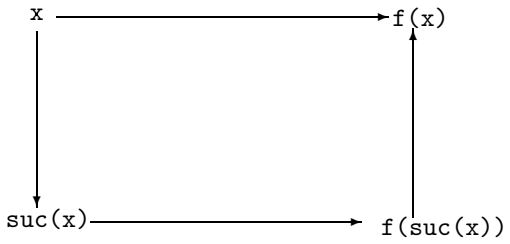


Figura 4.3: Esquema de funcionament d'una funció recursiva

Direm que la funció recursiva de la figura 4.2 és correcta sii es compleixen les següents condicions:

1. $P(x) \Longrightarrow b_t(x) \vee b_r(x)$
2. $P(x) \wedge b_r(x) \Longrightarrow P(suc(x))$
3. $P(x) \wedge b_t(x) \Longrightarrow Q(x, tr(x))$
4. $P(x) \wedge b_r(x) \wedge Q(suc(x), f(suc(x))) \Longrightarrow Q(x, co(x, f(suc(x)))$
5. *Existeix una funció tamany, especificada:*
   $tamany: T_1 \longrightarrow naturals$
   i que compleix:
   - $P(x) \wedge b_r(x) \Longrightarrow tamany(suc(x)) < tamany(x)$.
   - $P(x) \wedge b_r(x) \Longrightarrow tamany(x) > 0$

Una manera més general que la que hem proposat nosaltres per a demostrar l'acabament d'una acció recursiva, consisteix en definir una mena d'ordenació entre els paràmetres de les successives crides recursives per la qual no existeixin successions descendents infinites. L'estructura algebraica que suporta això s'anomena *preordre ben fonamentat*. Així doncs, per tal de garantir l'acabament d'una acció recursiva cal trobar una relació $\leq_{pbf}$ sobre els paràmetres de les successives crides recursives que sigui un preordre ben fonamentat.

4.3 Exemples

4.3.1 Càlcul del m.c.d. usant l'algorisme d'Euclides

Anàlisi

L'Algorisme d'Euclides estableix:

$$mcd(x,y) = \begin{cases} mcd(x-y,y) & \text{si } x > y \\ mcd(x,y-x) & \text{si } x < y \\ x & \text{si } x = y \end{cases}$$

De forma natural, l'Algorisme d'Euclides condueix a un disseny recursiu.

Disseny

A partir de l'anàlisi feta a l'apartat anterior podem plantejar el següent algorisme:

natural(x, y)
$P(x, y) = \{x > 0 \wedge y > 0\}$

m:=maxcomdiv(x, y);
$Q(x, y, m) = \{m = mcd(x, y)\}$

funció maxcomdiv (**ent** x:natural, **ent** y:natural) **retorna** natural **és**
  si
    x>y $\longrightarrow$ **retorna** maxcomdiv(x−y,y);
    x<y $\longrightarrow$ **retorna** maxcomdiv(x,y−x);
    x=y $\longrightarrow$ **retorna** x;
  fsi
ffunció

Verificació

Veiem que l'algorisme que hem presentat és efectivament correcte. Per això comprovarem les cinc propietats que caracteritzen els algorismes recursius correctes.

1. $P(x, y) \Longrightarrow x > y \vee x < y \vee x = y$ Trivial.

2. Per a comprovar que les crides recursives es fan amb paràmetres correctes, haurem de comprovar dues coses:

   - $P(x, y) \wedge x > y \Longrightarrow P(x - y, y) = \{x - y > 0 \wedge y > 0\}$
   - $P(x, y) \wedge x < y \Longrightarrow P(x, y - x) = \{y - x > 0 \wedge x > 0\}$

   Les dues coses són certes.

3. Al cas trivial, el resultat de la funció és l'esperat.

   $P(x, y) \wedge x = y \Longrightarrow x = mcd(x, y)$

   La qual cosa també és certa ja que el mcd de dos naturals iguals és qualsevol dels dos, segons estableix l'algorisme d'Euclides.

4. Al cas recursiu, si la crida recursiva generada produeix el resultat esperat, aleshores la crida inicial també el produeix.

   Com hi ha dos casos recursius, caldria fer dues comprovacions molt semblants. Així doncs, només farem la primera:

   $P(x, y) \wedge x > y \wedge maxcomdiv(x - y, y) = mcd(x - y, y) \Longrightarrow maxcomdiv(x, y) = mcd(x, y)$

   Efectivament això és cert ja que si $P(x, y) \wedge x > y$, aleshores, el valor que retorna $maxcomdiv(x, y)$ és precisament $maxcomdiv(x-y, y)$. Per hipòtesi suposem que aquest valor és igual al $mcd(x-y, y)$ i, segons l'algorisme d'Euclides, si $x > y$, $mcd(x - y, y) = mcd(x, y)$.

   En conseqüència, si $x > y$, certament $maxcomdiv(x, y) = mcd(x, y)$. De forma similar obtindríem el cas $x < y$.

   Amb això concloem que si la funció acaba, ho fa correctament. Anem ara a justificar que efectivament la funció *maxcomdiv* acabarà.

5. Cada nova crida recursiva es fa amb uns paràmetres estrictament menors *en algun sentit* que els paràmetres de la crida anterior.

   En aquest cas, la funció *tamany* podria ser:

   $tamany(x, y) = màxim(x, y)$.

   La qual compleix de forma evident:

   $P(x, y) \wedge x > y \Longrightarrow tamany(x - y, y) < tamany(x, y)$
   i també

$P(x,y) \wedge x < y \Longrightarrow tamany(x, y-x) < tamany(x,y)$

A més a més, $P(x,y) \Longrightarrow maxim(x,y) > 0$

Cost

El cas pitjor per l'algorisme d'Euclides fóra un que es fes amb paràmetres: $maxcomdiv(n,1)$. En aquest cas caldria fer un total de $n-1$ crides recursives fins a arribar a un cas trivial ja que cada nova crida decrementaria el primer paràmetre en una unitat i deixaria intacte el segon (1). Així doncs, el cost de l'algorisme d'Euclides és proporcional al tamany del màxim dels seus paràmetres. Dit d'una altra manera, el seu cost és $O(n)$, on n és el màxim dels valors dels paràmetres de la funció.

4.3.2 La divisió entera

Es tracta de fer una divisió entera en una màquina que permet només les operacions $+, -, *2,$ **div** 2.

L'especificació de la funció seria la següent:

n, d, q, r:natural
$P(n,d) = \{n \geq 0 \wedge d > 0\}$
divisio(n, d, q, r)
$Q(n,d,q,r) = \{n = q * d + r \wedge 0 \leq r < d\}$

A la postcondició queda clar que n és el dividend, d el divisor, q el quocient i r el residu. n i d són paràmetres d'entrada, q i r són paràmetres de sortida.

Anàlisi

Obtenir el quocient(q) d'una divisió entera vol dir *obtenir el nombre de vegades que el dividend conté al divisor*. Obtenir el residu(r) d'una divisió entera vol dir *esbrinar quantes unitats del dividend "sobren" després de restar-li q vegades el divisor.* A partir d'aquesta reflexió (prou bàsica, per altre costat) podem concloure la següent anàlisi recursiva:

$$q = n \textbf{ div } d = \begin{cases} (n-d) \textbf{ div } d + 1 & \text{si } n \geq d \\ 0 & \text{si } n < d \end{cases}$$

pel que fa al quocient i

$$r = n \textbf{ mod } d = \begin{cases} (n-d) \textbf{ mod } d & \text{si } n \geq d \\ n & \text{si } n < d \end{cases}$$

pel que fa al residu.

Disseny

A partir de l'anàlisi feta a l'apartat anterior es pot pensar amb el següent disseny:

acció divisió (n:**ent** natural, d:**ent** natural, q:**sort** natural, r:**sort** natural) **és**
  var r_1, q_1:natural **fvar**
  si
    $n < d$ **llavors** $q := 0$; $r := n$;
    $n \geq d$ **llavors** divisió $(n-d, d, q_1, r_1)$;
                $q := q_1 + 1$;
                $r := r_1$;

fsi
facció

Verificació

Veiem ara que l'acció proposada a l'apartat anterior és correcta:

1. Sempre es compleix un dels casos de la instrucció condicional: $P(n,d) \Longrightarrow n \geq d \vee n < d$ Evident.
2. Els paràmetres de cada nova crida compleixen la precondició de la crida:
   $P(n,d) \wedge d \leq n \Longrightarrow P(n-d,d)$. O sigui:
   $n \geq 0 \wedge d > 0 \wedge d \leq n \Longrightarrow n - d \geq 0 \wedge d > 0$
   És cert.
3. Si se satisfà el cas trivial, el resultat de la funció assoleix la postcondició.
   $P(n,d) \wedge d > n \Longrightarrow n = 0 * d + n \wedge 0 \leq n < d$
4. Si se satisfà el cas recursiu, també s'obté la postcondició.
   Hem de veure:
   $P(n,d) \wedge n \geq d \wedge n - d = d * q_1 + r_1 \wedge 0 \leq r_1 < d \Longrightarrow n = d * (q_1 + 1) + r_1 \wedge 0 \leq r_1 < d$
   Notem que, de les dues coses que ens cal demostrar, la segona ($0 \leq r_1 < d$) ja es compleix a la hipòtesi. Per tant només caldrà demostrar la primera: $n = d * (q_1 + 1) + r_1$. Però és certa ja que $n - d = d * q_1 + r_1$ per hipòtesi.
5. Cada crida es fa amb uns paràmetres "menors" que l'anterior.
   $tamany(n,d) = n$.
   Aquesta funció és estrictament decreixent ja que la precondició de la crida garanteix que $d > 0$ i aleshores:
   $tamany(n,d) = n > tamany(n-d,d) = n - d$
   $P(n,d) \wedge n \geq d \Longrightarrow \quad tamany(n,d) > 0$

Després de la verificació de cadascuna de les 5 condicions, concloem que l'acció *divisió* és correcta.

Cost

Cada nova crida recursiva incrementa en una unitat el quocient començant per 0. Per tant, es faran $O(q)$ crides recursives. O sigui, $O(n/d)$ crides recursives cadascuna de les quals tindrà un cost constant. Notem que el cas pitjor es produïra per $d = 1$ ($O(n)$).

En aquest punt ens podem plantejar la següent qüestió: Què passaria si a cada nova crida recursiva decrementéssim el quocient a la meitat? La resposta sembla senzilla: Arribaríem més ràpidament a un cas trivial. Quant més de ràpid? Quantes crides recursives ens caldria fer sota aquesta hipòtesi? La resposta és de l'ordre de $O(log(q)) = O(log(n/d))$. Efectivament, si suposem que q és una potència de 2, i que en cada nova crida recursiva reduïm el seu valor a la meitat, necessitarem $O(log(q))$ crides recursives per a arribar al cas trivial $q = 0$ (notem que després de fer aproximadament $log(q)$ crides, es complirà que $q/2^i \approx 1$, si fem una crida recursiva més, arribarem al cas trivial).

En conseqüència passaríem d'un cost lineal $O(n/d)$ a un altre de logarítmic $O(log(n/d))$. En el cas pitjor $O(log(n))$.

Si el decrement del tamany dels paràmetres en una funció recursiva d'una sola crida és lineal i la part no recursiva de la crida té un cost constant, el nombre de crides recursives que es generen és logarítmic respecte el tamany dels paràmetres i, en conseqüència, el cost de la funció és molt més baix que en el cas en que el tamany dels paràmetres decreix d'una forma constant.

El repte que ens plantejarem ara és dissenyar una nova acció recursiva per implementar la divisió entera que tingui un cost logarítmic respecte el tamany dels paràmetres.

4.3.3 La divisió entera amb cost logarítmic

Mantindrem la mateixa especificació que en el cas anterior, o sigui:

$natural(n, d)$
$P(n, d) = \{n \geq 0 \wedge d > 0\}$

$divisio(n, d, q, r)$

$Q(n, d, q, r) = \{n = q * d + r \wedge 0 \leq r < d\}$

L'anàlisi

- $n < d \longrightarrow q := 0; r := n.$

  Usarem el mateix cas trivial que a l'exemple anterior.

- $n \geq d$

  Aquest serà el cas recursiu. Per tal d'arribar al cas trivial podem optar per dues possibilitats:

  1. Incrementar d
  2. Decrementar n.

  A l'exemple anterior hem triat decrementar n. Ara optarem per incrementar d. Aquest increment el podem fer de forma constant o lineal. Optarem per incrementar d linealment per tal d'aconseguir un cost logarítmic tal com ens hem proposat. Així doncs, podem pensar en fer una crida recursiva de la forma:

  divisio $(n, d * 2, q_1, r_1)$

  que, per hipòtesi, obtindria la postcondició esperada:

  $Q = \{n = (d * 2) * q_1 + r_1 \wedge 0 \leq r_1 < d * 2\}$ [1]

  A partir del resultat d'aquesta crida recursiva i de la postcondició que suposem que es complirà [1] hem d'arribar a la postcondició del programa:

  $Q = \{n = d * q + r \wedge 0 \leq r < d\}.$ [2]

  Per fer això, establirem unes relacions entre el parell (r_1, q_1) i el parell (r, q) per tal que es compleixi [2].

  La primera d'aquestes relacions lliga q_1 amb q i és molt senzilla d'establir:

  $q := 2 * q_1$. Després de fer això, [1] es converteix en:

  $Q = \{n = d * q + r_1 \wedge 0 \leq r_1 < d * 2\}$ [3]

  Ara l'única diferència entre l'enunciat que tenim [3] i el que volem aconseguir [2] està en el segon conjuntant:$0 \leq r_1 < d * 2$.

  - Si $0 \leq r_1 < d$, l'expressió [3] coincideix amb la postcondició de l'acció [2].
    En aquest cas, per tant, cal fer simplement: $r := r_1$.
  - Si $d \leq r_1 < 2 * d$ ens caldrà afegir una unitat més al quocient q i treure en conseqüència, d unitats al residu r_1.
    Efectivament, de l'expressió [3] obtenim:
    $\{n = d * q + \mathbf{d} + (r_1 - \mathbf{d}) \wedge 0 \leq r_1 - d < d\}$
    I, arranjant-ho una mica:
    $\{n = d * (q + 1) + (r_1 - d) \wedge 0 \leq r_1 - d < d\}$,
    que coincideix amb la postcondició [2] fent: $q := q + 1$ i $r := r_1 - d$

El Disseny

acció divisió (n:**ent** natural, d:**ent** natural, q:**sort** natural, r:**sort** natural) **és**
  var r_1, q_1:natural **fvar**
  si
    n<d **llavors** q:=0; r:=n;
    n≥d **llavors** divisió $(n, d*2, q_1, r_1)$;
                $q := q_1 * 2$;
                si
                  $r_1 \geq d \longrightarrow q := q + 1; r := r_1 - d$;
                  $r_1 < d \longrightarrow r := r_1$;
                fsi
fsi
facció

El cost d'aquesta acció és $O(log(n/d))$ ($O(log(n/d))$ crides i un cost constant cada crida).

Notem que, de la mateixa manera que al capítol de disseny d'algorismes iteratius raonàvem sobre la correctesa d'un algorisme en el mateix moment de la seva construcció, també podem fer el mateix quan construim algorismes recursius. L'algorisme que acabem de desenvolupar n'és un exemple. Ens queden, però encara alguns detalls per a acabar-ne de garantir la correctesa tals com el seu acabament o el fet que els paràmetres de cada crida recursiva compleixin la precondició d'aquella crida. La verificació d'aquests petits detalls queda com a exercici.

4.3.4 Algorisme xinès de la multiplicació

A partir de dos nombres enters positius o zero, voldrem obtenir, utilitzant l'algorisme xinès de la multiplicació, el seu producte.

$natural(x, y, z)$
$P(x, y) = \{x \geq 0 \wedge y \geq 0\}$

z:=producte(x, y);

$Q(x, y, z) = \{z = x * y\}$

Anàlisi

Les dues operacions aritmètiques que un ordinador pot fer amb major rapidesa són el producte i la divisió per 2. Això és així perquè en la representació en complement dos que l'ordinador fa dels enters, aquestes operacions equivalen a fer un desplaçament a l'esquerra o dreta respectivament. Aprofitant aquesta propietat podem reescriure el producte de dos naturals x i y en termes de multiplicacions i divisions per 2 exclusivament. Veiem-ho:

$x * y = (2 * x) * (y/2) =$

$$\begin{cases} (2*x)*(y \textbf{ div } 2) & \text{si } y \text{ parell} \\ (2*x)*((y \textbf{ div } 2) + 1/2) & \text{si } y \text{ senar} \end{cases} =$$

$$\begin{cases} (2*x)*(y \textbf{ div } 2) & \text{si } y \text{ parell} \\ (2*x)*(y \textbf{ div } 2) + x & \text{si } y \text{ senar} \end{cases}$$

Com a cas trivial podem establir que *si* $y = 0$ *aleshores* $x * y = 0$.

Notem que fent aquesta anàlisi hem reescrit el producte de x i y en termes del mateix producte amb dos arguments diferents: $x * 2$ i y **div** 2. Heus ací la recursivitat de la solució.

Disseny

L'anàlisi anterior ens porta ràpidament al següent disseny recursiu:

funció producte (x:**ent** natural, y:**ent** natural) **retorna** natural **és**
  si
    y=0 **retorna** 0
    senar(y) **retorna** producte(2*x, y **div** 2) +x
    y>0 ∧ parell(y) **retorna** producte(2*x, y **div** 2)
  fsi
ffunció

Hi ha problemes, com aquest mateix, que tenen una formulació recursiva natural. En aquests casos, les solucions recursives solen ser més clares i elegants que les seves equivalents iteratives. Malauradament, a vegades resulten més ineficients. Un bon exemple d'això es troba a l'apartat 4.4.1

Verificació

1. $x \geq 0 \wedge y \geq 0 \Longrightarrow y = 0 \vee senar(y) \vee (y > 0 \wedge parell(y))$

   Clarament és cert.

2. Hem de veure que es compleixen dues condicions, una per cada cas recursiu:

   - $P(x, y) \wedge senar(y) \Longrightarrow P(2 * x, y \textbf{ div } 2) = \{2 * x \geq 0 \wedge y \textbf{ div } 2 \geq 0\}$
   - $P(x, y) \wedge (y > 0 \wedge parell(y)) \Longrightarrow P(2 * x, y \textbf{ div}2) = \{2 * x \geq 0 \wedge y \textbf{ div } 2 \geq 0\}$

   Ambdues condicions són també clarament certes.

3. $P(x, y) \wedge y = 0 \Longrightarrow x * y = 0$

   Si estem en un cas trivial clarament assolim el resultat esperat ja que en aquell cas $y = 0$ i, en conseqüència, $x * y = 0$ que coincideix amb el resultat que retorna la funció en aquest cas.

4. Tindrem dos casos recursius que verificar:

   - $P(x, y) \wedge \wedge(y > 0 \wedge parell(y)) \wedge producte(2 * x, y \textbf{ div } 2) = 2 * x * (y \textbf{ div } 2) \Longrightarrow 2 * x * (y \textbf{ div } 2) = (x * y)$

     ja que en aquest cas la funció retorna exactament producte$(2 * x, y \textbf{ div } 2)$.

     Això és cert ja que $2 * x * (y \textbf{ div } 2) = 2 * x * (y/2) = x * y$ si y és parell.
   - $P(x, y) \wedge senar(y) \wedge producte(2 * x, y \textbf{ div } 2) = 2 * x * (y \textbf{ div } 2) \Longrightarrow producte(2 * x, y \textbf{ div } 2) + x = (x * y)$

     Fent un raonament similar al del cas anterior podem verificar aquesta condició:

     $producte(2 * x, y \textbf{ div } 2) + x = 2 * x * (y \textbf{ div } 2) + x =_{(1)} 2 * x * (y/2 - 1/2) + x = x * y$

     (1):Ja que y és senar.

   Notem que les passes 3 i 4 es deriven de l'anàlisi feta.

5. La funció *tamany* en aquest cas podria ser: $tamany(x, y) = y$

   Efectivament, la successió de tamanys és decreixent estrictament ja que $tamany(x, y) > tamany(2 * x, y \textbf{ div } 2)$ si $y \neq 0$.

   A més a més si estem en un cas recursiu ($y > 0$), aleshores, $tamany(y) > 0$.

6. La successió de tamanys: y, $y \textbf{ div}2$, $(y \textbf{ div } 2) \textbf{ div } 2, ...$ té un mínim en el valor 0 que coincideix amb l'únic cas trivial.

El cost

Com al problema de la divisió, els paràmetres decreixen linealment (a cada crida el tamany dels paràmetres és la meitat que en la crida anterior). Per tant es faran $log(y)$ crides recursives, cadascuna de les quals tindrà un cost constant. En definitiva, el cost de la funció serà $O(log(y))$

4.3.5 El Mètode de Horner per a avaluar polinomis de grau n

El mètode de Horner és un mètode per a avaluar polinomis que evita de calcular les potències successives de la variable. El mètode consisteix en el següent:

sigui $p(x) = a_0 + a_1 * x + a_2 * x^2 + a_3 * x^3 + ... + a_n * x^n$

$p(x) = a_0 + x * (a_1 + a_2 * x + a_3 * x^2 + ... + a_n * x^{n-1})$

I això es pot continuar de la mateixa manera:

$p(x) = a_0 + x * (a_1 + x * (a_2 + a_3 * x + ... + a_n * x^{n-2}))$

Cada cop dins del parèntesi queda un polinomi més senzill per a avaluar.

La representació del polinomi

Sigui $p(x)$ un polinomi a coeficients enters.

Sigui v un vector que representa el polinomi p. v està declarat de la següent manera:

var v:**vector**[0..MAX_GRAU] **de** enter

i_ini:0..MAX_GRAU; i_fi:0..MAX_GRAU;

fvar

El polinomi p es representarà al vector v seguint el següent criteri:

El polinomi es representarà entre els índexos i_ini i i_fi del vector. A l'índex i_ini del vector s'hi representarà el coeficient del terme independent del polinomi. A líndex i_fi del vector s'hi representarà el coeficient del terme de major grau del polinomi.

En definitiva, l'índex k ($i\_ini \leq k \leq i\_fi$) del vector v representa el coeficient del terme x^{k-i_ini} del polinomi.

Per exemple, el polinomi: $p(x) = 2x^3 + x^2 - 5$ es representaria a v des de l'índex 0 a l'índex 3 de la següent manera:

$v[0] = -5$, $v[1] = 0$, $v[2] = 1$, $v[3] = 2$, $v[i] = indefinit$ per $i > 3$.

Si aquest mateix polinomi p el volem representar en un vector v des de l'índex 2 fins l'índex 5, el contingut del vector seria el següent:

$v[0] = v[1] = indefinit$, $v[2] = -5$, $v[3] = 0$, $v[4] = 1$, $v[5] = 2$, $v[i] = indefinit$ per $i > 5$.

Es tracta d'avaluar aquest polinomi per un valor determinat de la x utilitzant el mètode de Horner. Concretament, l'especificació de la funció que cal dissenyar és la següent:

$natural(i\_ini, i\_fi, x)$
$v : vector[0..MAXGRAU]$ de enter
$val : enter$
$P = \{0 \leq i\_ini \leq i\_fi \leq MAXGRAU \wedge v$ representa el polinomi $p\}$
$val := avaluar(v, i\_ini, i\_fi, x)$
$Q = \{val = p(x)\}$

L'anàlisi

$a_0 + a_1 * x + a_2 * x^2 + ... + a_n * x^n = a_0 + x * (a_1 + a_2 * x + ... + a_n * x^{n-1})$

Amb l'expressió anterior descrivim l'avaluació d'un polinomi de grau n en termes de l'avaluació d'un altre polinomi de grau $n-1$. Així doncs, aquesta expressió constitueix el cas recursiu del problema. El cas trivial l'obtindrem quan el polinomi que calgui avaluar sigui de grau zero (estigui format només pel terme independent).

Veiem-ho més formalment:

$avaluar(v, i\_ini, i\_fi, x) =$

- $v[i\_ini] + x * avaluar(v, i\_ini + 1, i\_fi, x)$ si $i\_ini < i\_fi$.
- $v[i\_ini]$ si $i\_ini = i\_fi$.

El disseny

```
funció avaluar (v:ent vector[0..MAXGRAU] de enter, i_ini:ent natural, i_fi:ent natural,
        x:ent enter) retorna enter és
  si i_ini<i_fi
    llavors retorna v[i_ini]+x*avaluar(v,i_ini+1,i_fi,x);
    sinó retorna v[i_ini];
  fsi
ffunció
```

La verificació la deixem com a exercici.

El cost

Cada nova crida recursiva es fa amb un polinomi de grau una unitat menor. Com el cas trivial s'obté per un polinomi de grau 0, concloem que caldrà fer $O(n)$ crides recursives (on n és el grau del polinomi) per a completar la seva avaluació.

4.4 La recursivitat amb vàries crides

Tots els exemples de funcions recursives que hem desenvolupat fins ara tenien una característica comuna: *Generaven una sola crida recursiva en cada execució* . Aquestes funcions donaven lloc una seqüència de crides recursives.

En aquest apartat estudiarem el comportament d'accions recursives que generen *més d'una crida recursiva en cada execució.* Veurem que aquestes accions són més potents que les primeres però, a canvi, tenen un comportament *pitjor* en algun sentit: ja no generaran una seqüència de crides recursives, com feien les anteriors, sinó un *arbre de crides recursives.* amb la qual cosa el seu cost gairebé sempre augmentarà sensiblement.

4.4.1 Exemple 1: La successió de Fibonnacci

La successió de Fibonnacci es defineix de la següent manera:

- $F_1 = 1$
- $F_2 = 1$

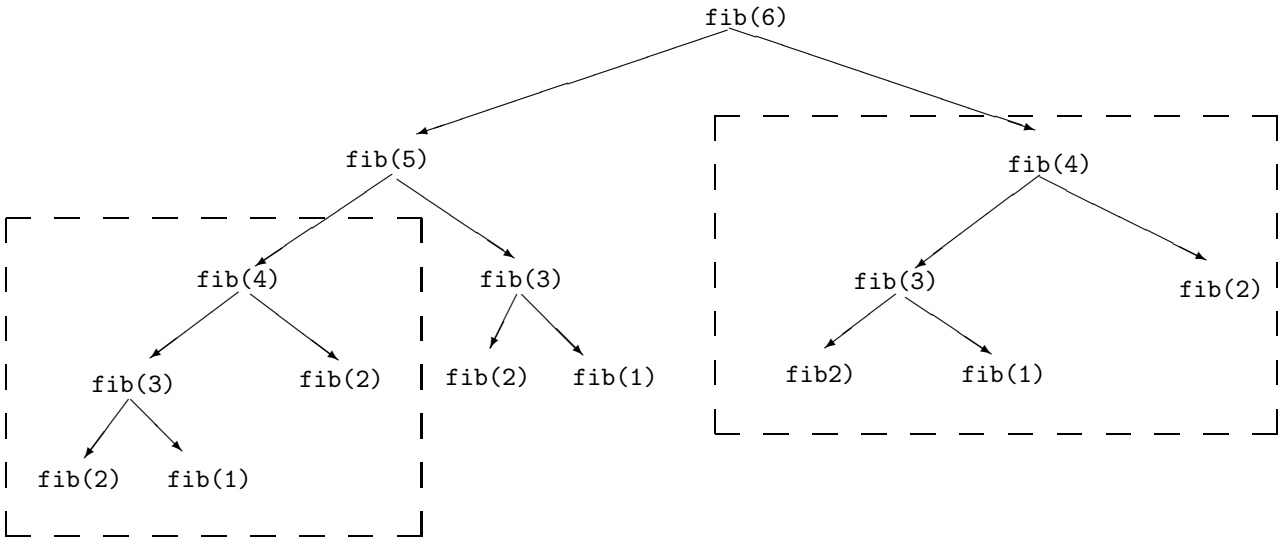


Figura 4.4: Arbre de crides de la funció *fibo*

- $F_n = F_{n-1} + F_{n-2}$ per $n > 2$.

I aquesta manera dóna lloc de forma molt natural a la següent funció recursiva per a generar el terme n-èssim de la successió de Fibonacci:

$natural(n)$
$P = \{n > 0\}$

funció fibo (n:natural) **retorna** natural **és**
  si
    n=1 **retorna** 1;
    n=2 **retorna** 1;
    n>2 **retorna** fibo$(n-1)$+fibo$(n-2)$;
  fsi
ffunció

$Q = \{fibo(n) = F_n\}$

Sense dubte aquesta és la manera més clara, més senzilla i més elegant de programar l'obtenció del terme n-èssim de la successió de Fibonacci. També és la més ineficient.

Per veure el comportament d'aquesta funció, simularem l'execució de la crida $fibo(6)$:

fibo(6) genera les crides fibo(5) i fibo(4). fibo(5) genera a la seva vegada fibo(4) i fibo(3)... A la figura 4.4 apareixen totes les crides que genera fibo(6).

El primer que podem observar a la figura és que la funció $fibo$ ja no genera una seqüència de crides recursives sinó més aviat un arbre de crides amb la qual cosa, el nombre de crides recursives ja no és proporcional a n, com passava fins ara en el pitjor dels casos, sinó a 2^n: Hem construït una funció de cost exponencial!!!

Efectivament el cost de la funció fibo és $O(2^n)$.

Si pensem en la solució iterativa òbvia al problema, ens adonarem que el seu cost és lineal respecte n ($(O(n))$:

```
funció fibo_i (n:natural) retorna natural és
  var fact,fant,i,temp:natural fvar
  fact:=1; fant:=1;
  per i:=3 a n fer
    temp:=fact+fant;
    fant:=fact;
    fact:=temp;
  fper
  retorna fact;
ffunció
```

En aquest punt pot ser interessant analitzar per quin motiu la solució recursiva augmenta tan ostensiblement el seu cost respecte la solució iterativa. La resposta és clara: *La solució recursiva repeteix vàries vegades el càlcul d'un terme de la successió.* A la figura 4.4, hem assenyalat dos arbres complets que es repeteixen més d'una vegada: els que fan el càlcul de $fibo(4)$. En definitiva podem observar que el càlcul de fibo(4) es repeteix dues vegades i el de fibo(3) es repeteix 3 vegades amb tot el cost addicional que això suposa.

En aquest exemple hem vist que la solució recursiva amb vàries crides era molt pitjor que la solució iterativa. Aquesta constatació ens podria fer caure en el desànim i portar-nos a pensar que si la solució que se'ns acudeix per a un problema utilitza una recursivitat múltiple (vàries crides recursives per execució) l'hem de rebutjar perquè el seu cost serà molt alt. Però això no és així:

- Per un costat existeixen problemes pels quals no es coneix una solució més eficient que l'exponencial. En aquests casos una solució recursiva exponencial que, a més resolgui el problema amb elegància, pot ser perfectament admissible.
- Per altre costat, funcions recursives amb vàries crides no sempre donen lloc a solucions de cost exponencial.

Tot seguit presentarem un exemple (les torres de Hanoi) en què la solució amb recursivitat múltiple és òptima. A l'apartat 5.1 descobrirem programes recursius amb recursivitat múltiple que no tenen un cost exponencial.

4.4.2 Exemple 2: Les Torres de Hanoi

El joc de les "Torres de Hanoi" consisteix en tres cilindres en els quals es poden col.locar peces circulars foradades de tamanys decreixents. Inicialment hi ha n peces al cilindre 1 i cap peça als cilindres 2 i 3. Es tracta de transportar totes les peces d'una en una des del cilindre 1 al cilindre 3 utilitzant el cilindre 2 per a ver els moviments intermitjos. Tot això fóra molt senzill si no hi hagués un altre requeriment:

En cap moment es pot posar en un cilindre una peça que sigui més gran que alguna de les peces que ja hi hagi en aquell cilindre.

A la figura 4.5 es mostra la situació inicial, la situació final i una situació prohibida.

Es tracta de generar els moviments mínims necessaris per a arribar a la posició final des de la posició inicial.

L'anàlisi del problema

Un raonament de caire recursiu que resol el problema és el següent:

Considerarem una torre de n peces com

- Una peça de base +

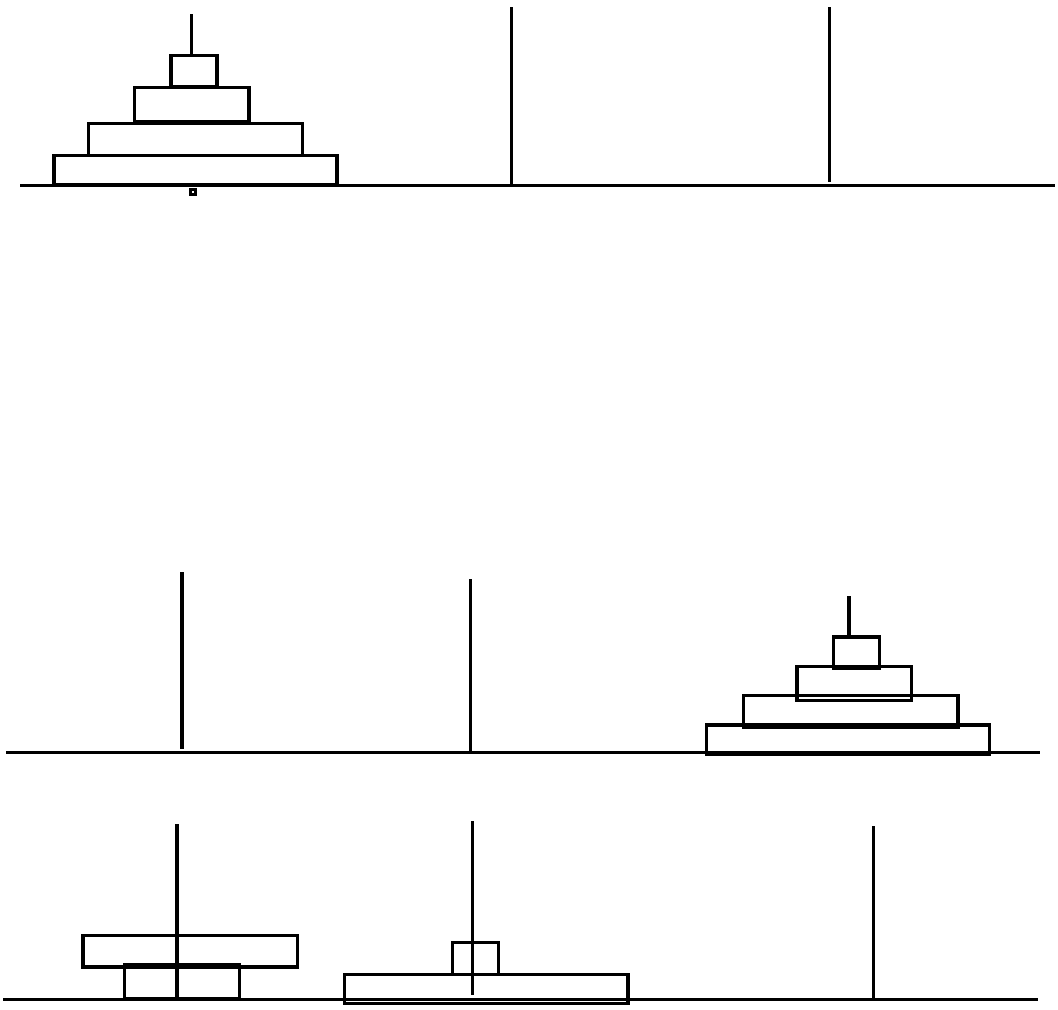


Figura 4.5: Les torres de Hanoi: situació inicial, situació final i situació prohibida.

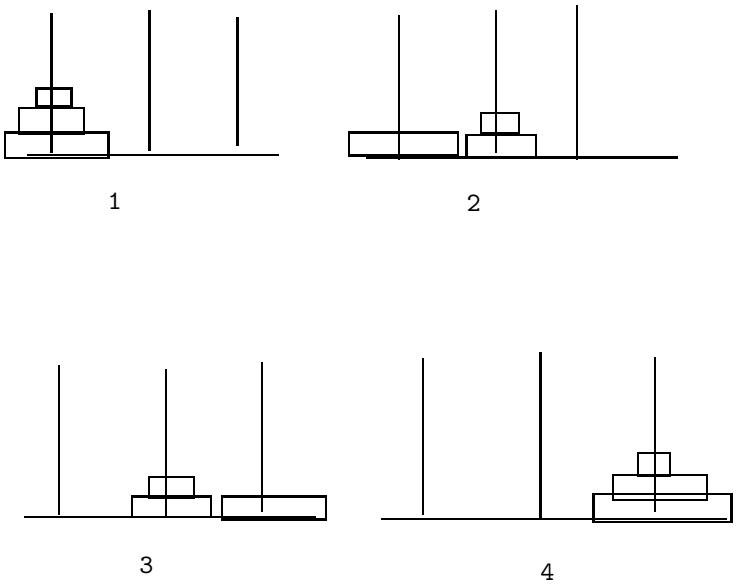


Figura 4.6: Exemple de resolució de les torres de Hanoi

- Una torre de $n-1$ peces.

Resoldre el problema voldrà dir:

- Si la torre del cilindre 1 té una sola peça, transportar-la al cilindre 3. El problema quedarà resolt de forma trivial.
- Si la torre del cilindre 1 té n peces ($n > 1$), aleshores:
  1. Transportar la torre formada per les $n-1$ peces superiors des del cilindre 1 fins al cilindre 2. (això és una crida recursiva). Aquí observem la utilitat del cilindre 2 com a cilindre de moviments.
  2. Moure la base del cilindre 1 al cilindre 3.
  3. Transportar la torre que hem col.locat al cilindre 2 cap al cilindre 3 (aquí tenim la segona crida recursiva).

Fet tot això queda resolt el problema i, per tant, transportada la torre des del cilindre 1 al 3.

Notem que l'algorisme implica la generació de dues crides recursives, amb la qual cosa estarem altre cop en un cas de recursivitat arborescent i, probablement, ens donarà lloc a un cost alt.

Aquest esquema de resolució es pot veure a la figura 4.6

El disseny

L'acció recursiva que implementaria l'anàlisi que hem fet a l'apartat anterior és la següent:

$P = \{n > 0 \wedge \text{mida}(origen) + \text{mida}(final) + \text{mida}(movim) = n\}$

acció hanoi (n:**ent** natural, origen:**ent** natural, final:**ent** natural, movim:**ent** natural) **és**

si n=1 **llavors** escriure("moviment ",origen,"a ", final);
sinó

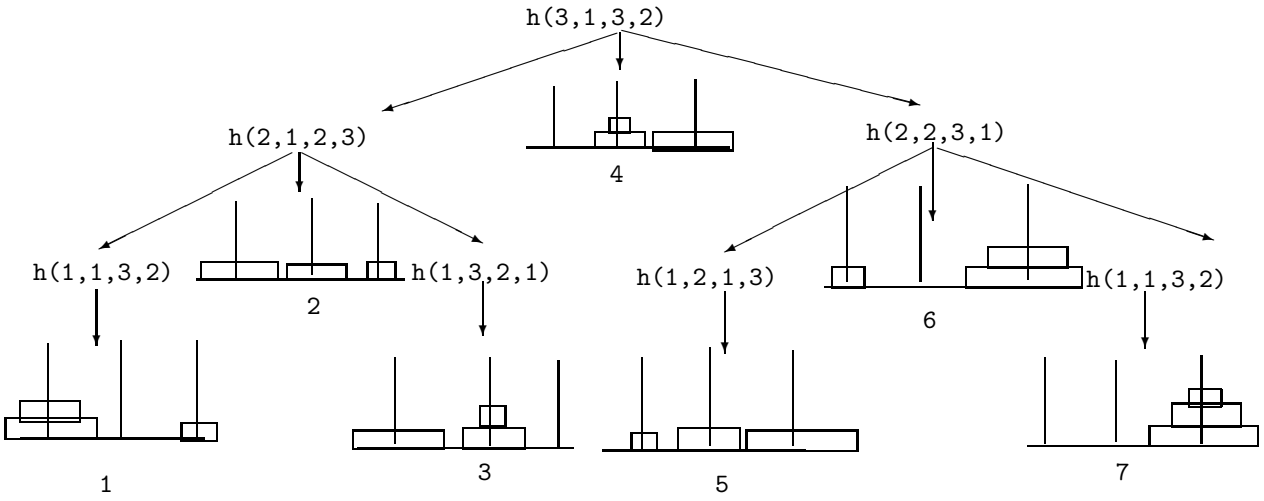


Figura 4.7: Exemple d'execució de les torres de Hanoi

hanoi(n-1, origen,movim, final);
escriure("moviment ",origen,"a", final);
hanoi(n-1,movim,final,origen);

fsi

facció

Q = {L'acció hanoi ha generat els mínims moviments necessaris per a portar n peces des del cilindre *origen* al cilindre $final$ utilitzant el cilindre *movim* tot respectant les regles del joc}

A tall d'exemple, a la figura 4.7 es presenta una execució de l'acció *hanoi* amb tres peces.

Optimalitat de la solució

Demostrarem per inducció sobre el nombre de peces amb què es fa el joc que l'acció *hanoi* resol el problema amb el mínim nombre de moviments possible.

- L'acció hanoi és òptima trivialment per $n = 1$.
- Suposant que l'acció *hanoi* és òptima per $n - 1$ peces cal demostrar que també ho és per n peces.

  Per a moure totes les n peces des de la torre 1 a la torre 3, necessàriament arribarà un moment en què haurem de moure la peça de la base de la torre 1 (la més gran) a la base de la torre 3. L'única posició en què això és possible és la que es mostra a la figura 4.8. Efectivament, el cilindre 1 no pot contenir cap altra peça, el cilindre 3 ha d'estar buit ja que la peça més gran ha d'anar a la base i la resta de les peces hauran d'estar col.locades al cilindre 2 en ordre creixent de tamanys. Anomenarem a la posició de la figura 4.8 *posició clau.*

  Per a resoldre el problema de forma òptima, considerant que en algun moment haurem d'arribar a la posició clau, necessàriament haurem d'utilitzar el següent algorisme:

  1. Moure les $n-1$ primeres peces del cilindre 1 al cilindre 2 amb el mínim nombre de moviments. L'acció $hanoi(n-1, 1, 2, 3)$ fa precisament això i ho fa, per hipòtesi d'inducció de forma òptima.
  2. Moure la peça de la base del cilindre 1 al cilindre 3.

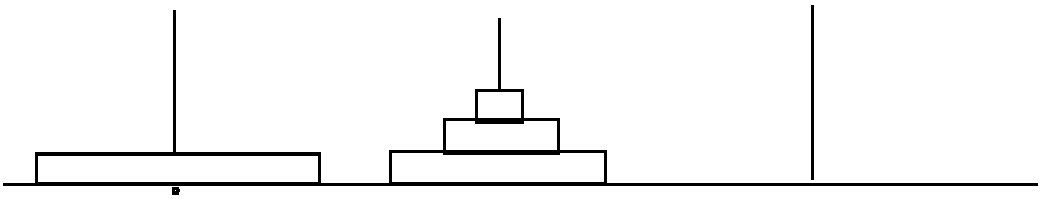


Figura 4.8: Posició clau a l'acció hanoi

3. Moure les $n-1$ peces del cilindre 2 al cilindre 3 amb el mínim nombre de moviments. L'acció $hanoi(n-1,2,3,1)$ fa això de forma òptima per hipòtesi d'inducció.

El comportament de l'acció $hanoi(n,1,3,2)$ és precisament el que acabem de descriure. Per tant concloem que l'acció *hanoi* resol el problema de forma òptima.

El cost

L'arbre de crides que genera l'acció $hanoi(n,1,3,2)$ té 2^n-1 vèrtex ja que en cada nivell es generen dues crides recursives i tenim un total de n nivells. Com cada crida farà un moviment, deduim que per resoldre el problema de tamany n calen 2^n-1 moviments.

Podem arribar a aquest mateix resultat d'una manera molt més elegant: utilitzant les equacions recurrents[2]. L'equació recurrent que descriu el nombre de moviments necessaris per a resoldre el problema usant l'acció presentada és la següent:

$T(1)=1$
$T(n)=2*T(n-1)+1$ si $n>1$

La qual es llegeix de la següent manera:

Si la torre de peces que hem de moure consta d'un sol cilindre, haurem de fer només un moviment per a resoldre el problema. Si consta de n cilindres, haurem de fer els moviments necessaris per a moure 2 torres de n-1 peces i un moviment addicional, tal i com es desprén de l'acció recursiva dissenyada.

Aquesta mateixa equació, per tal de resoldre-la, la podem posar en funció, no de la crida anterior $(n-1)$ sinó d'una crida qualsevol $n-i$ $(1\leq i\leq n-1)$:

$T(1)=1$
$T(n)=2^i*T(n-i)+2^i-1$ per $1\leq i\leq n-1$ [2]

Aquesta transformació és molt senzilla de realitzar desenvolupant un parell de cops $T(k)$ a l'expressió [1] i veient quin patró segueixen aquestes transformacions.

La gràcia de l'expressió [2] és que ens permet posar $T(n)$ en funció de $T(1)$ que, com té una resolució trivial, elimina la recursivitat de l'expressió de $T(n)$. Efectivament, si prenem $i=n-1$, obtenim:

$T(n)=2^{n-1}*T(1)+2^{n-1}-1=2^n-1$

que coincideix amb el nombre de moviments esperat per a resoldre el problema de Hanoi.

[2]Les equacions recurrents es descriuen en l'assignatura de Matemàtica discreta.

Concloem, doncs, que l'acció *hanoi* genera $2^n - 1$ moviments per a moure una torre de n cilindres. Però acabem de demostrar que aquesta acció és òptima, i que, per tant, resol el problema amb el mínim nombre de moviments. Per tant, la intuïció de que els programes amb vàries crides recursives donaven lloc sempre a solucions dolentes o, com a mínim, millorables s'esbaeix en aquest exemple. Estem davant una solució recursiva amb vàries crides molt elegant i òptima per a resoldre el problema de les torres de Hanoi. Aquesta falsa intuïció l'acabarem de dissipar a l'apartat 5.1.

4.4.3 Exemple 3: La generació de permutacions

El problema que ens plantegem tot seguit és el de generar i escriure les $n!$ permutacions de n elements continguts en un vector. L'especificació de l'acció fóra la següent:

P={v és un vector de N enters que conté n enters diferents en els seus n primers índexos $\wedge n \leq N$}

permutacions(v,n);

Q={S'han generat les $n!$ permutacions dels n elements de v}

Exemple: Si $v = (1, 2, 3)$, després de l'execució de *permutacions(v,3)* s'haurà escrit (per exemple a la pantalla):

1 2 3
2 1 3
1 3 2
3 1 2
2 3 1
3 2 1

L'Anàlisi

En aquest punt hem de decidir com podem plantejar el problema de la generació de permutacions de n elements en termes de sí mateix.

Si observem les permutacions de 3 elements llistades a l'apartat anterior, ens adonarem que hi ha algunes permutacions que acaben en 1, altres acaben en 2 i altres en 3. Podríem, doncs, pensar en fixar un dels n elements i generar totes les permutacions de la resta dels elements que acabin en aquell fixat. Aquest procés caldria repetir-lo per tots els n elements del vector.

Amb aquesta reflexió hem reduït el problema a calcular les permutacions de $n - 1$ elements.

Per altre costat, i com a cas trivial, la generació de les permutacions de 1 element es reduirà a fer una llistat del vector v.

Veiem un exemple amb les permutacions de 4 elements:

Sigui v=(1 2 3 4).

permu((1 2 3 4), 4) ho descomposarem en:

permu((1 2 3 4), 3) Aquesta acció escriurà les permutacions de (1 2 3 4) que deixen el 4 fixat al final.

permu((4 2 3 1), 3) Aquesta acció escriurà les permutacions de (4 2 3 1) que deixen el 1 fixat al final.

permu((4 1 3 2), 3) Aquesta acció escriurà les permutacions de (4 1 3 2) que deixen el 2 fixat al final.

permu((4 1 2 3), 3) Aquesta acció escriurà les permutacions de (4 1 2 3) que deixen el 3 fixat al final.

Seguint aquest esquema recursiu, es generaran $n!$ crides que correspondran a casos trivials i que tindran el següent aspecte:

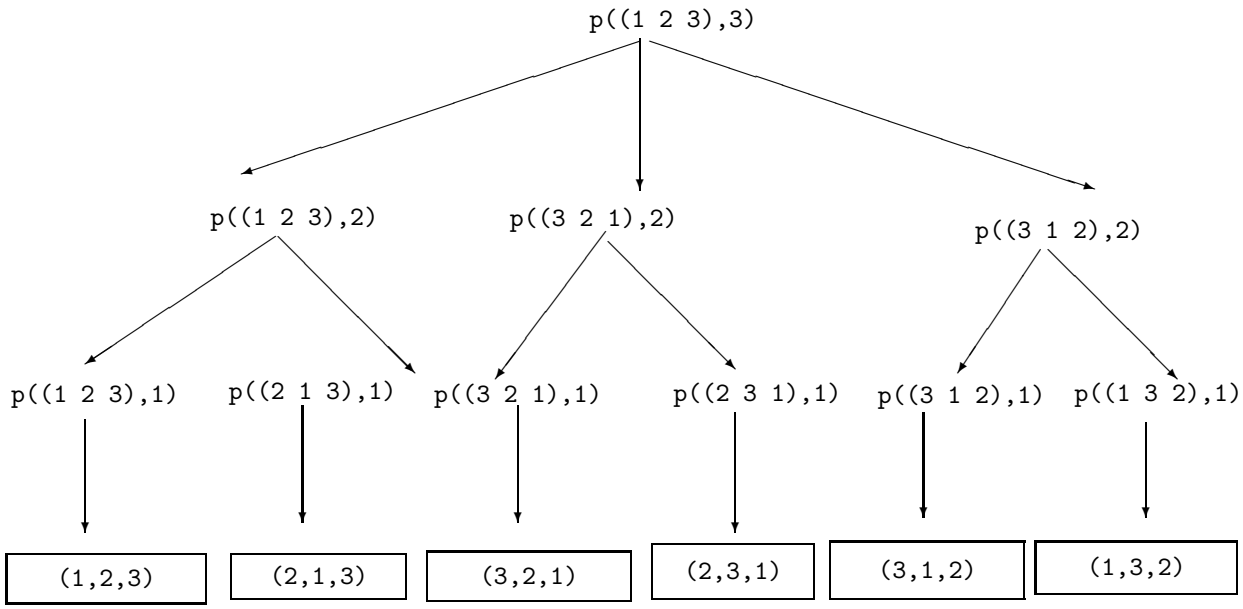


Figura 4.9: Execució de l'acció permu(v,4)

permu((1 2 3 4), 1) Aquesta acció escriu totes les permutacions de (1 2 3 4) que deixen (2 3 4) fixats al final. Però només n'hi ha una: (1 2 3 4). Queda, doncs, clar perquè quan es genera una crida a *permu* amb $n = 1$, ens trobem davant un cas trivial.

El disseny

```
acció permu(v: ent vector[1..N] de enter, n:ent natural) és
  var i,temp:natural fvar
  si
    n=1 llavors escriure(v);
    n> 1 llavors
      per i:=1 a n fer
        permu(v,n-1);
        temp:=v[i];
        v[i]:=v[n];
        v[n]:=temp;
      fper
  fsi
facció
```

El cost

El cost assimptòtic de l'acció *permu* es pot plantejar segons la següent equació recurrent:

$T(1) = 1$
$T(n) = n * T(n-1) + n$

on $T(n)$ representa el cost assimptòtic de l'acció *permu* amb un paràmetre n. El sentit d'aquesta equació

és el següent:

Una crida a l'acció *permu* amb paràmetre n llença n crides a *permu* amb paràmetre $n-1$ i fa n intercanvis entre variables amb cost $O(1)$ per cadascun. Finalment, la crida amb paràmetre 1 no genera noves crides recursives i té un cost total de $O(1)$.

Aquesta equació està plantejada en funció del terme anterior. Si la plantegem en funció de qualsevol terme anterior $(n-i)$, obtindrem:

$T(n) = n*(n-1)*...*(n-i+1)*T(n-i)+$
$\sum_{k=1}^{k=i} n*(n-1)*...*(n-k+1)$ [1]

Per $i = n-1$ assolirem el cas trivial i l'expressió [1] és $O(n!)$

Aquest cost és altament exponencial (molt superior a 2^n per $n > 3$).

4.5 Les immersions

4.5.1 La motivació

Freqüentment, durant el disseny d'una funció recursiva, ens adonem que estem seguint un camí que no és prou adequat. A vegades arribem a una solució correcta però ineficient, com en el cas de la funció que calculava el terme n-èssim de la successió de Fibonacci. Altres vegades la situació és més greu perquè ni tan sols som capaços de trobar una solució recursiva al problema plantejat.

Una possibilitat per a abordar el problema quan ens trobem en algun d'aquests casos és *generalitzar la funció f que volem calcular*, això és, *trobar una nova funció g de la qual f en sigui un cas particular i que sigui més senzilla de calcular que f o, simplement, més eficient.* Aleshores, evidentment, caldrà fer un disseny recursiu per a g.

Definició 11

Direm que una immersió d'una funció f és una altra funció g de la que f n'és un cas particular.

Exemple 1

Considerem la funció arrel que obté l'arrel quadrada entera d'un número natural, especificada de la següent manera:

$P(n) = \{n \geq 0\}$
$r := arrel(n);$
$Q(n,r) = \{r^2 \leq n < (r+1)^2\}$

Tot seguit presentem una de les moltes possibles generalitzacions d'aquesta funció:

$P(n,d) = \{n \geq 0 \wedge d > 0\}$
$r := arrel\_gen(n,d);$
$Q(n,r,d) = \{r^2 \leq n < (r+d)^2\}$

Efectivament arrel és un cas particular de arrel_gen:

$arrel(n) = arrel\_gen(n,1)\ \forall n \geq 0.$

I, a més a més, arrel_gen és més senzilla de dissenyar que arrel com veurem més endavant.

Amb tot plegat, el procés general per a dissenyar algorismes recursius utilitzant la tècnica de la immersió és el següent:

Sigui f una funció de la qual es desitja trobar un disseny recursiu,

1. Proposar una funció g que compleixi:

- g és una generalització de f.
- g és més senzilla de dissenyar que f

2. Fer un disseny recursiu per la funció g.
3. Implementar la funció f simplement com una crida a g amb els paràmetres adequats.

En general hi ha dos motius pels quals la immersió (generalització) g d'una funció f pot ser més senzilla de calcular que la funció original:

- Perquè la funció generalitada g ens aporti més informació que l'original f.
- Perquè la funció generalitzada g ens demani menys coses que l'original f.

Podem enumerar almenys tres casos en què una immersió pot introduir alguna millora al disseny d'una funció recursiva:

1. *Millora d'eficiència.* Una immersió pot permetre augmentar l'eficiència d'una solució recursiva.
2. *Obtenció de recursivitat final.* És una altra manera d'aconseguir augmentar l'eficiència d'una solució. Com veurem a l'apartat 4.7, les funcions recursives finals són més eficients que les seves equivalents no finals i es poden transformar automàticament a iteratives.
3. *Facilitar el disseny d'una funció recursiva.* A vegades fer una immersió és l'única manera d'obtenir un disseny recursiu per a una determinada funció.

En els apartats següents raonarem de quina manera les immersions ens poden ajudar a aconseguir aquestes millores.

4.5.2 Immersions per a millorar l'eficiència d'una funció recursiva

Suposem que hem dissenyat una funció recursiva que segueix l'esquema general de la que apareix a la figura 4.2:

$P(x)$
$res := f(x);$
$Q(x, res)$

És possible que aquest disseny hagi resultat ineficient i que, en revisar-lo fem la següent reflexió:

Per tal de calcular $f(x)$ més eficientment em resultaria molt pràctic disposar d'una certa dada v.

Podem proposar una immersió (o generalització) g de la funció f que incorpori aquesta informació v. Hi ha dues maneres de fer això:

- Afegint v com a resultat de la següent crida recursiva.
- Afegint v com a paràmetre de l'acció.

Seguidament presentem un parell d'exemples on s'exploren aquestes possibilitats.

La successió de Fibonacci

Partim del disseny de la funció que calcula el n-èssim valor de la successió de Fibonacci desenvolupat a l'apartat 4.4:

$P = \{n \geq 1\}$

funció fibo (n:natural) **retorna** natural **és**
  si
    $n = 1 \longrightarrow$ **retorna** 1;
    $n = 2 \longrightarrow$ **retorna** 1;
    $n > 2 \longrightarrow$ **retorna** fibo(n-1)+fibo(n-2);
  fsi
ffunció

$Q = \{fibo(n) = F_n\}$

On F_n representa el valor del terme n de la successió de Fibonacci.

Ja vam justificar en aquell moment que aquesta funció era molt ineficient perquè la doble crida recursiva li feia repetir càlculs. Intentem generalitzar-la per tal d'aconseguir una funció amb recursivitat simple que en millori l'eficiència.

Per això fem la següent reflexió:

Si al càlcul del cas recursiu ($fibo(n) = fibo(n-1) + fibo(n-2)$) coneguéssim algun dels dos sumands (posem per cas, $fibo(n-2)$), podíem calcular directament $fibo(n)$ sense la crida a $fibo(n-2)$. Ara bé, com ens podem assabentar del valor de $fibo(n-2)$? Ens el pot dir $fibo(n-1)$ com a resultat addicional. Aquesta és precisament la immersió que fem:

$P = \{n \geq 2\}$

funció fibo2 (n:natural) **retorna** <natural,natural>

$Q = \{fibo2(n) =< F_n, F_{n-1} >\}$

La immersió feta ha generalitzat $fibo(n)$ convertint-la en una funció que retorna com a resultat una parella de naturals: els valors de la successió de Fibonacci F_n i F_{n-1}. Notem que això ens ha obligat a enfortir lleugerament la precondició ja que la funció $fibo2(n)$ no té sentit per $n = 1$.

Ara es compleix que per a tot $n \geq 2$, $fibo2(n) =< fibo(n), fibo(n-1) >$.

Com ja hem dit, la ventatja de $fibo2$ és que la seva expressió recursiva involucra únicament recursivitat simple i resulta més eficient:

$P = \{n \geq 2\}$

funció fibo2 (n:natural) **retorna** <natural,natural>
  var f1,f2:natural; **fvar**
  si
    $n = 2 \longrightarrow$ **retorna** $< 1, 1 >$
    $n > 2 \longrightarrow < f1, f2 >:= fibo2(n-1)$;
        retorna $< f1 + f2, f1 >$;
  fsi
ffunció

$Q = \{fibo2(n) =< F_n, F_{n-1} >\}$

El disseny de $fibo$ és ara molt senzill i eficient si el fem en termes de $fibo2$:

$P = \{n \geq 1\}$

funció fibo (n:natural) **retorna** natural
  var f1,f2:natural; **fvar**
  si

$n = 1 \longrightarrow$ **retorna** 1
$n > 1 \longrightarrow < f1, f2 >:= fibo2(n);$
retorna $f1;$
fsi
ffunció

$Q = \{fibo(n) = F_n\}$

En aquest exemple hem vist com el fet d'afegir un resultat addicional a la funció generalitzada pot fer més eficient un disseny recursiu. Al proper exemple veurem que també es pot fer més eficient aquest disseny afegint paràmetres a la funció generalitzada.

La cerca dicotòmica

Sigui v un vector ordenat d'enters definit entre els índexos 1 i N. No és difícil de dissenyar una funció recursiva que cerqui un determinat enter dins de v entre els índexos 1 i n.

v:vector[1..N] de enter
x:enter
n, j:natural
$P = \{0 \leq n \leq N \wedge ordenat(v)\}$
j:=cerca(v, x, n);
$Q = \{\forall k : 1 \leq k \leq j : v[k] \leq x \wedge \forall k : j + 1 \leq k \leq n : v[k] > x \wedge 0 \leq j \leq n\}$

funció cerca (v, x, n) **retorna** natural
si $n = 0 \longrightarrow$ **retorna** 0;
$v[n] \leq x \longrightarrow$ **retorna** n;
$v[n] > x \longrightarrow$ **retorna** cerca$(v, x, n - 1)$;
fsi
ffunció

Aquesta funció recursiva té un cost $O(n)$. Aquest cost el podem millorar fins a $O(log(n))$ amb la idea presentada a 3.4 de la cerca dicotòmica. Però per a poder fer el disseny recursiu d'una acció que faci una cerca dicotòmica, necessàriament hem de canviar l'especificació de l'acció *cerca* que acabem de descriure: *hem de fer una immersió d'aquesta funció* tot i afegint a *cerca* el paràmetre de líndex del vector allà on comença el rang de la cerca. Sense aquesta immersió no podem desenvolupar una acció recursiva que implementi l'algorisme de la cerca dicotòmica.

v:vector[1..N] de enter
x:enter
ini, fi, j:natural
$P = \{1 \leq ini \leq fi + 1 \leq N + 1 \wedge ordenat(v)\}$
j:=cerca_dicot(v,x,ini,fi);
$Q = \{\forall k : ini \leq k \leq j : v[k] \leq x \wedge \forall k : j + 1 \leq k \leq fi : v[k] > x \wedge 0 \leq j \leq n\}$

cerca_dicot és efectivament una generalització de *cerca* doncs $cerca(v, x, n) = cerca\_dicot(v, x, 1, n)$. Notem que hem fet la generalització, en aquest cas, afegint un paràmetre a la funció inicial.

El disseny de la funció generalitzada serà:

funció cerca_dicot (v, x, ini, fi) **retorna** natural
si $ini > fi \longrightarrow$ **retorna** fi;
$ini \leq fi \longrightarrow m := (ini + fi)$**div** 2;
si $v[m] \leq x \longrightarrow$ **retorna** cerca_dicot$(v, x, m + 1, fi)$;
$v[m] > x \longrightarrow$ **retorna** cerca_dicot$(v, x, ini, m - 1)$;
fsi
fsi
ffunció

Un exercici interessant és el de verificar totes dues accions: *cerca* i *cerca_dicotom*.

4.5.3 Immersions per a obtenir recursivitat final (mètode del desplegament-plegament

Definició 12

> *Direm que una funció recursiva és final si el valor que retorna la crida recursiva que conté la funció coincideix amb el valor que retorna la funció.*

Això es produirà si a l'esquema de funció recursiva de la figura 4.2, *co* és la funció buida i, en conseqüència, l'acció de retorn de resultat té l'aspecte:

return $f(suc(x))$

Un exemple de funció recursiva final pot ser la que calcula el màxim comú divisor de dos naturals.

Les funcions recursives finals són interessants perquè es pot automatitzar la seva transformació a iteratives. D'aquesta transformació en parlarem a l'apartat 4.7.

En aquest apartat parlarem d'immersions que ens permetran transformar funcions recursives no finals en finals.

Aproximació intuitiva

Considerem la funció recursiva *factorial* (la nostra amiga exemplar de sempre):

$natural(n)$
$P = \{n \geq 0\}$

funció factorial(n:natural) **retorna** natural
  si $n = 0$ **llavors** retorna 1;
    sinó retorna n*factorial($n-1$);
  fsi
ffunció

$Q = \{\text{factorial}(n) = n!\}$

Aquesta funció recursiva **no és final**. Efectivament, en l'execució de *fact(n)*, un cop haguem obtingut el resultat de la crida recursiva *fact(n-1)*, encara haurem de multiplicar aquest resultat per n per tal de completar-la. *El nostre propòsit serà el d'obtenir una funció recursiva final per a calcular el factorial utilitzant tècniques d'immersió.*

La forma natural de fer-ho serà *trobar una funció* g que compleixi:

1. g és una generalització de $fact$
2. g admet una expressió recursiva final.

La forma més senzilla d'aconseguir aquestes dues propietats és construint la funció g generalitzant l'expressió de *fact* en el cas recursiu ($n * fact(n-1)$). Veiem-ho:

Considerem $g(t, r) = t * fact(r)$

Aquesta funció compleix les dues propietats que li demanàvem:

- g és una generalització de $fact$ perquè $fact(n) = g(1, n)$, $\forall n \geq 0$.

  Notem que per a obtenir la generalització utilitzem l'element neutre del producte dels naturals.

- g admet una expressió recursiva final perquè podem desenvolupar-la de la següent manera:

  $g(t,r) = t * fact(r) =$ si $r > 0$, utilitzant la descomposició recursiva de $fact$ (a aquesta operació l'anomenem *desplegament*)
  $t * r * fact(r-1) =$ agrupant termes
  $(t * r) * fact(r-1) =$ utilitzant la definició de g (a aquesta operació l'anomenem *plegament*)
  $g(t * r, r-1)$

  Notem que en fer aquest desplegament i plegament que ens ha permés obtenir una expressió recursiva final per a g hem utilitzat l'associativitat del producte dels naturals. Notem també que aquesta expressió no és vàlida per $r = 0$.

Amb el raonament que hem fet, obtenim la següent funció per g:

$\forall r, t \geq 0$ naturals:

$$g(t,r) = \begin{cases} g(t*r, r-1) & \text{si } r > 0 \\ t & \text{si } r = 0 \end{cases}$$

Que dóna lloc al següent disseny:

$P(t,r) = \{r \geq 0 \wedge t \geq 0\}$
funció g(t,r:natural) **retorna** natural **és**
  si $r = 0$ **retorna** t
    $r > 0$ **retorna** $g(t * r, r-1)$
  fsi
ffunció

i la funció $fact$ quedaria:

funció fact(n:natural) **retorna** natural **és**
  retorna g(1,n);
ffunció

Mètode

El mètode que hem esbossat en l'apartat anterior pel cas concret de la funció factorial, el podem estendre a una funció recursiva general com la següent:

$P(x)$
funció $f(x : T_1)$ **retorna** T_2 **és**
  si
    $b_t(x)$ $\longrightarrow$**retorna** $tr(x)$
    $b_r(x)$ $\longrightarrow$**retorna** $co(f(suc(x)), x)$
  fsi
ffunció
$Q(x, f(x))$

Sigui g la funció d'immersió. Proposem $g(r,t) = co(f(r), t)$ i suposem, pel desenvolupament del mètode, que $T_1 = T_2$[3]

- Si T_1 té un element neutre per l'operació co,[4] existirà un valor e que complirà: $co(f(r), e) = f(r)$, per la qual cosa g serà un cas particular de f.

[3] Si $T_1 \neq T_2$ les consideracions que fem tot seguit no són vàlides però abans de desistir en l'aplicació del mètode, és aconsellable fer un estudi d'aquell cas particular

[4] co és una funció definida $co : (T_1 * T_1) \longrightarrow T_1$ i, per tant, es tracta d'una operació.

- Si l'operació co és associativa, complirà $co(co(f(x), y), z) = co(f(x), co(y, z))$. Aquesta propietat ens permetrà trobar una expressió recursiva final per a g de la següent manera:

  $g(r, t) = co(f(r), t)$
  $=$\{si es compleix $b_r(r)$, despleguem\}
  $co(co(f(suc(r)), r), t)$
  $=$(Aplicant associativitat de co)
  $co(f(suc(r)), co(r, t))$
  $=$(pleguem)
  $g(suc(r), co(r, t))$.

  Si es compleix $b_t(r)$, calcularem el cas trivial fent: $g(r, t) = co(tr(r), t)$.

En total, la funció recursiva final que calcula g queda:

$P(x)$
funció $g(x : T_1, y : T_1)$ **retorna** T_1 **és**
  si
    $b_t(x, y)$ $\longrightarrow$retorna $co(tr(x), y)$
    $b_r(x, y)$ $\longrightarrow$retorna $g(suc(x), co(x, y))$
  fsi
ffunció
$Q(x, y, g(y, x))$

En aquest disseny també queda clar que la funció afitadora per a g és la mateixa que la que teníem per a f.

La successió de Fibonacci

En un apartat anterior hem vist com, a partir de la definició recursiva ineficient de la successió de Fibonacci, podíem derivar per immersió una altra funció capaç de calcular el n-èssim terme d'aquella successió utilitzant recursiviatat simple amb un cost molt inferior. Ara donarem un pas més endavant i presentarem un disseny recurrent *final* d'una funció que calcula el terme n-èssim de la successió de Fibonacci.

Partim de la definició de la successió de Fibonacci per a tot natural $n > 0$:

$$fib(n) = \begin{cases} 1 & \text{si } n = 1 \vee n = 2 \\ fib(n-1) + fib(n-2) & \text{si } n > 2 \end{cases}$$

L'objectiu serà ara el de trobar una generalització g de la funció fib que admeti una expressió recursiva final i de la qual fib en sigui un cas particular. Per a trobar aquesta funció d'immersió g ens inspirarem en l'expressió recursiva de fib. Proposem la següent funció g, definida per a tot $n \geq 2$ i per a tot w i $t \geq 0$:

$g(n, w, t) = w * fib(n) + t * fib(n-1)$

Aquesta funció compleix les dues propietats que demanàvem a les funcions d'immersió:

- fib n'és un cas particular:

  $fib(n) = g(n, 1, 0)\ \forall n \geq 2$

- g admet una expressió recursiva final:

  $$g(n, w, t) = \begin{cases} w + t & \text{si n=2} \\ g(n-1, w+t, w) & \text{si } n > 2 \end{cases}$$

  Aquesta expressió s'obté desplegant (amb una mica de seny) l'expressió donada per g i plegant posteriorment. El seu desenvolupament queda com a exercici.

El màxim d'un vector

La següent funció recursiva obté el màxim d'un vector t[1..N] de naturals entre dos índexos $1 \leq i1 \leq i2 \leq N$:

$P(t, i1, i2) = \{1 \leq i1 \leq i2 \leq N\}$
funció $maxim(t : vector[1..N], i1 : natural, i2 : natural)$ **retorna** $natural$ **és**
si
$i1 = i2 \longrightarrow$**retorna** $t[i1]$
$i1 < i2 \longrightarrow$**retorna** $mes\_gran(t[i1], maxim(t, i1 + 1, i2))$
fsi
ffunció
$Q(t, i1, i2, maxim(t, i1, i2)) = \{\forall k : i1 \leq k \leq i2 : maxim(t, i1, i2) \geq t[k]\}$

L'operació $mes\_gran : nat * nat \longrightarrow natural$ compleix la propietat associativa. A més a més, dins del conjunt dels naturals amb el zero, existeix un element neutre per aquesta operació $e = 0$. Com l'operació *més_gran* compleix aquestes dues propietats, no hem de tenir cap problema per a desenvolupar una immersió g de *màxim* que sigui recursiva final. Fem-ho:

A partir del resultat del cas recursiu, proposem la següent g:

$g(t, r, w1, w2) = mes\_gran(r, maxim(t, w1, w2))$

- g generalitza *màxim* ja que $g(t, 0, i1, i2) = maxim(t, i1, i2)$.
- g admet una expressió recursiva final:
  $g(t, r, w1, w2) = mes\_gran(r, maxim(t, w1, w2))$
  $\equiv (w1 < w2)$
  $mes\_gran(r, mes\_gran(t[w1], maxim(t, w1 + 1, w2)))$
  $\equiv$ (associativitat de *més_gran*)
  $mes\_gran(mes\_gran(r, t[w1]), maxim(t, w1 + 1, w2)))$
  $\equiv$ (definició de g)
  $g(t, mes\_gran(r, t[w1]), w1 + 1, w2)$

Per altra banda, pel cas $w1 = w2$ $g(t, r, w1, w2) = mes\_gran(r, t[w1])$

I aquesta anàlisi duu al següent disseny:

$P(t, w1, w2) = \{1 \leq w1 \leq w2 \leq N\}$
funció $g(t : vector[1..N], r : natural, w1 : natural, w2 : natural)$ **retorna** $natural$ **és**
si
$w1 = w2 \longrightarrow$**retorna** $mes\_gran(r, t[w1])$
$w1 < w2 \longrightarrow$**retorna** $g(t, mes\_gran(r, t[w1]), w1 + 1, w2)$
fsi
ffunció
{Q}=$\{g(t, r, w1, w2) = mes\_gran(r, maxim(t, w1, w2))\}$

4.5.4 Immersió d'especificacions per a facilitar el disseny d'una funció recursiva

Encetem ara l'últim dels grups d'immersions que estudiarem: aquelles encaminades a possibilitar el disseny d'una funció recursiva que, d'altra manera, resultaria molt complicada o impossible. Per això, proposarem una funció per la qual sigui difícil trobar una recurrència i la generalitzarem de vàries formes utilitzant diferents estratègies i donant lloc a solucions diverses.

L'arrel

Considerem el problema de trobar la part entera de l'arrel quadrada d'un número natural. És a dir:

funció arrel (n:enter) **retorna** natural ;

$P(n) = \{n \geq 0\}$
$r := arrel(n);$
$Q(n, r) = \{r^2 \leq n < (r+1)^2\}$

El disseny recursiu de la funció *arrel* no és gens senzill. No és fàcil trobar una relació de recurrència que permeti calcular-la fàcilment. La millor manera d'abordar el problema és fer una immersió de la funció *arrel*. És a dir: *Si no som capaços de trobar una recurrència per la funció arrel, potser podrem trobar-ne una per una funció més general.* Aquest problema és especialment interessant perquè permet fer moltíssimes immersions, cadascuna partint d'una consideració diferent i arribant a una solució també diferent.

Així doncs, utilitzarem aquest problema com a catàleg de possibilitats d'immersió a considerar davant d'una especificació.

Enfortir la precondició

Podem proposar una funció més general que *arrel* afegint més paràmetres a aquesta i reduint el domini d'aplicació de la funció. La forma natural de fer això serà enfortint la precondició, de forma que satisfaci ja una part de la postcondició: Si la precondició ja satisfà una part de la postcondició, el treball de la funció per a assolir la postcondició haurà de ser més petit. Veiem-ho:

Considerem la següent immersió de la funció arrel:

$P(n, a) = \{n \geq a^2\}$
$r := arrel2(n, a);$
$Q(n, r) = \{r^2 \leq n < (r+1)^2\}$

$arrel2$ és una generalització d'arrel ja que $arrel(n) = arrel2(n, 0)$. El significat intuitiu d'aquest paràmetre a que hem afegit és *el resultat parcial que portem calculat fins ara i que ens acosta a la postcondició.* Notem que aquest raonament té moltes semblances al que usem en el disseny iteratiu d'algorismes (on en cada nova iteració avancem cap a la postcondició actualitzant el valor d'un resultat parcial). Després comentarem més en detall les relacions *sorprenents* entre aquest tipus d'algorismes recursius i els seus iteratius equivalents. De moment, però, conformem-nos amb desenvolupar el disseny recursiu suggerit per l'especificació:

$P(n, a) = \{n \geq a^2\}$

funció arrel2 (n, a:natural) **retorna** natural **és**
  si $n < (a+1)^2 \longrightarrow$ **retorna** a;
    $n \geq (a+1)^2 \longrightarrow$ **retorna** arrel2$(n, a+1)$;
  fsi
ffunció
r:=arrel2(n, a)

$Q(n, a, r) = \{r^2 \leq n < (r+1)^2\}$

La seva verificació queda com a exercici.

Abans de deixar aquest exemple, fem una observació certament interessant que aportarà molta llum respecte les relacions entre les estructures recursives i les iteratives: Considerem la transformació de la funció *arrel2* a iterativa:

funció arrel2_it(n:natural) **retorna** natural
  $a := 0$;
  mentre $n \geq (a+1)^2$ **fer**

$a := a + 1$;
fmentre
retorna a;
ffunció
r:=arrel2_it(n)

$Q(n, r) = \{r^2 \leq n < (r + 1)^2\}$

$I = \{n \geq a^2\}$

Notem que l'invariant de la iteració coincideix amb la precondició de la funció recursiva *arrel2*, que la condició de continuació de la iteració és la mateixa que la guarda del cas recursiu i que la funció iterativa retorna el mateix resultat que el cas trivial de la funció recursiva.

Els dissenys recursius amb recursivitat final i tals que la postcondició no depèn d'allò que varia en cada nova crida recursiva s'anomenen dissenys recursius amb *postcondició constant* i tenen la característica que la seva transformació a iteratiu és immediata i verifica les propietats que hem vist en aquest exemple. Des d'un punt de vista intuitiu, es pot apreciar que aquell paràmetre de què depèn cada nova crida recursiva però no la postcondició final (al cas de la funció *arrel2*, aquest paràmetre era a) representa un resultat intermig que ens acosta cada cop més a la postcondició (filosofia idèntica a la que regeix el disseny iteratiu). Podem dir que cada nova crida recursiva significa i representa una nova volta a l'estructura iterativa.

No entrarem en l'estudi més detallat d'aquestes funcions però proposem com a exercici la transformació de les funcions recursives $fibo$ i $fact$ a recursives finals amb postcondició constant utilitzant immersions que reforcin les seves respectives precondicions.

Afeblir la postcondició

Una altra manera de fer una immersió que sigui més senzilla de calcular és especificar una funció $arrel3$ que generalitzi la funció *arrel* tot i demanant *menys coses* a la postcondició (i, per tant, possibilitant un disseny més senzill). La forma natural d'aconseguir això és *afeblint la postcondició.*

Una forma força intuitiva de fer més feble la postcondició és no demanar que n estigui entre r^2 i $(r + 1)^2$ sinó una cosa menys forta: que n estigui entre r^2 i $(r + d)^2$, amb $d > 0$. Això ens portarà a la següent especificació:

$P(n, d) = \{n \geq 0 \wedge d > 0\}$
$r := arrel3(n, d);$
$Q(n, r, d) = \{r^2 \leq n < (r + d)^2\}$

Queda clar que $arrel3(n, 1) = arrel(n)$ i, per tant, $arrel3$ és una generalització de $arrel$.

Aquesta especificació ens facilita el desenvolupament d'un disseny recursiu. N'hi ha prou amb prendre $n < d^2$ com a cas trivial (en aquest cas, $r = 0$):

$\{P(n, d)\} = \{n \geq 0 \wedge d > 0\}$

funció arrel3(n:natural,d:natural) **retorna** natural
si $n < d^2 \longrightarrow$ **retorna** 0
$n \geq d^2 \longrightarrow$???
fsi
ffunció

$r := arrel3(n, d);$

$Q(n, r, d) = \{r^2 \leq n < (r + d)^2\}$

Si utilitzem la funció limitadora natural: $t(n, d) = n - d^2$, queda clar que al cas recursiu haurem de fer una crida decrementant n o incrementant d. Proposem dos dissenys diferents, obtinguts tots dos incrementant d:

- $arrel31(n, d+1)$, amb cost lineal.
- $arrel32(n, d*2)$, amb cost logarítmic.

Desenvoluparem el primer i deixarem el segon com a exercici:

$\{P(n,d)\} = \{n \geq 0 \wedge d > 0\}$

funció arrel31(n:enter,d:enter) **retorna** natural
  si $n < d^2 \longrightarrow$ **retorna** 0
    $n \geq d^2 \longrightarrow$
      $r1 := arrel31(n, d+1)$;
      [1]
      ???
  fsi
ffunció

$r := arrel31(n, d)$;

$Q(n,r,d) = \{r^2 \leq n < (r+d)^2\}$

La hipòtesi de la correctesa de la crida recursiva, ens permet de satisfer a [1] la postcondició d'aquesta crida:

$[1] \equiv \{r1^2 \leq n < (r1+d+1)^2\}$
$\equiv$
$\{r1^2 \leq n < (r1+d)^2\} \vee \{(r1+d)^2 \leq n < (r1+1+d)^2\}$
$\Rightarrow (\ d \geq 1)$
$\{r1^2 \leq n < (r1+d)^2\} \vee \{(r1+1)^2 \leq n < (r1+1+d)^2\}$

D'aquesta apreciació se'n desprèn el següent disseny recursiu:

$P(n,d) = \{n \geq 0 \wedge d > 0\}$

funció arrel31(n:enter,d:enter) **retorna** natural
  si $n < d^2 \longrightarrow$ **retorna** 0
    $n \geq d^2 \longrightarrow$
      $r1 := arrel31(n, d+1)$;
      [1]
      si $(r1+1)^2 \leq n \longrightarrow$ **retorna** $r1+1$;
        $(r1+1)^2 > n \longrightarrow$ **retorna** $r1$;
      fsi
  fsi
ffunció

Com ja hem explicat, el cost d'aquesta funció és lineal. Es pot trobar una funció similar amb cost logarítmic duplicant d a cada nova crida recursiva. Aquest disseny queda com a exercici.

Per a obtenir aquests dissenys hem optat per afeblir la postcondició de la funció *arrel* d'una manera molt determinada:

$P(n,d) = \{n \geq 0 \wedge d > 0\}$
$r := arrel3(n, d)$;
$Q(n,r,d) = \{r^2 \leq n < (r+d)^2\}$

Podem pensar en altres immersions que utilitzen maneres diferents d'afeblir la postcondició. Suggerim-ne una altra:

$P(n,d) = \{n \geq 0 \wedge d > 0\}$
$r := arrel4(n, d)$;
$Q(n,r,d) = \{r^2 * d^2 \leq n < (r+1)^2 * d^2\}$

Notem que efectivament és una immersió de *arrel* ja que $arrel4(n, 1) = arrel(n)$.

Com a exercici podeu desenvolupar una funció que satisfaci aquesta especificació.

4.6 Algunes consideracions sobre la recursivitat

El disseny recursiu aporta solucions gairebé sempre molt elegants però, mal utilitzat, pot donar lloc a problemes molt més importants que els que tracta de solucionar.

En aquest apartat sintetitzarem els punts crítics més importants que poden ser un obstacle a l'hora de fer un disseny recursiu per a la resolució d'un problema i farem algunes consideracions generals respecte la recursivitat.

4.6.1 Costos típics dels programes recursius

Programes amb una sola crida recursiva

Aquests programes generen una seqüència de crides recursives.

La forma de decrement del tamany de les dades determina, en gran mesura el cost d'un programa recursiu. En programes recursius amb una sola crida recursiva i amb cost constant de la part no recursiva ($O(1)$) obteníem els costos:

- **Lineal ($O(n)$)** si el decrement de les dades es feia de forma constant a cada nova crida recursiva (i.e.: a cada nova crida recursiva el tamany de les dades era k unitats menors que a l'anterior).
- **Logarítmic ($O(log(n))$)** si el decrement de les dades es feia en forma lineal a cada nova crida recursiva (i.e.: a cada nova crida recursiva reduíem a n/k el tamany de les dades).

Evidentment la segona família és millor que la primera però no sempre és possible trobar solucions de cost logarítmic per als problemes plantejats.

Programes amb vàries crides recursives

Aquests programes generen un arbre de crides recursives. Degut a això, el seu cost serà típicament més alt que el dels anteriors. El nombre de nodes d'un arbre binari complet és $2^n - 1$ essent n el nombre de nivells de l'arbre. Per tant, el cost d'un programa recursiu amb dues crides serà sovint exponencial ($O(2^n)$).

Aquest fet no anul.la en absolut la utilitat dels programes recursius amb vàries crides. Efectivament, hi ha problemes pels quals l'algorisme recursiu amb vàries crides no té un cost exponencial (§5.1) i n'hi ha d'altres pels quals s'ha demostrat que la solució de cost exponencial és òptima . El problema de les Torres de Hanoi és un exemple d'aquest últim grup d'algorismes.

Finalment, en altres casos, no es coneix cap algorisme millor que un amb cost exponencial per a resoldre un problema.

4.6.2 Repetició de les crides recursives

Certs programes que utilitzen recursivitat múltiple (amb vàries crides recursives) tenen el problema que repeteixen crides recursives. Aquest fet és absolutament dramàtic ja que la repetició d'una crida recursiva implica, òbviament, la repetició de tot el subarbre de crides que aquesta genera. El cost del programa es fa aleshores absolutament intractable i cal buscar una solució alternativa que el redueixi. Aquest problema l'hem vist a les solucions recursives dels problemes *fibonacci*.

4.6.3 L'ús de variables globals i de paràmetres d'entrada/sortida

És perillós utilitzar variables globals o paràmetres d'entrada/sortida en els programes recursius. Tot seguit veurem un exemple d'això.

Exemple:

var gl:natural; **fvar**

funció fibo (n:natural) **retorna** natural **és**
  si n=1 **retorna** 1;
     n=2 **retorna** 1;
     n>2 gl:=n−1;
          retorna fibo(gl)+fibo(gl−1);
  fsi
ffunció

Aquesta funció, aparentment, obté el n-èssim número de Fibonacci de forma recursiva i utilitzant una variable global (gl). Aquesta variable global, com és lògic, és del tot innecessària però aparentment no influeix en el resultat de la funció. Res més lluny de la realitat.

El problema és que quan fem la crida $fibo(gl-1)$, la variable gl pot tenir un valor diferent d'aquell que esperem que tingui (n−1). Efectivament, com gl és una variable global, l'operació que fem immediatament abans ($fibo(gl)$) pot modificar el seu valor (com efectivament fa). Un exemple d'aquest problema el tenim amb la crida $fibo(4)$. Aquesta executa el següent codi:

gl:=n−1; $(gl = 3)$
retorna fibo(gl)+fibo(gl−1);

La crida $fibo(gl)$, com a efecte lateral, modifica el valor de gl. Quan calculem el valor de $gl-1$ per a fer la següent crida recursiva, $gl = 2$ i no 3 com necessitem.

4.6.4 El cost addicional de les crides recursives

La recursivitat té, de forma inherent, un cost afegit: el que suposa la creació del bloc d'activació de les successives crides recursives. Podem enumerar dues conseqüències immediates de l'elevat nombre de crides recursives que es poden generar fins arribar al cas trivial:

- Alt cost en temps degut al procés de creació dels successius blocs d'activació de les crides.
- Alt cost en espai degut a que aquests blocs d'activació coexisteixen *apilant-se* un al damunt de l'altre. Com a conseqüència d'aquest fet, programes altament recursius poden donar lloc fàcilment a un error de *sobreeiximent de la pila del sistema.*

4.7 Transformació de programes recursius en programes iteratius

4.7.1 Motivació

A l'apartat anterior hem enumerat una llista de dificultats que fan perillós l'ús de la recursivitat. Per altra banda, la recursivitat ens dóna lloc a unes solucions conceptuals, extraordinàriament elegants i molt clares. En definitiva *els programes recursius solen resultar molt més elegants que els seus equivalents iteratius però, en general, són més ineficients.*

Com preservar, aleshores, l'elegància sense renunciar a l'eficiència?

Una forma molt evident pot ser la de generar mètodes que permetin transformar programes recursius en altres d'iteratius equivalents. Fins i tot, podem automatitzar alguns d'aquests mètodes i fer que

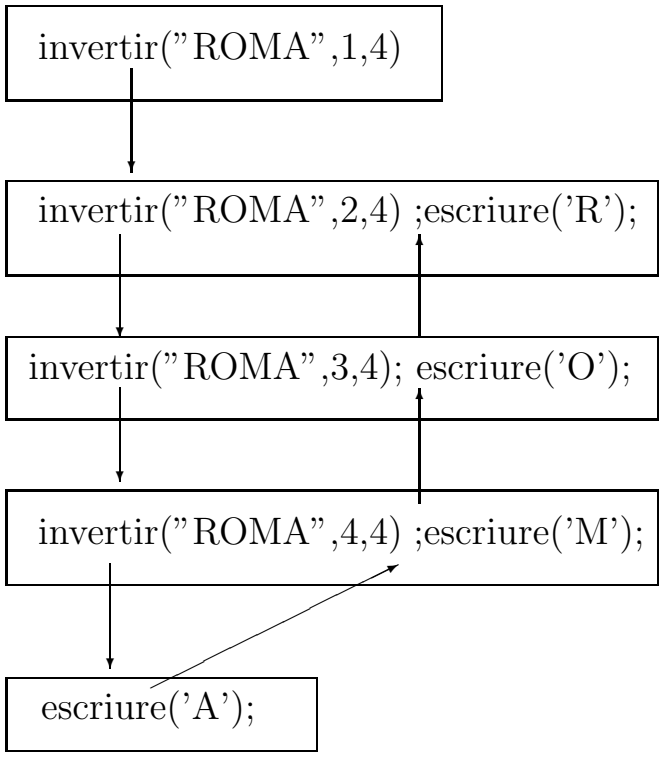


Figura 4.10: Seqüenciació d'instruccions en l'execució de l'acció *invertir*

siguin utilitzats per un compilador a l'hora de generar el codi corresponent a la traducció d'un programa recursiu.

4.7.2 Mètode de recorregut de la seqüència de crides

Presentació del mètode. Exemple

Recordem ara l'acció recursiva que invertia una cadena de caracters acabada en $\mathcal{M}$ i que ha estat presentada a l'apartat 4.1:

par:vector[1..n] de caràcter
$natural(i_1, i_2)$
$P = \{1 \leq i_1 \leq i_2 \leq n\}$

acció invertir (par:paraula, i_1:natural, i_2:natural) **és**
  si $i_1 = i_2$ **llavors** escriure($par[i_1]$);
    sinó
      invertir($par, i_1 + 1, i_2$);
      escriure($par[i_1]$);
  fsi
facció

Si fem una simulació de l'execució de l'acció *invertir*, o sigui, si seguim la seqüència d'instruccions que executa l'ordinador fins acabar l'acció, ens trobarem amb el que es presenta a la figura 4.10.

En aquesta seqüència podem distingir dos recorreguts:

- Un *descendent* fins a assolir el cas trivial ($invertir("ROMA", 4, 4)$).
- I un altre *ascendent* durant el qual es completen les funcions tals que la seva execució havia quedat interrompuda per una crida recursiva.

L'acció iterativa en què transformarem aquesta recursiva constarà, precisament, d'aquests dos recorreguts. Veiem-ho:

```
acció invertir_i (par:paraula, i1:natural, i2:natural)
  var k:natural fvar
  k:=i1;
  mentre k ≠ i2 fer
    k:=k+1;
  fmentre
  escriure(par[k]); /*acció que cal fer en el cas trivial*/
  mentre k ≠ i1 fer
    k:=k-1;
    escriure(par[k]);
  fmentre
facció
```

El primer dels recorreguts (el descendent) consisteix simplement en una reiterada aplicació de la funció *successor* fins a arribar al cas trivial.

Al segon dels recorreguts (el recorregut *ascendent*) hi ha un punt que resulta del tot essencial: *Per a poder completar l'execució de l'anterior crida recursiva ens cal recuperar els seus paràmetres. Dit d'una altra manera: ens cal tenir una forma de calcular l'anterior.*

A l'exemple de l'acció *invertir*, és clar que la crida anterior a $invertir("ROMA", i, j)$ és $invertir("ROMA", i-1, j)$, si $i > 1$. El càlcul de l'anterior és trivial en aquest cas. Ara bé, hi ha altres casos en què no existeix forma automàtica de calcular l'anterior. La solució per aquells casos serà anar guardant ordenadament en memòria els paràmetres de les successives crides que es van obtenint al primer dels recorreguts. La forma natural d'emmagatzemar en memòria aquests paràmetres és utilitzant una pila.

Formalització

Aquest mètode ens permetrà transformar programes recursius *amb una sola crida recursiva* en altres d'iteratius equivalents. Considerarem com a forma general d'una funció recursiva la que apareix a la figura 4.2.

El comportament d'aquesta funció recursiva es pot esquematitzar de la forma presentada a la figura 4.11

La idea de la transformació d'aquesta funció en una iterativa serà simular el doble recorregut que es presenta a la figura 4.11. Per això ens caldrà:

1. Identificar la seqüència de paràmetres de les successives crides recursives.

   Això bàsicament vol dir:

   (a) Decidir quin és el primer element de la seqüència x_0. (O sigui, quins són els arguments de la crida inicial).

   (b) Veure com es pot calcular el *successor* d'un element qualsevol x de la seqüència: $suc(x)$. (O sigui, establir la relació entre els paràmetres d'una crida i els de la següent).

   (c) Veure si es pot calcular l'*antecessor* d'un element qualsevol x: $ant(x)$.
   Queda clar que, donat un element de la seqüència de paràmetres de les crides $suc^k(x)$, es compleix que: $ant(suc^k(x)) = suc^{k-1}(x)$ si $k > 0$.

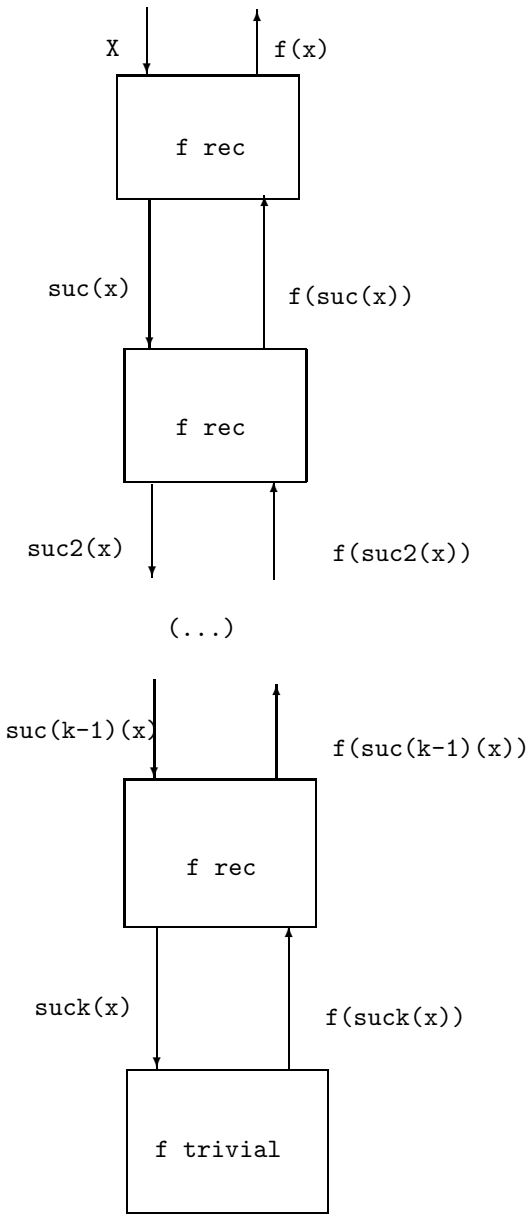


Figura 4.11: Comportament d'una funció recursiva

L'antecessor d'un element el necessitarem quan haguem de fer el segon dels recorreguts (el que completa l'execució de cada crida recursiva). Si no el podem calcular, haurem de guardar en el primer dels dos recorreguts la seqüència de paràmetres, per a poder-la recuperar després, quan es faci el segon recorregut. Notem que, com el segon recorregut es fa en sentit contrari, caldrà recuperar en primer lloc els darrers elements guardats. D'aquest fet es desprén que l'estructura de dades adient per a emmagatzemar els elements de la seqüència és una pila. (Veure exemples i apèndix).

(d) Identificar el darrer element de la seqüència. Aquell que ja no generarà cap nova crida recursiva (el cas trivial): $b_t(x)$.

2. Fer el primer recorregut de la seqüència de crides fins arribar al cas trivial.

   Aquest recorregut tindrà bàsicament la següent forma:

   mentre $suc^k(x_0) \neq x_{trivial}$ **fer**
   calcular $suc^{k+1}(x_0)$
   fmentre

3. Trobar el resultat del cas trivial:

   Calcular $tr(suc^n(x_0))$ on $b_t(suc^n(x_0)) = cert$.

4. Fer un segon recorregut de la seqüència de crides en sentit contrari. Es tracta d'anar calculant els resultats parcials successius $f(suc^k(x_0))$ fins a obtenir el resultat total $f(x_0)$.

   Aquest segon recorregut completa l'execució de la funció. (O sigui, realitza totes les operacions que queden pendents després d'obtenir el resultat de la crida recursiva que conté la funció). Per tant, a cada volta caldrà obtenir el resultat parcial corresponent a aquella volta ($f(suc^k(x))$) a partir del resultat de la crida recursiva generada ($f(suc^{k+1}(x))$) i del *paràmetre d'aquella volta* ($suc^k(x)$).

   $$\left.\begin{array}{c} suc^k(x) \\ f(suc^{k+1}(x)) \end{array}\right\} \rightarrow f(suc^k(x))$$

   Això mateix expressat en l'exemple del factorial significa:

   $$\left.\begin{array}{c} \text{n} \\ factorial(\text{n-1}) \end{array}\right\} \rightarrow factorial(\text{n}) = \text{n}^* factorial(\text{n-1})$$

   Per recuperar $suc^k(x)$, necessitarem poder-lo calcular a partir de $suc^{k+1}(x)$: $suc^k(x) = ant(suc^{k+1}(x))$ o bé tenir-lo emmagatzemat en una pila.

En conseqüència, l'esquema del mètode en pseudocodi és el següent:

```
P(x)
funció f_i (x_0 : T_1) retorna T_2 és
  var z : T_1; y : T_2 fvar;
  z := x_0
  mentre b_r(z) fer
    z:=suc(z);
  fmentre
  y := tr(z);
  mentre z ≠ x_0 fer
    z := ant(z);
    y := co(y, z);
```

fmentre
retorna y;
ffunció
$Q(x, f(x))$

L'invariant del primer recorregut és:

$I_1 = \{P(x) \wedge (\exists k : k \geq 0 \bullet suc^k(x) = z)\}$

L'invariant del segon recorregut:

$I_2 = \{I_1 \wedge y = f(z)\}$

Com a conseqüència de I_2 a la sortida de la segona iteració es complirà que $y = f(x_0)$, que és precisament el que volíem.

Per a demostrar que I_2 és efectivament invariant del segon bucle, veurem allò què es compleix en cada moment del segon bucle.

$\{y = f(suc^k(x_0)) \wedge z = suc^k(x_0) \wedge k > 0\}$

z :=ant(z);

$\{y = f(suc^k(x_0)) \wedge z = suc^{k-1}(x_0) \wedge k > 0\}$

y :=co(y, z);

$\{y = co(f(suc^k(x_0)), suc^{k-1}(x_0)) \wedge z = suc^{k-1}(x_0) \wedge k > 0\} =$

$\{y = f(suc^{k-1}(x_0)) \wedge z = suc^{k-1}(x_0) \wedge k > 0\}$

Per tant a cada nova iteració del bucle obtenim un nou resultat parcial sobre y ($y = f(z)$). A la fi obtindrem $z = x_0$ i llavors $y = f(x_0)$.

Si volem que la descripció del mètode resulti completa, hem de fer referència al càlcul de l'anterior, el qual, com ja hem dit abans, en general no és automàtic perquè la funció *successor* no sempre és injectiva. En el cas en què no tinguem una forma de calcular l'anterior (veure problema de l'algorisme xinès del producte), haurem de guardar la seqüència paràmetres de les successives crides recursives a mesura que les anem generant en el primer recorregut (el descendent). Ja hem justificat que l'estructura de dades oportuna per a guardar aquesta seqüència és una pila (perquè la recuperació dels elements de la seqüència en el recorregut ascendent es fa en ordre invers a com van ser carregats en el recorregut descendent). A l'apèndix es presenta l'estructura de dades pila, el seu comportament i una possible implementació.

L'algorisme presentat anteriorment per a transformar una acció recursiva en una altra iterativa equivalent a la primera queda de la següent manera quan el càlcul de l'anterior força l'ús d'una pila:

P(x)
funció f_i_amb_pila ($x_0 : T_1$) **retorna** T_2 **és**
  var $z : T_1$; $y : T_2$; $p : pila$ **fvar**;
  $z := x_0$
  $crear(p)$;
  mentre $b_r(z)$ **fer**
    $empilar(p, z)$;
    $z := suc(z)$;
  fmentre
  $y := tr(z)$;
  mentre $z \neq x_0$ **fer**
    $z := cim(p)$; $desempilar(p)$;
    $y := co(y, z)$;
  fmentre
  retorna y;
ffunció
$Q(x, f(x))$

Hem encetat l'apartat de transformació de funcions recursives en iteratives amb un exemple. Ara caldria fer-ne més per aclarir el mètode que hem desenvolupat. Abans, però, és interessant d'estudiar dues simplificacions d'aquest mètode general.

Simplificacions del mètode general.

A l'apartat anterior hem presentat el cas general de transformació d'una funció recursiva amb una sola crida a una altra d'iterativa. Hem vist que aquesta transformació involucrava dos recorreguts de la seqüència de crides. En aquest apartat veurem com, en alguns casos particulars, es pot eliminar un dels dos recorreguts.

Eliminació de la primera iteració La primera iteració té un doble objectiu:

1. Obtenir el $x_{trivial}$.
2. Emmagatzemar en una pila la seqüència de crides en el cas que no sigui possible calcular l'anterior a un element de la seqüència.

Aquells casos en què $x_{trivial}$ es pugui calcular directament i pels quals no calgui guardar l'element anterior de la seqüència, no necessitaran de la primera iteració. Un exemple és la versió iterativa de la funció $factorial$ (§4.7.2).

Tot seguit presentem l'esquema d'aquesta simplificació:

P(x)
funció f_i $(x_0 : T_1)$ **retorna** T_2 **és**
  var $z : T_1;\ y : T_2$ **fvar**;
  $z := cas\_trivial$;
  $y := tr(z)$;
  mentre $z \neq x_0$ **fer**
    $z := ant(z)$;
    $y := co(y, z)$;
  fmentre
  retorna y;
ffunció
$Q(x, f(x))$

Eliminació de la segona iteració: Recursivitat final. La segona iteració serveix, com ja s'ha explicat, per a completar el càlcul del resultat de la crida $f(suc^k(x))$ quan ja s'ha obtingut el resultat de la crida $f(suc^{k+1}(x))$. Hi ha casos en què no cal fer aquesta completació (i.e. el resultat de la crida $f(suc^{k+1}(x))$ i el de la crida $f(suc^k(x))$ són idèntics). En aquests casos ens estalviarem la segona iteració. Un exemple és la versió iterativa de la funció que calcula el màxim comú divisor (§4.7.2).

En definitiva, direm que una funció recursiva és final si la darrera acció que executa en cadascun dels casos recursius és una crida recursiva. L'esquema general d'una funció recursiva final (amb una sola crida recursiva) és el següent:

$P(x)$

funció $f(x : T_1)$ **retorna** T_2 **és**
  si
    $b_t(x)$ $\longrightarrow$retorna $tr(x)$
    $b_r(x)$ $\longrightarrow$retorna $f(suc(x))$
  fsi
ffunció

$Q(x, f(x))$

Notem que la funció co ha desaparegut (s'ha transformat en la identitat).

L'esquema de transforamció d'una funció recursiva final en iterativa és el següent:

P(x)
funció f_i $(x_0 : T_1)$ **retorna** T_2 **és**
  var $z : T_1;\ y : T_2$ **fvar**;
  $z := x_0$
  mentre $b_r(z)$ **fer**
    z:=suc(z);
  fmentre
  $y := tr(z)$;
  retorna y;
ffunció
$Q(x, f(x))$

Queda clar que un programa recursiu amb recursivitat final serà sempre més costós que el seu equivalent iteratiu. Molts compiladors, en la fase d'optimització del codi, són capaços de detectar la recursivitat final i de transformar-la en un esquema iteratiu equivalent. En aquest punt comprovem la utilitat de les immersions que ens permetien d'obtenir una acció recursiva final (§4.5.3)

Exemples

- **La funció factorial**

  funció factorial (n:natural) **retorna** natural **és**
    si
      n=0 **retorna** 1
      n>0 **retorna** n*factorial(n-1)
    fsi
  ffunció

  1. Identificació de la seqüència:
     Primer element $\longrightarrow n$
     $suc(n) \longrightarrow n - 1$
     $ant(n) \longrightarrow n + 1$
     $b_t(n) \longrightarrow n = 0$
     $b_r(n) \longrightarrow n > 0$
     $co(n, factorial(n-1)) \longrightarrow n * factorial(n-1)$

     Notem que en aquest cas és senzill calcular l'element anterior.
  2. Funció iterativa equivalent.
     funció factorial_i (n:natural) **retorna** natural **és**
       var z,y:natural; **fvar**
       z:=n;
       mentre z>0 **fer**
         z:=z-1;
       fmentre
       y:=1;
       mentre z≠ n **fer**
         z:=z+1;
         y:=y*z;
       fmentre
       retorna y;
     ffunció

La primera iteració és del tot innecessària en aquest cas. El cas trivial sempre és el mateix ($z = 0$) i, per altre costat, l'anterior és calculable directament. Podem, doncs, substituir la primera iteració per l'assignació **z:=0;**.

- **El màxim comú divisor**

funció maxcomdiv (m:natural,n:natural) **retorna** natural **és**
si
m>n **retorna** maxcomdiv(m−n,n);
m<n **retorna** maxcomdiv(m,n−m);
m=n **retorna** m;
fsi
ffunció

1. Identificació de la seqüència:
   Primer element$\longrightarrow (m, n)$
   $suc(m, n) \longrightarrow \begin{cases} (m-n, n) & si\ \ m > n \\ (m, n-m) & si\ \ m < n \end{cases}$
   $b_t(m, n) \longrightarrow m = n$
   $b_r(m, n) \longrightarrow m \neq n$
   $co() \longrightarrow \emptyset$

   Aquest és un típic cas de recursivitat final. Quna arribem al cas trivial ja no ens queden noves coses per fer. Ja hem calculat el mcd de tots dos paràmetres. Així doncs ens podrem estalviar el segon recorregut i, conseqüentment, el càlcul de l'anterior.

2. Funció iterativa equivalent.
   funció maxcomdiv_i (m:natural, n:natural) **retorna** natural **és**
   mentre m≠n **fer**
   si m>n **llavors** m:=m−n;
   sinó n:=n−m;
   fmentre
   retorna m;
   ffunció

- **El Producte**

funció producte (m:natural,n:natural) **retorna** natural **és**
si
n=0 **retorna** 0;
senar(n) **retorna** producte(2*m,n div 2)+m;
parell(n) **retorna** producte(2*m,n div 2);
fsi
ffunció

1. Identificació de la seqüència
   Primer element$\longrightarrow (m, n)$
   $suc(m, n) \longrightarrow (2 * m, n \text{ div } 2)$
   $b_t(m, n) \longrightarrow n = 0$
   $b_r(m, n) \longrightarrow n \neq 0$
   $co(m, n, prod(2 * m, n \text{ div } 2) \longrightarrow \begin{cases} producte(2 * m, n \textbf{ div } 2) & \text{si n parell} \\ producte(2 * m, n \textbf{ div } 2) + m & \text{si n senar} \end{cases}$
   $ant(m, n) = (m \text{ div } 2, pila)$
   En intentar calcular l'anterior de n tenim un problema: pot ser $2 * n$ o bé $2 * n + 1$ (ja que la funció *suc* no és injectiva). L'única manera de saber si és l'un o l'altre és haver guardat

en el primer recorregut tota la successió de n's. Al segon recorregut, a cada volta, anirem recuperant les n's en l'ordre invers al que s'han emmagatzemat (heus ací la necessitat de la pila).

2. Funció iterativa equivalent.

```
funció producte (m:natural, n:natural) retorna natural és
  var p:pila; z,prod:natural; fvar
  z:=n;
  crear(p);
  mentre z≠0 fer
    empilar(p,z);
    m:=2*m;
    z:=z div 2;
  fmentre
  prod:=0;
  mentre z≠ n fer
    z:=cim(p);
    desempilar(p);
    m:=m div 2;
    si senar(z) llavors prod:=prod+m;
  fmentre
  return prod;
ffunció
```

Les operacions que són *pròpies* de la pila, i també la forma que triarem per a representar en memòria aquesta estructura de dades estan descrites a l'apèndix A.

Cal notar que aquesta representació i operacions ens serviran, fent petitíssims retocs, per qualsevol altre programa recursiu que haguem de transformar en un d'iteratiu equivalent i, en general, per qualsevol programa que necessiti emmagatzemar dades en una pila. D'aquesta idea en parlarem més endavant al capítol que presenta els tipus abstractes de dades (§6).

4.8 Problemes

Nota: Encara que no s'indiqui a l'enunciat, cada programa s'haurà d'*especificar* , de *verificar* i de transformar a iteratiu.

Problema 32

Fer un programa recursiu que calculi el quocient i el residu de la divisió de dos nombres enters. Donar dues solucions, intentant millorar el cost.

Problema 33

Fer un programa que calculi l'enter a^b. Com abans, donar vàries solucions per tal de millorar el cost.

Problema 34

Dissenyar recursivament un programa que calculi la part entera del logaritme en base b d'un enter positiu.

Problema 35

Donat un vector d'enters $t[1..n]$ amb $n > 0$, dissenyar una funció recursiva que calculi el màxim dels seus components. Fer una versió amb recursivitat simple i una altra dividint el vector en dues meitats i activant la funció per cada meitat.

Problema 36

Dissenyar un programa recursiu que decideixi si un array és capicua (o sigui, si els elements a la mateixa distància dels extrems són iguals). Es demana trobar una solució que talli l'execució tan aviat com es detecti que l'array no és capicua.

Problema 37

Dissenyar un programa recursiu que trobi el vector simètric a un altre a de donat, o sigui, aquell que té a distància i de l'extrem esquerre l'element que a té a distància i del seu extrem dret.

Problema 38

Dissenyar un programa recursiu que calculi la matriu transposada a una donada. Us serà útil l'acció del problema anterior.

Problema 39

Dissenyar un algorisme recursiu que calculi el número n-èssim de Fibonacci. Criticar l'eficiència de la solució obtinguda. Dissenyar finalment, un altre algorisme que calculi el mateix però amb una sola crida recursiva.

Problema 40

Construir un algorisme recursiu que calculi el número combinatori $\begin{pmatrix} m \\ n \end{pmatrix}$.

Problema 41

Es disposa d'un vector d'enters $t[1..n]$ amb $n \geq 0$. Es desitja construir un programa que, donat aquell vector, retorni un valor booleà que indiqui si algun dels seus elements és igual a la suma dels que el precedeixen.

Problema 42

Un programador ha decidit representar un polinomi d'una variable mitjançant un vector de manera que l'element i-èssim del vector correspon al coeficient de la potència i-èssima. A la representació hi afegeix un enter que representa el grau del polinomi. Amb aquesta representació, dissenyar un programa que

avalui el polinomi en un punt qualsevol.

Problema 43

Donada una constant natural n i un vector $a[1..n]$ de naturals, decidir si existeix la manera de descomposar-lo en una part esquerra i una altra dreta que sumin el mateix. Aquestes parts poden ser de tamany desigual.

Problema 44

Donat un vector de enters deiferents ordenat de forma creixent, es desitja decidir si algun dels elements del vector coincideix amb l'índex allà on està situat.

Problema 45

Dissenyar recursivament una funció tal que, donats dos vectors ordenats de forma creixent: $a[1..n]$ i $b[1..m]$ amb n i m diferents i $m \geq 0, n \geq 0$ retorni un booleà que indiqui si la seva intersecció és no buida.

Problema 46

Dissenyar una funció recursiva que faci una cerca dicotòmica sobre un cert array t.

Problema 47

La funció d'Ackerman està definida per a tota parella de valors enters no negatius de la següent forma:

$A(0, n) = n + 1$

$A(m, 0) = A(m - 1, 1) \qquad (m > 0)$

$A(m, n) = A(m - 1, A(m.n - 1)) \qquad (m, n > 0)$

Fer un programa recursiu que calculi $A(m, n)$. Provar el programa. Davant l'èxit de la prova, intentar fer una versió iterativa del programa.

Problema 48

Trobar totes les combinacions diferents de nombres naturals que en sumen un de donat.

Fer-ho de dues maneres:

1. Permetent que hi puguin haver nombres repetits en una combinació.

   Exemple:

   Podem sumar 5 de les següents maneres:

   (1,1,1,1,1),(1,1,1,2),(1,1,3),(1,2,2),(1,4),(2,3),(5)

2. Imposant que tots els nombres que apareguin en una combinació siguin diferents.

   Exemple:

Amb aquesta restricció podem sumar 5 de les següents maneres:

(1,4),(2,3),(5)

Ajut: Fer una immersió per a trobar totes les combinacions que sumin n però contenint només valors més grans o iguals que i.

Problema 49

A l'apartat d'immersions han quedat proposats alguns problemes relatius a diversos desenvolupaments de la funció *arrel* utilitzant diferents hipòtesis. Fer aquells desenvolupaments.

Problema 50 (Examen juny 95)

Sigui $\{a, b\}$ un alfabet amb el qual es pot construir paraules tals com: *aabb, ab, a, b, ababba, aaa,...* i també la paraula buida λ. Es tracta de fer una acció recursiva que tracti amb paraules construïdes únicament amb lletres d'aquest alfabet (a, b). Concretament, l'acció tindrà la següent especificació:

$\{n \geq 0 \wedge n < MAXC\}$

mots_senars_a(n:natural,p:**sort** pila_de_paraules)

$\{\forall m \in p : (\mid m \mid = n \wedge \mid m_a \mid = 2 * k + 1) \wedge$
$\forall m \notin p : (\mid m \mid \neq n \vee \mid m_a \mid = 2 * k\}$

On MAXC és el nombre màxim de lletres que pot emmagatzemar una paraula, $\mid m \mid$ vol dir el nombre de lletres que conté el mot m i $\mid m_a \mid$ vol dir el nombre de lletres a que conté el mot m.

Aquesta postcondició es pot expressar en llenguatge natural aproximadament de la següent manera: **p** *conté totes les paraules de l'alfabet de longitud* **n** *que contenen un nombre senar de lletres 'a'.*

[1] *(2,5 punts)* Dissenyar una acció recursiva que compleixi l'especificació proposada.

[2] *(1 punt)* Verificar aquesta acció, parant especial esment en l'argumentació del fet que la relació recurrent trobada sigui correcta.

Problema 51 (Examen setembre 95)

Sigui v un vector d'enters definit entre 1 i N: v:vector[1..N] de enter.

Direm que aquest vector conté una seqüència bítona si està dividit en una primera subseqüència ascendent, seguida immediatament d'una segona subseqüència descendent i de forma que el darrer element de la primera subseqüència sigui, a la vegada, el primer de la segona. Una seqüència bítona es composa d'almenys un element.

Entre les següents seqüències, les 4 primeres són bítones, mentre que la darrera no ho és:

1 3 5 6 8 6 4 2
1 2
2 1
1
1 3 5 6 8 7 9 5 4

Volem dissenyar una funció tal que donat un vector i uns límits inferior (ini) i superior (fi) del mateix retorni l'índex del vector entre ini i fi que serveix de darrer element de la seqüència ascendent i de primer de la descendent o bé -1, si el vector no conté una seqüència bítona entre ini i fi.

$\{P\} = \{1 \leq ini \leq fi \leq N\}$

r:=bitona(v,ini,fi)
$\{Q\}$

[1] Proposa una postcondició per a la funció *bítona*.
[2] Dissenya una funció *bítona* recursiva final que satisfaci les especificacions donades. Verifica-la.
[3] Transforma la funció recursiva en una equivalent iterativa.

Problema 52 (Examen juny 94)

[1] En primer lloc, es tracta de fer el disseny verificat d'una funció recursiva amb la següent especificació:

$\{P\} = \{0 < b \leq 10 \wedge n \geq 0\}$
canvi(n:natural, b:natural) *retorna* z:natural
{Q}={z és un natural tal que la seva expressió decimal coincideix amb el natural n escrit en base b}

Com a exemple, $canvi(14, 2)$ retorna el natural 1110, els dígits del qual coincideixen amb l'expressió de 14 en base 2. Seguint el mateix raonament, $canvi(14, 3)$ retorna el natural 112 i $canvi(14, 4)$ retorna 32.

Aquest disseny l'heu de fer en ANSI-C.*(2 punts)*

[2] Per tal de fer una verificació més rigorosa, podries formalitzar la postcondició Q que hem expressat intuitivament a l'especificació de la funció? *(0,5 punts).*

Ara ens proposem de transformar la funció recursiva *canvi* a una funció *canvi2* iterativa i equivalent, seguint el mètode del *recorregut de la seqüència de crides*. Hi havia casos en els quals, aquesta transformació era possible de fer sense utilitzar una pila.

[3] Et sembla que aquest és un d'aquells casos? Justifica-ho. *(0,25 punts).*

Si a la pregunta **[3]** has cregut necessària la utilització d'una pila, ens haurem de posar d'acord sobre quines operacions pròpies de les piles es podran utilitzar en la transformació que us he proposat i quina serà l'especificació de tals operacions. Això equival a considerar l'estructura de dades *pila* com un tipus abstracte de dades i a fer-ne la seva especificació. Tot seguit us la presento:

tipus abstracte pila **especificació**

- crear() $\longrightarrow$ p:pila
  Pre={cert}
  Post={p és la pila buida}
- empilar(p:pila, e:element) $\longrightarrow$ p2:pila
  Pre={p no està plena}
  Post={$p2$ és la pila formada pels mateixos elements que p i un nou element e al capdamunt de la pila. }
- cim(p:pila) $\longrightarrow$ e:element
  Pre={p no és buida}
  Post={e és l'element del capdamunt de p (és a dir, el darrer element empilat a p)}
- desempilar(p:pila) $\longrightarrow$ p2:pila
  Pre={p no és la pila buida}

Post={$p2$ conté els mateixos elements que p llevat de l'element del capdamunt de p que ha estat eliminat.}

fi_especificació_tad pila;

[4] Amb l'ajut de l'especificació del t.a.d. *pila*, fes una funció *canvi2* iterativa i equivalent a la funció recursiva *canvi* usant el mètode de recorregut de la seqüència de crides. *(1 punt).*

Problema 53 (Examen juny 96)

Els professors de primer curs de l'EUP ens hem proposat que acabeu essent uns experts en el puzzle anomenat *torres de hanoi*. És per aquest motiu que us plantejo un altre problema relacionat.

Donats els tres eixos habituals del puzzle de les torres de Hanoi, es tracta de dissenyar l'acció *hanoi_dreta* que té com a objectiu transportar totes les peces d'un cilindre al seu veí dret utilitzant el tercer eix com a eix de moviments i tenint en compte les següents regles:

1. Només es pot moure una peça en cada moviment.
2. En un eix mai no hi pot haver una peça gran damunt d'una altra de més petita.
3. Només estan permesos els moviments d'una peça al seu eix veí dret (i.e. moure una peça des de l'eix 0 a l'eix 1, des de l'eix 1 a l'eix 2 i, també, des de l'eix 2 a l'eix 0. Cap altre moviment és considerat legal). Notem que aquesta regla és nova respecte la formulació tradicional del passatemps.

Direm que un moviment és *legal* si s'ajusta a aquestes tres regles.

Es demana:

1. (2 punts) Fes un disseny verificat de l'acció *hanoi_dreta* amb la següent especificació:

   hanoi_dreta(n:natural, i1:natural, i2:natural, mov:natural)
   Pre:{$n \geq 0 \wedge i2 = (i1 + 1)\ mod\ 3$}
   Post:{S'han generat i escrit per pantalla una seqüència de moviments legals per a transportar tota una torre de n peces des de l'eix $i1$ fins l'eix $i2$ utilitzant l'eix *mov* per a fer els moviments}

2. (*)(1,5 punts) El nombre de moviments generats per l'acció que has dissenyat a l'apartat anterior, és el mínim possible?

   Si ho és, demostra-ho. Si no, troba un disseny per *hanoi_dreta* que generi el mínim nombre de moviments. Concretament, l'especificació d'aquesta acció haurà de ser la següent:

   hanoi_dreta_min(n:natural, i1:natural, i2:natural, mov:natural)
   Pre:{$n \geq 0 \wedge i2 = (i1 + 1)\ mod\ 3$}
   Post:{S'ha generat i escrit per pantalla la seqüència **més petita** de moviments legals per a transportar tota una torre de n peces des de l'eix $i1$ fins l'eix $i2$ utilitzant l'eix *mov* per a fer els moviments}

   Suggeriments:

   - Per a fer aquest disseny mínim et serà útil disposar de l'acció *hanoi_esquerra_min* amb l'objectiu de transportar n peces des d'un eix al seu veí esquerre amb el mínim nombre possible de moviments legals.
   - Pensa amb quines són les *situacions clau* per les que s'ha de passar necessàriament en la resolució de *hanoi_dreta_min* i *hanoi_esquerra_min*. Utilitza llavors la inducció per a determinar com arribar a aquelles situacions clau amb el mínim nombre de moviments.

Problema 54 Examen setembre-96

Considerem la següent especificació d'una certa acció f:

a, b:**vector**$[1..N]$ **de** enter

a està inicialitzat.

Pre=$\{1 \leq ini \leq N \wedge \exists i, ini \leq i \leq N : a[i] = 0\}$

$f(a, ini) \longrightarrow < b, fi >$

Post=$\{0 \leq fi < N \wedge \forall j : 0 \leq j < fi : a[ini + j] = b[fi - j] \wedge a[ini + fi] = 0 \wedge b[fi + 1] = 0\}$

a i ini són paràmetres d'entrada que f no modifica.

1. (0,25) Expressa en llenguatge natural el què creus que fa l'acció f. Siguis rigorós[5]. En particular, parla de la marca de final pels vectors a i b. Dóna un altre nom per f relacionat amb el seu objectiu.
2. (1,25) Fes un disseny de f.
3. (1) Verifica'l. Para especial esment en assegurar que qualsevol indexació dels vectors a o b es fa dins del seu rang (1..N).
4. (0,75) Transforma f a iteratiu seguint el procediment del recorregut de la seqüència de crides.
5. (0,25) Quina informació necessitaries per a eliminar un dels dos bucles que apareixen a la versió iterativa?

Problema 55 El viatjant de comerç

Es té un mapa de carreteres que uneixen un conjunt de ciutats i també es coneixen les distàncies entre dues qualssevol d'aquelles ciutats. Un viatjant de comerç les ha de visitar totes i està pensant en fer un itinerari que passi per totes elles de forma que s'estalviï la major quantitat de benzina possible.

[5]Òbviament no us estic demanant que respongueu: "fi està entre 0 i N (estrictament menor) i per a tot j entre 0 i fi es compleix que a de ini+j és igual a ...." sinó que interpreteu amb tot rigor el sentit de la postcondició.

Capítol 5

Esquemes algorítmics

5.1 Esquema divideix_i_venç

5.1.1 Idea de l'esquema

L'estratègia *divideix_i_venç* per a resoldre un problema P consisteix en el següent:

1. Dividir aquell problema P en un conjunt de subproblemes $P_1, P_2, ..., P_n$.
2. Trobar la solució de cadascun dels P_i: $S_1, ..., S_n$.
3. Combinar adequadament les solucions parcials $S_1, .., S_n$ per a obtenir la solució total S del problema.

Aquest esquema el podem escriure en forma de funció general:

funció divideix_i_venç(D) **és**
  $\{D_1, ..., D_n\}$ ⟵fer_partició(D);
  per $i := 1$ **fins a** n **fer**
    S_i :=divideix_i_venç(D_i);
  fper
  retorna construir_solució_total($S_1, S_2, ..., S_n$);
ffunció

On l'acció *fer_partició(D)* retorna una llista de subconjunts $D_1, D_2, ..., D_n$ tals que:

1. $D_i \neq D_j$ per a tot $i, j \in 1..n$.
2. $D_1 \cup D_2 \cup ... \cup D_n = D$.

I l'acció *construir_solució_total*($S_1, S_2, ..., S_n$) combina les solucions parcials S_i per a construir la solució total del problema S.

5.1.2 Exemples

Màxim i mínim

El problema consisteix en trobar el màxim i el mínim d'una seqüència d'enters emmagatzemada en un vector entre dos índexos i_{esq} i i_{dreta}.

Una manera d'abordar aquest problema usant l'estratègia *divideix_i_venç* consisteix en:

1. Dividir la seqüència S en dues subseqüències iguals (aproximadament) S_1 i S_2.
2. Calcular recursivament el màxim i el mínim de cadascuna de les dues subseqüències. Anomenem-los max_1, max_2 i min_1, min_2.
3. El màxim total de la seqüència serà el $MAXIM(max_1, max_2)$. Mentre que el mínim total de la seqüència serà el $MINIM(min_1, min_2)$.

La següent acció en pseudocodi realitza aquest procés:

$P = \{1 \leq i_{esq} \leq i_{dreta} \leq N \wedge N > 0\}$

acció màxim_mínim(T:**ent vector**[1..N] **de** enter;
i_{esq}, i_{dreta}: **ent** natural;
min, max:**sort** enter)
si $i_{esq} = i_{dreta}$ **llavors** $min := T[i_{esq}]$; $max := T[i_{esq}]$;
sinó
$i_{mig} := (i_{dreta} + i_{esq})$ **div** 2;
max_min$(T, i_{esq}, i_{mig}, min1, max1)$;
max_min$(T, i_{mig} + 1, i_{dreta}, min2, max2)$;
max:=maxim$(max1, max2)$;
min:=minim$(min1, min2)$;
fsi
facció

$Q = \{min = minim(T[i_{esq}]...T[i_{dreta}]) \wedge max = maxim(T[i_{esq}]...T[i_{dreta}]\}$

Cost de la solució

Per a obtenir el cost de l'algorisme que acabem de proposar plantejarem una equació recurrent en la que diferenciarem els costos del cas trivial i del recursiu. El cost el mesurarem com el nombre de comparacions necessàries per a acabar l'algorisme. Anomenem $nc(n)$ al nombre de comparacions necessàries per arribar al final de l'algorisme partint de n elements a tractar. Suposem, per simplificar que n és una potència de 2.

- $nc(1) = 0$

  En el cas recursiu (número d'elements a tractar=1) no cal fer cap comparació.

- $nc(n) = 2 * nc(n/2) + 2$ [1]

  Aquest fóra el nombre de comparacions a fer en el cas recursiu.

El cost de la crida recursiva amb tamany n ($nc(n)$) ve donat en [1] en termes del cost de la crida recursiva amb tamany $n/2$ ($nc(n/2)$). Podríem posar el cost de la crida amb tamany n ($nc(n)$) en termes del cost d'una crida qualsevol amb tamany $n/2^i$ on $i \geq 0$. Per aconseguir això farem el que s'anomena un *desplegament*. O sigui, substiuirem a [1] el cost $nc(n/2)$ pel seu valor segons l'equació recurrent:

$nc(n) = 2 * (2 * nc(n/4) + 2) + 2 = 4 * nc(n/4) + 6$

Si continuem sustituint cada $nc(n/2^i)$ per $2 * nc(n/2^{i+1}) + 2$ podrem observar la següent relació de recurrència que posa un terme n en funció d'un altre qualsevol $n/2^i$:

$nc(1) = 0$

$nc(n) = 2^i * nc(n/2^i) + 2^{i+1} - 2$ [2]

On i és precisament el nombre de recurrències fetes per tal d'arribar a unes dades de tamany $n/2^i$.

El cas trivial s'assoleix per $n/2^i = 1$, o sigui, per $i = log(n)$. Sustituint en [2] obtindrem el nombre total de comparacions que haurem fet després de $i = log(n)$ recurrències:

$nc(n) = 2^{log(n)} * nc(1) + 2^{log(n)+1} - 2 = 2 * n - 2.$

Aquest nombre de comparacions coincideix exactament amb el que cal fer amb el programa iteratiu habitual per a resoldre aquest problema. Per tant, podem pensar que aquesta tècnica no ens ha ajudat a reduir el cost del problema.

Què passaria, però, si avancéssim un pas el cas trivial? O sigui, què passaria si consideréssim com a cas trivial un vector amb dos elements?

A la següent acció podem veure el resultat:

$P = \{1 \leq i_{esq} \leq i_{dreta} \leq N \wedge N > 0\}$

acció màxim_mínim(T:**ent vector**[1..N] **de** enter;
i_{esq}, i_{dreta}: **ent** natural;
min, max:**sort** enter)
si
$i_{esq} = i_{dreta}$ **llavors** $min := T[i_{esq}]$; $max := T[i_{esq}]$;
$i_{dreta} - i_{esq} = 1$ **llavors**
si T[i_{dreta}]< T[i_{esq}] **llavors** max:=T[i_{esq}];min:=T[i_{dreta}];
sinó max:=T[i_{dreta}];min:=T[i_{esq}];
fsi
$i_{dreta} - i_{esq} > 1$ **llavors**
$i_{mig} := i_{dreta} + i_{esq}$ **div** 2;
max_min($T, i_{esq}, i_{mig}, min1, max1$);
max_min($T, i_{mig} + 1, i_{dreta}, min2, max2$);
max:=maxim($max1, max2$);
min:=minim($min1, min2$);
fsi
facció

$Q = \{min = minim(T[i_{esq}]...T[i_{dreta}]) \wedge max = maxim(T[i_{esq}]...T[i_{dreta}]\}$

La diferència entre aquesta acció i l'anterior rau en el fet que mentre que en l'anterior feiem dues comparacions per a esbrinar el màxim i el mínim d'un vector de dos elements (doncs era un cas recursiu com un altre qualsevol), en aquesta només en fem una:

si T[i_{dreta}]< T[i_{esq}] **llavors** max:=T[i_{esq}];min:=T[i_{dreta}];
sinó max:=T[i_{dreta}];min:=T[i_{esq}];
fsi

Si plantegem i resolem l'equació recurrent adient per aquest cas obtenim:

$nc(2) = 1$

$nc(n) = 2^i * nc(n/2^i) + 2^{i+1} - 2$

Que, per un cas en que $n = 2^k$, arribaríem al cas trivial quan es complís la condició $n/2^i = 2$, o sigui, per $i = log(n) - 1$.

Per aquesta i, el nombre de comparacions a fer fóra de:

$nc(n) = 2^{log(n)-1} * nc(2) + 2^{log(n)} - 2 = 3/2 * n - 2$.

Aquest resultat, si bé no redueix l'ordre de magnitud (aquest continua essent $O(n)$), si que decrementa sensiblement el cost.

Al capítol 3 presentem un altre exemple d'aplicació d'aquest mètode.

5.2 Esquema de tornada enrera (backtracking)

5.2.1 Idea de l'esquema

Sovint ens trobem davant problemes en què cal trobar una solució dins un espai de cerca sense que tinguem cap mètode algorísmic concret per a trobar-la. En aquests casos, l'única alternativa és l'exploració exhaustiva d'aquell espai de cerca. Quan arribem a un punt en què tinguem l'evidència que no podrem trobar una solució a partir d'aquell punt, abandonarem aquella branca de la cerca, tornarem enrera i en reprendrem una altra que ens pugui portar a la solució.

Un exemple molt clar d'aquest plantejament és el joc dels escacs. Avui per avui no es coneix cap *mètode* per a guanyar als escacs. Si el coneguéssim tal vegada podríem fer un programa que jugués als escacs, programant aquell mètode algorísmic. En canvi, el que fem, és, a partir de la situació actual del tauler, explorar totes les situacions a les que s'arriba després de la realització de totes les jugades *prometedores* i totes les situacions a les que s'arribarà després de les jugades que pot fer l'oponent com a reposta a cadascuna de les nostres possibles jugades... Quan arribem a una posició que ens sigui desfavorable, rebutjarem aquella branca i tornarem enrera per a continuar l'exploració d'alguna altra branca més *prometedora*.

Un altre exemple pot ser la cerca de la sortida d'un laberint. Si a l'entrada trobéssim un full de paper amb la seqüència de moviments que ens portessin a la sortida de forma òptima, aquell full fóra el programa que, de la forma més exacta i eficient, ens permetria sortir del laberint. Però si això no és així no ens quedarà altre remei que fer una exploració exhaustiva del laberint tot i recordant les sales que ja hem visitat. Quan entrem per segona vegada a una sala, recordarem que ja l'hem visitada i tornarem enrera tot i buscant un altre camí cap a la sortida.

Aquest mètode de resolució de problemes s'anomena de tornada enrera (*backtracking*) perquè té la capacitat de descartar aquelles possibilitats de les quals es té una evidència de que no conduiran enlloc i, en aquell cas, reprendre'n una altra.

En definitiva, el *backtracking* es basa en l'exploració exhaustiva d'un espai de cerca fins el descobriment de la solució. Aquesta exploració, sovint ve guiada per funcions que avaluen si la situació actual està aprop o no de la solució final (heurístiques).

Originàriament, aquests mètodes de cerca se'ls va emmarcar dins l'àmbit de la Intel.ligència Artificial i es van utilitzar (amb versions més sofisticades) com a mètodes generals per a resoldre aquells problemes de cerca de solucions pels quals no es coneixia cap algorisme de resolució més específic i eficient. Se'ls va anomenar *mètodes febles de resolució de problemes.*

Definició 13

Un algorisme que utilitza l'esquema de tornada enrera o backtracking per trobar la solució a un determinat problema (pel qual no es coneix cap estratègia més específica de solució) consisteix a partir d'un inici de solució (o solució parcial) S_p a aquell problema (S_p pot ser inicialment buida) i intentar-la completar de totes les maneres possibles ($c_1, c_2, ..., c_n$).

- *Si una d'aquestes maneres ($S_p \cup \{c_i\}$) dóna lloc a una solució completa S_c l'algorisme acaba amb èxit (o bé continua cercant la resta de les solucions).*
  Sinó, es desestima i es prova de completar S_p amb la següent continuació possible ($S_p \cup \{c_{i+1}\}$). Notem que aquest és el pas de tornada enrera.
- *Si, després de provar totes les possibilitats de completar S_p ($\{c_1, ..., c_n\}$), cap d'elles permet trobar una solució S_c, es conclou que S_p no es pot completar fins a generar una solució completa.*

Tot seguit presentem dos exemples.

5.2.2 Exemple 1: Les vuit dames

Aquest problema consisteix en trobar una manera de disposar vuit dames en un tauler d'escacs de forma que no es matin entre elles. Una extensió del problema intentaria cercar totes les possibles disposicions de vuit dames en un tauler d'escacs de manera que no es matin entre elles. De moment nosaltres ens centrarem en trobar una única manera.

Com que no es coneix cap estratègia que permeti situar les dames ràpidament, ens veiem obligats a plantejar el problema com la cerca exhaustiva d'una solució dins l'espai de cerca constituït per totes les maneres possibles de col.locar vuit dames en un tauler. Quan trobem alguna manera de col.locar-les que compleixi que cap dama en mati cap altra, haurem trobat una solució del problema i podrem acabar la cerca. Aquest plantejament del problema ens porta directament a dissenyar un algorisme de *tornada enrera o backtracking* que realitzi aquesta cerca exhaustiva.

$P = \{\forall i : 1 \leq i \leq 8 \bullet f[i] = 0\}$

acció vuit_dames(f:**vector**[1..8] **de** *enter*)

$Q = \{\forall i : 1 \leq i \leq 8 \bullet 1 \leq f[i] \leq 8 \wedge$
$\forall i : 1 \leq i \leq 8, \forall j : 1 \leq j \leq 8 \bullet no\_mata(i, f[i], j, f[j])\}$

Ens cal explicar dos aspectes d'aquesta especificació:

- El vector f té el següent significat intuïtiu:

$$\forall i : 1 \leq i \leq 8 \bullet f[i] = \begin{cases} 0 & \text{no hi ha cap dama assignada a la columna } i \\ k \ \ (1 \leq k \leq 8) & \text{La dama de la columna } i \text{ està situada a la filera } k \end{cases}$$

- El predicat $no\_mata(i, j, k, m)$ val *cert* si una dama situada a la posició (i, j) (filera i, columna j) del tauler no mata a una altra dama situada a la posició (k, m) del tauler.

Si bé l'especificació que acabem de fer és força intuïtiva, no resulta gens apropiada per fer un disseny d'un algorisme recursiu que resolgui el problema de les vuit dames usant la tècnica de la tornada enrera. Recordem que a la introducció hem establert que els algorismes de *tornada enrera* es basen en la *compleció exhaustiva*[1] *d'una solució parcial*. Així doncs, haurem de permetre que el paràmetre f contingui, a l'inici de l'acció, una solució parcial (això és, algunes dames col.locades a algunes columnes del tauler). L'objectiu de l'acció serà llavors el de col.locar dames a la resta de les columnes de totes les maneres possibles (tot deixant immòbils les de les primeres) fins a trobar una combinació de vuit dames que no es maten entre elles. D'això, en direm *completar una solució parcial*.

Aquesta idea porta, ara sí a una bona especificació per l'acció *vuit_dames*:

$P = \{\forall i : 1 \leq i \leq n \bullet 1 \leq f[i] \leq 8 \wedge$
$\forall i : n < i \leq 8 \bullet f[i] = 0 \wedge$
$0 \leq n \leq 8 \wedge f[1..n] = F[1..n]\}$

acció vuit_dames_g(n : *enter*, f:**vector**[1..8] **de** *enter*, *trobat* : *boolea*)

$Q = \{(trobat \Longrightarrow \forall i : 1 \leq i \leq 8 \bullet 1 \leq f[i] \leq 8 \wedge$
$\forall i : 1 \leq i \leq 8, \forall j : 1 \leq j \leq 8 \bullet \neg mata(f[i], i, f[j], j) \wedge f[1..n] = F[1..n]) \wedge$
$(\neg trobat \Longrightarrow \forall f[1..8] : f[1..n] = F[1..n] \bullet$
$\exists i, j : 1 \leq i, j \leq 8 \bullet \text{mata}(f[i], i, f[j], j))\}$

La precondició la podem llegir de la manera següent: *El vector f conté una dama col.locada a cadascuna de les columnes des de la 1 a la n, ambdues incloses*. Per la seva banda, la postcondició es pot llegir de la manera següent: *Si trobat val cert, f conté una combinació de dames que no es maten entre elles*

[1] Quan diem *exhaustiva* fem referència a *totes les maneres possibles.*

que és una continuació del valor inicial de f (i.e. $f[1..n] = F[1..n]$). Si trobat vals $fals$, no hi ha cap combinació possible de vuit dames al tauler que deixin intactes les n primeres de forma que no es matin entre elles. Com es pot apreciar, el vector f actua com a solució parcial.

Per fer el disseny seguirem fil per randa el procediment que hem donat a la introducció: Partirem d'una solució parcial de longitud n (n dames col.locades al tauler) i la intentarem allargar afegint-hi una nova dama ($n+1$ dames) a la columna $n+1$ de totes les maneres possibles fins que s'arribi a alguna de les dues situacions següents:

- A partir d'alguna d'aquestes solucions parcials de longitud $n+1$ que acabem de construir, s'ha pogut obtenir (recursivament) la solució final (formada per vuit dames col.locades al tauler sense amenaçar-se dues a dues).
- Després d'intentar completar (recursivament) totes les solucions parcials de longitud $n+1$ que hem construït no s'ha pogut arribar a cap solució amb les vuit dames correctament situades. En aquell cas, i considerant que hem estat exhaustius a l'hora d'afegir una dama a la solució parcial de longitud n, podem concloure que no és possible allargar aquesta solució parcial per a trobar una solució completa.

Aquesta reflexió porta al següent disseny:

acció vuit_dames_g($n, f, trobat$) **és**
  si $n = 8 \longrightarrow trobat := cert$;
    $n < 8 \longrightarrow$
      $lcan$:=llis_disponibles($f, n+1$);
      $trobat := fals$;
      mentre ($\neg$ buit($lcan$) $\wedge \neg trobat$) **fer**
        $f[n+1]$:= primer($lcan$);
        $lcan$:=cua($lcan$);
        vuit_dames_g($n+1, f, trobat$);
      fmentre
  fsi
facció

L'acció *llis_disponibles*(f, n) retorna una llista amb totes les fileres i ($1 \leq i \leq 8$) tals que una dama col.locada al tauler a la posició (i, n) (per qualsevol $i \in l$) no amenaça a cap de les situades a les columnes $1..n-1$.

L'acció *primer* retorna respectivament el primer element de la llista que es passa per paràmetre. L'acció *cua* retorna la llista formada per tots els elements llevat del primer.

Encara una millora

Un darrer aspecte interessant d'aquest algorisme és el manteniment de la llista que conté en tot moment les fileres *disponibles* o *candidates* per col.locar una dama en una determinada columna. Conceptualment, aquesta llista és molt aclaridora però resulta poc pràctic de mantenir-la. Això és així per dues raons: En primer lloc, ocupa espai i, en segon lloc, necessita d'una estructura de dades (que pot ser perfectament una pila) per a ser implementada, la qual cosa afegeix una complexitat addicional a l'algorisme. Us proposo de *no utilitzar aquesta llista llevat del cas que sigui estrictament necessari*. I quan serà estrictament necessari d'utilitzar-la? La resposta és simple: *quan no sigui possible calcular el següent candidat a partir de l'actual*.

Per veure si és aquest el cas que ens ocupa al problema de les dames ens haurem de preguntar si és possible implementar una funció *següent_disponible*(f, n, f_actual) que retorni la propera filera (a partir de la filera f_actual) en la qual es pot col.locar una dama a la columna n sense amenaçar les que ja estan

col.locades a les coumnes anteriors (i que es troben a $f[1..n-1]$). En el cas que no existeixi cap filera (a partir de f_actual) que compleixi aquesta condició per la columna n, la funció podria retornar 0.

Sembla clar que aquesta funció *següent_disponible* és fàcilment implementable:

funció següent_disponible(f, n, f_actual) **retorna** natural
  $trobat := fals$; $i := f\_actual + 1$;
  mentre $i \leq 8 \wedge \neg trobat$ **fer**
    $trobat :=$ *posicio_no_amenacada*(f, i, n);
    $i := i + 1$;
  fmentre
  si $trobat$ **retorna** $i - 1$;
    sinó retorna 0;
  fsi
ffunció

posició_no_amenaçada(f, i, n) retorna *cert* si la posició (i, n) (filera i, columna n) no està amenaçada per cap altra dama present al tauler i retorna $fals$ en cas contrari.

Amb l'ajut de la funció *següent_disponible* podem prescindir de la lliusta de candidats. Veiem-ho:

acció vuit_dames_g_2($n, f, trobat$) **és**
  si $n = 8 \longrightarrow trobat := cert$;
    $n < 8 \longrightarrow$
      $f_actual :=$seguent_disponible$(f, n + 1, 0)$;
      $trobat := fals$;
      mentre $(f\_actual \neq 0 \wedge \neg trobat)$ **fer**
        $f[n + 1] := f\_actual$;
        vuit_dames_g_2$(n + 1, f, trobat)$;
        $f_actual :=$seguent_disponible$(f, n + 1, f_actual)$;
      fmentre
  fsi
facció

5.2.3 Exemple 2: La sortida del laberint

Plantejament del problema

Imaginem un laberint com un seguit d'habitacions unides per passadissos. Podem passar d'una habitació a l'altra si hi ha un passadís que les comunica.

Es tracta de dissenyar una acció que trobi un recorregut d'habitacions del laberint, que no en repeteixi cap i que permeti passar des d'una habitació d'entrada a una altra de sortida. Si l'habitació de sortida no és accessible des de la d'entrada, l'acció ho ha d'indicar.

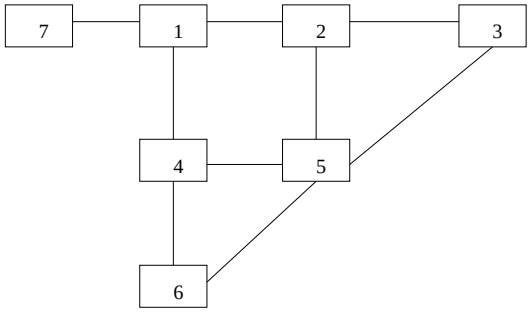
Per exemple, si al laberint de la figura volem trobar un recorregut des de l'habitació 1 a la 6, podem fer:

1,2,5,4,6

1,2,3,5,6

1,4,6

...



Representació del laberint

El laberint el podem representar com una matriu *lab* tal que:

- *lab*[*i*][*j*]= cert si hi ha un passadís que connecta les habitacions *i* i *j*.
- *lab*[*i*][*j*]= fals si no hi ha cap passadís que connecta les habitacions *i* i *j*.

Considerarem que $lab[i][i] = false$ per a tota habitació *i*.

Com els passadissos es poden recórrer en ambdós sentits, la matriu *lab* serà simètrica.

El laberint dibuixat a la figura anterior es podria representar amb una matriu *lab* com la següent (1 indica cert i 0 indica fals):

	1	2	3	4	5	6	7
1	0	1	0	1	0	0	1
2	1	0	1	0	1	0	0
3	0	1	0	0	1	0	0
4	1	0	0	0	1	1	0
5	0	1	1	1	0	0	0
6	0	0	0	1	1	0	0
7	1	0	0	0	0	0	0

Especificació de la solució

Plantejarem la solució a aquest problema seguint l'esquema de backtracking. Cal recordar que les accions que segueixen aquest esquema tenen l'objectiu de **completar de totes les maneres possibles una solució incompleta o parcial** fins trobar una solució completa, fins trobar-les totes o fins deduir que no n'hi ha cap.

Tenint això en ment, plantejarem una acció `sortidaLaberint` amb l'especificació següent:

acció sortidaLaberint(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **e/s** vector[1..N] de natural,
nsol: **e/s** natural, *hsort*: **ent** natural, *trobat*: **sort** booleà)

- **Pre:**
  - *lab*[1..*nhab*][1..*nhab*] està inicialitzada i representa un laberint de *nhab* habitacions tal com s'ha explicat a la secció 2
  - $1 \leq nhab \leq N$ (el nombre màxim d'habitacions d'un laberint és N)

  - $sol[1..nsol]$ representa la solució parcial que hem de completar.
    $sol[1..nsol]$ conté una llista d'habitacions tals que:
    * $sol[i]$ és veïna de $sol[i+1]$ (o sigui: $lab[sol[i]][sol[i+1]] = 1$, per a tot $i : 1 \leq i < nsol$)
    * $sol[1..nsol]$ no conté habitacions repetides
  - $1 \leq nsol \leq nhab$ (la solució parcial conté almenys una habitació que serà l'habitació d'entrada).
  - $1 \leq hsort \leq nhab$ ($hsort$ és l'habitació de sortida del laberint)

- **Post:**
  - Si $trobat = cert$, aleshores:
    * $sol[1..nsol]$ conté una llista d'habitacions tals que:
      · $sol[i]$ és veïna de $sol[i+1]$ (o sigui: $lab[sol[i]][sol[i+1]] = 1$, per a tot $i : 1 \leq i < nsol$)
      · $sol[1..nsol]$ no conté habitacions repetides
      · $sol[nsol] = hsort$ (la darrera habitació de la solució és l'habitació de sortida, per tant, la solució és completa)
      · $sol[1..nsol]$ és una compleció de $sol'[1..nsol']$ (sol completa la solució parcial que teníem a la precondició)[2].
    * $1 \leq nsol \leq nhab$ (el nombre d'habitacions de la solució completa és, com a màxim el nombre d'habitacions del laberint, ja que no admetem recorreguts amb habitacions repetides).
  - Si $trobat = fals$, aleshores:
    * No existeix cap recorregut pel laberint que completi la solució parcial $sol'[1..nsol']$ i que acabi a $hsort$
    * $nsol = nsol'$ (el valor de $nsol$ no ha sigut modificat per l'acció *sortidaLaberint*).
      Si no incloem aquesta condició no tenim controlat el valor que acaba prenent $nsol$ en una crida recursiva que acaba amb $trobat = fals$.

Disseny de l'acció *sortidaLaberint*

L'acció *sortidaLaberint*

acció sortidaLaberint(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **e/s** vector[1..N] de natural,
nsol: **e/s** natural, *hsort*: **ent** natural, *trobat*: **sort** booleà) **és**

var *seghab*: natural; **fvar**

si $sol[nsol] = hsort \longrightarrow trobat := cert$;
si no $\longrightarrow$
  $nsol := nsol + 1$; //*****[1]
  $seghab$:=seguentVeina($lab, nhab, sol, nsol - 1, 0$);
  mentre $seghab \leq nhab \wedge \neg trobat$ **fer**
    $sol[nsol] := seghab$;
    sortidaLaberint($lab, nhab, sol, nsol, hsort, trobat$);
    $seghab$:=seguentVeina($lab, nhab, sol, nsol - 1, seghab$);
  fmentre
  si $\neg trobat \longrightarrow nsol := nsol - 1$;
fsi //*****[2]

[2]Recordem que sol' fa referència al valor del vector sol a la precondició i $nsol'$ fa referència al valor de $nsol$ a la precondició

facció

Si considerem el laberint de les seccions 1 i 2, la crida inicial a aquesta acció podria ser:

sortidaLaberint(lab, 7, *sol*, 1, 6, *trobat);*

on:

- *lab* podria ser la matriu presentada a la secció 2
- $nhab = 7$
- $sol = \{1\}$ (agafant 1 com a habitació d'entrada al laberint). Partim d'una solució parcial formada únicament per l'habitació d'entrada.
- $hsort = 6$ (6 és l'habitació de sortida)
- *trobat* és un booleà de sortida.

L'acció *seguentVeina*

La crida:

seguentVeina($lab, nhab, sol, nsol - 1, habant$)

Retorna la següent veïna encara no explorada de la darrera habitació de la solució parcial *sol* (per tant, retorna la següent candidata a continuar la solució parcial).

Tinguem en compte dues coses respecte aquesta crida:

- El nombre d'habitacions de la solució parcial és $nsol - 1$ (ja que a l'inici de l'acció, concretament a [1], incrementem *nsol* per tal de poder-lo passar adequadament com a paràmetre de la crida recursiva).
- *habant* és la darrera habitació explorada com a candidata a continuar la solució parcial (si volem obtenir la següent candidata, hem de prendre com a referència quina va ser l'anterior).

Si fem una especificació més exacta de l'acció seguentVeina, obtindrem:

funció seguentVeina(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **ent** vector[1..N] de natural,
ns: **ent** natural, *hant*: **ent** natural) **retorna** natural

que es crida:

seghab:=seguentVeina($lab, nhab, sol, ns, hant$)

- **Pre:** $sol[1..ns]$ conté una solució parcial i *hant* és una habitació que no està a $sol[1..ns]$ i que ja ha estat explorada.
- **Post:** *seghab* és l'habitació del laberint *lab* veïna de $sol[ns]$ que és la més petita entre totes les veïnes de $sol[ns]$ que són més grans que *hant* i que no apareixen a la solució parcial $sol[1..ns]$.

  Si no hi ha cap habitació que compleixi aquesta condició (i.e., ja hem exhaurit totes les habitacions candidates a continuar la solució parcial $sol[1..ns]$), aleshores $seghab = nhab + 1$.

Codi de `seguentVeina`

funció seguentVeina(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **ent** vector[1..N] de natural,
ns: **ent** natural, *hant*: **ent** natural) **retorna** natural **és**

$seghab := hant + 1$;
$trobat := fals$;
mentre $seghab \leq nhab \wedge \neg trobat$ **fer**
si $repetit(seghab, sol, ns) \vee lab[ns][seghab] = fals \longrightarrow seghab := seghab + 1$;
si no $trobat := cert$;
fsi
fmentre
retorna *seghab*;
ffunció

$repetit(seghab, sol, ns)$ retorna cert si *seghab* es troba al vector $sol[1..ns]$. La implementació d'aquesta acció implicarà fer una cerca de *seghab* al vector $sol[1..ns]$.

Si no us agrada `seguentVeina`

En lloc d'usar *seguentVeina* podem donar una solució basada en una llista de candidats, com ja vam fer a la primera versió del problema de les vuit dames. A la llista de candidats aquí l'anomenem *lveines* (llista d'habitacions veïnes):

acció sortidaLaberint2(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **e/s** vector[1..N] de natural,
nsol: **e/s** natural, *hsort*: **ent** natural, *trobat*: **sort** booleà) **és**

var *lveines*: vector[1..N] de natural; **fvar**
si $sol[nsol] = hsort \longrightarrow trobat := cert$;
si no $\longrightarrow$
$lveines$:=llistaHabVeines($lab, nhab, sol, nsol$);
$nsol := nsol + 1$;
mentre $\neg buida(lveines) \wedge \neg trobat$ **fer**
$sol[nsol] := primer(lveines)$;
sortidaLaberint2($lab, nhab, sol, nsol, hsort, trobat$);
$treurePrimer(lveines)$;
fmentre
si $\neg trobat \longrightarrow nsol := nsol - 1$;
fsi
facció

Aquesta acció potser resulta més entenedora però és més ineficient: recordeu que cada nova crida recursiva que es faci generarà una còpia de la variable local *lveines*, que és un vector de N naturals. Això malmetrà molt espai i temps.

L'increment de *nsol*

Els paràmetres d'entrada-sortida o de sortida d'una acció sempre s'han de cridar amb variables. Mai no poden ser expressions.

Seria incorrecta una crida tal com:

sortidaLaberint(lab, nhab, sol, $\mathbf{nsol + 1}$*, hsort, trobat);*

ja que el quart paràmetre d'aquesta acció és d'entrada-sortida.

Per aquest motiu, a la instrucció [1] de l'acció *sortidaLaberint* incrementem *nsol*

D'altrabanda, com l'especificació ens demana que si no hem trobat una solució, *nsol* ha de valdre el mateix que a la precondició, incorporem la instrucció [2] i així restaurem el valor que inicialment tenia *nsol*.

Les instruccions [1] i [2] són conseqüència del fet que el paràmetre *nsol* és d'e/s.

nsol compleix dues funcions:

1. A l'inici de l'acció ens indica **quantes habitacions té la solució parcial**
2. A la fi de l'acció ens indica **quantes habitacions té la solució completa**

Si la mida d'una solució completa del problema del laberint fos constant (com ho era en el cas de les 8 dames: tota solució completa tenia exactament $n = 8$ dames col.locades), *nsol* no hauria de fer la segona funció. En conseqüència seria d'entrada i les instruccions [1] i [2] no serien necessàries.

La solució òptima

Podem plantejar ara el problema de trobar la solució al problema del laberint que passi per un menor nombre d'habitacions. En el cas del laberint de les seccions 1 i 2, aquesta solució òptima seria:

$sol = \{1, 4, 6\}$

Efectivament, no hi ha cap altra manera d'anar des de l'habitació 1 a la 6, visitant un nombre menor d'habitacions.

Una primera solució

Proposem una primera solució que calcula la solució òptima en un vector de sortida anomenat *solopt*.

$solopt[1..nsolopt]$ contindrà la compleció més curta de $sol[1..nsol]$ que acabi a *hsort*.

En cas que $solo[1..nsol]$ no es pugui completar fins *hsort*, $nsolopt = nhab + 1$.

```
acció sortidaLaberintOpt( lab: ent matriu[1..N][1..N] de booleà,
                          nhab: ent natural, sol: ent vector[1..N] de natural,
                          nsol: ent natural, hsort: ent natural,
                          solopt: sort vector[1..N] de natural, nsolopt: sort natural) és

   var soloptaux: vector[1..N] de natural; naux: natural;
      seghab: natural;
fvar

   si sol[nsol] = hsort ⟶ copiar(solopt, nsolopt, sol, nsol);
   si no ⟶
        nsolopt := nhab + 1;
        seghab:=seguentVeina(lab, nhab, sol, nsol, 0);
        mentre seghab ≤ nhab fer
           sol[nsol + 1] := seghab;
           sortidaLaberintOpt(lab, nhab, sol, nsol + 1, hsort, soloptaux, naux);
           si naux < nsolopt ⟶
                 copiar(solopt, nsolopt, soloptaux, naux);
           fsi
           seghab:=seguentVeina(lab, nhab, sol, nsol, seghab);
        fmentre
   fsi
facció
```

Aquesta acció introdueix alguns elements nous respecte *sortidaLaberint*:

1. La solució cercada **no** la posem a *sol* sinó a *solopt*. Això fa que *sol* i *nsol* puguin ser paràmetres d'entrada i no haguem de fer les instruccions [1] i [2] de *sortidaLaberint*.

2. Usem *nsolopt* com a substitut de la variable booleana *trobat*. $trobat = fals$ s'indica en aquest cas pel fet que $nsolopt > nhab$. Notem, en aquest sentit, que no aturem la nostra cerca en el moment en què trobem una solució (com féiem a *sortidaLabertint*), sinó que exhaurim tots els candidats, tot i esperant que algun d'ells ens doni la solució més curta. Això es nota a la condició de continuació del **mentre**:

   - **mentre** $seghab \leq nhab$ **fer**
     en el cas de *sortidaLaberintOpt*
   - **mentre** $seghab \leq nhab \wedge \neg trobat$ **fer**
     en el cas de *sortidaLaberint*

   En general, sempre que es vulguin trobar **totes** les solucions (i no aturar-se a la primera), cal treure la condició $\neg trobat$ del mentre.

3. Usem un condicional, immediatament després de la crida recursiva, per copiar la solució que aquesta crida acaba de trobar sobre *solopt*, en cas que sigui millor que la *solopt* que teníem fins ara.

Una altra versió amb modificacions a l'especificació

L'acció *sortidaLaberintOpt* té el problema que, novament, tenim un vector (*soloptaux*) com a variable local de l'acció recursiva. Aquesta variable local es replica a cada nova crida recursiva a *sortidaLaberintOpt*.

Si proposem una simple modificació a l'especificació de l'acció *sortidaLaberintOpt*, podem oblidar-nos d'aquesta variable local:

La idea és convertir *solopt* i *nsolopt* en paràmetres d'entrada-sortida:

acció sortidaLaberintOpt2(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **ent** vector[1..N] de natural,
nsol: **ent** natural, *hsort*: **ent** natural,
solopt:**e/s** vector[1..N] de natural, *nsolopt*: **e/s** natural)

Amb aquesta modificació, l'acció *sortidaLaberintOpt2* retorna a $solopt[1..nsolopt]$ la solució més curta entre totes les que:

- Completen $sol[1..nsol]$ i, a més a més:
- Milloren $solopt'[1..nsolopt']$, o sigui, són més curtes que $solopt'$ (que és la solució òptima a l'inici de l'acció)

Disseny

acció sortidaLaberintOpt2(*lab*: **ent** matriu[1..N][1..N] de booleà,
nhab: **ent** natural, *sol*: **ent** vector[1..N] de natural,
nsol: **ent** natural, *hsort*: **ent** natural,
solopt: **e/s** vector[1..N] de natural, *nsolopt*: **e/s** natural) **és**

var *seghab*: natural; **fvar**

si $sol[nsol] = hsort \longrightarrow$
si $nsol < nsolopt \longrightarrow$ copiar($solopt, nsolopt, sol, nsol$); **fsi**
si no $\longrightarrow$
si $nsol < nsolopt - 1 \longrightarrow$ ***[3]
$seghab$:=seguentVeina($lab, nhab, sol, nsol, 0$);
mentre $seghab \leq nhab$ **fer**
$sol[nsol+1] := seghab$;
sortidaLaberintOpt2($lab, nhab, sol, nsol+1, hsort, solopt, nsolopt$);
$seghab$:=seguentVeina($lab, nhab, sol, nsol, seghab$);
fmentre

fsi
fsi
facció

Comentaris a la solució

- Com ara *solopt* és també d'entrada, podrem passar a cada crida recursiva la solució més curta trobada fins el moment, i això estalvia haver de definir un vector local amb qui fer les comparacions.
- La instrucció condicional [3] té una funció simplement optimitzadora: si el nombre d'habitacions de la solució parcial *sol* és igual o un unitat menor que el nombre d'habitacions de la solució òptima *solopt* no té sentit intentar completar *sol* perquè està clar que no millorarem la que, de moment, és l'òptima.

  En tot cas, l'algorisme seguiria funcionant correctament sense aquesta instrucció condicional.
- La crida inicial pel laberint que mostràvem a les seccions 1 i 2 pot ser la següent:

  sortidaLaberintOpt2(lab,7,sol,1,6,solopt,8);

  On:

- *lab* és la matriu presentada a la secció 2
- 7 és el nombre d'habitacions del laberint (*nhab*)
- *sol* és $\{1\}$, prenent com a habitació d'entrada la 1.
- 1 és el nombre d'habitacions de *sol* (*nsol*).
- *solopt* ho podem deixar indefinit
- 8 és $nhab + 1$

  O sigui, posem com a solució òptima inicial, la pitjor possible: una que té $nhab+1$ habitacions (recordem que tota solució tindrà un màxim de *nhab* habitacions).

  Si a la fi de l'execució de *sortidaLaberintOpt2*, *nsolopt* continua essent $nhab+1$ voldrà dir que no s'ha pogut trobar cap solució per arribar a *hsort* a partir de l'habitació 1.

Capítol 6

Algoritmes d'ordenació

6.1 L'algorisme de la bombolla

Ens plantegem de derivar formalment l'algorisme de la bombolla per a ordenar un vector d'enters. L'algorisme consisteix en el següent:

En cada pas, el vector es considera dividit en dues parts: una primera ordenada i una segona no ordenada i tal que cada element d'aquesta segona part és més gran que qualsevol dels elements de la primera. Heus ací un exemple:

1 3 4 7 10 9 23 8 20

En aquest exemple la primera part constaria dels 4 primers elements i la segona dels 5 darrers.

L'objectiu de cada pas serà el d'afegir un nou element a la fi de la primera part, en concret, d'afegir-hi el menor dels elements de la segona part[1]. Així obtindrem un vector amb un nou element ordenat:

1 3 4 7 **8** 10 9 23 30

6.1.1 L'especificació

En primer lloc, especifiquem l'acció

t: vector$[1..N]$ de *enter*;
n : *enter*;

$P = \{t[1..n] = T[1..n] \wedge 1 \leq n \leq N\}$

bombolla(t, n);

$Q = \{ordenat(t, 1, n) \wedge t[1..n] = PERM(T[1..n])\}$

Com sempre, el predicat *ordenat* ve definit de la manera següent:

$ordenat(t, i, j) = \forall k : i \leq k < j \bullet t[k] \leq t[k + 1]$.

La postcondició Q estableix, a més a més de l'ordenació de t entre els índexos 1 i n, el fet que els valors que conté t al final són una permutació dels que contenia a l'inici (i.e. no hem modificat els antics valors de t en el procés d'ordenació del vector. Només hem modificat les seves posicions). A partir d'ara, obviarem aquesta propietat a tots els algorismes d'ordenació de vectors que trobarem en aquest text.

[1]El nom de *bombolla* ve del fet que a cada pas, l'element més *lleuger* de la segona part puja a afegir-se al final de la primera.

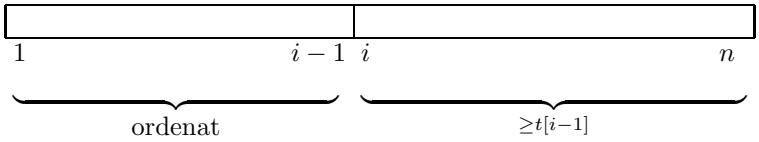


Figura 6.1: Expressió gràfica de l'invariant de *bombolla*

6.1.2 L'obtenció de l'invariant

La postcondició Q és molt estricta i estableix que la segona part del vector a la que feiem referència és buida (tot el vector està ordenat). La forma d'afeblir aquesta postcondició per a obtenir l'invariant (I_1) serà afegir la variable i que indicarà el punt de tall entre ambdues parts: l'ordenada ($1..i-1$) i la no ordenada ($i..n$). No ens hem d'oblidar de caracteritzar la part no ordenada tot dient que tots els seus elements són més grans que els de la part ordenada. Tot seguit el proposem:

$I_1 = \{ordenat(t, 1, i-1) \wedge 1 \leq i \leq n \wedge \forall p : 1 \leq p \leq i-1, \forall k : i \leq k \leq n \bullet t[p] \leq t[k]\}$

Gràficament, aquest invariant es mostra a la figura 6.1.

L'esquema de l'acció que hem de desenvolupar és el següent:

{P}
acció bombolla(t, n) **és**
  inicialització;
  mentre B_1 **fer**
    fer_pujar; (reestablir)
    avançar;
  fmentre
facció
{Q}

6.1.3 La condició de continuació del bucle

Amb una senzilla reflexió sobre I, ens adonarem que n'hi ha prou amb sortir del bucle quan $i = n$(i.e. $B_1 = (i < n)$) per tenir el vector t ordenat. Efectivament, si $\neg B1 = (i = n)$, tindrem:

$I \wedge \neg B_1 \Longrightarrow$

$ordenat(t, 1, n-1) \wedge \forall p : 1 \leq p \leq n-1 \bullet t[p] \leq t[n]$

(substituint en I_1 i per n, ja que $\neg B_1 \Longrightarrow i = n$)

$\Longrightarrow ordenat(t, 1, n)$

6.1.4 Inicialització, fita per la iteració i mode d'avançar

- Inicialització: $i := 1$.
- Fita iteració: $t = n - i$.
- Avançar: $i := i + 1$;

La inicialització $i := 1$ fa cert I_1 a l'inici de la primera iteració per domini nul d'aplicació.

Per altra banda, I_1 estableix que t està ordenat fins a $i-1$, per B_1 i per Q deduïm que i haurà d'avançar fins a n. Aquestes consideracions fan automàtiques les propostes tant de fita com de procediment per a avançar.

L'acció *bombolla* amb totes les consideracions fetes queda:

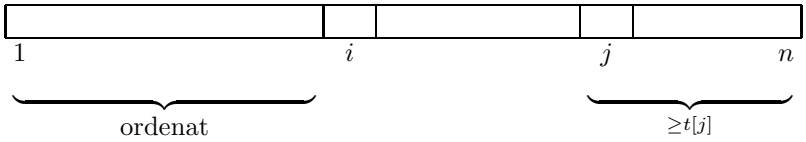


Figura 6.2: Expressió gràfica de I_2.

acció bombolla (t, n) **és**
  $i := 1$;
  mentre $i < n$ **fer**
    fer_pujar;
    $i := i + 1$;
  fmentre
facció

6.1.5 L'acció *fer_pujar*

Ara ja ens podem concentrar en l'acció *fer_pujar*. L'objectiu intuïtiu d'aquesta acció hauria de ser el de reestablir I tot fent pujar l'element menor dels de la segona part del vector $(i..n)$ a la posició i. D'aquesta manera obtindrem un vector ordenat des de la posició 1 fins la i. Això és precisament el que s'estableix a la següent especificació:

$P_2 = \{I_1 \wedge B_1\}$=$\{ordenat(t, 1, i-1) \wedge 1 \leq i \leq n-1 \wedge$
$\forall p : 1 \leq p \leq i-1, \forall k : i \leq k \leq n \bullet t[p] \leq t[k]\}$

fer_pujar;

$Q_2 = I_{1\,i+1}^{\,i} = \{ordenat(t, 1, i) \wedge 1 \leq i+1 \leq n \wedge \forall p : 1 \leq p \leq i, \forall k : i+1 \leq k \leq n \bullet t[p] \leq t[k]\}$

Aquest enunciat Q_2 el podem reescriure de la següent forma un xic més senzilla:

$Q_2 = \{ordenat(t, 1, i) \wedge 1 \leq i+1 \leq n \wedge \forall k : i \leq k \leq n \bullet t[i] \leq t[k]\}$

Notem que abans de fer_pujar, tenim t ordenat des de 1 fins a $i-1$ i tots els elements del vector des de i fins a n són més grans que els anteriors. L'objectiu de fer_pujar serà doncs, afegir un nou element a la part ordenada del vector (que passarà a tenir una longitud i). Per a que es compleixi la Q_2, aquest element haurà de ser el més petit de la part no ordenada.

fer_pujar no és una acció trivial. Ja es veu que per fer pujar l'element menor fins la posició i necessitarem plantejar un nou bucle.

Per a obtenir un invariant I_2 per aquest segon bucle, afeblirem Q_2 centrant-nos en la propietat que ha de complir la part no ordenada del vector. En lloc de demanar que $t[i]$ sigui menor o igual que tots els elements des de $i+1$ fins a n (com fa Q_2), demanarem només que, en cada moment, el valor associat a un cert índex j del vector t ($t[j]$) sigui el més petit de tots els valors que el segueixen (i que ja haurem revisat). Evidentment, j és un índex que es mou entre i i n.

$I_2 = \{P_2 \wedge \forall k : j \leq k \leq n \bullet t[j] \leq t[k] \wedge i \leq j \leq n\}$

Hem inclòs també P_2 dins de l'invariant perquè aquest enunciat fa referència a propietats de la part ordenada del vector, la veracitat de les quals **no** s'altera per l'execució de la iteració. L'expressió gràfica de I_2 es dóna a la figura 6.2.

Notem que quan j arribi a valdre i, tindrem a la posició i del vector el menor de tots els de la segona part. Com sabem que els valors $t[1], ..., t[i-1]$ són menors que els $t[i], ..., t[n]$, tindrem que el vector estarà ordenat fins la posició i, tal com estableix Q_2.

El codi de *fer_pujar* tindrà aquest aspecte:

$\{P_2\}$

$inic_2$;
mentre B_2 **fer**
  S;
  avançar;
fmentre
$\{Q_2\}$

- $B_2 = (j > i)$ Obtinguda a partir de $I_2 \wedge \neg B_2 \Longrightarrow Q_2$.
- Fita $t = j - i$
- Inicialització: $j := n$; Per garantir I_2 abans de la primera iteració.
- Avançar: $j := j - 1$;

Per a desenvolupar el codi de S convé tenir clares quines són les seves precondició i postcondició:

$\{I_2 \wedge B_2\} S \{I_2{}_{j-1}^{j}\}$

$I_2 \wedge B_2 = \{P_2 \wedge \forall k : j \leq k \leq n \bullet t[j] \leq t[k] \wedge i \leq j \leq n \wedge j > i\}$

S

$I_2{}_{j-1}^{j} = \{P_2 \wedge \forall k : j - 1 \leq k \leq n \bullet t[j-1] \leq t[k] \wedge i \leq j - 1 \leq n\}$

La precondició indica que tots els valors de t a partir de j són més grans o iguals que $t[j]$. La postcondició demana el mateix però a partir de $j - 1$. Per tant, l'única cosa que cal fer és comparar $t[j]$ amb $t[j-1]$ i intercanviar-los si $t[j-1]$ és més gran que $t[j]$. Així doncs, S serà:

si $t[j-1] > t[j]$ **llavors** intercanvi$(t[j], t[j-1])$

Per acabar, cal dir que la certesa de P_2 no es veu alterada per l'acció S i que la condició $i \leq j - 1$ ve garantida per la precondició $(j > i)$ amb la qual cosa podem concloure que s'assolirà la postcondició després de S.

acció bombolla (t, n) **és**
  $i := 1$;
  mentre $i < n$ **fer**
    $j := n$;
    mentre $j > i$ **fer**
      si $t[j-1] > t[j]$ **llavors** intercanvi$(t[j], t[j-1])$;
      $j := j - 1$;
    fmentre
    $i := i + 1$;
  fmentre
facció

6.2 Algorisme d'ordenació per selecció directa

Una manera molt intuïtiva (encara que, com veurem, no massa eficient) d'ordenar un vector consisteix en imaginar-lo dividit en dues parts: una primera ordenada i una segona no ordenada (i amb tots els seus elements més grans o iguals que els de la part ordenada). En cada pas de l'algorisme caldrà afegir el més petit dels elements de la part no ordenada a la fi de la part ordenada.

Com a exemple proposem el següent vector:

índex	1	2	3	4	5	6	7	8	9	10	11	12
valor	1	3	5	6	10	30	40	35	32	50	55	34

La part ordenada s'estén entre els índexos 1 i 6. A partir de l'índex 7 comença la part desordenada. Si ara donem un pas més a l'algorisme, prendrem el valor més petit de la part desordenada (32 situat a l'índex 9) i l'intercanviarem amb el que ocupa l'índex 7. D'aquesta forma farem créixer la part ordenada en un element.

EL vector després d'haver donat aquest pas quedaria de la següent manera:

índex	1	2	3	4	5	6	7	8	9	10	11	12
valor	1	3	5	6	10	30	32	35	40	50	55	34

6.2.1 L'especificació

L'especificació d'aquest algorisme és la mateixa que la de l'algorisme de la bombolla (v. tema 1) i, en general, coincidirà amb l'especificació de la gran majoria dels mètodes d'ordenació de vectors.

t:vector[1..N] de enter inicialitzat

$P = \{1 \leq n \leq N\}$

selecció(t,n);

$Q = \{\text{ordenat } (t, 1, n)\}$

ordenat $(t, i, j) = \forall k : i \leq k < j \bullet t[k] \leq t[k + 1]$.

6.2.2 L'invariant

L'invariant s'obté bàsicament afeblint la postcondició. De tota manera, caldrà afegir-hi alguna propietat addicional que es deriva directament del mètode amb el qual volem fer l'ordenació. De fet, la pròpia descripció que hem fet del mètode ens suggereix l'invariant:

> *Si aturem l'execució a l'inici de qualsevol volta del bucle tindrem el vector dividit en dues parts: la primera ordenada fins a j i la segona no ordenada però amb la propietat que tots els seus elements seran més grans que els de la part ordenada.*

Aquest és l'invariant que cal mantenir cert a cada volta. Ara només cal formalitzar-lo:

$I = \{ordenat(t, 1, j) \wedge 0 \leq j \leq n \wedge \forall i_1 : 1 \leq i_1 \leq j, \forall i_2 : j + 1 \leq i_2 \leq n \bullet t[i_1] \leq t[i_2]\}$

6.2.3 La inicialització, la condició de continuació i la fita

La inicialització Per a fer que I es compleixi a l'inici de la primera volta, cal fer la inicialització: $j := 0$.

Cal notar que $j := 1$ no funcionaria perquè no podem garantir que $t[1]$ sigui el menor de tots els elements del vector.

La condició de continuació (B) L'obtenim a partir de la propietat $I \wedge \neg B \Longrightarrow Q$. Aparentment $\neg B = (j = n)$ és la bona, però si optem per $\neg B = (j = n - 1)$ ens estalviem una volta i també arribem a la postcondició (com ens garanteix el darrer conjuntant de l'invariant).

Per tant: $B = (j \neq n - 1)$.

Notem que ara podríem ser restringir una mica més l'àmbit de j a l'invariant ($j \leq n - 1$). Concretament podríem canviar-lo:

$I_2 = \{ordenat(t, 1, j) \wedge 0 \leq j \leq n - 1 \wedge \forall i_1 : 1 \leq i_1 \leq j, \forall i_2 : j + 1 \leq i_2 \leq n \bullet t[i_1] \leq t[i_2]\}$

La fita Trivialment, $t = (n - j)$.

Fins ara hem derivat l'algorisme:

acció selecció(t,n) és
  j:=0;
  mentre $(j \neq n-1)$ **fer**
    reestablir;
    avançar;
  fmentre
facció

$I_2 = \{ordenat(t,1,j) \wedge 0 \leq j \leq n-1 \wedge \forall i_1 : 1 \leq i_1 \leq j, \forall i_2 : j+1 \leq i_2 \leq n \bullet t[i_1] \leq t[i_2]\}$

$t = n - j$.

6.2.4 El cos del bucle

L'acció *avançar* és clarament $j := j + 1$ i, pel que fa a restablir, si volem que a la sortida de la volta, el vector estigui ordenat fins l'índex $j+1$ (com així serà després d'avançar) hem de garantir que $t[j+1]$ sigui el més petit de tots els elements de la segona part del vector (des de $j+1$ fins a n). Per això proposem la següent composició:

j_{min}:=mínim_índex$(t, j+1, n)$;
intercanvi$(t[j_{min}], t[j+1])$;

on i_min:=mínim_índex(t, i, k) té la següent especificació:

t :**vector** $[1..n]$ **de** enter;

$P = \{1 \leq i \leq k \leq n\}$
$Q = \{\forall \mathrm{j}, i \leq j \leq k \bullet t[i\_min] \leq t[j] \wedge i \leq i\_min \leq k\}$

Notem que aquesta especificació és la versió pel mínim de l'acció que troba el màxim dels elements d'un vector i que vam presentar al capítol 1 [2].

Tot plegat ens queda:

acció selecció(t,n) és
  j:=0;
  mentre $(j \neq n-1)$ **fer**
    j_{min}:=mínim_índex$(t, j+1, n)$;
    intercanvi$(t[j_{min}], t[j+1])$;
    j:=j+1;
  fmentre
facció

$I_2 = \{ordenat(t,1,j) \wedge 0 \leq j \leq n-1 \wedge \forall i_1 : 1 \leq i_1 \leq j, \forall i_2 : j+1 \leq i_2 \leq n \bullet t[i_1] \leq t[i_2]\}$

$t = n - j$.

El cost d'aquest algorisme és $O(n^2)$ (per què?), que és equiparable al de la *bombolla* i, en general, a tots els algorismes poc astuts d'ordenació. Al següent apartat presentarem un algorisme *astut* que rebaixa el seu cost a $O(n * log(n))$ en el cas mitjà.

[2] Hi ha la petita diferència que en aquella acció trobàvem l'element màxim del vector i en aquesta **l'índex** allà on es troba l'element mínim del mateix. Però, com es evident, els canvis als que ens obliga aquesta circumstància són insignificants.

6.3 Ordenació ràpida (*quicksort*)

El *quicksort* es un algoritmo inventado por Hoare que permite ordenar de forma en promedio muy eficiente un vector de tamaño grande rebajando el coste $O(n^2)$ de los algoritmos poco astutos a $O(n * log(n))$ en el caso medio. En los siguientes apartados vamos a desarrollar una versión recursiva del algoritmo[3], a continuación la transformaremos en otra iterativa equivalente y, finalmente, calcularemos su coste.

6.3.1 Desarrollo del algoritmo recursivo

El algoritmo del *quicksort* es una aplicación del esquema *dividir_y_vencer*. No es el único: algún otro algoritmo de ordenación como el *mergesort* aplica ese mismo esquema aunque de un modo distinto.

La estrategia global que seguiremos para el desarrollo del algoritmo del *quicksort* es la siguiente: *Dividiremos el vector en dos partes de tal manera que la ordenación, por separado, de cada una de ellas dé como resultado la ordenación total del vector.* Por supuesto, deberemos establecer qué propiedad deben cumplir esas dos partes para que su ordenación separada dé lugar a la ordenación total del vector. Este es el punto más importante del *quicksort.*

Consideremos un vector $t[1..N]$ y dos índices inf y sup de este vector con la condición $1 \leq inf$ y $sup \leq N$. Especifiquemos la acción *quicksort* que deberá ordenar t entre los índices inf y sup:

acción quicksort (t: **e/s** vector, inf:**ent** natural, sup: **ent** natural)
$\{P\} = \{1 \leq inf \wedge sup \leq N\}$
$\{Q\} = \{\forall i : inf \leq i \leq sup, \forall j : inf \leq j \leq i \bullet t[j] \leq t[i]\}$

La estrategia global que hemos apuntado más arriba nos lleva a un primer esbozo de la solución que buscamos:

```
acción quicksort (t: e/s vector, inf:ent entero, sup: ent entero)
  si ??? ⟶ ???
       ??? ⟶
         ???
         {R}
         quicksort(t,inf,???);
         quicksort(t,???,sup);
  fsi
  facción
facción
```

El plan a seguir para completar este algoritmo será el siguiente:

1. Decidir una función limitadora conveniente para el algoritmo y unos parámetros para cada llamada recursiva que la hagan decrecer y que cumplan la precondición de esa llamada.
2. Calcular unas protecciones adecuadas para la instrucción condicional que nos permitan separar el caso de resolución trivial de aquél que necesitará un cálculo recursivo.
3. Calcular R, es decir, aquella propiedad que deberá cumplir el vector t para que las dos ordenaciones parciales realizadas recursivamente sean suficientes para ordenar el vector entre las posiciones inf y sup.
4. Desarrollar las acciones que permitirán alcanzar R.

[3]Para la derivación de esta acción utilizamos la formulació de J.L. Balcázar en su libro *Programación Metódica* McGraw-Hill, Madrid 1993, la cual nos parece muy elegante.

1. Función limitadora

$f(inf, sup) = sup - inf + 1$.

Como esta expresión debe ser mayor o igual a cero, deberemos añadir un nuevo conjuntando a la precondición que imponga esta condición:

$sup + 1 \geq inf$.

Ahora la precondición queda del siguiente modo:

$\{P\} = \{1 \leq inf \leq sup + 1 \leq N + 1\}$

Con la cual hacemos explícita la posibilidad de tratamiento del vector nulo si $sup = inf + 1$.

Para decrementar la función limitadora, haremos las siguientes llamadas recursivas:

```
quicksort(t,inf,k−1);
quicksort(t,k+1,sup);
```

2. Protecciones del condicional

Para poder realizar las llamadas recursivas, necesitamos $inf \leq sup$. Así pues, será ésta la condición que exigiremos. Para el caso no recursivo, tendremos $inf > sup$. Bajo esta condición ya tendremos garantizada la postcondición Q por nulidad de dominio.

Con todas las consideraciones hechas, la acción *quicksort* queda ahora del siguiente modo:

```
acción quicksort (t: e/s vector, inf:ent entero, sup: ent entero)
  si inf > sup ⟶ no_op;
     inf ≤ sup ⟶
       ???
       {R}
       quicksort(t,inf,k-1);
       quicksort(t,k+1,sup);
  fsi
facción
```

3. Cálculo de R

Vamos ahora a ocuparnos de la propiedad R que deberá cumplir t para que las dos ordenaciones parciales realizadas recursivamente sean suficientes para ordenar el vector entre las posiciones inf y sup.

La postcondición que el algoritmo *quicksort* tiene que establecer la podemos descomponer del siguiente modo:

$$\{Q\} = \{\forall i : inf \leq i \leq k-1, \forall j : inf \leq j \leq i \bullet t[j] \leq t[i] \wedge$$
$$\forall i : k+1 \leq i \leq sup, \forall j : k+1 \leq j \leq i \bullet t[j] \leq t[i] \wedge$$
$$\forall i : k \leq i \leq sup, \forall j : inf \leq j \leq k \bullet t[j] \leq t[i]\}$$

- La primera de ellas establece la ordenación del vector t entre los índices inf y $k-1$, estará garantizada por el resultado de la primera de las llamadas recursivas.
- La segunda establece la ordenación del vector entre los índices $k+1$ y sup y estará garantizada por el resultado de la segunda de las llamadas recursivas.
- Finalmente, la tercera, establece que los valores de los índices de la izquierda de k son inferiores o iguales a los valores de los índices de la derecha. Esta condición no está garantizada por nada de lo que hemos desarrollado hasta el momento. Para ver qué necesitamos para garantizarla, escribámosla de un modo que resulte más intuitivo:

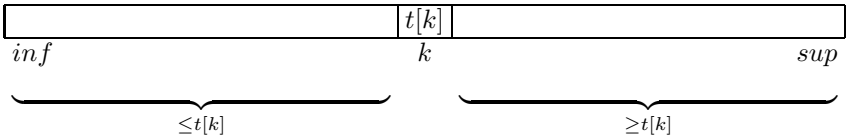


Figura 6.3: Condición previa R que garantiza la ordenación total del vector tras las llamadas recursivas.

$\{R\} = \{inf \leq k \leq sup$
$\forall j : inf \leq j \leq k \bullet t[j] \leq t[k] \wedge$
$\forall i : k < i \leq sup \bullet t[k] \leq t[i]\}$

En la figura 6.3 aparece el significado de esta condición gráficamente.

La importancia de esta condición es que si encontramos una manera razonable de satisfacerla habremos dividido el problema de ordenar un vector en dos subproblemas de ordenar un subvector. Ocupémonos ahora de encontrar aquellas acciones que nos permitirán satisfacer R

Con las consideraciones hechas hasta el momento, hemos diseñado el siguiente algoritmo:

acción quicksort (t: **e/s** vector, inf:**ent** natural, sup: **ent** natural)
$\{P\} = \{1 \leq inf \leq sup + 1\}$
si inf>sup $\longrightarrow$ no_op;
inf≤sup $\longrightarrow$
???
$\{R\} = \{inf \leq k \leq sup \wedge$
$\forall j : inf \leq j \leq k \bullet t[j] \leq t[k] \wedge$
$\forall i : k < i \leq sup \bullet t[k] \leq t[i]\}$
quicksort(t,inf,k-1);
quicksort(t,k+1,sup);
fsi
facción

4. Acciones necesarias para obtener R

El enunciado R que se debe satisfacer para que, después de las dos llamadas recursivas, obtengamos Q nos impone dividir el vector t en dos partes($inf..k$ y $k+1..sup$). Este proceso de partición o división del vector se sustenta en un valor x que se establece como elemento discriminador de las dos partes en que habremos dividido el vector t y que, al final de la partición, deberá aparecer en la posición k (x=t[k]). Pero, ¿ Cuál puede ser ese valor discriminador x? Idealmente sería aquél que nos permitiera dividir el vector t en dos partes (aproximadamente) iguales. Pero como no tenemos ninguna información sobre los valores que contiene el vector t, no podemos hacer ninguna hipótesis sobre dónde se encuentra este valor. Por tanto, podremos elegir cualquiera de los que aparecen en el vector entre las posiciones $inf..sup$, por ejemplo, haremos $x = t[inf]$.

Con lo que llevamos dicho hasta ahora, podemos pensar en una acción *partición* que nos permitiría descomponer el vector en dos partes (la primera desde inf hasta k con los elementos menores o iguales que x y la segunda desde $k+1$ hasta sup con los mayores o iguales que x). A continuación presentamos esta acción especificada formalmente:

$\{P'\} = \{1 \leq inf \leq sup + 1 \leq N + 1\}$
partición(t:**e/s** vector[1..N],inf:**ent** entero,sup **ent** entero,x:**ent** entero,k:**sal** entero)
$\{Q'\} = \{inf - 1 \leq k \leq sup \wedge$
$\forall j : inf \leq j \leq k \bullet t[j] \leq x \wedge$
$\forall i : k < i \leq sup \bullet x \leq t[i]\}$

Si, como hemos dicho, llamamos a la acción *partición* con el valor $x = t[inf]$ y utilizamos como límite

inferior del vector $inf + 1$ (ya sabemos que $t[inf] \leq x$):

$particion(t, inf + 1, sup, t[inf], k)$;

Obtendremos como postcondición:

$$\{Q'\} = \{inf \leq k \leq sup \wedge \\ \forall j : inf + 1 \leq j \leq k \bullet t[j] \leq t[inf] \wedge \\ \forall i : k < i \leq sup \bullet t[inf] \leq t[i]\}$$

Que podemos ampliar ya que $t[inf] \leq t[inf]$:

$$\{Q''\} = \{inf \leq k \leq sup \wedge \\ \forall j : inf \leq j \leq k \bullet t[j] \leq t[inf] \wedge \\ \forall i : k < i \leq sup \bullet t[inf] \leq t[i]\}$$

Finalmente, para obtener R, sólo necesitamos intercambiar los valores $t[inf]$ y $t[k]$. Notemos que $t[k]$ existe (k es un índice válido para el vector t) por el rango de los valores de k. Así pues:

$intercambio(t[inf], t[k])$.

En total, la acción *quicksort* quedaría del siguiente modo:

$\{P\} = \{1 \leq inf \leq sup + 1 \leq N + 1\}$

acción quicksort (t: **e/s** vector, inf:**ent** entero, sup: **ent** entero)
  si $inf > sup \longrightarrow$ no_op;
    inf≤sup $\longrightarrow$
      partición$(t, inf + 1, sup, t[inf], k)$;
      intercambio$(t[inf], t[k])$;
      quicksort$(t, inf, k - 1)$;
      quicksort$(t, k + 1, sup)$;
  fsi
facción

$\{Q\} = \{\forall i : inf \leq i \leq sup, \forall j : inf \leq j \leq i \bullet t[j] \leq t[i]\}$

La acció *partición* que necesitamos para concluir el diseño del algoritmo coincide exactamente con la que desarrollamos en el capítulo 1 con el mismo nombre (v. cap. 1).

6.3.2 Transformación a iterativo

El algoritmo *quicksort* quicksort utiliza recursividad múltiple, por lo cual genera un árbol de llamadas recursivas. De cara a transformar este algoritmo en otro iterativo equivalente será necesario identificar qué tipo de recorrido realiza el algoritmo sobre el árbol generado y, posteriormente, proponer la versión iterativa de ese recorrido convenientemente modificada para que implemente el *quicksort*. Hagámoslo:

En primer lugar debemos identificar el tipo de recorrido que realiza la ejecución de la acción sobre el árbol de llamadas recursivas. Se trata, de un recorrido en preorden, cuyo algoritmo iterativo, adaptado, es el siguiente:

```
acción quicksort_i (t:vector[1..N],inf:natural,sup:natural)
  var p:pila;
  si inf≤sup push(p, < 0, 0 >); fsi
  mientras ¬p_vacía(p) hacer
    partición(t,inf+1,sup,t[inf],k);
    intercambio(t[inf],t[k]);
    si inf≤ sup push(p,<k+1,sup>); fsi
    si inf≤ sup <inf,sup>:=<inf,k-1>;
    sinó <inf,sup>:=top(p); pop(p);
```

```
    fsi
  fmientras
facción
```

Aprovechando la propiedad de que el árbol tiene dos hijos o ninguno y que el proceso de empilado no modifica para nada los valores *inf* y *sup* podemos hacer una simplificación al algoritmo anterior:

```
acción quicksort_i (t:vector[1..N],inf:natural,sup:natural)
  var p:pila;
  si inf≤sup push(p, < 0, 0 >); fsi
  mientras ¬p_vacía(p) hacer
    partición(t,inf+1,sup,t[inf],k);
    intercambio(t[inf],t[k]);
    si inf≤ sup
      push(p,<k+1,sup>);
      <inf,sup>:=<inf,k-1>;
    sinó <inf,sup>:=top(p); pop(p);
    fsi
  fmientras
facción
```

6.3.3 Cálculo del coste

El propósito de este apartado es el de evaluar en orden de magnitud el tiempo de ejecución del algoritmo *quicksort* en su versión recursiva (el orden de magnitud de la versión iterativa del *quicksort* sería idéntico). Para ello, plantearemos en primer lugar una ecuación recurrente con la expresión del coste temporal del algoritmo:

Si llamamos $t(n)$ al coste en tiempo del algoritmo *quicksort* en su versión recursiva para un vector a ordenar de longitud $n \geq 0$, obtendremos el siguiente esquema de ecuación:

$t(n) \leq c_1 + c_2(n-1) + t(k-1) + t(n-k) \qquad 1 \leq k \leq n \qquad [1]$
$t(0) = c_3$

Donde n es el número de elementos a ordenar, k el número de elementos que quedan en la parte izquierda de la partición y c_1, c_2 y c_3 son constantes positivas que no van a influir en el cálculo del orden de magnitud del algoritmo.

- El caso peor ($T_p(n)$) se tendrá cuando hagamos una partición absolutmante desigual. En el límite, cuando consideremos un subvector vacío y otro con $n-1$ elementos. Esto sucede para $k = 1$ o para $k = n$. Ambos casos obtienen el mismo resultado. Hagamos $k = 1$:

  $T_p(n) \leq c_1 + c_2(n-1) + T_p(0) + T_p(n-1)$

  Como $T_p(0)$ es invariablemente igual a c_3, tenemos

  $T_p(n) \leq c_1 + c_2(n-1) + c_3 + T_p(n-1)$

  Como solución a esta recurrencia se obtiene que

  $T_p(n)$ es $O(n^2)$ en el caso peor. Por tanto, observamos que el quicksort es tan malo como los métodos tradicionales de ordenación en el caso peor.

- El caso mejor lo obtendremos cuando la partición divida en dos partes iguales el vector. Esto lo conseguiremos para $k = n/2$. En este caso llegamos a la recurrencia:

  $T_m(n) \leq c_1 + c_2(n-1) + T_m(n/2-1) + T_m(n/2)$

  Resolviendo esta recurrencia se obtiene que $T_m(n)$ es $O(nlog(n))$ en el caso mejor. La ventaja de este método es que en el caso medio, su coste conserva este orden de magnitud y, por tanto, mejora sensiblemente el coste de los métodos tradicionales.

- Para evaluar el caso medio supondremos que los valores están dispuestos en el vector de forma aleatoria y, en consecuencia, todas las particiones son igualmente probables. En ese caso podemos escribir la recurrencia:

  $T_{medio}(n) \leq 1/n \sum_{k=1}^{n} c_1 + c_2(n-1) + T_{medio}(k-1) + T_{medio}(n-k)$

  Si conjeturamos $T_{medio}(n) \leq cnlog(n)$ para alguna constante positiva c, comprobaremos que efectivamente se cumple la desigualdad para todo n mayor o igual que 2 y $c >$máx$(2c_1 + 4c_3, 4c_2)$.

6.4 Altres algorismes d'ordenació de vectors

6.4.1 Ordenació per fusió

Hem vist que l'algorisme *quicksort* utilitzava el principi *divideix i venç*. La idea era, en cada pas, dividir el vector en dues parts i tractar cadascuna d'elles separadament. D'aquesta manera, es generava un arbre que, en promig, tenia $log_2(n)$ nivells (essent n el nombre d'elements del vector que cal ordenar). Com entre totes les crides que es feien en un nivell qualsevol es tractaven els n elements, resultava un cost total promig de $O(n * log(n))$ que avantatjava d'una manera molt clara el $O(n^2)$ típic dels algorismes menys astuts d'ordenació.

La veritat és que aquest mateix esquema de *dividir i véncer* el podem utilitzar com a filó per a generar algun altre algorisme d'ordenació basat en el mateix principi.

Suposem, per exemple, que decidim dividir en dues parts iguals el nostre vector i ordenar cadsucuna d'elles separadament (recursivament). Un cop feta aquesta doble ordenació tindrem les dues meitats del vector ordenades independentment. Si ara fem una fusió d'aquestes dues meitats (utilitzant les idees de l'algorisme de fusió de dues seqüències presentat a **??**) obtindrem un únic vector ordenat.

Fent el mateix raonament que en el cas del *quicksort*, el cost d'aplicar aquest algorisme d'ordenació és $O(n * log_2(n))$, essent n el nombre d'elements del vector a ordenar. Hi ha, però, dues diferències:

- El cost del *quicksort* és $O(n * log(n)$ en promig i $O(n^2)$ en el cas pitjor, mentre que el cost de l'algorisme d'ordenació per *fusió* és sempre $O(n * log(n))$. Per què?
- Tot i tenir, aparentment, un cost global millor, l'algorisme d'ordenació per *fusió* no sol ser tan eficient com el *quicksort*.

6.4.2 Ordenació per selecció modificat (*heapsort*)

El *heapsort* és un altre dels que podríem anomenar *algorismes astuts* d'ordenació, amb un cost final $O(n * log(n))$. Per a descriure'l ens inspirarem en l'algorisme de selecció directa.

Recordem que en aquell algorisme, sempre disposàvem d'un vector dividit conceptualment en dues parts: Una primera, ordenada i una segona, per ordenar i amb tots els seus elements més grans que els de la primera part. Per tant, a cada pas, havíem de prendre el més petit dels elements de la segona part i incorporar-lo al final de la primera. Aquesta operació de prendre el mínim de la segona part tenia un cost de $O(n)$ en el cas pitjor[4]. Com l'operació de cercar el mínim calia repetir-la n vegades, teníem un cost total $O(n^2)$.

Imaginem ara que disposem d'una estructura de dades especial (anomenem-la h) que tingui el següent rendiment (suposem que abans de fer cada operació h conté ja n elements):

- Cost d'inserir un element en h: $O(log(n))$.

[4]Si tenim un vector de n elements i ens cal trobar el seu mínim, haurem de recórrer, un per un, tots els elements d'aquell vector i veure quin és, finalment, el més petit. Això ens donarà un cost $O(n)$.

- Cost de consultar l'element mínim de h: $O(1)$.
- Cost d'eliminar l'element mínim de h: $O(log(n))$.

Considerem ara la següent modificació de l'algorisme de selecció directa per a ordenar un vector v amb n elements:

acció heapsort(v:vector[1..n])
  Col.locar tots els n elements de v a h.
  per i:=1 **a** n **fer**
    $v[i]$:=mínim(h);
    eliminar_minim(h);
  fper
facció

Raonem ara quin seria el cost d'aquesta acció:

- Col.locació $v \rightarrow h$ (n elements): $O(n * log(n))$
- Repetició n vegades d'obtenció del mínim de h i eliminació posterior d'aquest element de h: $O(n * (1 + log(n))$.

Total: $O(n * log(n) + n + n * log(n)) = O(n * log(n))$.

Aquesta estructura de dades *misteriosa* que hem anomenat h és, en realitat, un *heap* i consisteix en un arbre binari complet tal que un node sempre és més petit que qualsevol dels seus fills. Aquesta estructura de dades acompanyada d'uns algorismes d'inserció i eliminació astuts permeten aconseguir els rendiments presentats a la taula anterior (a l'assignatura d'Edalg es veuen més detalls d'això).

6.4.3 Ordenació utilitzant arbres binaris de cerca

Els *heaps* no són l'única estructura de dades que ens permet generar un algorisme d'ordenació astut. Els *arbres binaris de cerca* en són una altra. Un *arbre binari de cerca* és un arbre tal que, per a qualsevol node, tots els valors del seu subarbre esquerre són més petits que el valor del node i tots els del seu subarbre dret són més grans que el valor del node.

Els arbres binaris de cerca tenen la propietat que en ser recorreguts utilitzant una determinada estratègia (que es coneix amb el nom de *inordre*[5]), s'obtenen els seus elements ordenats. A més a més, els algorismes per a inserir i per a treure un element d'un arbre binari de cerca carregat amb n elements tornen a tenir un cost (mitjà, en aquest cas) de $O(log(n))$. Això fa que el següent algorisme tingui novament un cost mitjà $O(n * log(n))$:

acció ordenació_abc(v:vector[1..n])
  a:arbre_binari_cerca;
  carregar els n elements de v a a;
  recórrer a en inordre i col.locar els elements obtinguts seqüencialment a v
facció

A les figures 6.4, 6.5, 6.6 i 6.7 es presenten gràficament tots aquests mètodes juntament amb el *quicksort*. A l'assignatura d'EDALG s'estudiaran amb més detall.

[5] A grans trets, podem dir que el recorregut en inordre d'un arbre binari és un recorregut en fondària de l'arbre que consisteix en recórrer primer el fill esquerre de l'arbre també en inordre, després visitar l'arrel i, finalment, recórrer en inordre el fill dret.

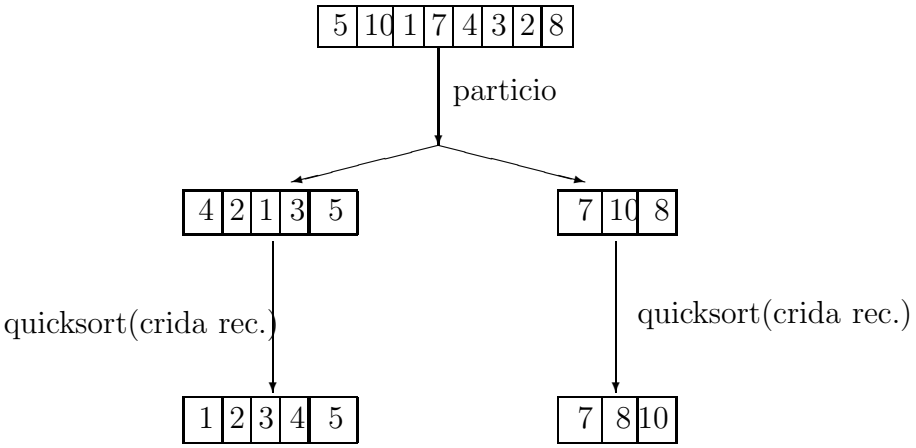


Figura 6.4: Mètode d'ordenació *quicksort*

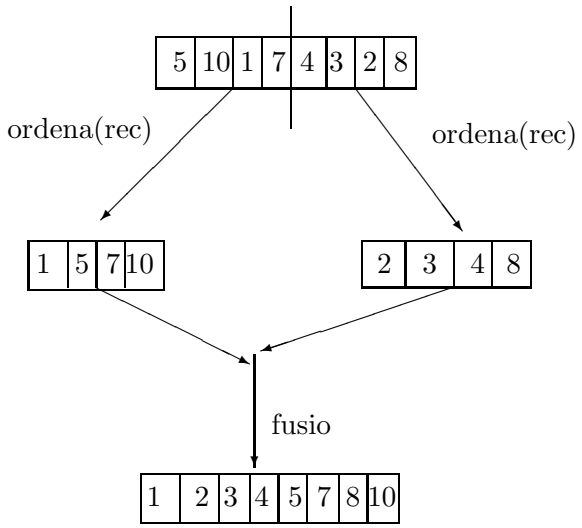


Figura 6.5: Mètode d'ordenació per fusió

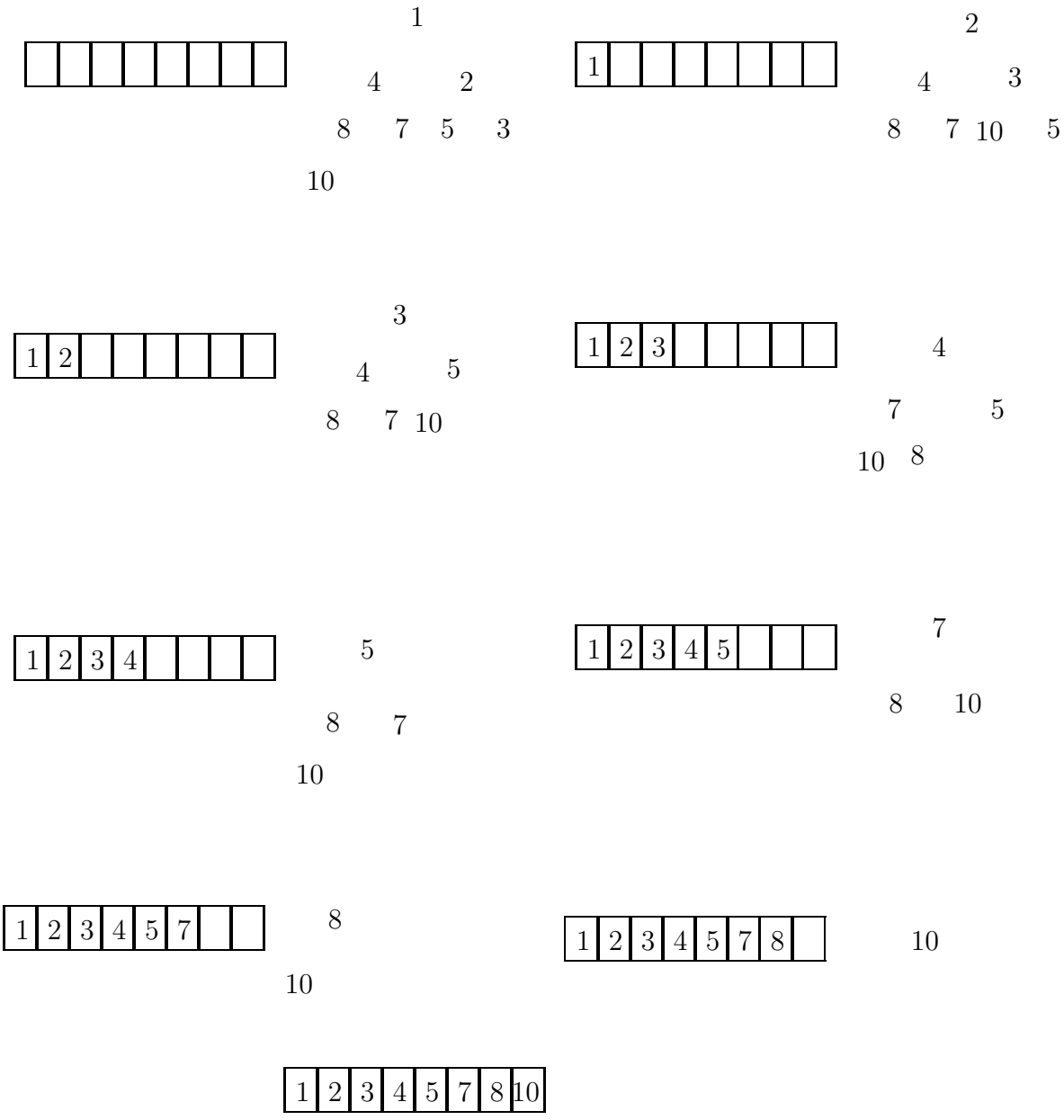


Figura 6.6: Mètode d'ordenació *heapsort*

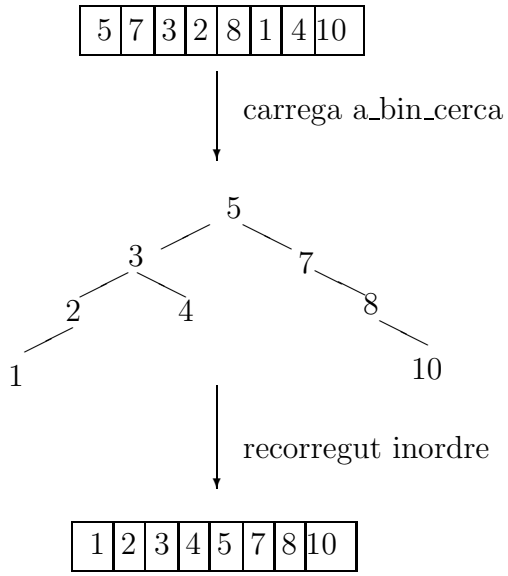


Figura 6.7: Mètode d'ordenació amb *arbres binaris de cerca*

6.5 Problemes

Problema 56

Recordem que a l'algorisme proposat d'ordenació d'un fitxer, l'acció *fusionar_sub* copiava l'estructura de *fusió4*. Què passaria si substituíssim l'acció *fusionar_sub* d'aquest algorisme per una altra que copiés l'estructura de *fusió3*? L'algorisme d'ordenació continuaria funcionant?

Problema 57

Escriu amb tot detall l'algorisme d'ordenació per fusió. No t'oblidis de donar l'invariant del bucle de l'acció de fusió.

Problema 58

L'algorisme *quicksort* no es comporta massa bé quan el volum d'elements a ordenar és petit. En aquest cas es veu superat fins i tot per un dels algorismes més dolents ($O(n^2)$). Modifica l'algorisme del *quicksort* per a que en el moment en què el vector que ha d'ordenar tingui menys de n_0 elements, cridi a un algorisme d'ordenació de cost $O(n^2)$.

Problema 59

Considerem la versió iterativa del *quicksort* (amb una pila manegada explícitament pel programador). Quin és el cas en què hi haurà un cost en espai més elevat? Com podríem fer per a alleugerir notablement aquest cost? Modifica l'algorisme *quicksort iteratiu* per a solucionar aquest problema.

Problema 60

Per cadascun dels algorismes d'ordenació presentats, elabora un vector per l'ordenació del qual, l'algorisme es comporti òptimament i un altre per l'ordenació del qual, l'algorisme es comporti de la pitjor manera.

Problema 61

Dissenya un programa que faci la fusió de n fitxers seqüencials ($n \geq 2$).

Problema 62 Examen Abril-96

L'algorisme d'ordenació *quicksort* necessita una acció per a *bipartir* el vector que cal ordenar en una part que tingui tots els elements més petits o iguals que un determinat valor x i una altra amb tots els seus elements més grans o iguals que x. El propòsit d'aquest problema és fer un disseny per l'acció *partició* però amb algunes diferències respecte el disseny proposat durant el curs.

En particular, aquesta és l'especificació de l'acció que volem construir:

$\{P\} = \{1 \leq inf \leq sup \leq N \wedge \exists i, inf \leq i \leq sup : x = t[i]\}$

partició(t:**e/s** vector[1..N] de enter,inf:**ent** natural, sup **ent** natural,x:**ent** natural,k:**sort** natural)

$\{Q\} = \{inf \leq k \leq sup \wedge$
$\forall j : inf \leq j \leq k : t[j] \leq x \wedge$
$\forall j : k+1 \leq j \leq sup : x \leq t[j]\}$

Notem que la precondició requereix que el *valor de tall* x, estigui present al vector entre els índexos inf i sup, la qual cosa exclou haver de considerar un vector buit.

Seguidament fem una proposta incompleta de l'acció que volem dissenyar:

```
1    acció partició(t,inf,sup,x,k)
2      var k1,k2:natural;
3      k1:=inf; k2:=sup;
4      mentre B fer
5        mentre t[k1]<x fer k1:=k1+1; fmentre
6        {Q1}
7        <codi1>
8        {Q2}
9        <codi2>
10     fmentre
11     k:=<exp>;
12   facció
```

- B representa la condició de continuació de la iteració principal (línia 4).
- <codi1> i <codi2> són dos trossos de codi que hauràs de descobrir.
- <exp> és l'expressió que cal assignar a k a la línia 11 per tal d'assolir la postcondició Q de l'acció.

Es demana:

1. (0,75)Invariant I i fita t de la iteració principal (línia 4).

2. (0,25) Condició de continuació B de la iteració principal.

3. (0,25) Expressió <exp> que cal assignar a k a la línia 11.

4. (0,75) Invariant I_1 i fita t_1 de la iteració de la línia 5 i postcondició Q_1 a la que s'arriba en acabar l'execució d'aquesta iteració (línia 6).

5. (0,5) Contingut de <codi1>

6. (0,25) Enunciat Q_2 que s'assoleix després de l'execució de [codi1].

7. (0,75) Contingut de <codi2> que faci que $\{Q_2\} < codi2 > \{I\}$ sigui correcte.

Suggeriment: Per a fer bé aquest problema cal tenir en compte la següent reflexió: A la línia 5, podem garantir que en algun moment t[k1]≥x. Per què? Qui ho ha de garantir? Pensa també en quin ha de ser el rang de les variables $k1$ i $k2$.

Problema 63 El quicksort republicà (Examen setembre-01)

En un moment en què l'hereu de tots els espanyols s'està construint una petita barraqueta de 700 milions de pessetes als terrenyets de papà, trobo necessari rescatar el vell problema de la bandera republicana[6].

Aquest problema consisteix a *tripartir* un vector que conté boles vermelles, grogues i violetes de manera que les vermelles se situen totes a la part esquerra del vector, les grogues a la part central i les violetes a la part dreta.

En particular, tot seguit us dono l'especificació i l'invariant de l'acció que resol aquest problema:

v:vector $[1..N]$ de natural. Suposarem que v està inicialitzat entre $1..n$ ($n \leq N$) amb valors naturals entre 1 i 3 (1="vermell", 2="groc", 3="violeta").

$\{P\} = \{v[1..n] = V[1..n] \wedge 0 \leq n \leq N\}$

banderaRepublicana(v,n)

$$\{Q\} = \{\forall i : 1 \leq i \leq k_1 \bullet v[i] = 1 \wedge \forall i : k_1 + 1 \leq i \leq k_2 \bullet v[i] = 2 \wedge \\ \forall i : k_2 + 1 \leq i \leq n \bullet v[i] = 3 \wedge 0 \leq k_1 \leq k_2 \leq n\}$$

Un invariant adequat per tal de desenvolupar el bucle de l'acció és el següent:

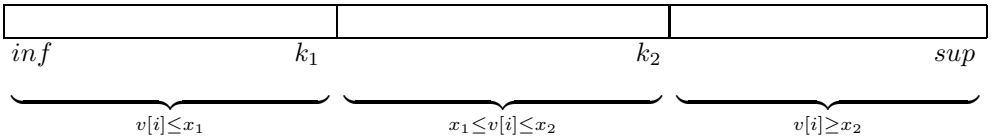
$$\{I\} = \{\forall i : 1 \leq i \leq k_1 \bullet v[i] = 1 \wedge \forall i : k_1 + 1 \leq i \leq j \bullet v[i] = 2 \wedge \\ \forall i : k_2 + 1 \leq i \leq n \bullet v[i] = 3 \wedge 0 \leq k_1 \leq j \leq k_2 \leq n\}$$

Utilitzarem la idea d'aquesta acció per tal d'oferir una variant del quicksort. La variant consisteix a agafar **dos** pivots del vector $v[inf..sup]$ que cal ordenar (x_1, x_2 de manera que $x_1 \leq x_2$) en lloc d'un i partir el vector en **tres** trams (i no en dos com feiem amb el quicksort clàssic):

1. Primer tram ($inf..k_1$): Format pels índexos i de v tals que $v[i] \leq x_1$.
2. Segon tram ($k_1 + 1..k_2$): Format pels índexos i de v tals que $x_1 \leq v[i] \leq x_2$.
3. Tercer tram ($k_2 + 1..sup$): Format pels índexos i de v tals que $v[i] \geq x_2$.

Gràficament ho podem expressar de la manera següent:

[6] Reformulació del clàssic problema de la bandera holandesa de Dijkstra. Està presentat als apunts de l'assignatura.



La manera de partir el vector en tres trams serà **exactament la mateixa** que l'emprada al problema de la bandera republicana per tal de separar els tres colors. I, en particular, seguirà pràcticament el mateix invariant.

Una vegada tenim el vector organitzat com a la figura, ja podem fer les crides recursives (que ara no seran dos....).

Nota: Una elecció simple dels pivots x_1 i de x_2 pot ser $v[inf]$ i $v[sup]$ (però compte amb l'ordre...).

Es demana: Implementar en pseudocodi l'acció *quickRepublic*(v, inf, sup) que ordeni els elements de v entre els índexos inf i sup ($inf \leq sup + 1$) i donar l'invariant del bucle.

Apèndix A

Correspondència pseudocodi-C++

A.1 Declaració de variables

Pseudocodi	C++
var $v_1 : T_1$; $v_2 : T_2$; **fvar**	`T1 v1;` `T2 v2;`

T_1, T_2 (`T1, T2`) són tipus (enter, caràcter...).

A.2 Declaració de constants

Pseudocodi	C++
const $c_1 = E_1$; $c_2 = E_2$; **fconst**	`const T1 c1=E1;` `const T2 c2=E2;`

`T1, T2` són tipus (enter, caràcter...) i E_1, E_2 (`E1, E2`) són expressions (e.g. $c_1 = 45$; c_2 ='a'; c_3 = "pepet";).

A.3 Estructura d'un algorisme/programa C++

Pseudocodi	C++

const
  <declaració constants>
fconst
var
  <declaració var. globals>
fvar
algorisme
  var
  <declaració var. locals>
  fvar
  <instruccions>
falgorisme
acció a1 (<paràmetres>) **és**
  <decl. var. locals i instrucc.>
facció
funció f1 (<paràmetres>) **retorna** T **és**
  <decl. var. locals i instrucc.>
  retorna x**;**
facció
	`#include<biblioteca>`
	
	`//Declaració constants:`
	`const T c=E;`
	`//...`
	
	`//Declaracio var.  globals:`
	`T v;`
	`//...`
	
	`//Capceleres d'accions i funcions:`
	`void a1(/*parametres*/);`
	`T f1(/*parametres*/);`
	
	`//Programa principal:`
	`void main(){`
	`  //instruccions`
	`}`
	
	`//Codi d'accions i funcions:`
	`void a1(/*parametres*/){/*instruccions*/}`
	`T f1(/*parametres*/){/*instruccions*/`
	`return x;}`

Anotacions:

- S'utilitzarà una directiva de precompilació `#include` per cada biblioteca que es vulgui utilitzar dins del programa. Les que utilitzarem més habitualment són les següents:

  ```
  #include <iostream.h>
  #include <vector>
  #include <string>
  ```

  Que ens permeten treballar respectivament amb operacions d'entrada/sortida, i els tipus `vector` i `string`.

- La col.locació de les capceleres de les accions i funcions immediatament després de les inclusions de biblioteques és imprescindible per les accions/funcions tals que es criden per primer cop abans de la definició del seu codi.

A.4 Tipus predefinits

A.4.1 Naturals

Pseudocodi	C++
i: natural;	`unsigned int i;` `unsigned long i;`
i: natural(35);	`unsigned int i(35);`
$k := i$ **div** j;	`k=i/j;`
$k := i$ **mod** j;	`k=i%j;`
$(i = j)$	`(i==j)`
$(i \neq j)$	`(i!=j)`
$(i \leq j)$	`(i<=j)`
$(i \geq j)$	`(i>=j)`

A.4.2 Enters

Pseudocodi	C++
i: enter;	`int i;` `long i;`
i: enter(35);	`int i(35);`
$k := i$ **div** j;	`k=i/j;`
$k := i$ **mod** j;	`k=i%j;`
$(i = j)$	`(i==j)`
$(i \neq j)$	`(i!=j)`
$(i \leq j)$	`(i<=j)`
$(i \geq j)$	`(i>=j)`

A.4.3 Booleans

Pseudocodi	C++
b:booleà;	`bool i;`
b: booleà(*cert*);	`bool b(true);`
cert, fals	`true, false`
$b_1 \wedge b_2$	`b1 && b2`
$b_1 \vee b_2$	`b1 || b2`
$\neg b$	`!b`
$(i = 10) \wedge b$	`(i==10) && b`

A.4.4 Caràcters

Pseudocodi	C++
c: caràcter;	`char c;`
c: caràcter('a');	`char c('a');`
codi(c)	c

Sovint utilitzarem tres caràcters especials:

- '\n' Canvi de línia i retorn de carro.
- '\t' Tabulador horitzontal.
- '\0' Caràcter de final de cadena de caràcters.

A.4.5 Vectors

Pseudocodi	C++
v :vector< T > (N) v :**vector**[1..N] **de** T;	`vector<T> v(N);`
v :vector<enter>(N,5)	`vector<int> v(N,5);`
$v[i]$	`v[i]`

T és un tipus (natural, enter...)
N és una constant(20, 1000....)

Anotacions:

- el rang de la declaració d'un vector en pseudocodi usualment va des de 1 fins a N.

  v: vector< T >(N) $\longrightarrow$ $v[1]..v[N]$.
  v: **vector** [1..N] **de** T $\longrightarrow$ $v[1]...v[N]$.

  En canvi, en C++, per definició, va des de `0..N-1`:

  `vector<T> v(N);` $\longrightarrow$ `v[0]...v[N-1]`.

- Compte!!! C++ **no avisa** si en temps d'execució fem un accés al vector fora del seu rang de definició.

- Per tal de representar cadenes de caràctes podem utilitzar el tipus `string` de C++ o bé podem usar vectors de caràcters acabats amb el caràcter especial '\0'.

- C++ també suporta la manera de definir vectors de C:

  `T v[N];` (exemple: `int v[40];`)

  Però recomanem `vector<T> v(N);` (exemple: vector<enter> v(40);).

A.4.6 Cadenes de caràcters

A.5 Estructures de control

Pseudocodi	C++
$i := j$;	`i=j;`
si B **llavors** I_1 **sino** I_2 **fsi**	`if (B) {I1}` `else {I2}`

si
$B_1 \longrightarrow I_1$
$B_2 \longrightarrow I_2$
...
$B_n \longrightarrow I_n$
fsi
	`if (B1) {I1}`
	`else if (B2) {I2}`
	`...`
	`else if (Bn) {In}`

mentre B **fer** S **fmentre**	`while (B) {S}`
per $i := 1$ **a** N **fer** S **fper**	`for (i=1;i<=N;i++) {S}`

I_k i S representen seqüències d'instruccions.
B_i representen expressions booleanes (guardes de condicionals i bucles).

Anotació:

- La correspondència entre l'estructura condicional de pseudocodi:

  si
  $B_1 \longrightarrow I_1$
  $B_2 \longrightarrow I_2$
  ...
  $B_n \longrightarrow I_n$
  fsi
  i la de C++:

```
if (B1) {I1}
else if (B2) {I2}
...
else if (Bn) {In}
```

  no és completa. En C++ es comença a avaluar des de dalt cap a baix (primer B_1, si B_1 avalua fals, B_2 i successivament fins que alguna B_k avalui cert o bé s'arribi a B_n). D'aquesta manera s'executa determinístament la seqüència d'instruccions associada a la primera guarda que ha avaluat cert. A més a més, si cap guarda avalua cert, no es produeix cap error.

  Recordem que en pseudocodi s'executa la seqüència d'instruccions associada a qualsevol guarda que avalua cert (de forma indeterminista). D'altra banda, en pseudocodi, exigim que almenys una guarda avalui cert.

A.6 Accions i funcions

Pseudocodi	C++

acció a (p_1: **ent** T_1,p_2: **sort** T_2,p_3: **e/s** T_3)
és
  S
facció
	`void a(T1 p1, T2& p2, T3& p3){`
	`  S;`
	`}`

funció f ($p_1 : T_1$) **retorna** T_2 **és**
  S
  retorna x;
ffunció
	`T2 f(T1 p1){`
	`  S;`
	`  return x;`
	`}`

T_1, T_2, T_3 (`T1, T2, T3`) són tipus.

p_1, p_2, p_3 (`p1, p2, p3`) sónm paràmetres dels tipus respectius.

x (`x`) és una variable de tipus $T - 2$ (`T2`).

Anotacions:

- Els paràmetres es passen en C++ de la següent manera:
  - (p: **ent** T) $\longrightarrow$ `(T p)` (pas de paràmetres per valor).
    Exemple: `(int p)`
  - (p: **sort** T) $\longrightarrow$ `(T& p)` (pas de paràmetres per referència).
    Exemple: `(int& p)`

  - (p: `e/s` T) $\longrightarrow$ `(T& p)` (pas de paràmetres per referència).
    Exemple: `(int& p)`

- Les funcions no han de produir efectes laterals i, per tant, no han de tenir paràmetres d'entrada/sortida o de sortida. C++ no prohibeix que les funcions tinguin paràmetres d'entrada/sortida o de sortida. Però no és convenient.

A.7 Entrada/Sortida

Pseudocodi	**C++**
escriure(x);	`cout<<x;`
escriure(c, x);	`c<<x;`
llegir(x);	`cin>>x;`
llegir(c, x);	`c>>x;`

Anotacions:

- `cout` és la constant de tipus `ostream` (canal de sortida) que es refereix a la sortida estàndar (monitor, usualment).
- `cin` és la constant de tipus `istream` (canal d'entrada) que es refereix a l'entrada estàndar (teclat, usualment).
- Es poden fer operacions d'entrada i de sortida sobre altres canals (dispositius o fitxers). Per aquest motiu hem inclòs les operacions llegir(c, x) o escriure(c, x) que indiquen que es fa una operació de lectura(escriptura) sobre el canal c. En C++, aquest canal hauria de ser una variable (constant) declarada de classe `istram` o `ostream`.

Apèndix B

Exemples de codificació en C++

B.1 Propòsit

Aquest annex presenta un seguit d'accions en C++ que us poden resultar útils com a exemples de codificació en C++.

B.2 Algun comentari addicional de C++

Tot seguit us comento algun aspecte del llenguatge que no hem vist a classe:

- La capcelera d'una acció és la seva declaració (sense el codi) acabada en ';'.

  Exemple:

  ```
  void intercanvi(int& a, int& b);
  void particio(vector<int>& v, int inf, int sup, int x, int& k);
  ```

  Aquestes dues són les capceleres de les accions *particio* i *intercanvi*.

  A les capceleres no cal posar el nom del paràmetre, però sí el seu tipus. Per tant, les anteriors capceleres també podrien ser escrites:

  ```
  void intercanvi(int&, int&);
  void particio(vector<int>& , int, int, int, int&);
  ```

  És una bona política de programació en C++ col.locar a l'inici del fitxer, abans del codi de les accions, funcions i programa principal les capceleres de les accions i funcions que es criden al llarg del fitxer. Un exemple d'això el trobareu a l'apartat B.4.

- C++ defineix un format de bucle anomenat *do-while* amb la forma següent:

  ```
  do{

  instruccio1;
  instruccio2;
  ...
  instruccion;
  }while(<condicio>);
  ```

Aquest bucle significa: Repetir les instruccions 1..n mentre es compleixi la `condicio`. Noteu que aqeust bucle s'executa almenys un cop (el bucle `while(c){...}` pot no executar-se cap cop si la condició de continuació és falsa a l'inici de la primera volta).

- C++ permet incloure comentaris al codi de dues maneres diferents:

```
//linia comentada

/* varies
  linies
  comentades

*/
```

B.3 Les accions

Les accions que presentem com a exemple en aquest document són les següents:

- **Adquisició**

```
void adquisicio(vector<int>& v, int max, int& inf, int& sup);
```

Llegeix del teclat una seqüència de com a màxim max d'enters i la col.loca entre els índexos inf i sup del vector v.

```
void adquisicio(vector<int>& v, int max, int& inf,
                int& sup)
{
  int i;

  do{
    cout<<"entra l'index incial del vector (0.."<<max-1<<")\n";
    cin>> inf;
  }while (inf<0 || inf>=max);

  do{
    cout<<"entra l'index final del vector ("<<inf-1<<".."<<max-1<<")\n";
    cin>> sup;
  }while (sup<inf-1 || sup>=max);

  for(i=inf;i<=sup;i++){

    cout<<"entra el valor associat a l'index "<<i<<" del vector\n";
    cin>>v[i];
 }

}
```

- **Intercanvi**

```
void intercanvi(int& a, int& b);
```

Intercanvia el valor de les variables enteres a i b.

```
void intercanvi(int& a, int& b)
{
  int aux;

  aux=a;
  a=b;
  b=aux;
}
```

- **Buidat**

```
void buidat(vector<int>& v,int inf,int sup);
```

Treu per pantalla els valors del vector v entre els índexos inf i sup.

```
void buidat(vector<int>& v,int inf,int sup)
{
  int i;

  cout<<"vector entre els indexos "<<inf<<" i "<<sup<<":\n";

  for(i=inf;i<=sup;i++){

    cout<<"("<<i<<":"<<v[i]<<")";
    if(i<sup) cout<<", ";
    else cout<<"\n";
  }
}
```

- **Partició**

```
void particio(vector<int>& v, int inf, int sup, int x, int& k);
```

Aquesta acció correspon a la que es presenta als apunts (cap. 3. La bipartició d'un vector)[1] :

acciópartició(t,inf,sup,x,k)
  var $k_2 : natural$;
  $k := inf - 1$;
  $k_2 := sup + 1$;
  mentre $k_2 \neq k + 1$ **fer**
    si $t[k+1] \leq x \longrightarrow k := k + 1$;
       $t[k_2 - 1] \geq x \longrightarrow k_2 := k_2 - 1$;
       $t[k+1] > x \wedge t[k_2 - 1] < x \longrightarrow$
          intercanvi$(t[k+1], t[k_2 - 1])$;
          $k := k + 1$;
          $k_2 := k_2 - 1$;
    fsi
  fmentre
facció

La traducció a C++ és la següent:

[1]Amb la diferència que en la traducció a C++ hem considerat el vector d'enters (no de naturals). Els índexos estan declarats `int` en lloc de `unsigned int` per simplicitat (algun d'ells en algun moment pot ser negatiu).

```
void particio(vector<int>& v, int inf, int sup, int x, int& k)
{
  int k2;

  k=inf-1;
  k2=sup+1;
  while(k2!=k+1){

   if(v[k+1]<=x){
       k=k+1;
   }
   else if(v[k2-1]>=x){
       k2=k2-1;
   }
   else{
       intercanvi(v[k+1],v[k2-1]);
       k=k+1;
       k2=k2-1;
  }
 }
}
```

- **Cerca dicotòmica**

```
void cerca_dicotomica(vector<int>& v, int inf,
                      int sup,   int x,   int& d);
```

Aquesta acció correspon a la que es presenta als apunts (cap. 3. La cerca dicotòmica)[2]:

acció cerca_dicotòmica(v, inf, sup, x, d)
  $e := inf; d := sup$;
  mentre $e \leq d$ **fer**
    $mig := (e + d)$ div 2;
    si $v[mig] \leq x \longrightarrow e := mig + 1;$
       $v[mig] > x \longrightarrow d := mig - 1;$
    fsi
  fmentre
facció

La traducció a C++ és la següent:

```
void cerca_dicotomica(vector<int>& v, int inf,
                      int sup,   int x,   int& d)
{
   int e, mig;

 e=inf; d=sup;
 while(e<=d){
   mig=(e+d)/2;
   if (v[mig]<= x) {e=mig+1;}
   else d=mig-1;
 }
}
```

[2]Amb les diferències que hem esmentat anteriorment

B.4 El fitxer sencer

En aquesta secció mostrem el fitxer *exemples.cpp* complet, amb les accions anteriors i un programa que les crida i utilitza. Aquest fitxer es pot compilar i executar fent:

```
$g++ exemples.cpp -o exemples
$exemples
```

Contingut del fitxer exemples.cpp

```
#include <iostream.h>
#include <vector.h>

const int N=20;

//*************CAPCELERES:

void cerca_dicotomica(vector<int>& , int , int ,int ,int&);
void particio(vector<int>& , int , int, int, int&);
void intercanvi(int&, int&);
void adquisicio(vector<int>&, int, int&, int&);
void buidat(vector<int>&,int,int);

//************CODIS DE LES ACCIONS


void particio(vector<int>& v, int inf, int sup, int x, int& k)
{
  int k2;

  k=inf-1;
  k2=sup+1;
  while(k2!=k+1){

   if(v[k+1]<=x){
       k=k+1;
   }
   else if(v[k2-1]>=x){
       k2=k2-1;
   }
   else{
       intercanvi(v[k+1],v[k2-1]);
       k=k+1;
       k2=k2-1;
  }
 }
}

void cerca_dicotomica(vector<int>& v, int inf,
                      int sup,   int x,   int& d)
{
```

```
   int e, mig;

 e=inf; d=sup;
 while(e<=d){
   mig=(e+d)/2;
   if (v[mig]<= x) {e=mig+1;}
   else d=mig-1;
 }
}

void intercanvi(int& a, int& b)
{
  int aux;

  aux=a;
  a=b;
  b=aux;
}

void adquisicio(vector<int>& v, int max, int& inf,
                int& sup)
{
  int i;

  do{
    cout<<"entra l'index incial del vector (0.."<<max-1<<")\n";
    cin>> inf;
  }while (inf<0 || inf>=max);

  do{
    cout<<"entra l'index final del vector ("<<inf-1<<".."<<max-1<<")\n";
    cin>> sup;
  }while (sup<inf-1 || sup>=max);

  for(i=inf;i<=sup;i++){

    cout<<"entra el valor associat a l'index "<<i<<" del vector\n";
    cin>>v[i];
 }

}

void buidat(vector<int>& v,int inf,int sup)
{
  int i;

  cout<<"vector entre els indexos "<<inf<<" i "<<sup<<":\n";

  for(i=inf;i<=sup;i++){

    cout<<"("<<i<<":"<<v[i]<<")";
    if(i<sup) cout<<", ";
    else cout<<"\n";
```

```
  }
}


//***********PROGRAMA PRINCIPAL:

void main()
{

 vector<int> v(N);
 int a,b,i,x;

 adquisicio(v,N,a,b);
 x=10;

 // cerca_dicotomica(v,a,b,x,i);


 particio(v,a,b,x,i);


 cout<<"L'index resultat es el "<<i<<"\n";

 cout<<"El vector modificat es el seguent:\n";

 buidat(v,a,b);

}
```