Programació 1

(Notes de curs)

Xavier Burguès, Jordi Cortadella, Juan Luis Esteban,
Nicola Galesi, Enric Martínez, Fernando Orejas i Albert Rubio

Setembre 2003

Índex

Prefaci

Aquest document no pretén ser un llibre de programació, tot i que no està lluny de ser-ho. Aquí es recullen els principals algorismes que s'imparteixen a les classes de teoria de l'assignatura *Programació 1* de la *Facultat d'Informàtica de Barcelona*.

L'objectiu principal del curs és ensenyar a l'alumne que la *Programació* és una disciplina molt relacionada amb la lògica matemàtica en la que cada parell *Especificació/Programa*, (E, P), pot ser considerat com un teorema en el que s'ha de demostrar que P és correcte respecte a E.

Tot i que la verificació formal no és l'objectiu d'un curs introductori a la programació, es pretén que l'estudiant sigui conscient que darrera de cada programa hi ha una formalització que permet demostrar la seva correctesa.

La inducció és l'eina fonamental que s'utilitza durant el curs per raonar. Tant en el disseny iteratiu com en el recursiu, la reducció d'un problema a un altre més senzill és la base per derivar programes correctes. En ambdós casos, es proposa una mecànica que permet sistematitzar el disseny dels algorismes. D'aquesta manera, es pretén que l'estudiant redueixi el risc de proposar dissenys incorrectes i tingui un mecanisme eficaç per a respondre preguntes que sovint apareixen:

- Com haig d'inicialitzar les meves variables abans d'entrar en un bucle?
- Com sé si l'índex no sortirà fora dels límits de la taula?
- Com puc garantir que una funció no generarà un nombre infinit de crides recursives?

De vegades, la manera de respondre a aquestes preguntes és mitjançant el mecanisme de *prova i error*. Massa sovint, la manera de resoldre els problemes detectats durant l'execució consisteix en posar "pegots" al codi que solucionen localment els casos que l'algorisme no havia tractat correctament. Moltes vegades això dona lloc a algorismes poc elegants, ineficients i difícils d'entendre, tot i que aparentment funcionin correctament.

L'aplicació sistemàtica de la inducció en la programació és la base d'aquest curs. La inducció fa que la correctesa dels algorismes sigui fruit d'un raonament i no pas d'un seguit de proves triades més o menys a l'atzar. Al final del curs es pretén que l'alumne sàpiga enfrontar-se a un problema de programació utilitzant les eines de raonament que s'han anat exercitant durant el curs, reduint el risc d'error i derivant programes eficients i elegants a la vegada.

Capítol 1

Nocions bàsiques de lògica

Aquest capítol presenta unes nocions bàsiques de lògica matemàtica necessàries per seguir el curs de programació. S'introdueix la lògica proposicional i alguns dels quantificadors que es faran servir més sovint.

1.1 Lògica proposicional

Una *proposició* és una afirmació de la que té sentit dir si és certa o falsa. Un *predicat* sobre n *variables lliures* és una relació entre n variables que es converteix en una proposició quan donem valors a les n variables. El *domini* d'una variable és el conjunt de valors que pot pendre.

Exemples:

- "x és parell", és un predicat sobre la variable x;
- "$x \times y = z$", és un predicat sobre les variables x, y i z;
- "3 és parell", és un predicat sobre zero variables, és a dir, una proposició, en aquest cas falsa;
- "$2 \times 4 = 8$" és una proposició verdadera.

Una variable *booleana* pot pendre només els valors *cert* i *fals*. Per abreviar, en comptes de *cert* escriurem 1 i en comptes de *fals* escriurem 0. Designarem les variables booleanes amb els noms $a, b, c, \ldots$. Les operacions sobre valors booleans es denominen *connectives*. Com només hi ha dos valors booleans, les connectives és poden definir llistant els resultats per totes les combinacions possibles de valors. Les connectives més importants són:

Taula 1.1: Defició de les connectives binàries

a	b	$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$a \Leftrightarrow b$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Taula 1.2: Comprobació d'equivalències

a	b	$\neg a$	$\neg a \vee b$	$a \Rightarrow b$	$b \Rightarrow a$	$(a \Rightarrow b) \wedge (b \Rightarrow a)$	$a \Leftrightarrow b$
0	0	1	1	1	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	0	0
1	1	0	1	1	1	1	1

- $\neg$: negació;
- $\wedge$: conjunció;
- $\vee$: disjunció;
- $\Rightarrow$: implicació;
- $\Leftrightarrow$: equivalència.

La connectiva $\neg$ és unària, és a dir, té només un operand, la resta són binàries. La negació de 0, és a dir $\neg 0$, es 1 i $\neg 1$ és 0. La definició de les conectives binàries es troba a la Taula 1.1. Aquesta mena de taules es denominen *taules veritatives*.

Es pot observar que $a \Rightarrow b$ és una abreviació de $\neg a \vee b$ i $a \Leftrightarrow b$ de $(a \Rightarrow b) \wedge (b \Rightarrow a)$. Es pot comprovar a la Taula 1.1 on hem calculat els valors pas a pas.

A l'expressió $a \Rightarrow b$, l'a es diu *antecedent* i la b *consecuent*. La implicació significa que sempre que es cumpleixi l'antecedent s'ha de complir el consequent, per tant la implicació és falsa només quan l'antecedent és cert, i el consequent és fals. Es pot observar que la connectiva $\Leftrightarrow$ es certa només quan els dos operands tenen el mateix valor.

Una *tautologia* és una expressió booleana que és sempre certa independentment dels valors que prenguin les seves variables, la seva taula veritativa serà per tant una columna d'uns. Dues expressions booleanes A i B són equivalents si les seves taules veritatives son iguals. Això ho denotarem amb el símbol $\equiv$, és a dir, $A \equiv B$. Això vol dir que sempre que A sigui certa, B també ho serà i sempre que A sigui falsa, B també ho serà. Així tenim que si $A \equiv B$, $(A) \Leftrightarrow (B)$ és una tautologia.

Algunes de les equivalències més importants són:

Commutativitat	$a \wedge b$	$\equiv$	$b \wedge a$
Commutativitat	$a \vee b$	$\equiv$	$b \vee a$
Associativitat	$a \wedge (b \wedge c)$	$\equiv$	$(a \wedge b) \wedge c$
Associativitat	$a \vee (b \vee c)$	$\equiv$	$(a \vee b) \vee c$
Idempotència	$a \wedge a$	$\equiv$	a
Idempotència	$a \vee a$	$\equiv$	a
Distributivitat	$a \wedge (b \vee c)$	$\equiv$	$(a \wedge b) \vee (a \wedge c)$
Distributivitat	$a \vee (b \wedge c)$	$\equiv$	$(a \vee b) \wedge (a \vee c)$
Element neutre	$a \wedge 1$	$\equiv$	a
	$a \wedge 0$	$\equiv$	0
	$a \vee 1$	$\equiv$	1
Element neutre	$a \vee 0$	$\equiv$	a

Absorció	$a \vee (b \wedge a)$	$\equiv$	a
Absorció	$a \wedge (b \vee a)$	$\equiv$	a
	$\neg(\neg a)$	$\equiv$	a
	$\neg 0$	$\equiv$	1
	$\neg 1$	$\equiv$	0
Lleis de De Morgan	$\neg(a \wedge b)$	$\equiv$	$(\neg a) \vee (\neg b)$
Lleis de De Morgan	$\neg(a \vee b)$	$\equiv$	$(\neg a) \wedge (\neg b)$
	$a \wedge \neg a$	$\equiv$	0
	$a \vee \neg a$	$\equiv$	1
	$a \Rightarrow b$	$\equiv$	$(\neg a) \vee b$
	$a \Rightarrow b$	$\equiv$	$(\neg b) \Rightarrow (\neg a)$
	$(a \wedge b) \Rightarrow a$	$\equiv$	1
	$a \Rightarrow (a \vee b)$	$\equiv$	1

Per construir predicats podem fer servir variables i operacions no booleanes i també comparacions. Per exemple $x + 7$ no és un predicat, i si substituim la x per un 3, tenim que $3 + 7$ no és una proposició perque no s'avalua a cert o fals, s'avalua a 10 que no és cap valor booleà. En canvi,

$x + 7 = 11$ sí és un predicat i quan donem un valor a la x es converteix en una proposició i podem avaluar si es certa o falsa.

Les operacions que podem fer servir per construir expressions booleanes són les operacions numèriques usuals com: $+,-,\times,/$. També es pot fer servir div (divisió entera) i mod (resta de la divisió entera). Les comparacions són també les usuals: $=$, $\neq$, $>$, $<$, $\geq$ i $\leq$. Els operands d'aquestes comparacions poden ser de qualsevol tipus, però ambdós del mateix. El resultat d'avaluar la comparació serà un valor booleà.

1.2 Quantificadors

Anem a esciure un predicat que expressi que un nombre n és primer, és a dir, el predicat s'avaluarà a cert si i només si el nombre és primer (n és primer si cap nombre enter més gran que 1 i més petit que el propi n el divideix). És fàcil escriure un predicat que expressi que un nombre n és divisible entre un nombre m, simplement cal que la resta de n div m sigui 0. El predicat és senzillament $n \bmod m = 0$. Podem anomenar a aquest predicat $Divideix(n, m)$ i dir $Divideix(n, m) \equiv n \bmod m = 0$. Així si hem de fer servir aquest predicat el podem escriure o usar el seu nom.

Per que n sigui primer ha de passar que $n \bmod 2 \neq 0$ i que $n \bmod 3 \neq 0$, i que $n \bmod 4 \neq 0$, etc. Tenim un nombre indeterminat de conjuncions. Per escriure un predicat d'aquest tipus farem servir un quantificador universal.

$$\forall i : D(i) : P(i)$$

La i és la variable *lligada* al quantificador. $D(i)$ és un predicat que ens indica el *domini* de la variable i, i $P(i)$ és el predicat que volem que compleixin els valors que pertanyen al domini. L'expressió $\forall i : D(i) : P(i)$ serà certa si tots el valors que estan al domini d'i (fan cert $D(i)$) també fan cert $P(i)$. Si hi ha un sol valor del domini que faci fals $P(i)$ llavors l'expressió es falsa. La forma usual de llegir l'expressió en llenguatge normal és: "per tot i que compleixi $D(i)$, $P(i)$ és cert", per aquest motiu el quatificador universal es coneix també com *per tot*.

Ara és fàcil escriure el predicat $Primer(n)$:

$$\forall i : 1 < i < n : n \bmod i \neq 0$$

En el cas que tinguem un disjunció amb un nombre indefinit d'operands farem servir el quantificador existencial:

$$\exists i : D(i) : P(i)$$

Aquesta expressió és certa si almenys un valor que cumpleix $D(i)$ també cumpleix $P(i)$. Per expressar que un nombre és composat, és a dir, no és primer:

$$\exists i : 1 < i < n : n \bmod i = 0.$$

Hi ha unes equivalències interessants entre els quantificadors (lleis de De Morgan generalitzades).

$$\neg\forall i : D(i) : P(i) \equiv \exists i : D(i) : \neg P(i) \qquad \neg\exists i : D(i) : P(i) \equiv \forall i : D(i) : \neg P(i)$$

Com que el fet de ser composat és el contrari de ser primer podriem haver escrit directament el predicat fent servir aquestes equivalències.

Si $\emptyset(i)$ el predicat que és sempre fals, independentment del valor d'i, és a dir, $\emptyset(i)$ defineix el domini buit, tenim:

$$\forall i : \emptyset(i) : P(i) \equiv 1 \qquad \exists i : \emptyset(i) : P(i) \equiv 0$$

1.2.1 Altres quantificadors

Hi ha altres quantificadors no booleans que es fan servir sovint com per exemple el *sumatori* i el *productori*. Si $E(i)$ és una expressió numèrica:

$$\sum_{i=1}^{n} E(i) = E(1) + \ldots + E(n) \qquad \prod_{i=1}^{n} E(i) = E(1) + \ldots + E(n)$$

Aquestes espressions també les podem escriure com:

$$\Sigma i : 1 \leq i \leq n : E(i) \qquad \Pi i : 1 \leq i \leq n : E(i)$$

En el cas en que el domini de la variable lligada sigui buit:

$$\Sigma i : \emptyset(i) : E(i) = 0 \qquad \Pi i : \emptyset(i) : E(i) = 1$$

Aquestes expressions no són predicats perque no s'avaluen a cert o fals, però es poden fer servir per construir predicats. Per exemple un predicat que expressi que la suma dels quadrats dels n primers nombres sigui m quedaria així:

$$Suma(n, m) \equiv \left(m = \sum_{i=1}^{n} i^2 \right)$$

Un altre quantificador no booleà és el *quantificador de comptatge*:

$$\#i : D(i) : P(i)$$

Compta quants elements del domini $D(i)$ fan cert el predicat $P(i)$. Per exemple, quants divisors diferents te un nombre n:

$$\#i : 1 \leq i \leq n : n \bmod i = 0$$

.

1.3 Problemes

1.1 Si els valors de les variables a, b i c són, respectivament, *fals*, *fals* i *cert*, determinar el valor de les expressions lògiques següents:

$$\begin{array}{ll} c \wedge \neg a \vee b & \neg(a \vee b) \wedge c \\ a \vee b \vee c & \neg a \wedge \neg b \wedge c \end{array}$$

1.2 Simplificar els predicats següents:

$$\begin{array}{llll} a) & P \wedge (Q \Rightarrow P) & b) & (P \vee Q) \wedge (\neg Q \vee R) \\ c) & \neg(P \vee Q) \wedge Q & d) & (Q \wedge R \wedge P) \vee \neg(R \vee S) \\ e) & (P \Rightarrow Q) \wedge (Q \Rightarrow P) & f) & (((P \Rightarrow Q) \Rightarrow R) \vee P) \\ g) & (P \vee Q) \wedge ((P \Rightarrow R) \vee (P \Rightarrow Q)) & & \end{array}$$

1.3 Simplificar els predicats següents:

$$\begin{array}{llll} a) & \neg(\neg((a = 7) \vee (4 * l = l * 4))) & b) & (a = b) \vee \neg(5 < 7) \\ c) & \neg((a > 5) \wedge (a < 2)) & d) & b = \neg(b \neq b) \\ e) & \neg((3 < a) \wedge \neg((4 = 6) \vee \mathit{fals})) & f) & b = (q \wedge (\neg r \vee (7 > 4))) \end{array}$$

1.4 Simplificar el següents predicats:

$$\begin{array}{ll} a) & \neg(x \text{ és parell} \wedge (x > 0)) \wedge \neg(x \text{ és senar} \wedge (x > 0)) \wedge (x \geq 0) \\ b) & (\neg\text{fet} \wedge \neg\text{acabat} \wedge (x = 0)) \Rightarrow ((x = -2) \vee (x = 0) \vee (x = 2)) \end{array}$$

1.5 Demostrar que els dos predicats següents són equivalents:

$$((x = 0) \vee (x = 2)) \wedge (\neg\text{fet} \Rightarrow (x = 0)) \qquad \equiv \qquad (\text{fet} \wedge ((x = 0) \vee (x = 2))) \vee (\neg\text{fet} \wedge (x = 0))$$

1.6 Si $Q(x)$ i $R(x)$ són els predicats *"x és un nombre racional"* i *"x és un nombre real"*, respectivament, formalitzar les frases següents:

- Tot racional és real.

- Alguns reals són racionals.
- No tot real és racional.

1.7 Expressar formalment els enunciats següents:

- Existeix un enter parell que és multiple de 13.
- Entre dos reals qualssevol, sempre n'existeix un tercer que rau entre els dos primers.
- La mitjana de dos enters qualssevol sempre resta inclosa en l'interval tancat definit per aquests.
- Per a tot enter positiu existeix arrel quadrada.
- La darrera xifra d'un enter parell ha de ser parella.

Capítol 2

Tipus de dades

2.1 Tipus bàsics

Els algoritmes informàtics que volem aprendre a dissenyar són tals que, en ser executats, calculen uns resultats a partir d'uns valors inicials. Més concretament, l'enfoc de programació que adoptem en aquesta assignatura és imperatiu. Això vol dir que hi ha una memòria i l'algoritme està composat per instruccions que, a mida que s'executen seqüencialment, manipulen la memòria (que en començar conté els valors inicials) de manera que al final contingui els resultats. Aquesta memòria la tenim estructurada en variables, cadascuna de les quals té un nom i és capaç de contenir un valor. Cada variable té aparellat un tipus (de dades) i només podrà contenir valors d'aquell tipus.

Els tipus de dades bàsics que considerem són els *booleans*, els *caràcters*, els *naturals*, els *enters* i els *reals*. Cada tipus de dades té un conjunt de valors (els que podran ser continguts per les variables d'aquest tipus) i està dotat d'unes operacions que permeten fer càlculs amb aquests valors (per exemple, la suma de dos enters que ens dóna un altre enter). Addicionalment, hi haurà operacions que barrejaran dos tipus diferents (per exemple, la comparació de dos enters que ens donarà un booleà).

2.1.1 Booleans

El tipus booleà conté dos valors: el valor *cert* i el valor *fals*. Les operacions són la negació ($\neg$), la conjunció ($\wedge$) i la disjunció ($\vee$).

2.1.2 Naturals

Conté tots els nombres naturals que pot representar la màquina. Com que cada ordinador i/o llenguatge de programació pot tenir un límit diferent, suposarem que el nom maxNat fa referència al natural més gran que es pot representar. Escriurem els valors de la manera habitual i les operacions disponibles són la suma (+), la resta (-), el producte (∗), la divisió entera (div) - que ens dóna el quocient de la divisió. Per exemple, 5 div 3 dóna 1 - i el mòdul (mod) - que ens dóna el residu de la divisió. Per exemple, 5 mod 3 dóna 2 -.

2.1.3 Caràcters

Conté tots els caràcters (lletres, signes de puntuació, etc.). Quan volguem denotar un valor d'aquest tipus en un algoritme, escriurem el caràcter corresponent entre cometes. Exemples: `'a'`, `'.'`, `'2'`.

2.1.4 Enters

Conté tots els nombres enters que pot representar la màquina. Com que cada ordinador i/o llenguatge de programació pot tenir un límit diferent, suposarem que els noms `minEnt` i `maxEnt` fan referència, respectivament, a l'enter més petit i al més gran que es poden representar. Suposarem que `minEnt` és negatiu i que `maxEnt` és més gran o igual que `maxNat`. Escriurem els valors de la manera habitual i les operacions disponibles són la suma (+), la resta (-), el producte (∗), la divisió entera (div), el mòdul (mod) i el canvi de signe (-).

2.1.5 Reals

Conté tots els nombres reals que pot representar la màquina. Com que cada ordinador i/o llenguatge de programació pot tenir un límit diferent, suposarem que els noms `minReal` i `maxReal` fan referència, respectivament, al real més petit i al més gran que es poden representar. Suposarem que `minReal` és menor o igual que `minEnt` i que `maxReal` és més gran o igual que `maxNat`. Escriurem els valors de la manera habitual (per exemple, 3.1415927 o 5.0 -per distingir-lo de 5, que serà un enter o un natural segons convingui en l'entorn on apareix-) i les operacions disponibles són la suma (+), la resta (-), el producte (∗), la divisió (/) i el canvi de signe (-).

2.1.6 Altres operadors

A més dels operadors que hem presentat per a cada tipus, podrem fer servir els següents:

- **Operadors relacionals**, que donats dos valors del mateix tipus ens tornen un booleà: $=, \neq, <, >, \leq, \geq$. En els caràcters, l'ordre ve donat per la codificació. En els booleans, de vegades, es considera *fals* $<$ *cert*.

- **Operadors de canvi de tipus**, que donat un valor d'un tipus ens tornen el valor corresponent d'un altre tipus: `natAEnter`, `enterANat` (aplicable només a enters entre 0 i `maxNat`), `enterAReal`, `arrod` (d'un real ens dóna l'enter més proper, aplicable només si el resultat és un enter representable), `trunc` (d'un real ens en dóna la part entera, aplicable en les mateixes condicions). Aquests operadors ens poden fer falta per efectuar operacions entre valors de tipus diferents. Per exemple, si tenim la suma de notes d'un grup d'alumnes com un valor real i el nombre d'alumnes com un valor natural i volem calcular la nota mitjana com a real, caldrà convertir el natural en real i efectuar la divisió de dos reals.

  En molts lleguatges, les conversions entre certs tipus no cal fer-les explícites. Per exemple, en la comparació entre un natural i un enter, s'assumeix implícitament que el natural ja és un enter. El mateix passaria en una comparació entre un enter i un real, on l'enter s'assumiria implícitament com real.

Hi ha, encara, els operadors que treballen amb la codificació dels caràcters. A cada caràcter li correspon un nombre natural que és el seu codi. Cada ordinador i llenguatge de programació pot tenir una codificació de caràcters diferent, però suposarem que en qualsevol codificació totes les lletres majúscules es codifiquen seguides i ordenades. Farem la mateixa suposició per a les lletres minúscules i per als dígits (els caràcters `'0'`, `'1'`,...`'9'`). Els operadors associats són codi (donat un caràcter, ens dóna un natural que és el seu codi) i car (donat un natural ens dóna el caràcter que el té per codi).

2.2 Expressions

Són combinacions de variables, valors, operacions, crides a funció i parèntesis. Exemple: $2 + 5 - (3 * 6)$.

Algunes expressions es podrien interpretar de maneres diverses si no fixem unes normes. Per exemple, $2+5-3*6$ es pot entendre com $(2+5)-(3*6)$, com $(2+(5-3))*6$ i d'altres maneres. La manera que considerarem correcta és aquesta: els operadors s'apliquen segons la seva prioritat i, en cas d'empat, d'esquerra a dreta. Les prioritats són aquestes:

Màxima prioritat:	canvi de signe, negació
Nivell 2:	$*$, div , mod , $/, \wedge$
Nivell 3:	$+, -, \vee$
Mínima prioritat:	$<, >, \leq, \geq, =, \neq$

Segons això, l'expressió $2+5-3*6$ s'ha d'entendre com $(2+5)-(3*6)$.

Les expressions, per ser sintàcticament correctes, han d'estar construïdes de manera que per a cada operador es respectin el nombre i tipus dels operands. Així 2.5 div 2 és incorrecta perquè l'operació div només es pot fer servir amb dos naturals o dos enters, però no amb un real i un enter. Tot i ser sintàcticament correcta, una expressió pot ser incorrecta per dos altres motius: per valors prohibits per algun operador (per exemple, per dividir per zero) o per sobreeiximent en el càlcul (el resultat de l'expressió o algun resultat parcial no són representables per la màquina).

Una confusió que es produeix sovint entre els qui són nous en el món de la programació és la no distinció entre els predicats (que apareixen en precondicions, postcondicions, invariants, etc.) i les expressions (que apareixen en l'algoritme). Una diferència important està en el repertori d'operadors. En un predicat hi poden aparèixer altres operadors a més dels que acabem de presentar (qualsevol operador conegut de forma generalitzada, com la implicació, l´arrel quadrada, ...) i també quantificadors. En una expressió, no (excepció feta de les funcions definides per l'usuari, tema que es presentarà més endavant.).

2.3 Exemples

Suposem que tenim les variables següents: m (enter que val 4), n (enter que val -5), a (booleà que val *cert*), b (booleà que val *fals*), x (real que val 3.2) i y (real que val 4.5).

- L'expressió $m * n < m \bmod -n$ s'interpreta com $(m * n) < (m \bmod (-n))$ que és sintàcticament correcta i equivalent a $(4 * (-5)) < (4 \bmod (-5))$ que equival a $-20 < (4 \bmod 5)$ que és $-20 < 4$, és a dir, *cert*.

- L'expressió $x = y = \mathit{cert}$ s'interpreta com $(x = y) = \mathit{cert}$ que equival a $(3.2 = 4.5) = \mathit{cert}$ que equival a $\mathit{fals} = \mathit{cert}$ que és *fals*.
- L'expressió $m * n = b \wedge \neg a$ s'interpreta com $(m * n) = (b \wedge (\neg a))$ que és un error donat que no es pot comparar un enter amb un booleà.
- L'expressió $x/m \geq 5$ és incorrecta perquè l'operador / s'ha d'aplicar a dos reals i m no n'és. La podem corregir de diverses maneres: $\texttt{arrod}(x)$ div $m \geq 5$, $\texttt{arrod}(x/\texttt{entAReal}(m)) \geq 5$, $x/\texttt{entAReal}(m) >= 5.0$, etc.
- L'expressió $\texttt{car}(\texttt{codi}(\text{'A'}) + 2)$ dóna com a resultat el caràcter 'C.

2.4 Problemes

2.1 Determinar el resultat de les expressions numèriques següents:

$3 + 7$ div $3 * 2 - 15$ $\qquad$ $9 - (86/4) * 3 + 4$
32 mod $4 + 12 - 4 * 3$ $\qquad$ 42 div $8 - 3 * 14 + 6$

2.2 Si $m = 5$, $n = -9$, $a = \mathit{fals}$ i $b = \mathit{cert}$, determinar el valor de les expressions següents, o dir si provoquen alguna mena d'error.

a) $m \geq n$ $\qquad$ b) $m * m < n * n$
c) $(m < n) \neq (a \vee b)$ $\qquad$ c) $a \vee b < b$
e) $\neg(m \geq 1$ div $(10 + n))$ $\qquad$ f) $\neg m \geq 1$ div $(10 + n)$
g) $\neg(m \geq 1$ div $(9 + n))$ $\qquad$ h) $m = n = a = b$
i) $m = (n = a) = b$ $\qquad$ j) $m = n = (a = b)$

Capítol 3

Semàntica axiomàtica

En aquest capítol es presenta la semàntica axiomàtica de les instruccions que es faran servir en els algorismes d'aquest llibre. Els objectius que es pretenen assolir en aquest capítol són:

- Donar una formalització de les instruccions del llenguatge algorísmic.
- Demostrar que tot programa pot ser considerat com un objecte matemàtic sobre el que es pot fer raonament formal.

Després de la presentació dels axiomes i regles, es proposaran un exemples senzills amb els que es farà raonament formal sobre la seva correctesa. No és l'objectiu d'aquesta assignatura arribar a la demostració de la correctesa dels algorismes prensentats, però sí fomentar el rigor en el raonament dels mateixos. Aquest exemples han d'ajudar a fomentar-lo.

3.1 Axiomes i regles

A continuació es presenten els axiomes i regles que formalitzen el comportament de les instruccions. La composició iterativa serà tractada amb molt més detall en el capítol 5 i, per aquesta raó, no proposarem cap exemple sobre ella en aquest capítol.

Axioma de l'assignació.

$$\{Q[x \leftarrow E]\}\ x := E\ \{Q\}$$

Regla de la composició seqüencial.

$$\frac{\{P\}\ S_1\ \{R\}\ ,\ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}}$$

Regles de la composició condicional.

$$\frac{\{P\ \wedge\ B\}\ S\ \{Q\}\ ,\ P\ \wedge\ \neg B\ \Rightarrow\ Q}{\{P\}\ \textbf{si}\ B\ \textbf{llavors}\ S\ \textbf{fsi}\ \{Q\}}$$

$$\frac{\{P\ \wedge\ B\}\ S_1\ \{Q\}\ ,\ \{P\ \wedge\ \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{si}\ B\ \textbf{llavors}\ S_1\ \textbf{sino}\ S_2\ \textbf{fsi}\ \{Q\}}$$

$$\frac{\begin{array}{l}\forall i : 1 \leq i \leq n :\ \{P\ \wedge\ \neg B_1\ \wedge\ \cdots\ \wedge\ \neg B_{i-1}\ \wedge\ B_i\}\ S_i\ \{Q\},\\ \{P\ \wedge\ \neg B_1\ \wedge\ \cdots\ \wedge\ \neg B_n\}\ S_{n+1}\ \{Q\}\end{array}}{\begin{array}{l}\{P\}\ \textbf{si}\ B_1\ \textbf{llavors}\ S_1\ \textbf{sinosi}\ B_2\ \textbf{llavors}\ S_2\ \ldots\\ \ldots\ \textbf{sinosi}\ B_n\ \textbf{llavors}\ S_n\ \textbf{sino}\ S_{n+1}\ \textbf{fsi}\ \{Q\}\end{array}}$$

Regla de la composició iterativa.

$$\frac{\{I\ \wedge\ B\}\ S\ \{I\}}{\{I\}\ \textbf{mentre}\ B\ \textbf{fer}\ S\ \textbf{fmentre}\ \{I\ \wedge\ \neg B\}}$$

Regla de conseqüència.

$$\frac{P\ \Rightarrow\ P'\ ,\ \{P'\}\ S\ \{Q'\}\ ,\ Q'\ \Rightarrow\ Q}{\{P\}\ S\ \{Q\}}$$

3.2 Exemples

Si sabem que es compleix

$$\{c \text{ és una lletra}\}\ S\ \{x > 0\}$$

llavors també és compleix:

- $\{c$ és una vocal$\}\ S\ \{x > 0\}$
- $\{c$ és una lletra$\}\ S\ \{x \geq 0\}$
- $\{c$ és una vocal$\}\ S\ \{x \geq 0\}$

Exercici 1

Sabem que

$$\{x > 0\}\ S\ \{y > 0\}.$$

Indicar per quins dels següents predicats es pot garantir la seva certesa.

$\{x \geq 0\}\ S\ \{y > 0\}$	NO
$\{x = 3\}\ S\ \{y \geq 0\}$	SI
$\{x > 0\ \wedge\ x$ és parell$\}\ S\ \{$cert$\}$	SI
$\{x > 0\ \wedge\ x$ és parell$\}\ S\ \{y$ és parell$\}$	NO
$\{x > 0\}\ S\ \{y = 3\}$	NO
$\{x^3 > 0\}\ S\ \{y^2 > 0\}$	SI
$\{x^2 > 0\}\ S\ \{y^3 > 0\}$	NO
$\{x > 0\}\ S\ \{y = -3\ \vee\ y \geq 0\}$	SI
$\{$cert$\}\ S\ \{y \geq 0\}$	NO

Exercici 2

Donat el següent fragment de codi,

$$\begin{array}{|l|}\hline \{\texttt{Pre:}\quad ?\} \\ y := 2 * x; \\ x := y + 6; \\ \{\texttt{Post:}\ x = 0\} \\ \hline \end{array}$$

trobar la precondició més feble que el faci correcte.

Per a resoldre l'exercici es pot fer servir l'axioma de l'assignació a partir de la postcondició i anant enrera.

$$\begin{array}{|l|}\hline \{\texttt{Pre:}\quad 2x + 6 = 0\} \equiv \{x = -3\} \\ y := 2 * x; \\ \{y + 6 = 0\} \\ x := y + 6; \\ \{\texttt{Post:}\ x = 0\} \\ \hline \end{array}$$

Exercici 3

Donat el següent fragment de codi,

$\{\texttt{Pre:}\quad ?\}$
$y := 2 * x;$
$z := y + 6;$
$y := z + x;$
$z := y \ div\ 3;$
$x := z - x;$
$\{\texttt{Post:}\ x = 2\}$

trobar la precondició més feble que el faci correcte. Es fa servir l'axioma de l'assignació a partir de la postcondició i anant enrera.

$\{\texttt{Pre:}\quad ((2x + 6 + x)\ div\ 3) - x = 2\}$
$y := 2 * x;$
$\{((y + 6 + x)\ div\ 3) - x = 2\}$
$z := y + 6;$
$\{((z + x)\ div\ 3) - x = 2\}$
$y := z + x;$
$\{(y\ div\ 3) - x = 2\}$
$z := y\ div\ 3;$
$\{z - x = 2\}$
$x := z - x;$
$\{\texttt{Post:}\ x = 2\}$

La precondició obtinguda es pot simplificar de la següent manera:

$\{((2x + 6 + x)\ div\ 3) - x = 2\} \equiv$
$\{((3x + 6)\ div\ 3) - x = 2\} \equiv$
$\{x + 2 - x = 2\} \equiv$
$\{2 = 2\} \equiv$
$\{cert\}$

D'on es dedueix que la postcondició $x = 2$ es complirà per a qualsevol precondició.

3.3 Problemes

3.1 Escriure expressions booleanes per a les següents especificacions:

1. Donades dues parelles d'enters que representen el vèrtex superior esquerre i el vèrtex inferior dret d'un rectangle amb costats paral·lels als eixos de coordenades, determinar si un tercer punt (x, y) és dins del rectangle.

2. Determinar si un nombre positiu representa un any de traspàs. Un any és de traspàs si és múltiple de 4, excloent-ne els múltiples de 100 que no ho són de 400; 1964, 2004 i 2400 són de traspàs, però 1977 i 2100 no.
3. Determinar si dos intervals tancats $[a, b]$ i $[c, d]$ s'intersequen. I si els intervals són oberts?
4. Determinar si un natural n de com a molt quatre xifres és cap-i-cua.
5. Determinar si tres punts del pla de coordenades (a, b), (c, d) i (e, f) són damunt d'una mateixa recta o no. Suposar que totes les coordenades són enteres i fer servir només operacions enteres.

3.2 Donar especificacions informals i/o formals per als problemes següents:

1. Donat un nombre enter x no negatiu que representa un periode de temps en segons, trobar l'equivalent en dies, hores, minuts i segons.
2. Donats dos enters positius a i b, calcular el màxim comú divisor.
3. Donats dos enters positius a i b, calcular el mínim comú múltiple.
4. Donat un natural n, determinar si és primer.
5. Donat un natural n, calcular el nombre primer més petit que sigui més gran o igual que n.
6. Ordenar tres nombres enters a, b i c i deixar el seu valor a les variables p, s i t.
7. Donat el natural x, determinar quantes xifres té.

Capítol 4

Assignació i composicions seqüencial i condicional

En aquest capítol es presenten alguns algorismes que es poden resoldre únicament amb sentències d'assignacio, composició seqüencial i composició condicional. Donada la seva senzillesa, s'intentarà demostrar la correctesa d'alguns d'ells.

4.1 Intercanvi de valors

Dissenyar un fragment de codi que intercanvii el valor de dues variables a i b de tipus T.

Especificació

var a, b: T;
{`Pre:` $a = A,\ b = B$}
{`Post:` $a = B,\ b = A$}

Solució

Es fa servir una variable auxiliar per emmagatzemar un valor intermedi.

var aux: T;
$aux := a$;
$a := b$;
$b := aux$;

Demostració de correctesa

La correctesa es pot demostrar aplicant l'axioma de l'assignació a partir de la postcondició i anant enrera cap a la precondició.

$\{b = B,\ a = A\}$
$aux := a;\quad \uparrow$
$\{b = B,\ aux = A\}$
$a := b;\qquad \uparrow$
$\{a = B,\ aux = A\}$
$b := aux;\quad \uparrow$
$\{\texttt{Post:}\ a = B,\ b = A\}$

D'aquí es veu immediatament que la precondició implica el predicat inicial obtingut (són equivalents).

Encapsulament amb una acció

El fragment anterior es fa servir sovint en programació. Es pot encapsular amb una acció de la següent manera:

acció *intercanvi (**e/s** a: T, **e/s** b: T)*
$\{\texttt{Pre:}\ \ a = A,\ b = B\}$
$\{\texttt{Post:}\ a = B,\ b = A\}$
    var *aux: T;*
    aux $:= a;\ a := b;\ b := aux;$
facció

4.2 Descomposició horària

Dissenyar un fragment de codi tal que donat un natural n que representa un nombre de segons, produeixi tres naturals h, m i s que representin el desglòs de n en hores, minuts i segons.

Especificació

var n, h, m, s*: nat;*
$\{\texttt{Pre:}\ \text{—}\}$
$\{\texttt{Post:}\ n = 3600h + 60m + s,\ 0 \leq m < 60,\ 0 \leq s < 60\}$

Solució

```
var t: nat;
s := n mod 60;
t := n div 60;
m := t mod 60;
h := t div 60;
```

Demostració de correctesa

La correctesa es pot demostrar aplicant l'axioma de l'assignació a partir de la postcondició. Comencem per les dues darreres instruccions:

$\{n = 3600(t \ div\ 60) + 60(t \ mod\ 60) + s,\ 0 \leq (t \ mod\ 60) < 60,\ 0 \leq s < 60\}$
$m := t \ mod\ 60;$ $\uparrow$
$\{n = 3600(t \ div\ 60) + 60m + s,\ 0 \leq m < 60,\ 0 \leq s < 60\}$
$h := t \ div\ 60;$ $\uparrow$
$\{\texttt{Post:}\ n = 3600h + 60m + s,\ 0 \leq m < 60,\ 0 \leq s < 60\}$

El predicat obtingut es pot simplificar. En primer lloc,

$$0 \leq (t \bmod 60) < 60 \ \equiv\ \text{cert}$$

donat que l'operació mòdul compleix aquesta propietat. Per una altra part tenim

$$3600(t \operatorname{div} 60) = 60 \cdot 60 \cdot (t \operatorname{div} 60) = 60(t - t \bmod 60)$$

i així ens queda

$$n = 60(t - t \bmod 60 + t \bmod 60) + s = 60t + s$$

amb el predicat

$$n = 60t + s,\ 0 \leq s < 60$$

Ara apliquem l'axioma de l'assignació per a la següent instrucció:

$\{n = 60(n \ div\ 60) + s,\ 0 \leq s < 60\}$
$t := n \ div\ 60;$ $\uparrow$
$\{n = 60t + s,\ 0 \leq s < 60\}$

i de nou podem simplificar fent la substitució:

$$60(n \text{ div } 60) = n - n \bmod 60$$

i ens queda

$$n = n - n \bmod 60 + s,\ 0 \leq s < 60$$

o també

$$s = n \bmod 60,\ 0 \leq s < 60$$

on veiem que el segon predicat és redundant donat que el primer ja l'implica. I així queda

$$s = n \bmod 60$$

Finalment apliquem l'axioma de l'assignació a la darrera instrucció:

$\{n\ mod\ 60 = n\ mod\ 60\}$
$s := n\ mod\ 60;$ $\uparrow$
$\{s = n\ mod\ 60 + s\}$

El predicat obtingut és equivalent, igual que la precondició del problema. Per tant, queda demostrada la correctesa de la descomposició horària.

4.3 Valor absolut

Dissenyar un fragment de codi que calculi el valor absolut d'un nombre enter.

Especificació

var x, abs: *enter;*
{`Pre:` —}
{`Post:` $abs = |x|$}

Solució

si $x \geq 0$ ***llavors*** *abs* := x; ***sino*** *abs* := -x; ***fsi***;

Demostració de correctesa

Cal aplicar la regla de la composició condicional. Això comporta demostrar dues coses:

a) $\{\texttt{Pre} \ \wedge \ x \geq 0\}$ $abs := x$ $\{\texttt{Post}\}$

b) $\{\texttt{Pre} \ \wedge \ x < 0\}$ $abs := -x$ $\{\texttt{Post}\}$

Demostrarem només la primera, donat que la segona és molt semblant. Cal demostrar

$$\boxed{\begin{array}{l}\{\texttt{Pre} \ \wedge \ x \geq 0\} \\ abs := x; \\ \{abs = |x|\}\end{array}}$$

Aplicant l'axioma de l'assignació a partir de la postcondició s'obté

$$\boxed{\begin{array}{l}\{x = |x|\} \\ abs := x; \\ \{abs = |x|\}\end{array}}$$

i d'aquí es demostra la correctesa aplicant la regla de la conseqüència donat que la precondició de l'assignació implica el predicat obtingut[1]:

$$x \geq 0 \ \Rightarrow \ x = |x|$$

Encapsulament en una funció

El càlcul del valor absolut s'utilitza sovint en programació. Es pot encapsular en una funció tal com es mostra a continuació:

```
función abs (x: enter) retorna a: enter
{Pre: —}
{Post: a = |x|}
    si x ≥ 0 llavors a := x; sino a := -x; fsi;
    retorna a;
ffunció
```

4.4 Màxim de dos nombres

Dissenyar un fragment de codi que calculi el màxim de dos nombre enters.

[1] Cal notar que ($\texttt{Pre} \ \wedge \ x \geq 0$) és equivalent a ($x \geq 0$).

Especificació

$$\boxed{\begin{array}{l}\textbf{var } a, b, m\text{: } enter; \\ \{\texttt{Pre: } —\} \\ \{\texttt{Post: } m = \max(a, b)\}\end{array}}$$

Solució

$$\boxed{\textbf{si } a > b \textbf{ llavors } m := a; \textbf{ sino } m := b; \textbf{ fsi};}$$

Demostració de correctesa

Cal aplicar la regla de la composició condicional. Això comporta demostrar dues coses:

a) $\{\texttt{Pre} \ \wedge \ a > b\}\ m := a\ \{\texttt{Post}\}$

b) $\{\texttt{Pre} \ \wedge \ a \leq b\}\ m := b\ \{\texttt{Post}\}$

Demostrarem només la primera, donat que la segona és molt semblant. Cal demostrar

$$\boxed{\begin{array}{l}\{\texttt{Pre} \ \wedge \ a > b\} \\ m := a; \\ \{m = \max(a, b)\}\end{array}}$$

Aplicant l'axioma de l'assignació a partir de la postcondició s'obté

$$\boxed{\begin{array}{l}\{a = \max(a, b)\} \\ m := a; \quad \uparrow \\ \{m = \max(a, b)\}\end{array}}$$

i d'aquí es demostra la correctesa aplicant la regla de la conseqüència donat que la precondició de l'assignació implica el predicat obtingut:

$$a > b \ \Rightarrow \ a = \max(a, b)$$

Encapsulament en una funció

El càlcul del màxim de dos nombres es pot encapsular en una funció tal com es mostra a continuació:

```
función màxim (a: enter, b: enter) retorna m: enter
{Pre: —}
{Post: m = max(a, b)}
    si a > b llavors m := a; sino m := b; fsi;
    retorna m;
ffunció
```

4.5 Ordenació de tres nombres

Dissenyar un fragment de codi que ordeni tres nombres enters a, b i c, i els assigni a les variables p, s i t (primer, segon i tercer, en ordre creixent).

Especificació

```
var a, b, c, p, s, t: enter;
{Pre:  cert}
{Post: (p, s, t) és una permutació de (a, b, c), p ≤ s ≤ t}
```

Solució

Fem servir l'acció *intercanvi* presentada a la pàgina 16.

```
p := a; s := b; t := c;
si p > s llavors intercanvi(p,s); fsi;
si s > t llavors
    intercanvi(s,t);
    si p > s llavors intercanvi(p,s); fsi;
fsi;
```

Es deixa al lector la demostració de la correctesa d'aquest algorisme.

Solució sense sentències condicionals

La següent solució només utilitza assignacions, operacions de suma i resta i crides a la funció *màxim* presentada a la pàgina 19. No utilitza cap sentència condicional, a excepció de la que conté la funció *màxim*.

```
p := −màxim(−a,màxim(−b,−c)); {Una manera de calcular el mínim}
t := màxim(a,màxim(b,c));
s := a + b + c − p − t; {Una manera de calcular l'element restant}
```

4.6 Problemes

4.1 Dissenyar un algorisme que, donat un nombre enter x no negatiu que representa un periode de temps en segons, calculi l'equivalent en dies, hores, minuts i segons.

4.2 Dissenyar un algorisme que calculi el valor absolut de la diferència de dos nombres enters donats.

4.3 Dissenyar un algorisme que, donada una expressió horària en el format (hores, minuts i segons), hi afegeixi un segon i retorni el resultat en el mateix format.

4.4 Dissenyar un algorisme que, donada una quantitat en pessetes, la descompongui en un nombre mínim de monedes de cinc-centes, cent, vint-i-cinc, cinc i una peseta.

4.5 Dissenyar un algorisme que, donat un nombre natural més petit que 128, ens doni la seva representació en binari. Cal tenir en compte que la representació binària no tindrà més de 7 dígits.

Capítol 5

Disseny inductiu

5.1 Introducció

Aquest capítol presenta una metodologia per a dissenyar programes recursius i iteratius. En les dues seccions que segueixen, es mostren de forma independent el disseny recursiu i l'iteratiu, amb el raonament inductiu com a factor comú.

L'estil d'ambdues parts és lleugerament diferent, pero l'objectiu és el mateix: usar la inducció en el disseny. A més, en ambdós casos es presenten aquells aspectes que l'estudiant ha de raonar/justificar en la seva solució, i com aquest raonament l'ajuda en el disseny. Aquests apartats venen marcats per la semàntica (axiomàtica) del llenguatge.

5.2 Disseny recursiu

Per simplificar, podem dir que un algorisme recursiu és senzillament un condicional amb una crida a funció. L'única particularitat és que la funció cridada és la que estem dissenyant. L'estructura de una solució recursiva té sempre tres apartats:

1. *Casos senzills*. Detectar un o més casos senzills (no recursius) i resoldre'ls.
2. *Casos recursius*. Considerar els casos recursius i resoldre'ls (usant inducció).
3. *Acabament*. Raonar sobre l'acabament del programa recursiu, és a dir, indicar què es va fent més petit en cada crida recursiva. Si no és massa complicat, es pot usar (informalment) el concepte de funció de fita.

Aquests tres apartats estan tots relacionats entre si. En molts casos, pensarem primer el/s cas/os recursiu/s i després els senzills, encara que finalment en la solució posem primer els casos senzills. Per altra banda, normalment, quan decidim quina crida recursiva fem, ja tenim al cap que és el que decreix.

De tota manera, és important explicitar aquests tres apartats per que indiquen què s'ha de comprovar/raonar en un programa recursiu.

5.2.1 Exemple: potència

Veguem un classic primer exemple de problema senzill amb la seva solució raonada informalment. L'exemple a tractar és el de calcular la potència de dos naturals mitjançant productes. Considerem la següent especificació.

funció *pot (a, b : nat)* ***retorna*** *p : nat*
{`Pre:` $a > 0$}
{`Post:` $p = a^b$}

1. *Cas senzill.* Agafem $b = 0$. En aquest cas tenim que a^b és 1.

2. *Cas recursiu.* Considerem com a cas recursiu el cas complementari, és a dir, $b > 0$. Així tenim:

funció *pot (a, b : nat)* ***retorna*** *p : nat*
{`Pre:` $a > 0$}
{`Post:` $p = a^b$}
    si *$b = 0$* ***llavors*** *p := 1;*
    sino *??*
    fsi*;*
    retorna *p;*
ffunció

   Pensem ara una crida recursiva, amb paràmetres que satisfacin la precondició (en aquest cas particular no cal satisfer res), que siguin correctes (com a expressió del tipus requerit) i que ens facin decréixer algun paràmetre (per a garantir l'acabament). Un candidat evident és: $p := pot(a, b-1)$. En aquest cas, com $b > 0$ tenim que $b-1$ és un natural, i el decreixement el tenim en el segon paràmetre. Així doncs, per inducció, després de la crida recursiva tenim que

$$p = a^{b-1}$$

i per tant, per obtenir a^b, el que ens queda per fer és multiplicar p per a.

El raonament podria ser el següent:

> Fem la crida recursiva $p := pot(a, b-1)$, que té paràmetres correctes, ja que $b > 0$ (i satisfà la precondició).
> Després de la crida (per inducció) tenim $p = a^{b-1}$ i per tant, el que ens queda per fer és multiplicar p per a.

La solució final queda així.

```
función pot ( a,b : nat) retorna p : nat
{Pre:  a > 0}
{Post: p = a^b}
    si b = 0 llavors p := 1;
    sino
        p := pot (a,b − 1);
        {p = a^(b−1)}
        p := p ∗ a;
    fsi;
    retorna p;
ffunció
```

3. *Acabament.* Com ja hem dit, decreix el segon paràmetre b.

5.2.2 Exemple: suma de dígits

El problema és, donat un número natural, retornar la suma dels seus dígits. L'especificació és

```
función sumadígits (n : nat) retorna s: nat
{Pre: —}
{Post:  s = "suma dels dígits d'n representat en base 10"}
```

En aquest cas, per trobar la solució és millor pensar primer el cas recursiu. Per això resulta molt útil fer-se una representació gràfica del problema. Considerem que n té com a dígits $a_m a_{m-1} \ldots a_1 a_0$, així el que volem aconseguir és

$$a_m + a_{m-1} + \ldots + a_1 + a_0$$

Això ens fa pensar en una possible crida recursiva:

$$\underbrace{a_m + a_{m-1} + \ldots + a_1}\, + a_0$$

que ens permetria obtenir la suma dels dígits $a_m a_{m-1} \ldots a_1$. És fàcil veure que el número que té aquest dígits és n div 10. Per tant, fem la crida recursiva *sumadígits*(n div 10), doncs per inducció obtenim

$$a_m + a_{m-1} + \ldots + a_1$$

i només ens cal sumar a_0, és a dir n mod 10, per tenir la suma de tots els dígits.

Ara, hem de fer notar que n div 10 és més petit que n només si $n > 0$, i per tant hem de tractar el cas $n = 0$ com a cas senzill, que té 0 com a suma dels dígits.

La solució raonada segueix els tres apartats esmentats anteriorment.

1. *Cas senzill*. Agafem $n = 0$. En aquest cas, la suma dels dígits és 0.

2. *Cas recursiu*. Agafem el complementari, per tant $n > 0$, i fem la crida recursiva $s :=$ *sumadígits*(n div 10). Així tenim:

```
función sumadígits (n : nat) retorna s: nat
{Pre: —}
{Post:  s = "suma dels dígits d'n representat en base 10"}
    si n = 0 llavors s :=0;
    sino
        s:= sumadígits(n div 10);
        {s conté la suma dels dígits de n div 10}
        ??
    fsi;
    retorna s;
ffunció
```

Com que tenim

$$a_m + a_{m-1} + \ldots + a_1$$

només ens cal sumar a_0, és a dir n mod 10, per tenir la suma de tots els dígits.

```
funció sumadígits (n : nat) retorna s: nat
{Pre: —}
{Post:  s = "suma dels dígits d'n representat en base 10"}
    si n = 0 llavors s :=0;
    sino
        s:= sumadígits(n div 10);
        {s conté la suma dels dígits de n div 10}
        s := s + (n mod 10);
    fsi;
    retorna s;
ffunció
```

3. *Acabament.* Com que $n > 0$, tenim que n div 10 és menor que n.

5.3 Disseny iteratiu

5.3.1 Composició iterativa

La sintaxi de la iteració és la clàssica.

$$\texttt{mentre}\ B\ \texttt{fer}\ S\ \texttt{fmentre}$$

La semàntica axiomàtica de la iteració està basada en el principi d'inducció. La forma més simple d'expressar la semàntica de la iteració es la següent. Si I és una propietat invariant, és a dir

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

i alguna cosa decreix quan executem el cos del bucle S llavors tenim que

$$\boxed{\{I\}\ \textbf{mentre}\ B\ \textbf{fer}\ S\ \textbf{fmentre}\ \{I\ \wedge\ \neg B\}}$$

El decreixement en cada volta ens permet garantir que sortirem del bucle en algun moment i, per tant, només farem un nombre finit de passes de la iteració. De fet, encara que està amagat, l'acabament de la iteració és el

que ens permet usar la inducció, i garantir que, si en un pas re-satisfem l'invariant, la resta de passes (que són una menys que abans) faran bé la seva feina, i ens portaran a la postcondició $\{I \wedge \neg B\}$.

Ara bé, per regla general, en un disseny iteratiu el que tenim és una precondició i una postcondició donada, i sempre cal fer algunes inicialitzacions. Així que, resulta molt útil per al disseny, i a més com a exercici, extreure, a partir de la semàntica de la iteració i de les altres instruccions del llenguatge, les condicions que s'han de satisfer en aquest cas. És a dir que s'ha de satisfer per que el següent algorisme iteratiu sigui correcte.

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

Segons la semàntica de la iteració cal

1. Trobar un invariant I i una condició B tals que $\{I \wedge B\}S\{I\}$.
2. Garantir que $I \wedge \neg B \Rightarrow Q$.
3. Garantir que $\{P\}Ini\{I\}$.
4. Garantir el decreixement a cada iteració del bucle.

5.3.2 Raonament inductiu

A partir de la semantica axiomàtica anteriorment presentada, el raonament inductiu dirigit al disseny iteratiu el podriem descomposar en les següent fases:

1. Definir l'estat del programa en un punt intermig del problema. Per exemple, suposem que hem fet una part de la feina que s'ha de fer. Podem caracteritzar l'estat ? Aquesta caracterització és la que tradicionalment coneixem com a *invariant*.
2. Deduir com podem caracteritzar l'estat en que la resolució del problema ha acabat. Això és el que ens determinarà la *condició d'acabament del bucle*.
3. Deduir que cal fer per *progressar* en la resolució del problema de manera que es *conservi l'invariant*. Això és el que ens determinarà el cos del bucle.

4. Demostrar que l'execució d'una iteració del bucle ens acosta a la condició d'acabament del bucle. Si cal, podem fer servir una funció de fita per formalitzar-ho. Això ens demostrarà que a la condició d'acabament s'hi arribarà amb un nombre finit d'iteracions (*el bucle acaba*).

5. Deduir com podem caracteritzar l'estat abans de començar el bucle de manera que es cumpleix l'invariant. Això és el que ens determinarà la *inicialització* del bucle.

Igualment que en el disseny recursiu, pot ser que l'ordre en el que es consideren aquests apartats no sigui el que s'indica. En general, la primera fase serà sempre la de trobar l'invariant, però també pot ser que al considerar les següents fases haguem d'afinar l'invariant.

La forma més natural i senzilla d'obtenir invariants, que expressin aquest càlcul intermig, és via la introducció de noves variables (locals). Aquesta és la part més creativa del procés inductiu.

5.3.3 Exemple: factorial

Dissenyar una funció que calculi el factorial d'un nombre $n > 0$.

Especificació:

funció *factorial (n : nat)* ***retorna*** *f : nat*
{`Pre:` —}
{`Post:` $f = n!$}

Raonament inductiu

Anem a aplicar el raonament inductiu al problema anterior.

- **Invariant**. Suposem que hem calculat el factorial fins a un cert nombre $i \leq n$. Tindrem el següent invariant:

$$f = i! \ \wedge \ 0 \leq i \leq n.$$

- **Condició d'acabament**. Haurem acabat quan tinguem el factorial de n, és a dir, $i = n$. Això farà que $f = n!$.

- **Cos del bucle**. Donat que $i \neq n$, cal progressar i mantenir l'invariant. El progrés es garanteix amb "$i := i{+}1$", mentre que l'invariant es manté amb "$f := f * i$".

- **Inicialització**. Cal fer $i = 0$, l'únic valor possible per que es compleixi l'invariant al principi per a qualsevol valor de n. Això comporta $f = 1$.
- **Demostració d'acabament**. A cada iteració s'incrementa i. En algun moment arribarà a ser n.

Solució

```
funció factorial (n : nat) retorna f : nat
{Pre: —}
{Post: f = n!}
    var i: nat;
    i := 0; f := 1;
    {Inv:  f = i! ∧ 0 ≤ i ≤ n}
    mentre i ≠ n fer
        i := i + 1; f := f * i;
    fmentre;
    retorna f;
ffunció
```

5.3.4 Exemple: factorització d'un nombre natural

Dissenyar una acció que escrigui la descomposició en nombres primers del paràmetre $n > 1$ que rep el programa. Per exemple, si $n = 24$, el programa ha d'escriure

```
2 2 2 3
```

Especificació

```
acció factoritzar (ent n: nat)
{Pre:  n > 1}
{Post: Ha escrit la descomposició en factors primers de n}
```

Raonament inductiu

1. **Invariant.** Considerem que ja hem escrit uns quants factors primers $f_1, f_2, \cdots, f_k$, tal que $f_1 \leq f_2 \leq \cdots \leq f_k$. A més tenim que $f_1 \cdot f_2 \cdots f_k \cdot x = n$, on x és la part que ens queda per factoritzar. A més, sabem que hem escrit tots els factors f_i tal que $f_i < f$, on f és una variable que ens recorda fins on hem provat de trobar factors.

2. **Condició d'acabament.** No ens ha de quedar res per factoritzar. Això vol dir que $x = 1$. La condició del bucle serà la complementari, $x \neq 1$.

3. **Cos del bucle.** Poden passar dues coses:

   - f és un factor de x. En aquest cas, podem escriure f i recalcular la part que ens queda per factoritzar (dividim x per f). D'aquesta manera hem progressat (hem escrit un factor més i hem reduit la part per factoritzar) i seguim mantenint l'invariant. Cal tenir en compte que f pot tornar a ser un factor del nou valor d'x. Per aquesta raó l'hem de mantenir al mateix valor.
   - f no és un factor d'x. En aquest cas, podem incrementar f. Això garanteix progrés (anem acostant f a x) i també manté l'invariant.

4. **Inicialització**. L'invariant ens diu que x és la part que ens queda per factoritzar i que s'han escrit tots els factors més petits que f. Al principi tindrem que ens queda per factoritzar tot el nombre ($x = n$) i que el factor més petit serà més gran o igual a 2 ($f = 2$).

5. **Demostració d'acabament.** En aquest cas, es garanteix o bé que x disminueix o bé que f augmenta. En tot cas, mai passarà que $f > x$ essent $x \neq 1$. Altrament, voldria dir que hi ha factors de n (els que té la x) que són menors que f i que no hem escrit, cosa que contradiu l'invariant. Per tant, sempre decreix la distància entre x i f, i aquesta distància és més gran o igual que zero mentre es satisfà la condició del bucle. De fet, quan f es trobi amb x sabem que el valor d'x es tornarà 1 i el programa acabarà.

Una observació important d'aquest algorisme és la següent: el fet que cada vegada escrivim el **factor més petit** de la part sense factoritzar garanteix que aquest factor sigui primer. La demostració és pot fer fàcilment per contradicció.

Solució

```
acció factoritzar (ent n: nat)
{Pre:  n > 1}
{Post: Ha escrit la descomposició en factors primers de n}
    var x: nat; {Nombre que ens queda per factoritzar}
        f: nat; {Factor actual}

    {Inicialitzem de manera que es cumpleixi l'invariant al principi}
    x := n; f := 2;

    {Inv:  S'han escrit els factors tals que multiplicats per x ens donen n.
           A més, s'han escrit tots els factors mes petits que f}
    mentre x ≠ 1 fer
        si x mod f = 0 llavors
            escriure(f); x := x div f;
            {Mantenim l'invariant i progressem (x disminueix)}
        sino
            f := f + 1;
            {Mantenim l'invariant i progressem (f s'acosta a x)}
        fsi;
    fmentre;
facció
```

Capítol 6

Recursivitat i iteració amb dades escalars

La utilització del mètode inductiu per al disseny d'algorismes recursius i iteratius és l'objectiu principal d'aquest capítol. Els problemes tractats només utilitzen dades escalars. Es presenten tant algorismes iteratius com recursius, incloent el raonament inductiu que permet derivar el disseny proposat.

6.1 Factorial

Dissenyar una funció que calculi el factorial d'un nombre $n > 0$. Una versió iterativa d'aquest problema es pot trobar a la Secció 5.3.3 (pàg. 28). Aquí es presenta una solució recursiva.

Especificació:

funció *factorial* $(n : nat)$ ***retorna*** $f : nat$
{`Pre:` —}
{`Post:` $f = n!$}

Solució recursiva

- Cas senzill: per $n \leq 1$, $n! = 1$.
- Cas recursiu: per $n > 1$, $n! = n \cdot (n-1)!$.
- Demostració d'acabament. A cada crida recursiva es decrementa el paràmetre. En algun moment arribarà a ser 1 i acabarà.

```
función factorial (n : nat) retorna f : nat
{Pre: —}
{Post: f = n!}
    si n ≤ 1 llavors f := 1;
    sino f := n∗ factorial(n − 1);
    fsi;
    retorna f;
ffunció
```

6.2 Nombres primers

Dissenyar una funció que determini si un nombre $n > 0$ és primer.

Especificació:

```
función primer (n : nat) retorna p : booleà
{Pre:  n > 0}
{Post: p = "n és un nombre primer"}
```

Comentaris

L'algorisme que es proposa va explorant divisors a partir del 2. La condició d'acabament de l'algorisme aprofita el fet que tot nombre no primer té un divisor que no és més gran que la seva arrel quadrada.

Solució iterativa

- Invariant: cap natural superior o igual a 2 i inferior a k divideix n.
- Condició d'acabament: quan trobi un divisor ($n \bmod k = 0$) o quan ja no en pugui trobar més ($k > \sqrt{n}$). La condició $k > \sqrt{n}$ es pot reescriure com $k * k > n$.
- Cos del bucle: donat que k no divideix a n i $k \leq \sqrt{n}$ (altrament no entraríem al bucle), podem fer "*k := k+1*" per mantenir l'invariant i progressar.
- Inicialització: $k = 2$ fa que l'invariant es compleixi a l'inici del bucle.
- Demostració de l'acabament: el valor de k creix a cada iteració. En cas de no trobar cap divisor, tard o d'hora s'arribarà a complir $k > \sqrt{n}$.

```
función primer (n : nat) retorna p : booleà
{Pre:  n > 0}
{Post: p = "n és un nombre primer"}
    var k : nat;
    k := 2;
    {Inv:  ∀d : 2 ≤ d < k :  n mod d ≠ 0}
    mentre k * k ≤ n  ∧  n mod k ≠ 0 fer
        k := k + 1;
    fmentre;
    retorna k * k > n;
ffunció
```

Solució recursiva

La solució recursiva es basa en una inducció sobre els posibles divisors de n. Per això es dissenya una funció auxiliar que ens diu si el nombre n té algun divisor més gran que d. Amb la següent especificació:

```
función té_divisors (n : nat, d : nat) retorna td : booleà
{Pre:  n > 0, d > 1, n no té cap divisor a l'interval [2, d − 1]}
{Post: td = "n té algun divisor a l'interval [d, n − 1]"}
```

El raonament per a dissenyar aquesta funció és el següent:

- Casos senzills:
  - $d > \sqrt{n}$. Donat que n no té cap divisor a $[2, d-1]$ tampoc en podrà tenir cap a l'interval $[d, n-1]$.
  - $n \bmod d = 0$. Ja hem trobat un divisor.
- Cas recursiu: $n \bmod d \neq 0$. Hi haurà divisors si n'hi ha a l'interval $[d+1, n-1]$.
- Demostració d'acabament. A cada crida s'incrementa d. Tard o d'hora arribarà al cas senzill $d > \sqrt{n}$

```
función té_divisors (n : nat, d : nat) retorna td : booleà
{Pre:  n > 0, d > 1, n no té cap divisor a l'interval [2, d − 1]}
{Post: td = "n té algun divisor a l'interval [d, n − 1]"}
    si d ∗ d > n llavors td := fals;
    sinosi n mod d = 0 llavors td := cert;
    sino td := té_divisors (n, d + 1);
    fsi;
    retorna td;
ffunció
```

Finalment cal fer la crida inicial per saber si un nombre és primer. Cal mirar si té divisors a l'interval $[2, n-1]$.

```
función primer (n : nat) retorna p : booleà
{Pre:  n > 0}
{Post: p = "n és un nombre primer"}
    retorna ¬ té_divisors (n,2);
ffunció
```

6.3 Màxim comú divisor

Dissenyar un algorisme que calculi el màxim comú divisor de dos nombres naturals.

Comentaris

La solució proposada es basa en l'algorisme d'Euclides:

$$mcd(a,b) = \begin{cases} b & \text{per } a = 0 \\ mcd(b \bmod a, a) & \text{per } a \neq 0 \end{cases}$$

Especificació

```
función mcd (a: nat, b: nat) retorna m: nat
{Pre: —}
{Post: m = mcd(a, b)}
```

Solució iterativa

- Invariant: $mcd(a, b) = mcd(A, B)$, on A i B són els valors inicials d'a i b, respectivament.
- Condició d'acabament: $a = 0$ (segons algorisme d'Euclides).
- Cos del bucle: apliquem l'algorisme d'Euclides per mantenir l'invariant. Per progressar, fem que a sempre sigui el més petit dels dos nombres.
- Inicialització: l'invariant és compleix a l'entrada de la funció. No cal afegir cap inicialització.
- Demostració de l'acabament: donat que $b \bmod a < a$ quan $a > 0$, després de la primera iteració sempre tenim que $a < b$. A partir de llavors a sempre decreix en fer-se l'assignació "$a := b \bmod a$". Això fa que, tard o d'hora, a arribi a ser zero.

```
funció mcd(a: nat, b: nat) retorna m: nat
{Pre:  a = A, b = B}
{Post: m = mcd(a, b)}
    {Inv:  mcd(a, b) = mcd(A, B)}
    mentre a ≠ 0 fer
        var aux: nat;
        aux := a; a := b mod a; b := aux;
    fmentre;
    retorna b;
ffunció
```

Solució recursiva

Fem l'anàlisi per casos:

- Cas senzill: $a = 0$, el mcd és b.
- Cas recursiu: $a \neq 0$, el mcd és $mcd(b \bmod a, a)$.
- Demostració d'acabament: semblant a la demostració pel cas iteratiu. El valor del primer paràmetre sempre decreix després de fer la primera crida. En algun moment s'arribarà al cas senzill $a = 0$.

```
función mcd(a: nat, b: nat) retorna m: nat
{Pre: —}
{Post: m = mcd(a, b)}
    si a = 0 llavors m := b;
    sino m := mcd (b mod a, a);
    fsi;
    retorna m;
ffunció
```

6.4 Nombres perfectes

Dissenyar una funció que determini si un natural $n > 0$ és perfecte. Un nombre n és perfecte si és igual a la suma dels seus divisors tret d'ell mateix (p.ex. $n = 6$ és perfecte perquè $n = 1 + 2 + 3$).

Especificació

```
función perfecte (n : nat) retorna p : booleà
{Pre:  n > 0}
{Post: p = "n és perfecte"}
```

Solució Iterativa

- Invariant: *sum* conté la suma dels divisors de n a l'interval $[1, d-1]$.
- Condició d'acabament: que no sigui possible trobar més divisors ($d \leq n$ div 2) o que la suma dels trobats fins ara sigui superior a n ($sum > n$).
- Cos del bucle: per mantenir l'invariant cal sumar d a *sum* si d divideix a n. Per progressar, cal incrementar d.
- Inicialització: d:=1 (no s'ha provat cap divisor) i $sum := 0$ (la suma dels divisors és zero). D'aquesta manera es compleix l'invariant al principi del bucle.
- Demostració de l'acabament: d creix en cada iteració. no pot creixer indefinidament sense arribar a $d > n$ div 2.

```
funció perfecte(n : nat) retorna p : booleà
{Pre:  n > 0}
{Post: p = "n és perfecte"}
    var sum, d : nat;
    sum := 0; d := 1;
    {Inv:  sum conté la suma dels divisors d'n a l'interval [1, d − 1]}
    mentre d ≤ n div 2 ∧ sum ≤ n fer
        si n mod d = 0 llavors sum := sum + d; fsi
        d := d + 1;
    fmentre;
    retorna sum = n;
ffunció
```

6.5 Suma de dígits

Dissenyar una funció que calculi la suma dels dígits d'un natural n representat en base 10. Una versió recursiva d'aquest problema es pot trobar a la Secció 5.2.2 (pàg. 25). Aquí es presenta una versió iterativa.

Especificació

```
funció sumadígits (n : nat) retorna s: nat
{Pre:  —}
{Post:  s = "suma dels dígits d'n representat en base 10"}
```

Solució iterativa

- Invariant: s conté la suma dels dígits de menys pes tractats fins ara i n conté els dígits de més pes que encara no s'han tractat.
- Condició d'acabament: $n = 0$ (no queden dígits per tractar).
- Cos del bucle: per tal de conservar l'invariant, en cada iteració s'actualitza el valor de s sumant un nou dígit que s'elimina de n.
- Inicialització: no s'ha sumat encara cap dígit ($s = 0$).
- Demostració de l'acabament: en cada iteració s'elimina un dígit amb l'operació *mod*, amb la qual cosa es garanteix que n es farà zero en algun moment.

```
función sumadígits (n : nat) retorna s: nat
{Pre: —}
{Post: s = "suma dels dígits d'n representat en base 10"}
    s := 0;
    {Inv: s conté la suma dels dígits de menys pes tractats fins ara
          i n conté els dígits de més pes encara no tractats}
    mentre n > 0 fer
        s := s + n mod 10;
        n := n div 10;
    fmentre;
    retorna s;
ffunció
```

6.6 Arrel digital

Dissenyar una funció que retorni l'arrel digital d'un nombre natural. L'arrel digital és un nombre de només un dígit decimal que s'obté de la suma successiva dels dígits del nombre inicial. Per exemple, l'arrel digital de 83756 és 2 (8+3+7+5+6=29, 2+9 = 11, 1+1 = 2).

Comentari

Per a resoldre aquest problema es fa servir la funció *sumadígits* presentada a la Secció 6.5.

Especificació

```
función arreldigital (n : nat) retorna a: nat
{Pre: —}
{Post: a = "arrel digital de n"}
```

Solució recursiva

- Cas senzill: quan n només té un dígit l'arrel digital és n.
- Cas recursiu: l'arrel digital de n és igual a l'arrel digital de n div 10 (dígits de més a l'esquerra) més n mod 10 (dígit de més a la dreta).
- Acabament: en cada crida recursiva disminueix el nombre de dígits.

```
función arreldigital (n : nat) retorna a: nat
{Pre: —}
{Post: a = "arrel digital d'n"}
    si n < 10 llavors a := n;
    sino a := arreldigital (sumadígits (n));
    retorna a;
ffunció
```

Una altra solució recursiva

La següent solució no necessita la funció *sumadígits*. Es deixa pel lector el raonament que justifica l'algorisme.

```
función arreldigital (n : nat) retorna a: nat
{Pre: —}
{Post: a = "arrel digital d'n"}
    si n < 10 llavors a := n;
    sino a := arreldigital (arreldigital (n div 10) + n mod 10);
    retorna a;
ffunció
```

6.7 Arrel quadrada

Dissenyar una funció que retorni la part entera de l'arrel quadrada d'un nombre natural n.

Especificació

```
función arrel_quadrada (n : nat) retorna a: nat
{Pre: —}
{Post: a = ⌊√n⌋}
```

Solució iterativa

Per a deduir l'invariant del bucle reescriurem la postcondició de la funció de la següent manera:

$$a^2 \leq n < (a+1)^2$$

La estratègia de l'algorisme consistirà en anat provant nombres naturals succesivament fins que en trobem un tal que el seu quadrat sigui superior a n. El natural anterior serà l'arrel quadrada que cerquem.

- Invariant. Tots els nombres que hem provat anteriorment tenen un quadrat no superior a n:

$$(a-1)^2 \leq n$$

- Condició d'acabament: quan trobem un nombre tal que el seu quadrat sigui superior a n ($a * a > n$).
- Cos del bucle: cal provar el natural següent ($a := a + 1$).
- Inicialització: donat que $n \geq 0$, el valor $a = 1$ ja fa complir l'invariant.
- Demostració d'acabament: el fet que a s'incrementi a cada iteració garanteix que la condició d'acabament es farà certa en algun moment.

```
función arrel_quadrada (n : nat) retorna a: nat
{Pre: —}
{Post: a^2 ≤ n < (a+1)^2}
    a := 1;
    {Inv:  (a-1)^2 ≤ n}
    mentre a * a ≤ n fer
        a := a + 1;
    fmentre;
    {(a-1)^2 ≤ n < a^2}
    retorna a − 1;
ffunció
```

Per a una versió recursiva veure el problema 16 d'aquest capítol.

6.8 Canvi de base

Dissenyar una acció que escrigui un nombre natural n en una base b.

Especificació

```
acció canvi_base (ent n, b : nat)
{Pre:  2 ≤ b ≤ 9}
{Post: Ha escrit el nombre n en base b sense cap zero al davant.}
```

Comentaris

L'especificació indica que la representació de n en base b s'escriu sense zeros al davant. Això vol dir que pel cas $n = 0$ no s'escriurà cap dígit. En el cas que es volgués escriure un '0', s'hauria de fer un tractament especial abans de cridar a l'acció.

La representació d'un nombre n en base b es pot caracteritzar amb la següent equació:

$$n = d_k \cdot b^k + d_{k-1} \cdot b^{k-1} + \cdot + d_2 \cdot b^2 + d_1 \cdot b + d_0$$

on d_k és el dígit de més pes i d_o el de menys pes. Per a resoldre aquest problema es farà servir l'acció "*escriure*(x)", que permet escriure un nombre natural x.

Solució recursiva

- Cas senzill ($n = 0$): no cal fer res.
- Cas recursiu ($n > 0$): podem transformar l'equació anterior en la següent:

$$n = b \cdot \underbrace{(d_k \cdot b^{k-1} + d_{k-1} \cdot b^{k-2} + \cdot + d_2 \cdot b + d_1)}_{n \text{ div } b} + \underbrace{d_0}_{n \bmod b}$$

  D'aquesta manera, l'escriptura de n en base b es pot desglosar en dos subproblemes:

  - l'escriptura de n div b (dígits de més pes) seguida de
  - l'escriptura de n mod b (dígit de menys pes)

- Demostració d'acabament: a cada crida recursiva es divideix el paràmetre n per b, eliminant un dígit en base b. En algun moment el paràmetre s'arribarà a fer zero (cas senzill).

```
acció canvi_base (ent n, b : nat)
{Pre:  2 ≤ b ≤ 9}
{Post: Ha escrit el nombre n en base b sense cap zero al davant.}
    si n > 0 llavors
        canvi_base (n div b, b);
        escriure (n mod b);
    fsi;
facció
```

6.9 Problemes

6.1 Dissenyar una funció anomenada `prod` que, donats dos nombres naturals a i b, retorni el producte $p = a * b$. Es pot fer servir la suma i la resta d'enters en l'algorisme, però no el producte (en els dos darrers casos es permetrà el producte i divisió per 2). Dissenyar les següents variants de la funció:

- Versió iterativa, amb invariant

$$\{\texttt{Inv} : p = a * k \ \wedge \ 0 \leq k \leq b\}$$

- Versió recursiva, fent la crida a $\texttt{prod}(a, b - 1)$
- Versió recursiva, fent la crida a $\texttt{prod}(2 * a, b \text{ div } 2)$
- Versió iterativa, similar a la darrera versió recursiva, amb invariant

$$\{\texttt{Inv} : a * b = c * d + p\}$$

  i inicialitzacions

$$c := a; \quad d := b; \quad p := 0;$$

6.2 Dissenyar un algorisme que, donats dos enters x i y, amb $y \geq 0$, calculi l'enter x^y. Es poden fer servir sumes, restes, productes i divisions enteres, però no operacions amb nombres reals ni exponenciació.

6.3 Dissenyar un algorisme que, donats dos enters positius, calculi el quocient i el residu de la seva divisió entera, utilitzant només els operadors aritmètics de suma i resta.

6.4 Dissenyar un algorisme que trobi el nombre de xifres d'un natural donat x.

6.5 Dissenyar un algorisme que retorni el número resultant d'invertir les xifres d'un natural donat x.

6.6 Donats dos nombres naturals x i y, dissenyar una funció que retorni un valor booleà que sigui *cert* si x és un sufix de y en la seva representació en base 10, i retorni *fals* altrament. Considerar que el 0 és un nombre "sense dígits" i, per tant, és sufix de qualsevol altre nombre. Exemples: 32 és sufix de 5632, però 514 no és sufix de 1524.

6.7 Dissenyar un algorisme que ens digui si un natural donat és múltiple de 3. El programa només pot usar la suma (sobre naturals) i el mòdul i la divisió per 10. Cal basar-se en la següent propietat:

> *Un número és múltiple de* 3 *quan la suma dels seus dígits és múltiple de* 3.

6.8 Donat un natural n, anomenem $sp(n)$ la suma dels dígits de les posicions parells i $si(n)$ la suma dels dígits de les posicions imparells, considerant el dígits de menys pes com el de la posició 0. Per saber si n és múltiple d'11 es pot fer servir la següent regla:

> *Un natural* n *és múltiple d'11 si* $|sp(n) - si(n)|$ *és múltiple d'11.*

Dissenyar una funció que digui si un natural és múltiple d'11 fent servir la regla anterior.

6.9 Dissenyar un algorisme que donat un natural n (considerat en base 10) ens retorni un número m (que nomes conté 0's i 1's) que representa n en base 2.

6.10 Generalitzar el problema anterior a qualsevol base b més gran o igual que 2.

6.11 Dissenyar un algorisme que calculi el quadrat d'un enter donat fent servir només sumes, aprofitant la igualtat $(x+1)^2 = x^2 + 2x + 1$. Fer servir una idea similar per a calcular el cub d'un enter donat.

6.12 Dissenyar un algorisme que, donats dos naturals a i un real b, trobi la part entera del logaritme d'a en base b. Recordem que el logaritme de x en base b és el número y tal que $b^y = x$.

6.13 Dissenyar un algorisme que, donat un natural diferent de zero, en trobi el divisor més gran, exclòs ell mateix, i el divisor més petit exclòs l'1. Cal tenir cura del cas que el número sigui primer.

6.14 Dissenyar un algorisme que calcula el mínim comú múltiple de dos naturals diferents de zero. Ajut: utilitzar el càlcul del màxim comú divisor.

6.15 Dissenyar algorismes que, donats enters no negatius n i m, calculin el nombre combinatori

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

Donar diverses solucions triant diversos casos bàsics per al raonament inductiu. Els algorismes han de ser tals que els resultats intermitjos no siguin molt més grans que el resultat final.

6.16 La següent funció calcula la part entera de l'arrel quadrada d'un nombre natural n (veure versió iterativa a la secció 6.7). Demostrar que la funció compleix l'especificació.

Suggeriment: fer la demostració calculant els predicats R, S i Q del cos de la funció.

```
funció arrel_quadrada (n : nat) retorna a: nat
{Pre: —}
{Post: a^2 ≤ n < (a + 1)^2}
    si n = 0 llavors a := 0;
    sino
        a := arrel_quadrada (n div 4);
        {R}
        a := 2 * a;
        {S}
        si (a + 1) * (a + 1) ≤ n llavors a := a + 1 fsi;
        {Q}
    fsi;
    retorna a;
ffunció
```

Capítol 7

Taules

En aquest capítol es presenten algorismes que treballen sobre taules. En molt d'ells, es proposen tant solucions iteratives com recursives.

7.1 Suma d'elements

Dissenyar una funció que calculi la suma dels elements d'una taula $t[1..N]$ d'enters.

Especificació

funció *suma_taula (t:* ***taula****[1..N]* ***de*** *enter)* ***retorna*** *s: enter*
{`Pre:` $N > 0$}
{`Post:` $s = \sum_{k=1}^{N} t[k]$}

Solució iterativa

- Invariant: s conté la suma dels elements de $t[1..i-1]$, amb $1 \leq i \leq N+1$.

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{s = \sum_{k=1}^{i-1} t[k]}\ t[i]\ \cdots\ t[N]$$

- Condició d'acabament: s'hauran sumat tots els elements quan $i = N+1$.

- Cos del bucle: cal progressar (i:=i+1) i mantenir l'invariant ($s := s + t[i]$, abans d'incrementar i).
- Inicialització: el valor $i = 1$ és l'únic que garanteix l'invariant per a qualsevol valor de N. Això comporta que $s = 0$.
- Demostració de l'acabament: l'increment de i a cada iteració garanteix que tard o d'hora arribarà a valer $N + 1$.

```
función suma_taula (t: taula[1..N] de enter) retorna s: enter
{Pre:  N > 0}
{Post: s = Σ_{k=1}^{N} t[k]}
    var i: nat;
    i := 1; s := 0;
    {Inv:  s = Σ_{k=1}^{i-1} t[k], 1 ≤ i ≤ N + 1}
    mentre i ≠ N + 1 fer
        s := s + t[i];
        i := i + 1;
    fmentre;
    retorna s;
ffunció
```

Solució recursiva

Per a una solució recursiva cal dissenyar una funció auxiliar que generalitzi el problema que volem resoldre i utilitzi una variable d'inducció. Així doncs, dissenyarem una funció que calculi la suma dels i primers elements de la taula. Aquesta funció tindrà la següent especificació.

```
función suma_taula_rec (t: taula[1..N] de enter, i: nat) retorna s: enter
{Pre:  N > 0, 1 ≤ i ≤ N}
{Post: s = Σ_{k=1}^{i} t[k]}
```

- Cas senzill: $i = 1$, només cal sumar un element.
- Cas recursiu: $i > 1$, la suma és pot expressar com

$$s = \sum_{k=1}^{i-1} t[k] \; + \; t[i]$$

$$\underbrace{t[1] + t[2] + \cdots + t[i-1]}_{suma\_taula\_rec(t,\, i-1)} + t[i]$$

- Demostració d'acabament: a cada crida es decrementa el valor del paràmetre i. En algun moment arribarà al cas senzill $i = 1$.

```
funció suma_taula_rec (t: taula[1..N] de enter, i: nat) retorna s: enter
{Pre:  N > 0, 1 ≤ i ≤ N}
{Post: s = Σ_{k=1}^{i} t[k]}
    si i = 1 llavors s := t[1];
    sino s := suma_taula_rec (t, i − 1) + t[i];
    fsi;
    retorna s;
ffunció
```

Finalment cal dissenyar la funció que calcula la suma de tots els elements de la taula.

```
funció suma_taula (t: taula[1..N] de enter) retorna s: enter
{Pre:  N > 0}
{Post: s = Σ_{k=1}^{N} t[k]}
    retorna suma_taula_rec (t,N);
ffunció
```

7.2 Element màxim d'una taula

Dissenyar una funció que calculi el valor de l'element màxim d'una taula $t[1..N]$ d'enters.

Especificació

```
funció màxim (t: taula[1..N] de enter) retorna m: enter
{Pre:  N > 0}
{Post: m = max_{k=1..N} t[k]}
```

Solució iterativa

- Invariant: $m = \max_{k=1..i-1} t[k],\ 1 \leq i \leq N+1$

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{m = \max\limits_{k=1..i-1} t[k]}\ t[i]\ \cdots\ t[N]$$

- Condició d'acabament: s'ha calculat el màxim per a tota la taula ($i = N + 1$).
- Cos del bucle: cal mantenir l'invariant per una posició més de la taula. Si l'element $t[i]$ és més gran que m s'ha d'actualitzar m. L'invariant s'ha de fer valer per una posició més (i:=i+1).
- Inicialització: cal inicialitzar m amb un valor de la taula, altrament podria ser més gran que qualsevol valor i donar un resultat erroni. Per això podem fer "$m := t[1]$" i "$i := 2$".
- Demostració de l'acabament: l'increment de i a cada iteració garanteix que tard o d'hora arribarà a valer $N + 1$.

```
función màxim (t: taula[1..N] de enter) retorna m: enter
{Pre:  N > 0}
{Post: m = max_{k=1..N} t[k]}
    m := t[1]; i := 2;
    {Inv:  m = max_{k=1..i-1} t[k], 2 ≤ i ≤ N + 1}
    mentre i ≠ N + 1 fer
        si t[i] > m llavors m := t[i]; fsi;
        i := i + 1;
    fmentre;
    retorna m;
ffunció
```

Solució recursiva

Per a una solució recursiva cal dissenyar una funció auxiliar que generalitzi el problema que volem resoldre i utilitzi una variable d'inducció. Així doncs, dissenyarem una funció que calculi el màxim dels i primers elements de la taula. Aquesta funció tindrà la següent especificació.

```
funció màxim_rec (t: taula[1..N] de enter, i: nat) retorna m: enter
{Pre:  N > 0, 1 ≤ i ≤ N}
{Post: m = max_{k=1..i} t[k]}
```

- Cas senzill: $i = 1$, el màxim és l'element $t[1]$.

- Cas recursiu: $i > 1$, el màxim es pot expressar com

$$m = \max(\max_{k=1..i-1} t[k],\ t[i])$$

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{\text{màxim\_rec}(t, i-1)}\ t[i]$$

- Demostració d'acabament: a cada crida es decrementa el valor del paràmetre i. En algun moment arribarà al cas senzill $i = 1$.

```
funció màxim_rec (t: taula[1..N] de enter, i: nat) retorna m: enter
{Pre:  N > 0, 1 ≤ i ≤ N}
{Post: m = max_{k=1..i} t[k]}
    si i = 1 llavors m := t[1];
    sino
        m := màxim_rec (t, i − 1);
        si t[i] > m llavors m := t[i]; fsi;
    fsi;
    retorna m;
ffunció
```

Finalment cal dissenyar la funció que calcula el màxim de tots els elements de la taula.

```
funció màxim (t: taula[1..N] de enter) retorna m: enter
{Pre:  N > 0}
{Post: m = max_{k=1..N} t[k]}
    retorna màxim_rec (t,N);
ffunció
```

7.3 Inversió d'una taula

Dissenyar una acció que inverteixi el contingut d'una taula. Invertir una taula vol dir posar el primer element a la darrera posició, el segon a la penúltima, etc.

Especificació

```
acció invertir (e/s t: taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s = invertir(t_e)}
```

on t_e i t_s fan referència als valors de la taula a l'entrada i la sortida de l'acció, respectivament. La postcondició es pot reescriure més formalment de la següent manera:

$$\forall k : 1 \leq k \leq N : t_s[k] = t_e[N - k + 1]$$

Solució iterativa

- Invariant: s'han invertit les posicions $t[1..i-1]$ amb les posicions $t[N-i+2..N]$, amb $1 \leq i \leq N \text{ div } 2 + 1$.

$$\underbrace{t[1] \ \cdots \ t[i-1]}_{\text{invertit}} \ t[i] \ \cdots \ t[N-i+1] \ \underbrace{t[N-i+2] \ \cdots \ t[N]}_{\text{invertit}}$$

- Condició d'acabament: quan $i \geq N - i + 1$. Això és equivalent a la condició $i \geq \frac{N+1}{2}$, que en el domini dels enters es pot demostrar que és equivalent a $i \geq N \text{ div } 2 + 1$.

- Cos del bucle: per mantenir l'invariant cal intercanviar els elements $t[i]$ i $t[N - i + 1]$ i fer créixer l'index i en una unitat.

- Inicialització: al principi no hi ha res invertit ($i = 1$).

- Demostració de l'acabament: l'increment de i a cada iteració garanteix que tard o d'hora es complirà la condició d'acabament.

A l'algorisme farem servir l'acció *intercanvi*. La implementació d'aquesta acció es pot trobar a la Secció 4.1 (pàg. 16).

```
acció invertir (e/s t: taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s = invertir(t_e)}
    var i: nat;
    i := 1;
    {Inv:  S'han invertit les posicions t[1..i − 1] amb les posicions t[N − i + 2..N],
           1 ≤ i ≤ N div 2 + 1}
    mentre i < N div 2 + 1 fer
        intercanvi (t[i],t[N − i + 1]);
        i := i + 1;
    fmentre;
facció
```

Solució recursiva

Per a una solució recursiva dissenyarem una acció que inverteixi qualsevol subinterval de la taula. Tindrà la següent especificació:

```
acció invertir_rec (e/s t: taula[1..N] de enter; ent i, j: nat)
{Pre:  N > 0, 1 ≤ i ≤ N, 1 ≤ j ≤ N}
{Post: t_s[i..j] = invertir(t_e[i..j])}
```

Cal notar que l'anterior especificació permet rebre subintervals buits (quan $i > j$). El raonament recursiu és com segueix:

- Cas senzill: l'interval té menys de dos elements ($i \geq j$). En aquest cas no cal fer res.
- Cas recursiu ($i < j$): en aquest cas es poden intercanviar els elements $t[i]$ i $t[j]$, i a continuació invertir el subinterval $t[i+1..j-1]$.
- Demostració d'acabament: a cada crida recursiva es decrementa el tamany del subinterval a invertir. Tard o d'hora s'arribarà a un interval amb menys de dos elements.

```
acció invertir_rec (e/s t: taula[1..N] de enter; ent i, j: nat)
{Pre:  N > 0, 1 ≤ i ≤ N, 1 ≤ j ≤ N}
{Post: t_s[i..j] = invertir(t_e[i..j])}
    si i < j llavors
        intercanvi (t[i], t[j]);
        invertir_rec (t, i + 1, j − 1);
    fsi;
facció
```

Finalment cal dissenyar l'acció que fa la crida inicial per a invertir tota la taula.

```
acció invertir (e/s t: taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s = invertir(t_e)}
    invertir_rec (t,1,N);
facció
```

7.4 Cerca d'un element

Dissenyar una funció que cerqui un element en una taula i ens digui si l'ha trobat i on l'ha trobat.

Especificació

funció *cerca (t:* ***taula****[1..N]* ***de*** *enter, x: enter)*
retorna *(trobat: booleà, i: nat)*
{`Pre:` $N > 0$}
{`Post:` $trobat \Leftrightarrow x \in t, trobat \ \Rightarrow \ t[i] = x$}

En un abús de notació, fem servir $x \in t$ per indicar

$$\exists k : 1 \leq k \leq N : t[k] = x$$

Solució iterativa

- Invariant: $x \notin t[1..i-1],\ trobat \ \Rightarrow \ t[i] = x,\ 1 \leq i \leq N+1$.

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{x \notin t[1..i-1]}\ t[i]\ \cdots\ t[N]$$

- Condició d'acabament: o bé trobem l'element (*trobat*=cert), o bé arribem al límit de la taula ($i = N+1$). Així doncs, la condició d'acabament és

$$trobat \ \vee \ i = N+1$$

- Cos del bucle: si $t[i] = x$, ja hem trobat l'element i cal actualitzar la variable *trobat*. Si $t[i] \neq x$, llavors cal progressar (i:=i+1). En ambdós casos es manté l'invariant. Cal notar que en trobar l'element no s'ha d'incrementar la variable i per mantenir l'invariant ($trobat \ \Rightarrow \ t[i] = x$).

- Inicialització: cal que $i = 1$ per complir l'invariant per a qualsevol valor de N. Fent $trobat = fals$ es compleix l'invariant.

- Demostració de l'acabament: a cada iteració, excepte una, s'incrementa i. Això fa que cada vegada s'acosti al cas $i = N+1$. En cas de no incrementar i, vol dir que s'ha trobat l'element i el bucle també acaba.

```
función cerca (t: taula[1..N] de enter, x: enter)
                retorna (trobat: booleà, i: nat)
{Pre:  N > 0}
{Post: trobat ⇔ x ∈ t, trobat ⇒ t[i] = x}
    var i : nat;
    i:= 1; trobat := fals;
    {Inv:  x ∉ t[1..i − 1], trobat ⇒ t[i] = x, 1 ≤ i ≤ N + 1}
    mentre ¬trobat ∧ i ≠ N + 1 fer
        si t[i] = x llavors trobat := cert;
        sino i:= i + 1;
        fsi;
    fmentre;
    retorna (trobat,i);
ffunció
```

Solució recursiva

Cal dissenyar una funció auxiliar que generalitzi el problema que volem resoldre.

```
funció cerca_rec (t: taula[1..N] de enter, x: enter, k: nat)
                retorna (trobat: booleà, i: nat)
{Pre:  N > 0, 1 ≤ k ≤ N}
{Post: trobat ⇔ x ∈ t[1..k], trobat ⇒ t[i] = x}
```

- Cas senzill: $k = 1$, la cerca és immediata (només cal mirar $t[1]$).
- Cas recursiu: $k > 1$, es pot desglosar en dues cerques
  - la cerca a l'element k, i en el cas de no trobar l'element
  - la cerca als elements $1..k-1$ (crida recursiva).
- Demostració d'acabament: a cada crida recursiva es decrementa el paràmetre k. Tard o d'hora arribarà al valor $k = 1$ (cas senzill).

```
función cerca_rec (t: taula[1..N] de enter, x: enter, k: nat)
                  retorna (trobat: booleà, i: nat)
{Pre:  N > 0, 1 ≤ k ≤ N}
{Post: trobat ⇔ x ∈ t[1..k], trobat ⇒ t[i] = x}
    si k = 1 llavors trobat := t[1] = x; i:=1;
    sino
        si t[k] = x llavors trobat := cert; i:=k;
        sino (trobat,i) := cerca_rec (t,x,k − 1);
        fsi;
    fsi;
    retorna (trobat,i);
ffunció
```

Finalment cal dissenyar la funció que resol el problema inicial.

```
función cerca (t: taula[1..N] de enter, x: enter)
              retorna (trobat: booleà, i: nat)
{Pre:  N > 0}
{Post: trobat ⇔ x ∈ t, trobat ⇒ t[i] = x}
    retorna cerca_rec(t,x,N);
ffunció
```

7.5 Cerca amb sentinella

La cerca amb sentinella és una variant de la cerca que permet un disseny iteratiu més eficient. Per poder fer una cerca amb sentinella cal garantir que l'element cercat es trobi a la taula. Si no és així, cal afegir-lo en una posició auxiliar, normalment la darrera de la taula.

Especificació

```
función cerca_sentinella (t: taula[1..N+1] de enter, x: enter)
                         retorna (trobat: booleà, i: nat)
{Pre:  N > 0}
{Post: trobat ⇔ x ∈ t[1..N], trobat ⇒ t[i] = x}
```

Solució iterativa

- Invariant: $t[N+1] = x,\ x \notin t[1..i-1],\ 1 \leq i \leq N+1$.

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{x \notin t[1..i-1]}\ t[i]\ \cdots\ t[N]\ \underbrace{t[N=1]}_{x}$$

- Condició d'acabament: quan trobem l'element ($t[i] = x$).
- Cos del bucle: com que estem dins del bucle, tenim que $t[i] \neq x$. Així doncs, podem mantenir l'invariant per una posició més fent "i:= i+1".
- Inicialització: per complir l'invariant cal posar l'element a la posició auxiliar ($t[N+1] := x$) i definir $i = 1$.
- Demostració de l'acabament: a cada iteració s'incrementa i. En el pitjor dels casos i arribarà a ser $N+1$ i es complirà la condició d'acabament, donat que $t[N+1] = x$.

```
funció cerca_sentinella (t: taula[1..N+1] de enter, x: enter)
                retorna (trobat: booleà, i: nat)
{Pre:  N > 0}
{Post: trobat ⇔ x ∈ t[1..N], trobat  ⇒  t[i] = x}
    t[N+1]:= x; i := 1;
    {Inv:  t[N+1] = x, x ∉ t[1..i-1], 1 ≤ i ≤ N+1}
    mentre t[i] ≠ x fer
        i:= i + 1;
    fmentre;
    retorna (i ≠ N+1, i);
ffunció
```

7.6 Cerca binària o dicotòmica

La cerca binària o dicotòmica és un algorisme eficient de cerca sobre taules ordenades, basat en la tècnica genèrica de disseny coneguda com *dividir i vèncer*.

Especificació

funció *cerca_binària (t:* ***taula****[1..N]* ***de*** *enter, x : enter)* ***retorna*** *i: nat*
{`Pre:` t *està ordenada creixentment,* $N > 0$}
{`Post:` $1 \leq i \leq N,\ x \in t \ \Leftrightarrow\ t[i] = x$ }

Solució iterativa

- Invariant: $1 \leq inf \leq sup \leq N,\ x \in t \ \Leftrightarrow\ x \in t[inf..sup]$
- Condició d'acabament: quan l'interval de cerca tingui només un element ($inf = sup$).
- Cos del bucle: es compara el valor del mig de l'interval de cerca amb l'element cercat i es redueix l'interval per la meitat, eliminant el subinterval on no pugui pertànyer l'element.
- Inicialització: *inf* i *sup* defineixen l'interval $[1..N]$.
- Demostració d'acabament: a cada iteració es redueix la distància entre *inf* i *sup*. Per tant, tard o d'hora arribaran a ser iguals.

```
funció cerca_binària (t: taula[1..N] de enter, x : enter) retorna i: nat
{Pre:  t està ordenada creixentment, N > 0}
{Post: 1 ≤ i ≤ N, x ∈ t ⇔ t[i] = x }
    var inf, sup: nat;
    inf := 1; sup := N;
    {Inv:  inf ≤ sup, x ∈ t ⇔ x ∈ t[inf..sup]}
    mentre inf ≠ sup fer
        var k: nat;
        k := (inf + sup) div 2;
        si t[k] < x llavors inf := k + 1;
        sino sup := k;
        fsi;
    fmentre;
    retorna inf;
ffunció
```

La demostració d'acabament de la cerca dicotòmica ha de tenir en compte el fet que la divisió d'enters és per defecte. Depenent de com es dissenyés el cos del bucle, l'acabament no es podria garantir. Per exemple, el següent cos del bucle no garantiria l'acabament.

```
k := (inf + sup) div 2;
si t[k] ≤ x llavors inf := k;
sino sup := k − 1;
fsi;
```

Suposem, per exemple, que $inf = 3$ i $sup = 4$. En aquest cas compararíem amb la posició $k = 3$. Si es dona el cas que $t[k] \leq x$, llavors s'executaria l'assignació "$inf := 3$", deixant *inf* i *sup* amb els mateixos valors que a la iteració anterior. Això faria que el bucle no acabés mai.

7.7 Problemes

7.1 Dissenyar un algorisme que compti quantes vegades es troba un enter donat en una taula d'enters.

7.2 Dissenyar un algorisme que compti quantes parells i quants senars hi ha en una taula de naturals.

7.3 Dissenyar un algorisme que calculi la mitjana d'una taula de nombres reals, i un altre que en calculi la variància. Recordem que la variància d'un conjunt d'elements $\{x_1, \ldots, x_n\}$ es defineix com $\sum_{i=1}^{n}(x_i - \mu)^2/n$, on μ és la mitjana.

7.4 Dissenyar un algorisme que calculi la mitjana i la variància d'una taula de reals consultant-ne només un cop cada component. No es permet utilitzar cap altra taula. Pista: reescriure la definició de la variància en una forma més convenient.

7.5 Dissenyar un altre algorisme que determini si la taula conté només elements parells.

7.6 Dissenyar un algorisme que digui si una taula d'enters està ordenada creixentment.

7.7 Dissenyar un algorisme que decideixi, per a una taula $[1..2N]$ d'enters, si els components d'índex parell de la taula estan ordenats en ordre creixent i els senars en ordre decreixent.

7.8 Dissenyar una acció que, donada una taula d'enters, la modifiqui sumant 1 a tots els seus element.

7.9 Dissenyar una funció que rebi una taula d'una certa mida i que determini si és un palíndrom (és el mateix llegir-la d'esquerra a dreta que de dreta a esquerra) o no.

7.10 Dissenyar una funció que, donada una taula d'enters amb el primer element diferent de zero, compti el nombre dels seus canvis de signe. Un canvi de signe és l'aparició de dos enters de signes diferents, separats com a màxim per zeros.

7.11 Dissenyar un algorisme que, donada una taula de naturals, tots entre 1 i M,

1. compti el nombre de vegades que apareix cadascun d'ells, és a dir, quants cops apareix l'1, quants el 2, i així fins al M.
2. en calculi la moda, és a dir, l'element que més cops hi apareix.

7.12 Dissenyar una funció que, donada una taula d'enters, determini quants canvis de pendent conté. Un canvi de pendent és un element que és l'últim d'una seqüència estrictament creixent i el primer d'una estrictament decreixent, o a l'inrevés. Els extrems de la taula no es consideren en cap cas canvi de pendent.

Per exemple, la taula

$$[1, 3, 7, 9, 6, 2, 4, 4, 5, 7, 3]$$

té tres canvis de pendent: 9, 2 i 7

7.13 Donada una taula a, un segment nul és un fragment de la taula tal que la suma dels seus elements és zero. Dissenyar un algorisme que trobi el segment nul més gran d'una taula ordenada creixentment.

7.14 Suposem que a i b són dues taules de nombres enters positius, amb els llocs buits marcats amb un 0. Escriviu algorismes que emmagatzemin en una altra taula:

1. la intersecció dels elements de a i b.
2. la unió dels elements de a i b.

Repetir els apartats anteriors suposant que les taules estan ordenades.

7.15 Suposar que a i b són dues taules (ordenades) estrictament creixents de nombres enters. Dissenyar un algorisme que calculi el nombre de valors que apareixen en totes dues taules.

7.16 Un polinomi d'una variable x amb coeficients reals i grau com a molt N es pot representar amb una taula $[0..N]$ de reals. La component $i$èssima de la taula és el coeficient de x^i en el polinomi. Dissenyar algorismes per a:

- donats dos polinomis p i q, calcular el polinomi $p + q$.
- donat un polinomi p, calcular el polinomi p' (derivada de p).
- donat un polinomi p, calcular el polinomi integral de p.
- donat un polinomi p, calcular el grau del polinomi.
- donat un polinomi p i un real x, avaluar $p(x)$.
- donats dos polinomis p i q, calcular el polinomi producte $p * q$.

7.17 Siguin $a[1..N]$ i $b[1..N]$ dues taules d'enters. Dissenyar un algorisme que calculi nombre de parells (i, j) tals que $a(i) + b(j) > 0$.

Suposar després que a i b són creixents, i intentar donar una solució molt més eficient.

7.18 Dissenyar un algorisme que, donada una taula creixent de nombres enters $a[1...N]$ i un nombre enter $k \geq 0$, calculi el nombre de parells (i, j) tals que

$$1 \leq i \leq j \leq n \ \wedge \ a(j) - a(i) \leq k.$$

7.19 Dissenyar una funció que donada una taula d'enters, determini si hi ha alguna posició que conté la suma dels elements de totes les posicions anteriors.

7.20 Dissenyar una funció que donada una taula de [1..N] d'enters, determini si hi ha algun α entre 0 i N tal que la suma dels elements entre 1 i α coincideix amb la suma dels elements entre $\alpha + 1$ i N.

7.21 Un nombre natural de com a molt N dígits es pot representar per una taula $[0..N - 1]$. El dígit de menys pes és el de la posició 0 i el de més pes el de la posició $N - 1$. Si la representació és en base B, cada component de la taula conté un natural entre 0 i $B - 1$.

L'interès d'aquesta és que permet representar nombres molt més grans que els del tipus predefinit sense provocar sobreeiximents, triant N prou gran.

Dissenyar algorismes que, amb naturals representats d'aquesta manera,

- calculi la suma de dos naturals, indicant si hi ha sobreeiximent.
- calculi la resta de dos naturals, indicant si hi ha error.
- donats naturals a i b, determini si $a = b$, $a < b$ o $a > b$.

7.22 Una taula està "mitjanament ordenada" si per a tot element de la taula es compleix que el seu valor és més gran o igual a la mitjana dels valors anteriors. Per exemple, la taula

```
1 5 3 5 4 6 5
```

està mitjanament ordenada. En canvi, la taula

```
1 5 3 5 3 6 5
```

no ho està degut al fet que la mitjana de `1 5 3 5` és més gran que `3`. Dissenyar una funció que digui si una taula està mitjanament ordenada.

Capítol 8

Ordenació de taules

Els algorismes que segueixen ordenen de manera *creixent* taules d'enters. Els dos primers (inserció i selecció) són algorismes bàsics de complexitat quadràtica, mentre que els dos últims són algorismes avançats força més eficients basats, com la cerca binarìra, en l'estratègia *dividir i vèncer*.

Tots els algorismes presentats en aquest capítol tenen la següent especificació:

acció *ordenar (**e/s** t: **taula** [1..N] **de** enter)*
{`Pre:` $N > 0$}
{`Post:` $t_s \in permut(t_e),\ ordenada(t_s)$}

Nomenclatura utilitzada en aquest capítol

La notació t_e i t_s la fem servir per referir-nos als valors de la taula a l'*entrada* i la *sortida* de l'acció, respectivament. Aquesta és una notació que utilitzarem sovint en les especificacions d'accions que tinguin paràmetres d'entrada/sortida.

La notació $x \in permut(y)$ la fem servir per indicar que x i y són taules del mateix tamany i que el contingut de x és el mateix que el de y amb un possible canvi d'ordre dels seus elements. Direm que x *és una permutació de* y.

El predicat $ordenada(t)$ indica que els elements de la taula t estan ordenats. Si no diem res més al respecte, suposarem que estan ordenats creixentment.

Fent un abús de notació, també farem servir predicats del següent tipus:

$$t[i..j] \leq t[p..q]$$

que serà una abrevació del predicat

$$\forall k, r : i \leq k \leq j, p \leq r \leq q : t[k] \leq t[r]$$

o dit en llenguatge natural: "*qualsevol dels elements de l'interval* $t[i..j]$ *és menor o igual que qualsevol dels elements de l'interval* $t[p..q]$". En particular, també podrem utilitzar la notació

$$x \leq t[i..j]$$

quan només ens referim a un element.

8.1 Ordenació per inserció

L'ordenació per inserció procedeix de manera molt similar a com ordenem un joc de cartes. El que fa l'algorisme és, per cada element no ordenat, inserir-lo al lloc que li pertoca a la part ja ordenada de la taula.

A continuació presentem el raonament per a un algorisme iteratiu.

- Invariant:

$$t \in permut(t_e),\ ordenada(t[1..i-1]),\ 1 \leq i \leq N+1$$

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{ordenada}\ t[i]\ \cdots\ t[N]$$

- Condició d'acabament: quan tota la taula estigui ordenada ($i = N+1$).
- Cos del bucle: farem que hi hagi un element més ordenat a cada iteració. Per això inserirem l'element $t[i]$ a lloc que li correspongui dins de $t[1..i-1]$, desplaçant els elements més grans un lloc cap a la dreta. A continuació, incrementarem i.
- Inicialització. Al principi ja tenim al menys un element ordenat. Per fer vàlid l'invariant n'hi ha prou en fer $i = 2$.
- Demostració d'acabament: a cada iteració s'incrementa i. Això garanteix que tard o d'hora arribarà a ser $N + 1$.

```
acció ordenar_inserció (e/s t: taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s ∈ permut(t_e), ordenada(t_s)}
    var i: nat;
    i := 2;
    {Inv:  t ∈ permut(t_e), ordenada(t[1..i-1]), 1 ≤ i ≤ N+1}
    mentre i ≠ N+1 fer
        inserir t[i] a t[1..i-1];
        i := i+1;
    fmentre;
facció
```

Refinament de "*inserir* $t[i]$ *a* $t[1..i-1]$"

L'inserció de l'element $t[i]$ a l'interval $t[1..i-1]$ es farà iterativament, desplaçant un element de l'interval una posició a la dreta si és més gran que $t[i]$. Inicialment, s'emmagatzemerà l'ement $t[i]$ a una variable auxiliar x.

- Invariant: tindrem una part de l'interval desplaçat una posició cap a la dreta. Aquest serà el subinterval $t[j+1..i]$, on $t[j]$ és l'element a tractar a la propera iteració. L'invariant és:

$$x < t[j+1..i],\ 1 < j \leq i$$

- Condició d'acabament: Ens aturarem quan trobem un element que sigui més petit o igual que x o quan arribem a l'inici de la taula. Per evitar l'avaluació de la posició inexistent "0" de la taula, ens aturarem una posició abans. Així, la condició d'acabament és:

$$j = 2 \ \vee \ t[j-1] \leq x$$

  Això comportarà un tractament especial pel darrer element de la taula que s'haurà de fer fora del bucle.

- Cos del bucle: donat que $t[j-1] > x$, desplacem aquest element un lloc a la dreta

$$t[j] := t[j-1];\ i := j-1;$$

- Inicialitzacio: extraiem l'element $t[i]$ i el guardem a x. Inicialment, l'interval desplaçat és buit ($j = i$). Demostració d'acabament: a cada iteració es decrementa el valor de j. Això garanteix que, en el pitjor dels casos, j arribarà a valer 2.

```
refinament inserir t[i] a t[1..i − 1]
    var j: nat; {Per recórrer la part ordenada de la taula}
        x: enter; {Per emmagatzemar temporalment l'element a inserir}
    x := t[i]; j := i;
    {Inv:  x < t[j + 1..i], 1 < j ≤ i}
    mentre j ≠ 2 ∧ t[j − 1] > x fer
        t[j] := t[j − 1]; j := j − 1;
    fmentre;
    {j = 2 ∨ t[j − 1] ≤ x}
    si t[j − 1] ≤ x llavors t[j] := x;
    sino t[2] := t[1]; t[1] := x;
    fsi;
```

L'anterior refinament es pot simplificar si utilizem una avaluació "curt-circuitada" de la condició d'acabament, tal com es fan en lleguatges com C o Java. L'invariant tindria ara la condició $j \geq 1$ en lloc de $j > 1$. Farem servir la notació $\wedge_c$ per denotar l'avaluació curtcircuitada de la conjunció. El refinament quedaria com segueix:

```
refinament inserir t[i] a t[1..i − 1]
    var j: nat, x: enter;
    x := t[i]; j := i;
    {Inv:  x < t[j + 1..i], 1 ≤ j ≤ i}
    mentre j ≠ 1 ∧c t[j − 1] > x fer
        t[j] := t[j − 1]; j := j − 1;
    fmentre;
    t[j] := x;
```

Finalment es presenta l'algorisme complet:

```
acció ordenar_inserció (e/s t: taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s ∈ permut(t_e), ordenada(t_s)}
    var i: nat;
    i := 2;
    {Inv:  t ∈ permut(t_e), ordenada(t[1..i − 1]), 1 ≤ i ≤ N + 1}
    mentre i ≠ N + 1 fer
        var j: nat, x: enter;
        x := t[i]; j := i;
        {Inv:  x < t[j + 1..i], 1 ≤ j ≤ i}
        mentre j ≠ 1  ∧_c  t[j − 1] > x fer
            t[j] := t[j − 1]; j := j − 1;
        fmentre;
        t[j] := x; i := i + 1;
    fmentre;
facció
```

8.2 Ordenació per selecció

L'ordenació per selecció es basa en triar en cada moment l'element més petit de la part no ordenada de la taula i situar-lo a continuació dels elements ja ordenats. L'invariant és semblant al de la ordenació per inserció, però afegint la condició de que qualsevol element de la part ordenada és més petit o igual que qualsevol element de la part no ordenada. Això fa que la part ordenada ja no s'hagi de modificar més durant la resta de l'algorisme.

- Invariant:

$$t \in permut(t_e),\ ordenada(t[1..i-1]), t[1..i-1] \leq t[i..N],\ 1 \leq i \leq N$$

$$\underbrace{t[1]\ t[2]\ \cdots\ t[i-1]}_{ordenada} \quad \leq \quad \underbrace{t[i]\ \cdots\ t[N]}_{no\ ordenada}$$

- Condició d'acabament: quan tota la taula estigui ordenada ($i = N$). Cal notar que la ordenació de l'interval $t[1..N-1]$ ja implica la ordenaci'o de la tula sencera, donat que l'element $t[N]$ no és més petit que l'element $t[N-1]$, segons l'invariant.

- Cos del bucle: farem que hi hagi un element més ordenat a cada iteració. Per això cal cercar l'element més petit a l'interval $t[i..N]$ i

posar-lo a la posició i. L'element que estava a la posició i el podem posar al lloc on estava l'element mínim. Una vegada fet això, podem incrementar i per fer valer l'invariant per una posició més.

- Inicialització. Al principi no hi ha cap element ordenat ($i = 1$).
- Demostració d'acabament: a cada iteració s'incrementa i. Això garanteix que tard o d'hora arribarà a ser N.

```
acció ordenar_selecció (e/s t : taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s ∈ permut(t_e), ordenada(t_s)}
    var i: nat;
    i := 1;
    {Inv:  t ∈ permut(t_e), ordenada(t[1..i − 1]),
           t[1..i − 1] ≤ t[i..N], 1 ≤ i ≤ N}
    mentre i ≠ N fer
        var k: nat;
        k := "posició de l'element mínim de t[i..N]";
        intercanvi (t[i], t[k]);
        i := i + 1;
    fmentre;
facció
```

Per l'acció *intercanvi* es pot fer servir l'algorisme presentat a la Secció 4.1 (pàg. 16). Per a refinar la cerca de l'element mínim de l'interval $t[i..N]$ es pot fer servir un algorisme semblant al descrit a la Secció 7.2 (pàg. 45) per a calcular l'element màxim d'una taula. En aquest cas, el que cerquem és la posició on es troba l'element en lloc del valor de l'element.

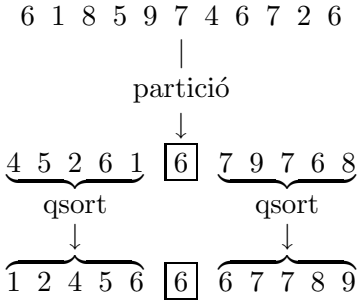
Deixem pel lector el raonament d'aquesta part de l'algorisme. A continuació es presenta l'algorisme d'ordenació per selecció complet.

```
acció ordenar_selecció (e/s t : taula[1..N] de enter)
{Pre:  N > 0}
{Post: t_s ∈ permut(t_e), ordenada(t_s)}
    var i: nat;
    i := 1;
    {Inv:  t ∈ permut(t_e), ordenada(t[1..i-1]),
           t[1..i-1] ≤ t[i..N], 1 ≤ i ≤ N}
    mentre i ≠ N fer
        var k, j: nat;
        k := i; j := i + 1
        {Inv:  t[k] = min(t[i..j-1]), i < j ≤ N + 1}
        mentre j ≠ N + 1 fer
            si t[j] < t[k] llavors k := j; fsi;
            j := j + 1;
        fmentre;
        intercanvi (t[i], t[k]);
        i := i + 1;
    fmentre;
facció
```

8.3 Quick sort

L'algorisme d'ordenació ràpida o *quicksort* es basa en la idea de dividir la taula en dues parts al voltant d'un element anomenat *pivot*, garantint que els elements d'una de les parts siguin tots ells inferiors o iguals al pivot, i que els elements de l'altra part siguin tots ells superiors o iguals al pivot. A continuació es procedeix recursivament a ordenar cadascuna de les parts.

El següent gràfic mostra un exemple de l'estratègia utilitzada.



En primer lloc, donarem l'especificació de l'algorisme de partició per poder-lo utilitzar en el raonament de l'algorisme de *quicksort*.

acció *partició (**e/s** t: **taula**[1..N] **de** enter; **ent** i, j: nat; **sort** k: nat)*
{Pre: $N > 0,\ 1 \leq i < j \leq N$}
{Post: $t_s[i..j] \in permut(t_e[i..j]),\ i \leq k \leq j,\ t[i..k-1] \leq t[k] \leq t[k+1..j]$}

I ara l'especificació de l'algorisme *qsort*. Cal notar que en aquesta especificació es permet cridar a *qsort* amb intervals buits ($i > j$).

acció *qsort (**e/s** t: **taula**[1..N] **de** enter; **ent** i, j: nat)*
{Pre: $N > 0,\ 1 \leq i \leq N+1,\ 0 \leq j \leq N$}
{Post: $t_s[i..j] \in permut(t_e[i..j]), ordenada(t_s[i..j])$}

Raonament recursiu de *qsort*

- Cas senzill: quan l'interval a ordenar sigui buit o només tingui un element ($i \geq j$), no caldrà fer res.
- Cas recursiu ($i < j$): cal partir l'interval en dos troços de tal manera que els elements del subinterval de l'esquerra siguin menors o iguals que els elements del subinterval de la dreta. Després cal ordenar cadascun d'ells.
- Demostració d'acabament: cadascun dels troços partits és més petit que el troç original. En algun moment s'arribarà a troços buits o d'un element.

acció *qsort (**e/s** t: **taula**[1..N] **de** enter; **ent** i, j: nat)*
{Pre: $N > 0,\ 1 \leq i \leq N+1, i; 0 \leq j \leq N$}
{Post: $t_s[i..j] \in permut(t_e[i..j]), ordenada(t[i..j])$}
    var *k: nat;*
    si $i < j$ ***llavors***
        partició (t, i, j, k);
        qsort (t, i, k − 1);
        qsort (t, k + 1, j);
    fsi;
facció

Únicament resta dissenyar l'acció principal d'ordenació que crida a *qsort*:

```
acció ordenar (e/s t: taula [1..N] de enter)
{Pre:  N > 0}
{Post: t_s ∈ permut(t_e), ordenada(t_s)}
    qsort (t,1,N);
facció
```

Raonament iteratiu de *partició*

Per a fer la partició, triarem com a element pivot el primer de l'interval, $t[i]$, i desglosarem l'algorisme en dos parts:

1. Partir l'interval $t[i..j]$ en dos subintervals, $t[i..k]$ i $t[k+1..j]$, tal que $t[i..k] \leq t[i] \leq t[k+1..j]$.
2. Posar el pivot, $t[i]$ a la posició k de l'interval

Cal indicar que la segona part de l'algorisme és necessària per garantir un acabament de *qsort*. En posar el pivot a la posició k, podem reduir la ordenació als intervals $t[i..k-1]$ i $t[k+1..j]$, garantint que les dues parts són més petites que l'interval original.

Per resoldre la primera part (iterativa) de l'algorisme farem servir un invariant descrit pel següent diagrama:

$$\underbrace{t[i] \ \cdots \ t[k]}_{\leq t[i]} \ t[k+1] \ \cdots \ t[q] \ \underbrace{t[q+1] \ \cdots \ t[j]}_{\geq t[i]}$$

- Invariant:
$$i \leq k \leq q \leq j, \ t[i..k] \leq t[i] \leq t[q+1..j]$$
Aquest invariant descriu la situació on els extrems esquerra i dret de l'interval jan han estat classificats i on només ens queda el subinterval del mig, $t[k+1..q]$, per classificar.

- Condició d'acabament: quan tots els elements hagin estat classificats ($k = q$).

- Cos del bucle: a cada iteració classificarem l'element $t[q]$. Si és més petit que $t[i]$, l'intercanviarem amb l'element $t[k+1]$, altrament el deixarem on està. En ambdós casos, un dels subintervals classificats creixerà, és a dir, caldrà incrementar k o decrementar q.

- Inicialització: per a complir l'invariant, n'hi ha prou amb tenir $k = i$ i $q = j$, és a dir, només l'element $t[i]$ ha estat classificat.

- Demostració d'acabament: a cada iteració s'incrementa k o es decrementa q. Per tant, en algun moment es trobaran i es complirà la condició d'acabament ($k = q$).

```
acció partició (e/s t: taula[1..N] de enter; ent i, j: nat; sort k: nat)
{Pre:  N > 0, 1 ≤ i < j ≤ N}
{Post: t_s[i..j] ∈ permut(t_e[i..j]), i ≤ k ≤ j, t[i..k − 1] ≤ t[k] ≤ t[k + 1..j]}
    var q: nat;
    k := i; q := j;
    {Inv:  i ≤ k ≤ q ≤ j, t[i..k] ≤ t[i] ≤ t[q + 1..j]}
    mentre k ≠ q fer
        si t[q] < t[i] llavors
            intercanvi (t[k + 1], t[q]); k := k + 1;
        sino
            q := q − 1;
        fsi;
    fmentre;
    {Ara posem el pivot a la posició k}
    intercanvi (t[i] t[k]);
facció
```

L'algorisme que s'ha presentat per a fer la partició de l'interval pot derivar-se amb un raonament relativament senzill. Tot i això, hi ha altres versions una mica més complicades que redueixen el nombre d'intercanvis a realitzar i, per tant, són més eficients.

8.4 Fusió de taules ordenades

Dissenyar una funció que rebi dues taules ordenades i retorni una altra taula que sigui la fusió ordenada de les dues taules.

Especificació

```
funció fusió (A: taula[1..N] de enter; B: taula[1..M] de enter)
                retorna C: taula[1..N+M] de enter
{Pre:  N > 0, M > 0, ordenada(A), ordenada(B)}
{Post: C ∈ permut(A ∪ B), ordenada(C)}
```

Solució iterativa

La solució que es presenta aquí realitzar la fusió en dues fases:

1. Fusiona elements de A i de B fins exhaurir els elements d'una de les taules.

2. Afegir els elements de la taula no completada.

D'aquesta manera, la primera fase resulta més senzilla. En aquesta secció només presentarem el raonament iteratiu per a la primera fase i comentarem la segona fase. L'invariant de la primera queda il·lustrat amb la següent figura.

$$\underbrace{A[1] \ \cdots \ A[i-1]}_{\text{fusionat}} \ A[i] \ \cdots \ A[N]$$

$$\underbrace{B[1] \ \cdots \ \cdots \ B[j-1]}_{\text{fusionat}} \ B[i] \ \cdots \ B[M]$$

$$\underbrace{C[1] \ \cdots \ \cdots \ \cdots \ \cdots \ C[i+j-2]}_{permut(A[1..i-1] \cup B[1..j-1])} \ C[i+j-1] \ \cdots \ C[N+M]$$

En tot moment, haurem fusionat uns quants elements de A i de B (els més petits) i els haurem posat ordenadament a C. Per progressar caldrà agafar l'element més petit, triat entre $A[i]$ i $B[j]$, i posar-el a la posició $C[k]$.

- Invariant:

  $C[1..k-1] \in permut(A[1..i-1] \cup B[1..j-1]),\ ordenada(C[1..k-1])$
  $C[1..k-1] \leq A[i..N],\ C[1..k-1] \leq B[j..M],\ k = i+j-1$

- Condició d'acabament: quan tots els elements d'alguna de les taules ja hagin estat fusionats ($i = N+1 \ \vee \ j = M+1$).

- Cos del bucle: caldrà triar l'element més petit entre $A[i]$ i $B[j]$ per posar-el a $C[k]$. Per mantenir l'invariant caldrà incrementar els indexos corresponents.

- Inicialització: al començament no hi ha res fusionat ($i = j = k = 1$).

- Demostració d'acabament: a cada iteració agafem un element de A o de B. Tard o d'hora s'exhauriran els elements d'una de les taules.

```
funció fusió (A: taula[1..N] de enter; B: taula[1..M] de enter)
                retorna C: taula[1..N+M] de enter
{Pre:  N > 0, M > 0, ordenada(A), ordenada(B)}
{Post: C ∈ permut(A ∪ B), ordenada(C)}
    var i, j, k: nat;
    i := 1; j := 1; k := 1;
    {Inv:  C[1..k − 1] ∈ permut(A[1..i − 1] ∪ B[1..j − 1]), ordenada(C[1..k − 1])
           C[1..k − 1] ≤ A[i..N], C[1..k − 1] ≤ B[j..M] ∧ k = i + j − 1}
    mentre i ≤ N ∧ j ≤ M fer
        si A[i] < B[j] llavors C[k] := A[i]; i := i + 1;
        sino C[k] := B[j]; j := j + 1;
        fsi;
        k := k + 1;
    fmentre;

    {Començament de la segona fase}
    {Inv ∧ (i = N + 1 ∨ j = M + 1)}
    mentre i ≤ N fer
        C[k] := A[i]; i := i + 1; k := k + 1;
    fmentre;

    {Inv ∧ i = N + 1}
    mentre j ≤ M fer
        C[k] := B[j]; j := j + 1; k := k + 1;
    fmentre;

    {(Inv ∧ i = N + 1 ∧ j = M + 1) ⇒ Post}
    retorna C;
ffunció
```

Cal notar que la segona fase té dos bucles, però només un d'ells executarà alguna iteració donat que una de les taules ja haurà estat fusionada completament. Els bucles de la segona fase només fan una còpia dels elements no fusionats a la taula C.

8.5 Ordenació per fusió (merge sort)

L'ordenació per fusió o *merge sort* es basa en la idea de dividir la taula en dues parts que ordenem per separat, i que a continució fusionem. El procés

d'ordenació de les parts és recursiu, mentre que es procedeix iterativament a l'hora de fusionar.

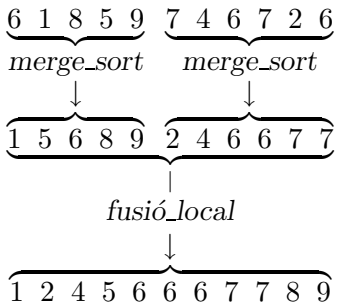
Per a realitzar aquesta ordenació, utilitzarem un algorisme de fusió de taules molt semblant al presentat a l'anterior secció. Per aquesta raó no es descriurà el raonament corresponent a la fusió i únicament es presentarà una adaptació de l'algorisme pel cas del *merge sort*. La característica principal d'aquest algorisme, que anomenarem *fusió_local* és que la fusió es realitza sobre el mateix interval a ordenar. Tot i això, cal una taula auxiliar. L'especificació de *fusió_local* és la següent:

acció *fusió_local (**e/s** t: **taula**[1..N] **de** enter; **ent** i, k, j: nat)*
{Pre: $N > 1,\ 1 \leq i \leq k < j \leq N,\ ordenada(t[i..k]),\ ordenada(t[k+1..j])$}
{Post: $t_s[i..j] \in permut(t_e[i..j]), ordenada(t[i..j])$}

A continuació detallem l'especificació de l'acció principal del *merge sort*, que és idèntica a la del *quick sort*.

acció *merge_sort (**e/s** t: **taula**[1..N] **de** enter; **ent** i, j: nat)*
{Pre: $N > 0,\ 1 \leq i \leq N+1,\ 0 \leq j \leq N$}
{Post: $t_s[i..j] \in permut(t_e[i..j]), ordenada(t[i..j])$}

El següent gràfic mostra un exemple de l'estratègia utilitzada.



Raonament recursiu de *merge sort*

- Cas senzill: quan l'interval a ordenar sigui buit o només tingui un element ($i \geq j$), no caldrà fer res.
- Cas recursiu ($i < j$): cal partir l'interval en dos troços, ordenar cadascun d'ells i fusionar-los.

- Demostració d'acabament: cadascun dels troços partits és més petit que el troç original. En algun moment s'arribarà a troços buits o d'un element.

```
acció merge_sort (e/s t: taula[1..N] de enter; ent i, j: nat)
{Pre:  N > 0, 1 ≤ i ≤ N + 1, 0 ≤ j ≤ N}
{Post: t_s[i..j] ∈ permut(t_e[i..j]), ordenada(t[i..j])}
    si i < j llavors
        var k: nat;
        k := (i + j) div 2;
        merge_sort(t, i, k);
        merge_sort(t, k + 1, j);
        fusió_local (t, i, k, j);
    fsi;
facció
```

Fusió de subintervals adjacents

Finalment només resta dissenyar l'acció *fusió_local* que fusiona dos subintervals adjacents i ordenats. El raonament iteratiu pel disseny d'aquesta acció és molt similar al de la funció *fusió* presentada a l'anterior secció (pàg. 66). Per aquesta raó no el descriurem aquí. En el cas de la *fusió_local* cal deixar el resultat a la mateixa taula. Per aquesta raó cal declarar una taula auxiliar i afegir una fase de còpia al final de la fusió.

```
acció fusió_local (e/s t: taula[1..N] de enter; ent i, k, j: nat)
{Pre:  N > 1, 1 ≤ i ≤ k < j ≤ N, ordenada(t[i..k]), ordenada(t[k + 1..j])}
{Post: t_s[i..j] ∈ permut(t_e[i..j]), ordenada(t[i..j])}
    var x: taula[1..N] de enter; {taula auxiliar per a la fusió}
        p, q, r: nat; {indexos de taules per a la fusió}

    {Primera fase: fusió fins exhaurir un dels intervals}
    p := i; q := k + 1; r := i;
    mentre p ≤ k ∧ q ≤ j fer
        si t[p] < t[q] llavors x[r] := t[p]; p := p + 1;
        sino x[r] := t[q]; q := q + 1;
        fsi;
        r := r + 1;
    fmentre;

    {Segona fase: afegit dels elements restants}
    mentre p ≤ k fer
        x[r] := t[p]; r := r + 1; p := p + 1;
    fmentre;
    mentre q ≤ j fer
        x[r] := t[q]; r := r + 1; q := q + 1;
    fmentre;

    {Tercera fase: còpia de la taula auxiliar a la taula original}
    p := i;
    mentre p ≤ j fer
        t[p] := x[p]; p := p + 1;
    fmentre;
facció
```

8.6 Problemes

8.1 Dissenyar una variant de l'ordenació per selecció on a cada iteració de l'algorisme es seleccionin els elements mínim i màxim i es posin al principi i al final de la part no ordenada.

8.2 Donada una taula t de $[1..N]$ d'enters i un enter x, dissenyar un algorisme que la reorganitzi de tal manera que primer apareguin tots els elements més petits que x, despres tots els iguals a x i finalment tots els més grans. Les dues úniques operacions possibles sobre la taula són

la consulta i l'intercanvi. Intentar que funcioni en temps proporcional a N i no a N^2.

8.3 Donada una taula $v[1..N]$ d'enters, es pot extraure una seqüència creixent de la següent manera:

- L'element $v[1]$ és el primer element de la seqüència
- L'element de la seqüència que segueix a $v[i]$ és $v[j]$ tal que j és l'element més proper a `i`, amb $j > i$, i $v[j] \geq v[i]$.

Exemple: donada la taula

```
1 3 2 8 5 4 8 7 12 8
```

la seqüència creixent que s'extrauria seria

```
1 3 8 8 12
```

Dissenyar una acció que reordeni els elements d'una taula d'enters de tal manera que la seqüència creixent quedi emmagatzemada ordenadament a l'esquerra de la taula i la resta d'elements quedin a la dreta de la taula (no necessàriament ordenats).

8.4 Dissenyar l'acció *partició* de l'algorisme de *quick sort* (pàg. 64) amb el següent invariant:

$$\{\texttt{Inv}: \ i \leq k \leq q \leq j, \quad t[i..k] \leq t[i] \leq t[k+1..q]\}$$

8.5 Modificar l'algorisme d'ordenació per selecció (pàg. 61) de tal manera que la taula contingui una permutació del seu contingut original, amb la part esquerra ordenada amb només una instància de cada element i la part dreta, sense ordenar, amb els elements repetits. Per exemple:

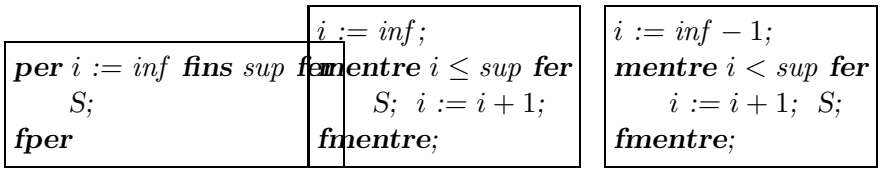
```
original:   8  3  5  8  9  5  6  5  7  2  7  1  2  3

ordenada:   1  2  3  5  6  7  8  9  5  3  7  5  8  2
            ----------------------  ----------------
                  ordenada              repetits
```

Capítol 9

Matrius

En aquest capítol es presenten algorismes bàsics d'àlgebra lineal que treballen principalment sobre matrius. Sovint, es farà servir la sentència "`per`" en les iteracions dels algorismes. Aquesta sentència és força apropiada per a fer recorreguts sobre els elements de vectors i matrius.

A efectes de raonar amb sentències iteratives de tipus "`per`", considerarem que els següents fragments tenen un comportament equivalent:



(a) (b) (c)

Per que això sigui així, hem de considerar que

- La sentència S no modifica la variable i.
- El valor de la variable i és desconegut en acabar el bucle i, per tant, no el podem fer servir en la postcondició.

Quan es doni el cas que $inf > sup$, no s'executarà cap iteració del bucle. Usualment, raonarem amb aquests bucles fent servir l'equivalència (b) amb el següent tipus d'invariant:

{`Inv:` *S'ha fet el tractament S per als elements* $inf..i-1$, $inf \leq i \leq sup+1$}
per $i := inf$ ***fins*** *sup* ***fer***
    S;
fper

Un avantatge dels bucles "`per`" és que l'acabament queda garantit per la pròpia semàntica i, per tant, no cal demostar-lo. La inicialització de l'índex del bucle també és quelcom que ve implícit amb la pròpia semàntica. El cos del bucle correspon normalment al tractament de l'element i, incloent les instruccions que fan que l'invariant es compleixi per un element més de l'estructura tractada.

9.1 Producte escalar

Dissenyar un programa que calculi el producte escalar de dos vectors, amb la següent especificació:

funció *prod_escalar* $(a, b$: ***taula*** *[1..N]* ***de*** *enter)* ***retorna*** *p:enter*
{`Pre:` $N > 0$}
{`Post:` $p = \sum_{k=1}^{N} a[k] \cdot b[k]$}

L'invariant es pot obtenir amb una generalització de la postcondició,

$$p = \sum_{k=1}^{i-1} a[k] \cdot b[k], \;\; 1 \leq i \leq N+1$$

donant lloc al següent algorisme.

funció *prod_escalar* $(a, b$: ***taula*** *[1..N]* ***de*** *enter)* ***retorna*** *p:enter*
{`Pre:` $N > 0$}
{`Post:` $p = \sum_{k=1}^{N} a[k] \cdot b[k]$}
    var *i: nat;*
    p := 0;
    {`Inv:` $p = \sum_{k=1}^{i-1} a[k] \cdot b[k], \; 1 \leq i \leq N+1$}
    per *i:=1* ***fins*** *N* ***fer***
        *p := p + a[i] * b[i];*
    fper*;*
    retorna *p;*
ffunció

9.2 Suma de matrius

La suma de matrius te ls següent especificació:

funció *suma* $(A, B$: ***taula*** *[1..N,1..M]* ***de*** *enter)* ***retorna*** C:***taula*** *[1..N,1..M]* ***de*** *enter*
{`Pre:` $N > 0,\ M > 0$}
{`Post:` $C = A + B$}

on l'operació "+" significa "suma de matrius". Com ja se sap d'àlgebra lineal, la suma de matrius es realitza element a element de tal manera que

$$C[i, j] = A[i, j] + B[i, j]$$

El recorregut dels elements de la matriu es pot fer de diverses maneres. Probablement, la més natural és aquella que fa un recorregut per files i, dins de cada fila, un recorregut per columnes[1].

D'aquesta manera es pot derivar un invariant que caracteritza el recorregut per files:

$$C[1..i-1, *] = A[1..i-1, *] + B[1..i-1, *],\ 1 \leq i \leq N+1$$

que ens diu que s'ha realitzat la suma per a tots els elements de les files $1..i-1$. El símbol '$*$' representa tots els elements d'aquella dimensió. En aquest cas en concret, '$*$' representa $1..M$. Dins del tractament de cada fila hi ha un tractament per columnes implementat per un altra bucle amb el següent invariant:

$$C[i, 1..j-1] = A[i, 1..j-1] + B[i, 1..j-1],\ 1 \leq j \leq M+1$$

que ens diu que s'han tractat les columnes $1..j-1$ dins la fila i. Això ens porta a un algorisme com el que segueix.

[1] El mateix tipus de recorregut es pot fer per columnes i files dins de cada columna.

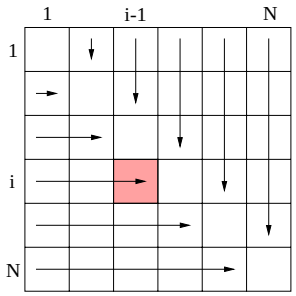


Figura 9.1: Recorregut de la matriu per a fer la transposició.

```
funció suma (A, B: taula [1..N,1..M] de enter) retorna C:taula [1..N,1..M] de enter
{Pre:  N > 0, M > 0}
{Post: C = A + B}
    var i, j: nat;
    {Inv:  C[1..i − 1, ∗] = A[1..i − 1, ∗] + B[1..i − 1, ∗], 1 ≤ i ≤ N + 1}
    per i:=1 fins N fer
        {Inv:  C[i, 1..j − 1] = A[i, 1..j − 1] + B[i, 1..j − 1], 1 ≤ j ≤ M + 1}
        per j:=1 fins M fer
            C[i, j] := A[i, j] + B[i, j];
        fper;
    fper;
    retorna C;
ffunció
```

9.3 Matriu transposta

L'especificació de l'algorisme és

```
acció transposar (e/s A: taula [1..N,1..N] de enter)
{Pre:  N > 0}
{Post: A_s = A_e^T}
```

on A^T representa la transposició de la matriu A. La transposició es realitza mitjançant l'intercanvi dels elements $A[i, j]$ amb els elements $A[j, i]$.

El recorregut dels elements es pot realitzar tal com mostra la Figura 9.1, utlitzant dos bucles per visitar cada una de les dimensions i tenint en compte que només els elements per sota la diagonal han de ser intercanviats amb els elements de sobre la diagona. Això dona lloc als invariants i algorisme que es presenten a continuació.

```
acció transposar (e/s A: taula [1..N,1..N] de enter)
{Pre:  N > 0}
{Post: A_s = A_e^T}
    var i, j: nat;
    {Inv:  S'han transposat les files 2..i-1 amb les columnes 2..i-1, 1 < i ≤ N+1}
    per i:=2 fins N fer
        {Inv:  S'han transposat els elements A[i, 1..j-1]
               amb els elements A[1..j-1, i], 1 < j ≤ i}
        per j:=1 fins i-1 fer
            intercanvi (A[i, j], A[j, i]);
        fper;
    fper;
facció
```

9.4 Matriu simètrica

Dissenyar una funció amb la següent especificació:

```
funció simètrica (A: taula [1..N,1..N] de enter) retorna sim: booleà
{Pre:  N > 0}
{Post: sim = "A és simètrica"}
```

Una matriu és simètrica si es compleix la següent propietat:

$$\forall i, j : \ 1 \leq i \leq N \ \wedge \ 1 \leq j \leq N \ : A[i, j] = A[j, i]$$

Tot i que el quantificador $\forall$ ens podria portar a pensar que s'ha de realitzar un recorregut per tots els elements de la matriu, l'algorisme que cal dissenyar es correspon millor a una estratègia de cerca: cal cercar un element $A[i, j]$ tal que $A[i, j] \neq A[j, i]$.

La cerca es pot desglosar de la següent manera:

- Cerca d'una fila que contingui algun element no simètric.
- Cerca, dins d'una fila, d'un element no simètric

Això dona lloc a una estructura amb dos bucles imbricats. El primer d'ells té l'invariant:

Les files $1..i-1$ són simètriques, $\neg fs \ \Rightarrow$ la fila i no és simètrica

on *fs* és una variable booleana que controla la cerca. De manera semblant, dins de cada fila i es pot derivar un invariant que caracteritza la cerca de la columna d'algun element no simètric:

Les columnes $1..j-1$ són simètriques, $\neg cs \Rightarrow$ la columna j no és simètrica

on *cs* és una variable booleana que controla la cerca dins de la fila.

El raonament associat a aquests invariants és molt semblant al de la cerca en taules (veure Secció 7.4, pàg. 48), i no es descriurà aquí. Cal notar que la condició d'acabament del bucle més imbricat conté la condició $j < i$. Això és degut a que, de manera semblant a la transposició de matrius (veure Figura 9.1), només cal comparar cada element per sota de la diagonal amb l'element corresponent de sobre la diagonal. L'algorisme resultant és el que segueix.

```
funció simètrica (A: taula [1..N,1..N] de enter) retorna sim: booleà
{Pre:  N > 0}
{Post: sim = "A és simètrica"}
    var i, j: nat; fs, cs: booleà;
    i := 2; fs := cert;
    {Inv:  Les files 1..i − 1 són simètriques, ¬fs ⇒ la fila i no és simètrica}
    mentre i ≤ N ∧ fs fer
        j := 1; cs := cert;
        {Inv:  Les columnes 1..j − 1 són simètriques, ¬cs ⇒ la columna j no és simètrica}
        mentre j < i ∧ cs fer
            si A[i, j] ≠ A[j, i] llavors cs := fals;
            sino j := j + 1;
            fsi;
        fmentre;
        si ¬cs llavors fs := fals;
        sino i := i + 1;
        fsi;
    fmentre;
    retorna fs;
ffunció
```

L'anterior algorisme es pot millorar tenint en compte dos fets:

- Les variables *fs* i *cs* tenen funcions semblants i poden ser compartides pels dos bucles.

- La funciona no necessita conèixer exactament on hi ha alguna asimetria, es a dir, els valors de i i j al final dels bucles és irrellevant.

Amb això, es pot arribar a un algorisme una mica més eficient, on els invariants dels bucles són una mica més difícils de especificar. Deixem al lector la tasca de definir aquests invariants. L'algorisme és el que segueix.

```
función simètrica (A: taula [1..N,1..N] de enter) retorna sim: booleà
{Pre:  N > 0}
{Post: sim = "A és simètrica"}
    var i, j: nat;
    i := 1; sim := cert;
    mentre i ≤ N ∧ sim fer
        j := 1;
        mentre j < i ∧ sim fer
            sim := A[i, j] = A[j, i];
            j := j + 1;
        fmentre;
        i := i + 1;
    fmentre;
    retorna sim;
ffunció
```

9.5 Cerca en matriu ordenada

Volem dissenyar un algorisme que faci la cerca d'un element en una matriu ordenada. L'ordenació ens ve donada per files i per columnes, és a dir, l'element $A[i+1, j]$ és més gran o igual que l'element $A[i, j]$ i l'element $A[i, j+1]$ és més gran o igual que l'element $A[i, j]$. L'especificació és la següent:

```
función cerca_ordenada (A: taula [1..N,1..M] de enter; x : enter)
                        retorna (trobat: booleà, i: nat, j:nat)
{Pre:  N > 0, M > 0,
       ∀i, j : 1 ≤ i < N, 1 ≤ j ≤ M : A[i, j] ≤ A[i + 1, j],
       ∀i, j : 1 ≤ i ≤ N, 1 ≤ j < M : A[i, j] ≤ A[i, j + 1]}
{Post: trobat ⇔ x ∈ A, trobat ⇒ A[i, j] = x}
```

El fet que la matriu estigui ordenada ens permet proposar un algorisme de complexitat proporcional a $N+M$, en lloc de la complexitat proporcional

a $N \cdot M$ que obtindriem amb una cerca convencional. L'invariant que ens permet fer una cerca eficient és el següent:

$$x \in A \Leftrightarrow x \in A[1..i, j..M],\ \mathit{trobat}\ \Rightarrow\ A[i,j] = x$$

Aquest invariant ens diu que la pertenença a la matriu es pot reduir a la pertenença a una submatriu. La següent figura il.lustra l'invariant per a una matriu concreta on s'esta fent la cerca de l'element $x = 8$.

		j			M	
1	4	5	7	10	12	1
2	5	8	9	10	13	
6	7	10	11	12	15	i
9	11	13	14	17	20	
11	12	19	20	21	23	
13	14	20	22	25	26	

El raonament amb aquest invariant és el següent:

- Condició d'acabament: quan es troba l'element (*trobat*) o bé quan no es pot trobar (la submatriu és buida: $i < 1\ \vee\ j > M$).
- Cos del bucle: cal comparar l'element a cercar amb l'element inferior esquerra ($A[i,j]$) de la submatriu. Poden passar tres coses:
  - $A[i,j] = x$. En aquest cas, ja hem trobat l'element.
  - $A[i,j] > x$. En aquest cas sabem que cap element de la fila i podrà ser igual a x. Per tant, podem reduir la submatriu en una fila.
  - $A[i,j] < x$. En aquest cas sabem que cap element de la columna j podrà ser igual a x. Per tant, podem reduir la submatriu en una columna.
- Inicialització. Al principi, la submatriu que pot contenir l'element és tota la matriu ($i = 1$, $j = M$). La variable *trobat* cal inicialitzar-la a fals per que l'invariant sigui vàlid.
- Demostració d'acabament. A cada iteració en la que no s'ha trobat l'element, s'elimina una fila o una columna de la submatriu. Això no pot executar-se infinitament sense que la submatriu acabi essent buida.

```
función cerca_ordenada (A: taula [1..N,1..M] de enter; x : enter)
                        retorna (trobat: booleà, i: nat, j:nat)
{Pre:  N > 0, M > 0,
       ∀i, j : 1 ≤ i < N, 1 ≤ j ≤ M : A[i, j] ≤ A[i + 1, j],
       ∀i, j : 1 ≤ i ≤ N, 1 ≤ j < M : A[i, j] ≤ A[i, j + 1]}
{Post: trobat ⇔ x ∈ A, trobat ⇒ A[i, j] = x}
    i := N; j := 1; trobat := fals;
    {Inv:  x ∈ A ⇔ x ∈ A[1..i, j..M], trobat ⇒ A[i, j] = x}
    mentre ¬trobat ∧ i ≥ 1 ∧ j ≤ M fer
        si A[i, j] < x llavors j := j + 1;
        sinosi A[i, j] > x llavors i := i − 1;
        sino trobat := cert;
        fsi;
    fmentre;
    retorna (trobat, i, j);
ffunció
```

Cal notar que la definició de la submatriu és important en aquest algorisme. El lector podrà comprovar que no totes les definicions de submatrius són apropiades per a fer l'algorisme eficient. Per exemple, si agafessim el següent invariant:

$$x \in A \Leftrightarrow x \in A[1..i, 1..j],\ trobat \ \Rightarrow\ A[i, j] = x$$

la cerca no es podria realitzar amb temps proporcional a $N + M$. Deixem que el lector trobi algun altre invariant que permeti fer la cerca eficient.

9.6 Multiplicació de matrius

Volem dissenyar una funció amb la següent especificació:

```
funció multiplicació (A: taula [1..N,1..M] de enter; B: taula [1..M,1..P] de enter)
                      retorna C:taula [1..N,1..P] de enter
{Pre:  N > 0, M > 0, P > 0}
{Post: C = A · B}
```

Per a calcular $C = A \cdot B$ cal realitzar tants productes escalars com elements té la matriu C. En aquest algorisme farem servir els indexos i, j i k per referenciar les dimensions $[1..N]$, $[1..M]$ i $[1..P]$, respectivament. El càlcul que cal fer per a cada element de C és el següent:

$$C[i,k] = \sum_{j=1}^{M} A[i,j] \cdot B[j,k]$$

L'algorisme proposat té tres bucles imbricats, cadascun d'ells associat a un dels indexos de la multiplicació. Els dos primers bucles fan un recorregut pels elements de la matriu C. Per a cada element es fa un producte escalar (veure Secció 9.1, pàg. 72). L'algorisme resultant és el que segueix.

```
funció multiplicació (A: taula [1..N,1..M] de enter; B: taula [1..M,1..P] de enter)
                        retorna C:taula [1..N,1..P] de enter
{Pre:  N > 0, M > 0, P > 0}
{Post: C = A · B}
    var i, j, k: nat;
    {Inv:  C[1..i − 1, ∗] = A[1..i − 1, ∗] · B, 1 ≤ i ≤ N + 1}
    per i:=1 fins N fer
        {Inv:  C[i, 1..k − 1] = A[i, ∗] · B[∗, 1..k − 1], 1 ≤ k ≤ P + 1}
        per k:=1 fins P fer
            var prod: enter;
            {Calculem el producte escalar A[i, ∗] · B[∗, k]}
            prod := 0;
            {Inv:  prod = A[i, 1..j − 1] · B[1..j − 1, k], 1 ≤ k ≤ M + 1}
            per j := 1 fins M fer
                prod := prod + A[i, j] ∗ B[j, k];
            fper;
            C[i, k] := prod;
        fper;
    fper;
    retorna C;
ffunció
```

9.7 Problemes

9.1 Dissenyar un algorisme que, donada una matriu $N \times N$ d'enters, determini si tots els seus elements són diferents de zero.

9.2 Escriure un algorisme que determini, per a una matriu a amb n files i m columnes, el nombre de components negatives.

9.3 Escriure tres algorismes que, donada una matriu quadrada a d'ordre $N \times N$ comprovin si és:

1. simètrica ($a_{ij} = a_{ji}$ per a tot i, j);
2. triangular ($a_{ij} = 0$ per a tot $i < j$);
3. diagonal ($a_{ij} = 0$ per a tot $i \neq j$).

Dissenyar-ne versions amb dues iteracions i amb una sola iteració.

9.4 Dissenyar una funció que comprovi que una matriu quadrada d'ordre N, representada en una taula t, és un quadrat màgic. Una matriu és un quadrat màgic si es donen les tres condicions següents:

- La suma S dels elements d'una fila és idèntica per a totes les files.
- La suma dels elements de qualsevol columna també és S.
- La suma dels elements de les dues diagonals principals és S.

La funció ha de retornar cert si t és un quadrat màgic, i fals en cas contrari.

Capítol 10

Seqüències

En aquest capítol es presenta un model abstracte per a tractar seqüències i algorismes bàsics sobre seqüències.

10.1 Operacions amb seqüències

A continuació s'especifiquen les operacions que es poden realitzar sobre les seqüències. Per a representar l'estat d'una seqüència utilitzarem la següent nomenclatura:

$$\underbrace{s_1 \ \cdots \ s_{k-1}}_{pe(s)} \bullet \underbrace{s_k \ \cdots \ s_n}_{pd(s)}$$

on $pe(s)$ representarà la part tractada de la seqüència (part esquerra) i $pd(s)$ la part per tractar (part dreta). El primer element de $pd(s)$ serà al que s'accedirà cada vegada que es faci una lectura. S'arribarà a final de seqüència quan $pd(s) = \emptyset$.

Sovint necessitarem raonar sobre l'acabament d'un algorisme en termes de la longitud de la seqüència o d'una part de la seqüència. Es farà servir la nomenclatura $|s|$, $|pe(s)|$ i $|pd(s)|$ per indicar longituds d'aquestes parts.

L'execució d'una operació en una seqüència quan es troba en un estat que no compleix la precondició de l'operació suposarà un error d'execució. Alguns exemples de errors serien els següents:

- Cridar a *actual*(s) o *avançar*(s) quan es compleix *fi_seq*(s).
- Fer una operació de lectura quan s no està oberta per a lectura.
- Tancar una seqüència quan no està oberta.

- Obrir una seqüència quan encara està oberta.

És important destacar que en una seqüència només es poden realitzar operacions de lectura o d'escriptura, però no d'ambdues a la vegada. El tipus d'operacions a realitzar dependrà de com s'hagi obert la seqüència (*obrir_lectura* o *obrir_escriptura*). L'operació de *tancar* una seqüència és comuna a tots dos modes d'accés. L'especificació de totes les operacions sobre seqüències es presenta a continuació, on es fa servir el tipus genèric T per indicar el tipus de dades dels elements de la seqüència.

OPERACIONS PER A LECTURA

acció *obrir_lectura (**e/s** s: **seq de** T)*
{`Pre:` *s està tancada*}
{`Post:` *s està oberta per a lectura,* $pe(s) = \emptyset$}

funció *fi_seq (s: **seq de** T)* ***retorna*** *f: booleà*
{`Pre:` *s està oberta per a lectura*}
{`Post:` $f = (pd(s) = \emptyset)$}

funció *actual (s: **seq de** T)* ***retorna*** *x: T*
{`Pre:` *s està oberta per a lectura,* $\neg$*fi_seq(s),* $s = pe(s) \bullet s_k \ \cdots$}
{`Post:` $x = s_k$}

acció *avançar (**e/s** s: **seq de** T)*
{`Pre:` *s està oberta per a lectura,* $\neg$*fi_seq(s),* $s = s_1 \cdots s_{k-1} \bullet s_k s_{k+1} \ \cdots$}
{`Post:` *s està oberta per a lectura,* $s = s_1 \cdots s_{k-1} s_k \bullet s_{k+1} \ \cdots$}

OPERACIONS PER A ESCRIPTURA

acció *obrir_escriptura (**e/s** s: **seq de** T)*
{`Pre:` *s està tancada*}
{`Post:` *s està oberta per escriptura,* $s = \bullet$}

acció *escriure (**e/s** s: **seq de** T; **ent** x: T)*
{`Pre:` *s està oberta per escriptura,* $s = s_1 \cdots s_k \bullet$}
{`Post:` *s està oberta per a escriptura,* $s = s_1 \cdots s_k \ x \bullet$}

OPERACIÓ PER A TANCAR

acció *tancar (**e/s** s: **seq de** T)*
{`Pre:` *s està oberta*}
{`Post:` *s està tancada*}

10.2 Comptar *a*s en una frase

Dissenyar una funció que compti el nombre de *'a'*s que hi ha en una seqüència de caràcters, amb la següent especificació:

```
funció comptar_as (s: seq de caràcter) retorna n: nat
{Pre:  s està tancada}
{Post: n = "nombre de as que hi ha a s"}
```

El raonament per a un disseny iteratiu podria ser com aquest:

- Invariant: n conté el nombre de *'a'*s que hi ha a $pe(s)$.

$$\underbrace{s_1 \ \cdots \ s_{k-1}}_{n = \#\ a\text{s}} \bullet \underbrace{s_k \ \cdots \ s_n}_{pd(s)}$$

- Condició d'acabament: *fi_seq(s)*. Això farà que al final n tingui el nombre de as de tota la seqüència.
- Cos del bucle: cal examinar l'element actual de s. En el cas que sigui una *'a'*, cal incrementar n. En qualsevol cas, cal avançar per mantenir l'invariant.
- Inicialització: cal obrir la seqüència per lectura ($pe(s)$ serà buida) i definir $n = 0$ (no hi ha cap *'a'* a $pe(s)$).
- Demostració d'acabament: a cada iteració s'avança i, per tant, $|pd(s)|$ decreix. En algun moment arribarà a ser zero i *fi_seq(s)* serà cert.

```
funció comptar_as (s: seq de caràcter) retorna n: nat
{Pre:  s està tancada}
{Post: n = "nombre de as que hi ha a s"}
    obrir_lectura (s);
    n := 0;
    {Inv:  n = "nombre de as que hi ha a pe(s)"}
    mentre ¬fi_seq (s) fer
        si actual (s) = 'a' llavors n := n + 1; fsi;
        avançar (s);
    fmentre;
    tancar (s);
    retorna n;
ffunció
```

10.3 Comptar paraules en una frase

S'ha de dissenyar una funció que compti el nombre de paraules d'una seqüència de caràcters, amb la següent especificació:

> ***funció*** *comptar_paraules (s:* ***seq de*** *caràcter)* ***retorna*** *n: nat*
> {`Pre:` *s està tancada*}
> {`Post:` *n = "nombre de paraules que hi ha a s"*}

Considerarem que una paraula és qualsevol fragment de la seqüència que no conté cap blanc (que representarem per ␣) i que està precedida i seguida d'algun ␣. Eventualment, la primera paraula pot no estar precedida de ␣, i la darrera paraula pot no estar seguida de ␣.

Així, el següent texte té cinc paraules:

La␣taula␣␣␣és␣␣de␣fusta

igual que el següent texte:

␣␣La␣taula␣␣␣és␣␣de␣fusta␣␣␣␣

El problema a resoldre es fa més senzill si observem propietat que hi han tantes paraules com començaments de paraula. I un començament de paraula el podem detectar quan trobem dos caraàcters adjacents on el primer és ␣ i el segon és diferent de ␣. Només hi ha una excepció en aquesta propietat: la primera paraula pot no estar precedida per un ␣. Això ho tindrem en compte en la inicialització de l'algorisme.

- Invariant: n conté el nombre de començaments de paraula trobats a *pe*(s), i c conté el darrer caràcter de *pe*(s), que correspon al caràcter llegit a la darrera iteració. Al principi de seqüència, $c =$␣.
- Condició d'acabament: *fi_seq(s)*.
- Cos del bucle: Si tenim $c =$␣ i *actual*$(s) \neq$␣, llavors hem detectat un nou començament de paraula i cal incrementar n. A més, cal actualitzar c amb el darrer caràcter llegit i avançar. Tot això fa que l'invariant es compleixi per a un element més de la seqüència.
- Inicialització: cal obrir s per a lectura. Com que *pe*(s) és buida, cal fer $n = 0$. A més, posarem $c =$␣ per contemplar el cas en que el primer caràcter de la seqüència no sigui ␣ i així poder detectar l'inici de la primera paraula. Això ho podem fer donat que l'afegiment de caràcters ␣ a l'inici del texte no canvia el seu nombre de paraules.

- Demostració d'acabament: a cada iteració s'avança i es decrementa $|pd(s)|$. En algun moment es farà zero i *fi_seq(s)* serà zero.

```
funció comptar_paraules (s: seq de caràcter) retorna n: nat
{Pre:  s està tancada}
{Post: n = "nombre de paraules que hi ha a s"}
    var c: caràcter;
    obrir_lectura (s);
    n := 0; c := '␣';
    {Inv:  n = "nombre d'inicis de paraula que hi ha a pe(s)",
           |pe(s)| > 0  ⇒  c = "darrer caràcter de pe(s)",
           |pe(s)| = 0  ⇒  c = '␣'}
    mentre ¬fi_seq (s) fer
        si c = '␣' ∧ actual (s) ≠ '␣' llavors n := n + 1; fsi;
        c := actual (s);
        avançar (s);
    fmentre;
    tancar (s);
    retorna n;
ffunció
```

10.4 Mitjana dels elements d'una seqüència

Volem dissenyar una funció que calculi la mitjana dels elements d'una seqüència s d'enters. Quan s no és buida, la mitjana es pot calcular com

$$m = 1/n \cdot \sum_{i=1}^{n} s_i$$

on $n = |s|$. Quan $|s| = 0$, llavors definirem la mitjana com zero. L'especificació de la funció és la següent:

```
funció mitjana (s: seq de real) retorna m: real
{Pre:  s està tancada}
{Post: |s| > 0  ⇒  m = 1/|s| · Σ_{i=1}^{|s|} s_i,
       |s| = 0  ⇒  m = 0}
```

Per a calcular la mitjana n'hi ha prou en saber el nombre d'elements de s i la seva suma. Això és el que caracteritzarà l'invariant del bucle.

- Invariant: $n = |pe(s)|$, $sum = \sum_{i=1}^{|pe(s)|} s_i$.

- Condició d'acabament: *fi_seq(s)*.
- Cos del bucle: cal incrementar n i sumar l'element actual a *sum* per mantenir l'invariant per una posició més de la seqüència.
- Inicialització: per a fer valer l'invariant al principi cal obrir s per a lectura, $n = 0$ i $sum = 0$ (donat que $|pe(s)| = 0$).
- Demostració d'acabament: a cada iteració s'avança una posició i $|pd(s)|$ decreix. Tard o d'hora es farà zero i *fi_seq(s)* serà cert.

```
funció mitjana (s: seq de real) retorna m: real
{Pre:  s està tancada}
{Post: |s| > 0 ⇒ m = 1/|s| · Σ_{i=1}^{|s|} s_i,
       |s| = 0 ⇒ m = 0}
    var n: nat; sum: real;
    obrir_lectura (s);
    n := 0; sum := 0;
    {Inv:  n = |pe(s)|, sum = Σ_{i=1}^{|pe(s)|} s_i}
    mentre ¬fi_seq (s) fer
        sum := sum + actual (s);
        n := n + 1;
        avançar (s);
    fmentre;
    tancar (s);
    {n = |s|, sum = Σ_{i=1}^{|s|} s_i}
    si n ≠ 0 llavors m := sum/n; sino m := 0; fsi;
    retorna m;
ffunció
```

10.5 Cerca d'un element

Dissenyar una acció que cerqui un element en una seqüència s i s'aturi quan el trobi. En el cas que no el trobi, l'acció ha d'arribar al final de la seqüència. L'acció ha de deixar la seqüència oberta per a lectura i, en el cas que l'element hi sigui, *actual(s)* ha de tenir el valor de l'element cercat. L'especificació de l'acció és la següent:

```
acció cerca (e/s s: seq de T; ent x: T)
{Pre:  s està tancada}
{Post: s està oberta, x ∈ s ⇒ actual(s) = x, x ∉ s ⇒ fi_seq(s)}
```

L'estratègia seguida per l'algorisme presentat és molt semblant a la de la cerca iterativa en una taula (veure Secció 7.4, pàg. 48). L'invariant que s'utilitza en aquest cas és el següent:

$$x \notin pe(s), \quad trobat \;\Rightarrow\; (actual(s) = x)$$

i deixem que el lector dedueixi la resta del raonament que ens porta a la solució iterativa.

```
acció cerca (e/s s: seq de T; ent x: T)
{Pre:  s està tancada}
{Post: s està oberta, x ∈ s ⇒ actual(s) = x, x ∉ s ⇒ fi_seq(s)}
    var trobat: booleà;
    obrir_lectura (s); trobat := fals;
    {Inv:  x ∉ pe(s), trobat ⇒ (actual(s) = x)}
    mentre ¬fi_seq(s) ∧ ¬trobat fer
        si actual (s) = x llavors trobat := cert;
        sino avançar (s);
        fsi;
    fmentre;
facció
```

10.6 Fusió de seqüències ordenades

L'algorisme presentat en aquesta secció realitza la fusió de dues seqüències ordenades i genera una altra seqüència ordenada que conté els elements de les dues anteriors. L'especificació és com segueix:

```
funció fusió (s1, s2: seq de enter) retorna s3: seq de enter
{Pre:  s1, s2 estan tancades i ordenades creixentment}
{Post: s3 = s1 ∪ s2, s3 està tancada i ordenada creixentment}
```

on, en un abús de notació, fem servir el símbol ∪ per indicar "fusió".

L'estratègia per a resoldre aquest problema és molt semblant a la utilitzada per fer la fusió de dues taules ordenades (veure Secció 8.4, pàg. 64) i, per aquesta raó, no presentarem les justificacions de l'algorisme iteratiu al voltant de l'invariant. Com en el cas de la fusió de taules, l'algorisme té dues fases:

1. Fusió dels elements de *s1* i *s2* fins exhaurir una de les dues seqüències.

2. Còpia dels elements de l'altra seqüència.

L'invariant utilitzat a la primera fase és el següent:

$$s3 = pe(s1) \cup pe(s2),\ s3 \text{ està ordenada creixentment},\ s3 \leq pd(s1),\ s3 \leq pd(s2)$$

on la notació $s3 \leq pd(s1)$ s'utilitza per indicar que qualsevol element de $s3$ és més petit o igual que qualsevol element de $pd(s1)$.

```
funció fusió (s1, s2: seq de enter) retorna s3: seq de enter
{Pre:  s1, s2 estan tancades i ordenades creixentment}
{Post: s3 = s1 ∪ s2, s3 està tancada i ordenada creixentment}
    obrir_lectura (s1); obrir_lectura (s2); obrir_escriptura (s3);
    {Inv:  s3 = pe(s1) ∪ pe(s2), s3 està ordenada creixentment, s3 ≤ pd(s1), s3 ≤ pd(s2)}
    mentre ¬fi_seq(s1) ∧ ¬fi_seq(s2) fer
        si actual (s1) ≤ actual (s2) llavors
            escriure (s3, actual (s1)); avançar (s1);
        sino
            escriure (s3, actual (s2)); avançar (s2);
        fsi;
    fmentre;

    {Inv ∧ (fi_seq(s1) ∨ fi_seq(s2))}
    mentre ¬fi_seq(s1) fer
        escriure (s3, actual (s1)); avançar (s1);
    fmentre;

    {Inv ∧ fi_seq(s1)}
    mentre ¬fi_seq(s2) fer
        escriure (s3, actual (s2)); avançar (s2);
    fmentre;

    {Inv ∧ fi_seq(s1) ∧ fi_seq(s2)}
    tancar (s1); tancar (s2); tancar (s3);
    retorna s3;
ffunció
```

10.7 Problemes

10.1 Escriure un algorisme que, donada una seqüència d'enters acabada en 0, decideixi si un valor donat pertany a la seqüència o no. Repetir el

problema suposant que la seqüència és creixent.

10.2 Dissenyar un algorisme que, donada una seqüència d'enters acabada en 0, determini si està formada només per valors positius.

10.3 Dissenyar un algorisme que, donada una seqüència d'enters positius, comprovi si la sèrie és creixent.

10.4 Donada una frase acabada en punt, comptar el nombre de caràcters no blancs que hi apareixen, sense comptar el punt finalitzador.

10.5 Dissenyar un algorisme que compti el nombre de lletres de la primera paraula d'un text.

10.6 Dissenyar un algorisme que, donada una frase acabada en punt, determini si té més lletres 'b' que 'c'.

10.7 Donada una frase acabada en punt, determinar quina és la primera vocal que hi apareix.

10.8 Donada una seqüència d'enters acabada en 0, calcular-ne la mitjana aritmètica.

10.9 Dissenyar un algorisme que, donada una seqüència d'enters z_0, z_1, z_2, ... escrigui la seqüència de sumes parcials z_0, $z_0 + z_1$, $z_0 + z_1 + z_2$, ...

10.10 Donada una seqüència no buida d'enters, trobar-ne el menor.

10.11 Donat un text, determinar si cada vocal hi apareix un cop com a mínim.

10.12 Donat un text, determinar quina és la vocal que més cops hi apareix.

10.13 Donada una sèrie d'enters amb com a mínim tres elements, determinar si formen una progressió aritmètica o no. Recordar que una progressió aritmètica és una sèrie de la forma b, $a + b$, $2a + b$, $3a + b$, $4a + b$, ... amb alguns valors d'a i b.

10.14 Escriure un algorisme que faci la còpia d'un text suprimint tots els espais en blanc.

10.15 Donada una frase acabada en punt, comptar el nombre de caràcters que apareixen a partir de la primera 'a', sense comptar-ne el punt final.

10.16 Dissenyar un algorisme que copiï un text sense els blancs del començament.

10.17 Suposem que F i G són dues seqüències estrictament creixents de nombres enters. Dissenyar un algorisme que calculi el nombre de valors que apareixen en totes dues seqüències.

10.18 Donades F i G seqüències creixents de nombres enters, calcular la distància entre totes dues. Definim la distància entre les seqüències com

$$\text{dist}(F, G) = \min\{|f - g| \;\; | f \in F, g \in G\}$$

10.19 Donades F i G seqüències creixents de nombres enters, dissenyar un algorisme que determini si existeixen $f \in F$ i $g \in G$ tals que $|f-g| < 9$.

10.20 Dissenyar un algorisme que, donades tres seqüències creixents d'enters, F, G i H, que com a mínim tenen un element comú, calculi el mínim element comú.

10.21 Dissenyar un algorisme que, donades dues seqüències ordenades d'enters, n'obtingui una altra, també ordenada, amb els elements comuns de les dues.

10.22 Tenim una seqüència d'enters c_0, c_1, c_2, ..., c_n que conté els coeficients del polinomi $P(x) = c_0 + c_1 x + \ldots + c_n x^n$. Dissenyar un algorisme que calculi eficientment $P(m)$ a partir de la seqüència i un enter m donat. L'algorisme no pot fer servir l'operació d'exponenciació. Repetir el problema suposant que la seqüència que ens donen és c_n, c_{n-1}, ..., c_1, c_0.

Capítol 11

Algorismes numèrics

Aquest capítol presenta alguns algorismes que tracten dades numèriques de tipus real. En aquest casos és important tenir en compte la precisió amb la que es pot treballar. Els problemes proposats són iteratius i el seu acabament depèn de la precisió amb la que es vol calcular la solució final.

En els problemes que es plantegen, es farà servir una constant ε que controlarà el grau de precisió de les solucions calculades. El valor d'aquesta constant sol ser petit, per exemple $\varepsilon = 10^{-10}$.

11.1 Exponencial

Dissenyar una funció que calculi un valor aproximat per e^x, essent $0 \leq x < 1$. L'especificació de la funció és la següent:

funció *exp (x: real)* ***retorna*** *s: real*
{`Pre:` $0 \leq x < 1$}
{`Post:` $s \approx e^x$}

Se sap que e^x es pot calcular mitjançant la suma dels elements d'una sèrie de Taylor:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots$$

Quan $0 \leq x < 1$, els termes d'aquesta sèrie són cada vegada més petits i la suma convergeix cap a e^x. Si en lloc de agafar infinits termes de la sèrie, se n'agafen només uns quants, es comet un error. Aquest error és menor com més termes de la sèrie s'agafin.

Anomenem t_i a l'i-èssim terme d'aquesta suma i s_i a la suma dels i primers termes de la suma. Considerarem que la solució obtinguda és prou aproximada quan

$$|t_i| < \varepsilon |s_i|.$$

El càlcul de t_i no és computacionalment senzill: cal calcular x^i i $i!$. Tot i això, es pot observar que aquest càlcul es pot fer incrementalment a partir de t_{i-1} de la següent manera:

$$t_i = t_{i-1} \cdot \frac{x}{i}$$

Tot això ens porta a un algorisme iteratiu que es justifica tal com segueix:

- Invariant:
$$s = \sum_{k=0}^{i} \frac{x^k}{k!}, \quad t = \frac{x^i}{i!}$$
- Condició d'acabament: quan el terme sumat ja sigui prou petit ($t < \varepsilon s$).
- Cos del bucle: cal calcular un nou terme i sumar-lo. Per això, cal incrementar i, calcular el nou terme en funció de l'anterior i afegir-lo a s.
- Inicialització: podem inialitzar la suma amb el terme zero ($i = 0,\ t = 1,\ s = 1$). D'aquesta manera es compleix l'invariant.
- Demostració d'acabament: la demostració es basa en la covergència d'aquests tipus de sèries que garanteix que, tard o d'hora, hi haurà un terme t_i més petit que εs_i.

```
función exp (x: real) retorna s: real
{Pre:  0 ≤ x < 1}
{Post: s ≈ e^x}
    var i: nat; t: real;
    i := 0; t := 1; s := 1;

    {Inv:  s = Σ_{k=0}^{i} x^i/i!, t = x^i/i!}

    mentre t ≥ ε * s fer
        i := i + 1;
        t := t * x/i;
        s := s + t;
    fmentre;
    retorna s;
ffunció
```

11.2 Cosinus

Dissenyar una funció que calculi un valor aproximat per $\cos x$, essent $0 \leq x < 1$. L'especificació de la funció és la següent:

```
función cos (x: real) retorna s: real
{Pre:  0 ≤ x < 1}
{Post: s ≈ cos x}
```

Se sap que *cosx* es pot calcular mitjançant la suma dels elements d'una sèrie de Taylor:

$$\cos x = \sum_{j=0}^{\infty}(-1)^j \frac{x^{2j}}{(2j)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + \frac{x^{2j}}{(2j)!} + \cdots$$

El comportament d'aquesta sèrie és semblant a la presentada a la secció anterior per a calcular e^x. Els termes també es poden calcular incrementalment amb la següent recurrència:

$$t_i = t_{i-1} \cdot \frac{x^2}{i(i-1)}$$

on $i = 2j$. D'aquí es pot dissenyar un algorisme basat en el següent invariant:

$$s = \sum_{k=0}^{j}(-1)^k \frac{x^i}{i!}, \quad t = (-1)^j \frac{x^i}{i!}$$

El raonament associat a aquest invariant és semblant al del problema anterior i es deixa com a exercici pel lector.

funció *cos (x: real)* ***retorna*** *s: real*
{`Pre:` $0 \leq x < 1$}
{`Post:` $s \approx \cos x$}
    var *i: nat; t: real;*
    i := 0; t := 1; s := 1;

    {`Inv:` $s = \sum_{k=0}^{j}(-1)^j \frac{x^i}{i!},\ t = (-1)^j \frac{x^i}{i!},\quad on\ i = 2j$}

    mentre $|t| \geq \varepsilon * s$ ***fer***
        $i := i + 2;$
        $t := -t * x * x/(i * (i-1));$
        $s := s + t;$
    fmentre*;*
    retorna *s;*
ffunció

11.3 Zero d'una funció

Dissenyar un algorisme que aproximi el zero d'una funció $f(x)$, contínua a l'interval $[a, b]$, i sabent que els signes de $f(a)$ i $f(b)$ són diferents. L'especificació és la següent:

funció *zero* $(a, b$*: real)* ***retorna*** *z: real*
{`Pre:` $f(a) \cdot f(b) < 0$, *f és contínua entre a i b*}
{`Post:` $|f(z)| < \varepsilon$}

El disseny d'algorisme segueix una estratègia semblant a la de la cerca binària. Iterativament anirà reduint l'amplada de l'interval a la meitat depenent del signe de la funció en el punt mig de l'interval, tal com es mostra en la Figura 11.1.

El raonament de l'algorisme és el següent:

- Invariant: els signes de la funció a a i b són diferents i z es troba entre a i b.
$$f(a) \cdot f(b) < 0,\ \ (a < z < b)\ \vee\ (b < z < a)$$
Cal observar que l'invariant no suposa que $a \leq b$.

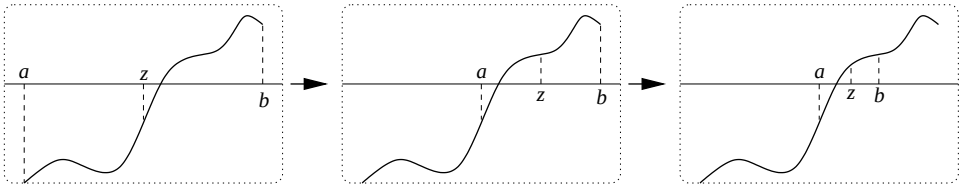


Figura 11.1: Càlcul iteratiu del zero d'una funció contínua.

- Condició d'acabament: $f(z) < \epsilon$.
- Cos del bucle: es calcula el valor de la funció al mig de l'interval (z) i es redueix l'interal per la meitat segons els signe de $f(z)$ per mantenir l'invariant.
- Inicialització: al principi a i b ténen els valors d'entrada de la funció i z és la mitjana dels dos.
- Demostració d'acabament: a cada iteració es redueix l'interval a la meitat. Arribarà un moment en que l'interval serà prou petit per tenir $f(z) < \varepsilon$.

```
funció zero (a, b: real) retorna z: real
{Pre:  f(a) · f(b) < 0, f és contínua entre a i b}
{Post: |f(z)| < ε}
    z := (a + b)/2;
    {Inv:  f(a) · f(b) < 0, (a < z < b) ∨ (b < z < a)}
    mentre |f(z)| ≥ ε fer
        si f(a) * f(z) > 0 llavors a := z; sino b := z; fsi;
        z := (a + b)/2;
    fmentre;
    retorna z;
ffunció
```

En el cas que el càlcul de $f(x)$ fos complexe, es podria proposar un algorisme que minimitzes les crides a la funció f. Suggerim que el lector proposi aquesta solució.

11.4 Problemes

11.1 Dissenyar una funció que calculi un valor aproximat per a la sèrie següent, per a valors de x a l'interval $[0, 1]$:

$$\int_0^x e^{-u^2} du = x - \frac{x^3}{3 \cdot 1!} + \frac{x^5}{5 \cdot 2!} - \frac{x^7}{7 \cdot 3!} + \cdots$$

11.2 El mètode de Newton-Raphson. Aquest mètode permet aproximar el zero d'una funció de manera més ràpida que una cerca binària. Donat un valor inicial x_0, genera una seqüència x_1, x_2, x_3, ... que cada vegada s'acosta més al zero. Per generar x_{i+1} cal conèixer la funció $f(x_i)$ i la seva derivada $f'(x_i)$ i aplicar la següent recurrència:

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

Es demana utilitzar el mètode de Newton per calcular $\sqrt{a}$. Per això cal trobar el zero de la funció $f(x) = x^2 - a$, que té per derivada $f'(x) = 2x$.

Dissenyar la funció que calculi $\sqrt{a}$ i comparar la seva convergència respecte a la cerca binària presentada a la secció 11.3.

11.3 El mètode de Newton-Raphson (veure problema anterior) també permet dissenyar un algorisme per a fer divisions sense utilitzar l'operació de divisió. Per a calcular $q = z/d$ es pot calcular $1/d$ i després multiplicar per z. Per a calcular $1/d$ podem trobar el zero de la funció

$$f(x) = \frac{1}{x} - d$$

que té el zero a $x = 1/d$. La derivada de la funció és

$$f'(x) = -\frac{1}{x^2}$$

Dissenyar una funció que calculi z/d utilitzant el mètode de Newton-Raphson i estudiar la seva convergència[1].

[1]Per a evitar problemes de convergència, suposar que $d \in [1/2, 1)$ i prendre $x_0 = 1.5$. Aquestes són les condicions que s'utilitzen per a realitzar divisions amb nombres reals normalitzats segons l'estàndard de IEEE.

Capítol 12

Generació de seqüències d'elements

En aquest capítol es presenten alguns algorismes de generació de seqüències d'elements, per exemple, la generació de totes les permutacions de n elements, o de totes les seqüencies creixents de nombres naturals que sumen n.

La manera més intuitiva de resoldre aquests problemes és mitjançant el raonament recursiu. En tots aquest algorismes s'utilitzarà una taula on s'aniran emmagatzemant les seqüències generades. Tot i que aquesta taula es pot passar com un paràmetre d'entrada a cada crida recursiva, utilitzarem el pas de paràmetres d'entrada/sortida com a mecanisme de transmissió d'informació.

El pas de paràmetres d'entrada és normalment implementat mitjançant la còpia completa del paràmetre. En el cas dels algorismes d'aquest capítol, el paràmetre transmés és una taula de n elements i la seva còpia podria representar una penalització important en el rendiment de l'algorisme. Per una altre cantó, el pas com a paràmetre d'entrada/sortida fa que els canvis realitzats en una acció siguin visibles a l'acció que l'ha cridat. Caldrà doncs anar en compte en caracteritzar aquests possibles canvis en l'especificació dels algorismes.

Els algorismes que seran presentats combinen la recursivitat amb la iteració. Per no ofegar la descripció de l'algorisme amb la seva formalització, no descriurem el raonament corresponent a les parts iteratives. Ens limitarem a mostrat els invariants que modelen el seu comportament. A més, el raonament recursiu de tots els algorismes és molt semblant. Per aquesta raó només analitzarem a una mica més de detall el primer algorisme i

discutirem les diferències principals en la resta d'algorismes.

Nomenclatura utilitzada

Per fer més senzilla l'especificació dels algorismes farem servir alguna nomenclatura apropiada. Per exemple, el predicat

$$t[1..l] \in \{1..k\}^l$$

ens indicarà que els elements de l'interval $t[1..l]$ contenen valors del conjunt $\{1..k\}$. El predicat

$$\mathit{creixent}(t[1..l])$$

indicarà que l'interval és estrictament creixent ($t[1] < t[2] < \cdots < t[l]$). El predicat

$$\mathit{diferents}(t[1..l])$$

indicarà que no hi ha cap parella d'elements a $t[1..l]$ que siguin iguals. Cal adonar-se que

$$\mathit{creixent}(t[1..l]) \;\Rightarrow\; \mathit{diferents}(t[1..l])$$

Finalment, a tots els algorismes caldrà escriure de tant en tant les seqüències generades i emmagatzemades a una taula. Per fer això suposarem que disposem d'una acció que ens escriu tot el contingut de la taula:

$$\mathit{escriure\_taula}(t)$$

12.1 Seqüències d'elements de $\{1..k\}$

L'especificació de l'algorisme és la següent:

acció *genseq1k (**ent** n, k: nat)*
{`Pre:` $n > 0,\ k > 0$}
{`Post:` *Ha escrit totes les seqüències de longitud n que existeixen agafant elements del conjunt* $\{1..k\}$}

Per exemple, si tinguessim $n = 2$ i $k = 3$, les seqüències generades serien les següents (no necessàriament en aquest ordre):

```
(1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3)
```

Per a resoldre el problema farem inducció amb la llargària de la seqüència i l'anirem construint a partir d'un prefix ja construit anteriorment. Per això dissenyarem la següent acció:

acció *genseq1k_rec (**ent** n, k: nat; **e/s** t: **taula**[1..n] **de** nat; **ent** l: nat)*
{`Pre:` $n > 0,\ k > 0,\ 0 \leq l \leq n,\ t[1..l] \in \{1..k\}^l$}
{`Post:` *Ha escrit totes les seqüències de longitud n que existeixen agafant elements del conjunt $\{1..k\}$ i que comencen pel prefix $t[1..l]$*}

I ara fem un raonament recursiu per dissenyar *genseq1k_rec*, fent inducció amb la llargària del prefix.

- Cas senzill ($l = n$): ja hem generat una seqüència completa i la podem escriure.
- Cas recursiu ($l < n$): cal definir una posició més del prefix posant tots els possibles valors del conjunt $\{1..k\}$ i generant les seqüències pel sufix que resta per definir.
- Demostració d'acabament: a cada crida recursiva incrementem la longitud del prefix. Això garanteix que sempre s'arribarà al cas senzill ($l = n$).

```
acció genseq1k_rec (ent n, k: nat; e/s t: taula[1..n] de nat; ent l: nat)
{Pre:  n > 0, k > 0, 0 ≤ l ≤ n, t[1..l] ∈ {1..k}^l}
{Post: Ha escrit totes les seqüències de longitud n que existeixen
       agafant elements del conjunt {1..k} i que comencen pel prefix t[1..l]}
    si l = n llavors escriure_taula(t);
    sino
        var i: nat;
        {Inv:  Ha escrit totes les seqüències amb prefix t[1..l]
               i amb els valors {1..i − 1} a la posició l + 1}
        per i := 1 fins k fer
            t[l + 1] := i; genseq1k_rec (n,k,t,l + 1);
        fper;
    fsi;
facció
```

Finalment cal dissenyar l'acció principal amb la que s'ha de començar la construcció de les seqüències. Cal declarar una taula on emmagatzemar les seqüències i fer la crida inicial amb un prefix de longitud zero.

```
acció genseq1k (ent n, k: nat)
{Pre:  n > 0, k > 0}
{Post: Ha escrit totes les seqüències de longitud n que existeixen
       agafant elements del conjunt {1..k}}
    var S: taula[1..n] de nat;
    genseq1k_rec (n, k, S, 0);
facció
```

12.2 Seqüències creixents d'elements de $\{1..k\}$

L'especificació de l'algorisme és la següent:

```
acció genseq1k_creix (ent n, k: nat)
{Pre:  n > 0, k > 0}
{Post: Ha escrit totes les seqüències creixents de longitud n
       d'elements agafats del conjunt {1..k}}
```

Per exemple, si tinguessim $n = 3$ i $k = 5$, les seqüències generades serien les següents (no necessàriament en aquest ordre):

```
(1 2 3) (1 2 4) (1 2 5) (1 3 4) (1 3 5)
(1 4 5) (2 3 4) (2 3 5) (2 4 5) (3 4 5)
```

La solució a aquest problema és molt semblant a l'anterior. Només hi ha dues diferències principals:

- En definir l'element $l + 1$ del prefix, cal fer-ho amb valors més grans que els que hi han a la posició l.
- Pel motiu anterior, no podem cridar a l'acció amb un prefix de longitud zero. Donat que com a mínim necessitem un prefix de longitud 1, farem la inicialització del primer element de la seqüència a l'acció principal.

Amb aquestes consideracions, l'algorisme queda de la següent manera:

```
acció genseq1k_creix_rec (ent n, k: nat; e/s t: taula[1..n] de nat; ent l: nat)
{Pre:  n > 0, k > 0, 1 ≤ l ≤ n, t[1..l] ∈ {1..k}^l, creixent(t[1..l])}
{Post: Ha escrit totes les seqüències creixents de longitud n
       d'elements agafats del conjunt {1..k} i que comencen pel prefix t[1..l]}
    si l = n llavors escriure_taula(t);
    sino
        var i: nat;
        {Inv:  Ha escrit totes les seqüències creixents amb prefix t[1..l]
               i amb els valors {t[l] + 1..i − 1} a la posició l + 1}
        per i := t[l] + 1 fins k fer {Agafem valors més grans que t[l]}
            t[l + 1] := i; genseq1k_creix_rec (n,k,t,l + 1);
        fper;
    fsi;
facció
```

```
acció genseq1k_creix (ent n, k: nat)
{Pre:  n > 0, k > 0}
{Post: Ha escrit totes les seqüències creixents de longitud n
       d'elements agafats del conjunt {1..k}}
    var S: taula[1..n] de nat; i: nat;
    {Generem seqüències amb prefixos d'un element}
    per i := 1 fins k fer
        S[1] := i; genseq1k_creix_rec (n,k,S,1);
    fper;
facció
```

12.3 Seqüències d'elements diferents de $\{1..k\}$

L'especificació de l'algorisme és la següent:

```
acció genseq1k_dif (ent n, k: nat)
{Pre:  n > 0, k > 0}
{Post: Ha escrit totes les seqüències de longitud n
       d'elements diferents del conjunt {1..k}}
```

Per exemple, si tinguessim $n = 2$ i $k = 4$, les seqüències generades serien les següents (no necessàriament en aquest ordre):

```
(1 2) (1 3) (1 4) (2 1) (2 3) (2 4) (3 1) (3 2) (3 4) (4 1) (4 2) (4 3)
```

Per a resoldre aquest problema farem servir una estratègia semblant a les anteriors. En generar el següent element del prefix, cal tenir em compte no repetir cap dels que ja han aparegut al prefix. Per evitar haver de consultar tots els elements apareguts, s'utilitzarà una taula auxiliar que contindrà informació sobre els element utilitzats. En particular, tindrem la següent taula:

$$\mathit{usat}\text{: } \textbf{taula}[1..k] \textbf{ de } \text{booleà}$$

que en tot moment indicarà els elements que han estat usats en el prefix construit. Es a dir,

$$\mathit{usat}[i] \Leftrightarrow \text{l'element } i \text{ ha estat usat en el prefix } t[1..l]$$

Aquesta serà l'especificació de l'algorisme recursiu:

acció *genseq1k_dif_rec (**ent** n, k: nat; **e/s** t: **taula**[1..n] **de** nat;*
*__e/s__: usat: **taula**[1..k] **de** booleà; **ent** l: nat)*
{Pre: $n > 0,\ k > 0,\ 0 \leq l \leq n,\ t[1..l] \in \{1..k\}^l,$ *diferents*$(t[1..l])$,
usat indica els elements utilitzats a $t[1..l]$}
{Post: *Ha escrit totes les seqüències de longitud n d'elements diferents del conjunt* $\{1..k\}$ *i que comencen pel prefix* $t[1..l]$, $usat_s = usat_e$}

Cal notar en aquesta especificació que el contingut de la taula *usat* és el mateix a l'entrada i a la sortida de l'acció ($usat_s = usat_e$). Aquest fet és importat a l'hora de mantenir la informació en aquesta taula coherent amb el contingut del prefix.

```
acció genseq1k_dif_rec (ent n, k:nat; e/s t: taula[1..n] de nat;
                        e/s: usat: taula[1..k] de booleà; ent l: nat)
{Pre:  n > 0, k > 0, 0 ≤ l ≤ n, t[1..l] ∈ {1..k}^l, diferents(t[1..l]),
       usat indica els elements utilitzats a t[1..l]}
{Post: Ha escrit totes les seqüències de longitud n d'elements diferents
       del conjunt {1..k} i que comencen pel prefix t[1..l], usat_s = usat_e}
    si l = n llavors escriure_taula(t);
    sino
        var i: nat;
        {Inv:  Ha escrit totes les seqüències d'elements diferents amb prefix t[1..l]
               i amb els valors {1..i − 1} a la posició l + 1}
        per i := 1 fins k fer
            si ¬usat[i] llavors {Només agafem valors no usats}
                t[l + 1] := i; usat[i] := cert;
                genseq1k_dif_rec (n,k,t,usat,l + 1);
                usat[i] := fals; {deixem d'usar l'element}
            fsi;
        fper;
    fsi;
facció
```

Finalment, l'acció que fa la crida inicial ha de generar un prefix buit amb cap element usat.

```
acció genseq1k_dif (ent n, k: nat)
{Pre:  n > 0, k > 0}
{Post: Ha escrit totes les seqüències de longitud n
       d'elements diferents del conjunt {1..k}}
    var S: taula[1..n] de nat;
        usat: taula[1..k] de booleà;
        i: nat;
    {Hem de cridar a l'acció recursiva amb un prefix de longitud zero
    i sense cap element usat}
    per i := 1 fins k fer usat[i] := fals; fper;
    genseq1k_dif_rec (n,k,S, usat,0);
facció
```

12.4 Permutacions de n elements

La generació de totes les permutacions de n elements es podria realitzar mitjançant la següent crida:

genseq1k_dif (n,n);

Per exemple, per $n = 3$ s'obtindria la següent sortida:

```
(1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)
```

En aquesta secció es presenta un algorisme lleugerament diferent a *genseq1k_dif* que té les següents propietats:

- No fa servir la taula *usat* per guardar informació sobre els elements utilitzats en el prefix.
- Permet generar permutacions dels elements d'una taula sobre la mateixa taula. Per tant, no cal que els elements estiguin agafats d'un conjunt predeterminat $\{1..k\}$.

L'especificació de l'algorisme és la següent:

acció *permutacions(**ent** n: nat)*
{`Pre:` $n > 0$}
{`Post:` *Ha generat totes les permutacions dels elements* $\{1..n\}$}

Per a realitzar una implementació recursiva dissenyarem una generalització de l'acció anterior:

acció *permutacions_rec (**ent** n: nat; **e/s** t: **taula**[1..n] **de** nat; **ent** l: nat)*
{`Pre:` $n > 0,\ 0 \leq l \leq n$}
{`Post:` *Ha generat totes les permutacions dels elements de t que tenen el prefix* $t[1..l]$, $t_s = t_e$}

Per al disseny d'aquest algorisme és importat garantir que el contingut de la taula t a la sortida serà el mateix que el de l'entrada.

```
acció permutacions_rec (ent n: nat; e/s t: taula[1..n] de nat; ent l: nat)
{Pre:  n > 0, 0 ≤ l ≤ n}
{Post: Ha generat totes les permutacions dels elements de t
       que tenen el prefix t[1..l], t_s = t_e}
    si l = n llavors escriure_taula(t);
    sino
        var i: nat;
        {Inv:  Ha generat totes les permutacions amb prefix t[1..l]
               i amb els valors {l + 1..i − 1} a la posició l + 1}
        per i := l + 1 fins n fer
            intercanvi (t[l + 1], t[i]); {Posem un nou element del sufix}
            permutacions_rec (n, t, l + 1); {Generem permutacions}
            intercanvi (t[l + 1], t[i]); {Tornem l'element al seu lloc}
        fper;
    fsi;
facció
```

Finalment, l'acció que fa la crida inicial ha de generar un prefix buit i la taula plena dels elements que s'han de permutar.

```
acció permutacions(ent n: nat)
{Pre:  n > 0}
{Post: Ha generat totes les permutacions dels elements {1..n}}
    var S: taula [1..n] de nat; i: nat;
    per i := 1 fins n fer S[i] := i; fper;
    permutacions_rec (n, S, 0);
facció
```

12.5 Seqüències creixents que sumen n

Suposem, per exemple, que $n = 10$. L'algorisme que volem dissenyat hauria de generar les següents seqüències d'elements:

```
(1 2 3 4) (1 2 7) (1 3 6) (1 4 5) (1 9) (2 3 5) (2 8) (3 7) (4 6) (10)
```

L'especificació de l'algorisme és la següent:

```
acció seqcreix_sum (ent n: nat)
{Pre:  n > 0}
{Post: Ha generat totes les seqüències creixents de nombres positius que sumen n}
```

De nou farem servir una estratègia semblant als problemes anteriors, fent inducció amb la longitud del prefix construit. Cal observar que la longitud de les seqüències generades mai serà superior a n. Per ser més precisos, podriem demostrar que mai serà superior a $\alpha\sqrt{n}$ on α és una certa constant. Per no complicar el problema, utilitzarem una taula de tamany n per guardar la seqüència generada.

L'especificació de l'acció recursiva més general és la següent:

```
acció seqcreix_sum_rec (ent n: nat; e/s t: taula[1..n] de nat; ent l, s: nat)
{Pre:  n > 0, 1 ≤ l ≤ n, creixent(t[1..l]), s = Σ_{k=1}^{l} t[k] ≤ n}
{Post: Ha generat totes les seqüències creixents de nombres positius que sumen n
       i que tenen com a prefix t[1..l]}
```

Per estalviar feina a l'hora de calcular la suma dels elements del prefix generat, s'utilitza el paràmetre s que va acumulant la suma del prefix. De manera semblant al que passava amb l'acció *genseq1k_dif_rec* (veure Secció 12.3), es necessita un prefix no buit per poder consultar l'element anterior del prefix i forçar que la seqüència sigui creixent. Per aquesta raó tenim $l \geq 1$ a la precondició.

```
acció seqcreix_sum_rec (ent n: nat; e/s t: taula[1..n] de nat; ent l, s: nat)
{Pre:  n > 0, 1 ≤ l ≤ n, creixent(t[1..l]), s = Σ_{k=1}^{l} t[k] ≤ n}
{Post: Ha generat totes les seqüències creixents de nombres positius que sumen n
       i que tenen com a prefix t[1..l]}
    si s = n llavors escriure_taula(t);
    sino
        var i: nat;
        i := t[l] + 1; {Agafem valors més grans que t[l]}
        {Inv:  Ha escrit totes les seqüències creixents que sumen n amb prefix t[1..l]
               i amb els valors {t[l] + 1..i − 1} a la posició l + 1}
        mentre s + i ≤ n fer {Ens aturem quan la suma sigui massa gran}
            t[l + 1] := i;
            seqcreix_sum_rec (n, t, l + 1, s + i);
            i := i + 1;
        fmentre;
    fsi;
facció
```

Ara només resta dissenyar l'acció principal que genera crides recursives amb prefixos de longitud 1.

```
acció seqcreix_sum (ent n: nat)
{Pre:  n > 0}
{Post: Ha generat totes les seqüències creixents de nombres positius que sumen n}
    var S: taula[1..n] de nat; i: nat;
    {Genera seqüències amb prefixos d'un element}
    per i := 1 fins n fer
        S[1] := i; seqcreix_sum_rec (n, S, 1, i);
    fper;
facció
```

12.6 Problemes

12.1 Dissenyar un algorisme que generi totes les seqüències de longitud k de nombres positius tal que el terme i de la seqüència no sigui superior a i per a qualsevol i.

12.2 Dissenyar un algorisme que generi totes les aplicacions injectives del conjunt $\{1 \dots k\}$ sobre el conjunt $\{1 \dots n\}$, suposant que $k \leq n$. Una aplicació és injectiva si no hi ha dos elements del conjunt $\{1 \dots k\}$ als que els hi correspongui el mateix element de $\{1 \dots n\}$. La generació de les aplicacions hauria de tenir una complexitat lineal sobre k.

12.3 Dissenyar un algorisme que generi totes les possibles particions d'un natural $n > 0$, és a dir, totes les representacions de n com la suma de naturals positius. Generar les sumes de manera que l'ordre dels sumands sigui no creixent. Per exemple, per $n = 4$, les particions són $1+1+1+1$, $2+1+1$, $2+2$, $3+1$ i 4.

12.4 Dissenyar un algorisme amb la mateixa especificació que el problema anterior, però ara generant les seqüències en ordre alfabètic invertit. Per exemple, per $n = 4$ generariem 4, $3+1$, $2+2$, $2+1+1$ i $1+1+1+1$.

12.5 Una seqüència de $2n$ nombres, on cada nombre pertany al conjunt $\{1 \dots k\}$, es diu que *té sort* si la suma dels n primers nombres és igual a la suma dels n restants. Dissenyar un algorisme que, donats n i k, escrigui totes les seqüències amb sort.

Capítol 13

Recursivitat avançada

13.1 Les torres de Hanoi

Les torres de Hanoi és un exemple excepcional que demostra el poder del raonament inductiu per a resoldre problemes complexos. L'enunciat del problema és el següent:

> *Els monjos d'un monestir havien de moure unes pedres circulars, totes de diferent diàmetre, des d'un lloc sagrat fins un altre lloc sagrat. Degut al seu pes, les pedres més petites sempre havien d'estar apilades sobre les més grosses i només es podia moure una pedra a la vegada. Per a moure les pedres, els monjos només disposaven d'un altre lloc sagrat on les podien dipositar temporalment. El problema a resoldre era el següent: quina era la seqüència de moviments que s'havien de realitzar per moure n pedres des d'un lloc fins un altre sense que mai una pedra més petita estigués per sota d'una de més grossa?*

Anomenem *origen*, *destí* i *auxiliar* els tres llocs sagrats on es poden dipositar les pedres i suposem que volem moure n pedres des d'*origen* a *destí*. Aquests llocs es poden identificar amb tres nombres naturals diferents, per exemple 1, 2 i 3. L'especificació de l'acció que volem dissenyar és la següent:

acció *hanoi (**ent** n, origen, desti, aux: nat)*
{`Pre:` *origen, desti i aux identifiquen tres llocs diferents*}
{`Post:` *S'han escrit els moviments per moure n pedres des de origen a desti utilitzant aux*}

El raonament inductiu que es pot realitzar per a resoldre aquest problema s'il·lustra a la Figure 13.1. Suposem que sabem resoldre el problema per $n-1$

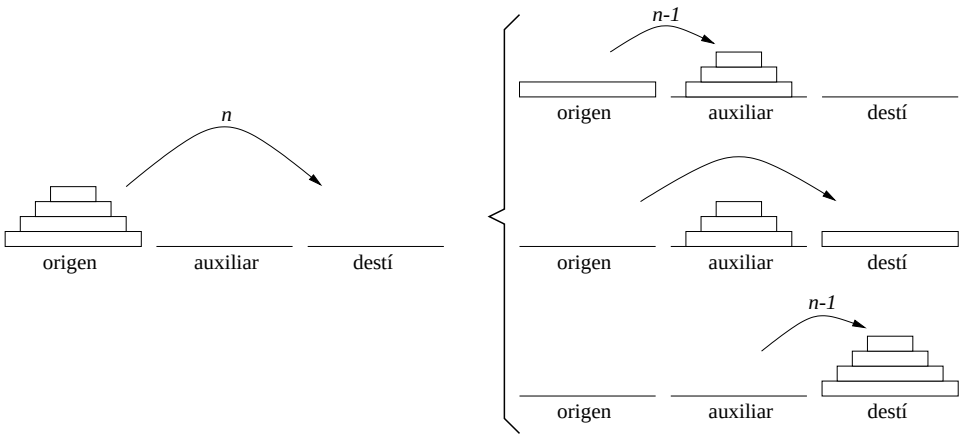


Figura 13.1: Inducció en el problema de les torres de Hanoi

pedres. Com ho podriem resoldre per n pedres?. La solució és la següent:

- Movem $n-1$ pedres des d'*origen* a *auxiliar*, fent servir *destí* com a lloc temporal per a dipositar pedres. Cal notar que la pedra més grossa restarà quieta a *origen* i, per tant qualsevol pedra que es mogui sobre d'ella sempre serà més petita.
- Movem la pedra grossa des d'*origen* a *destí*.
- Movem les $n-1$ pedres d'*auxiliar* a *destí*, fent servir *origen* per a dipositar pedres temporalment.

El raonament recursiu que ens deriva l'algorisme és el següent (fem inducció amb el nombre n de pedres a moure):

- Cas senzill: $n = 0$, no cal moure cap pedra.
- Cas recursiu: $n > 0$, apliquem les tres fases explicades anteriorment (moure $n-1$, moure 1 i moure $n-1$).
- Demostració d'acabament: cada crida recursiva decremanta el nombre de pedres a moure. Sempre s'arribarà al cas senzill $n = 0$.

```
acció hanoi (ent n, origen, desti, aux: nat)
{Pre:  origen, desti i aux identifiquen tres llocs diferents}
{Post: S'han escrit els moviments per moure n pedres des de origen a desti utilitzant aux}
    si n > 0 llavors
        hanoi (n − 1, origen, aux, desti);
        escriure ("Moure de ", origen, " a ", desti);
        hanoi (n − 1, aux, desti, origen);
    fsi;
facció
```

Finalment ens podriem preguntar,

Quants moviments serien necessaris per a moure n pedres?

Si anomenem $mov(n)$ a aquest nombre, podem demostrar per inducció que $mov(n) = 2^n - 1$. Efectivament, sabem que $mov(0) = 2^0 - 1 = 0$ (no cal fer cap moviment). Ara suposem, com a hipòtesi d'inducció, que $mov(n-1) = 2^{n-1} - 1$, i observem l'algorisme proposat anteriorment. El nombre de moviments per a n pedres és

$$mov(n) = mov(n-1) + 1 + mov(n-1) = (2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$$

Per exemple, si tinguessim 10 pedres i poguessim fer un moviment cada minut, trigariem 17 hores i 3 minuts a moure les 10 pedres d'un lloc a l'altre. Què trigariem si tinguéssim 30 pedres?. Més de ... 2000 anys!

13.2 Camí en un laberint (2 moviments)

El problema a resoldre és el de trobar un camí en un laberint que vagi des d'una posició origen (O), amb coordenades (x_o, y_o), a una posició destí (D), amb coordenades (x_d, y_d), sense passar per cap obstacle. Representarem el laberint amb una matriu $L[1..N, 1..M]$ que inicialment conté 0's (per indicar les caselles lliures) i 1's (per indicar les caselles amb obstacles). Després de trobar el camí, volem deixar-lo marcat amb 2's a la mateixa matriu.

En aquesta primera versió del problema, només permetrem fer moviments cap a la dreta (augmentant la coordenada y) i cap abaix (augmentant la coordenada x). L'especificació del problema és la següent:

```
acció trobar_camí (e/s L: taula [1..N,1..M] de nat; ent x_o, y_o, x_d, y_d: nat; sort trobat: booleà)
{Pre:  L conté un laberint, 1 ≤ x_d ≤ N, 1 ≤ y_d ≤ M, L[x_d, y_d] no és un obstacle}
{Post: trobat ⇔ existeix un camí amb només moviments ↓ i → des de (x_o, y_o) a (x_d, y_d),
       trobat ⇒ L té el camí trobat marcat amb "2"}
```

Farem un raonament recursiu per derivar l'algorisme. En primer lloc cal notar que l'especificació del problema permet que (x_o, y_o) indexi una casella fora de la matriu. Això ens servirà per tractar els casos senzills de manera més elegant.

El paradigma recursiu d'aquest problema el podem explicar de la següent manera:

> *Si volem anar de (x_o, y_o) a (x_d, y_d) cal veure si ens podem moure a la dreta i trobar un camí de $(x_o + 1, y_o)$ a (x_d, y_d). Si no el trobem, cal veure si ens podem moure cap abaix i trobar un camí de $(x_o, y_o + 1)$ a (x_d, y_d). Si tampoc el trobem, llavors no hi ha camí.*

Es a dir, el problema de trobar un camí és redueix al problema de fer un moviment i trobar un camí des de la nova posició, provant tots els possibles moviments des de la casella on ens trobem.

Degut a que només es poden realitzar moviments del tipus $\rightarrow$ i $\downarrow$, podem observar que l'existència d'un camí implica que les caselles visitades sempre estaran en la submatriu $L[x_o..x_d, y_o..y_d]$.

El raonament recursiu es pot fer de la següent manera:

- Casos senzills:
  - (x_o, y_o) es troba fora de la submatriu $L[1..x_d, 1..y_d]$. En aquest cas, no existeix camí.
  - $L[x_o, y_o]$ és un obstacle. Tampoc existeix camí.
  - $x_o = x_d \ \wedge \ y_o = y_d$. Ja hem trobat camí, doncs l'origen és el propi destí.
- Cas recursiu (l'únic que no cobreixen els casos senzills). Ens trobem dins de la matriu, en una casella que no és un obstacle, però que tampoc és la casella destí. Cal provar de fer el moviment $\rightarrow$ i trobar el camí des de (x_o+1, y_o). En cas de no trobar-lo, cal provar el moviment $\downarrow$ i trobar el camí des de (x_o, y_o+1). Si en algun cas es troba el camí, cal marcar la casella amb un '2'. Per inducció, el camí trobat ja haurà estat marcat.
- Demostració d'acabament. En cada crida recursiva poden passar dues coses:
  - Que es faci un moviment i s'arribi a un cas senzill. En aquest cas, la recursivitat acabarà.

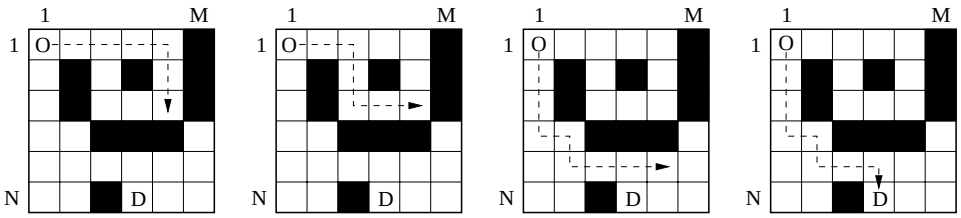


Figura 13.2: Camins explorats per l'algorisme del laberint amb moviments $\rightarrow$ i $\downarrow$.

– Que es faci un moviment i s'arribi a una casella sense obstacle dins de la submatriu $L[1..x_d, 1..y_d]$. En aquest cas, sempre ens aproparem a (x_d, y_d) en alguna de las dimensions de la matriu. Això no pot passar un nombre infinit de vegades sense que arribem a la casella (x_d, y_d), altrament hauriem sortit de la submatriu $L[1..x_d, 1..y_d]$ en algun moment. Això demostra que qualsevol cadena de crides recursives no pot ser infinita.

L'evolució de l'algorisme s'il·lustra a la Figura 13.2, mostrant diferents camins explorats durant la cerca.

```
acció trobar_camí (e/s L: taula [1..N,1..M] de nat; ent x_o, y_o, x_d, y_d: nat; sort trobat: booleà)
{Pre:  L conté un laberint, 1 ≤ x_d ≤ N, 1 ≤ y_d ≤ M, L[x_d, y_d] no és un obstacle}
{Post: trobat ⇔ existeix un camí amb només moviments ↓ i → des de (x_o, y_o) a (x_d, y_d),
       trobat ⇒ L té el camí trobat marcat amb "2"}
    si x_o < 1 ∨ x_o > x_d ∨ y_o < 1 ∨ y_o > y_d llavors trobat := fals; {Fora del laberint}
    sinosi L[x_o, y_o] = 1 llavors trobat := fals; {Obstacle}
    sinosi x_o = x_d ∧ y_o = y_d llavors L[x_o, y_o] := 2; trobat := cert; {Hem arribat}
    sino {Encara no hem arribat. Intentem avançar.}
        trobar_camí (L, x_o + 1, y_o, x_d, y_d, trobat);
        si ¬trobat llavors
            trobar_camí (L, x_o, y_o + 1, x_d, y_d, trobat);
        fsi;
        {trobat ⇔ ha trobat un camí}
        si trobat llavors L[x_o, y_o] := 2; fsi;
    fsi;
facció
```

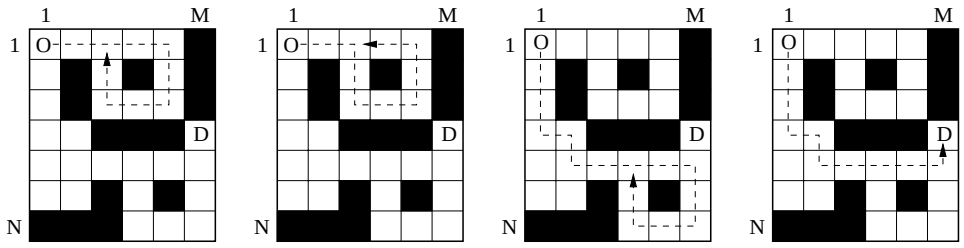


Figura 13.3: Camins explorats per l'algorisme del laberint amb moviments $\rightarrow$, $\leftarrow$, $\downarrow$ i $\uparrow$.

13.3 Camí en un laberint (4 moviments)

En aquesta secció es presenta un algorisme semblant al de la secció anterior. Però en aquest cas es permeten moviments cap a les quatre direccions: dreta, esquerra, amunt i abaix. Això fa que la resolució sigui una mica més complicada. Més en concret, el problema que pot aparèixer es mostra a la Figura 13.3, on es veu que la cerca podria entrar en un cicle de caselles visitades sense obstacles. Això podria comportar una cadena infinita de crides recursives que no trobessin mai el camí ni sortissin mai de la matriu.

L'especificació de l'algorisme que volem dissenyar ara és:

acció *trobar_camí (**e/s** L: **taula** [1..N,1..M] **de** nat; **ent** x_o, y_o, x_d, y_d: nat; **sort** trobat: booleà)*
{`Pre:` *L conté un laberint,* $1 \leq x_d \leq N$, $1 \leq y_d \leq M$, $L[x_d, y_d]$ *no és un obstacle,*
L conté el camí construit fins a (x_o, y_o) *marcat amb "2"*}
{`Post:` *trobat* $\Leftrightarrow$ *existeix un camí des de* (x_o, y_o) *a* (x_d, y_d),
trobat $\Rightarrow$ *L té el camí trobat marcat amb "2"*}

A continuació s'expliquen les principals diferències amb l'algorisme presentat a la secció anterior:

- Donat que es poden realitzar quatre moviments, quasevol casella de la matriu que no sigui un obstacle pot acollir un camí valid. Per aquesta raó, el primer cas senzill únicament comprova que (x_o, y_o) estigui dins de la matriu.

- En el cas recursiu, es proven quatre possibles moviments sempre i quan no es trobi un camí abans. En el problema anterior només se'n provaven dos. La diferència principal està en que la casella visitada es marca (amb un '2') *abans* de provar els moviments. Això permet detectar el cas en que un camí visita una casella ja visitada pel mateix

camí. Finalment, en el cas que no es trobi camí, la casella es desmarca per deixar la matriu tal com estava abans de cercar aquest camí.

- El segon cas senzill també comprova que la casella no hagi estat visitada. Les caselles visitades són considerades com obstacles, amb l'objectiu d'evitar camins cíclics. Per aquesta raó, la condició és $L[x_o, y_o] \neq 0$.
- La condició d'acabament és més difícil de demostrar. En aquest cas, cal observar que no sempre els moviments ens garanteixen l'apropament a la casella destí. En canvi, sí que cada vegada que es fa una crida recursiva, abans es marca una casella més de la matriu amb un '2' (casella visitada). Com que el nombre de caselles és finit, mai podrà haver una cadena infinita de crides recursives sense trobar una casella ja visitada. Això garanteix l'acabament.

La Figura 13.3 mostra l'evolució de l'algorisme en diverses fases de la seva execució per un exemple concret. A continuació es presenta l'algorisme.

```
acció trobar_camí (e/s L: taula [1..N,1..M] de nat; ent x_o, y_o, x_d, y_d: nat; sort trobat: booleà)
{Pre: L conté un laberint, 1 ≤ x_d ≤ N, 1 ≤ y_d ≤ M, L[x_d, y_d] no és un obstacle,
     L conté el camí construit fins a (x_o, y_o) marcat amb "2"}
{Post: trobat ⇔ existeix un camí des de (x_o, y_o) a (x_d, y_d),
     trobat ⇒ L té el camí trobat marcat amb "2"}
    si x_o < 1 ∨ x_o > N ∨ y_o < 1 ∨ y_o > M llavors trobat := fals; {Fora del laberint}
    sinosi L[x_o, y_o] ≠ 0 llavors trobat := fals; {Obstacle o casella visitada. Tirem enrera.}
    sinosi x_o = x_d ∧ y_o = y_d llavors L[x_o, y_o] := 2; trobat := cert; {Hem arribat}
    sino {Encara no hem arribat. Intentem avançar.}
        L[x_o, y_o] := 2; {Marquem per no tornar a passar.}
        trobar_camí (L, x_o + 1, y_o, x_d, y_d, trobat);
        si ¬trobat llavors
            trobar_camí (L, x_o − 1, y_o, x_d, y_d, trobat);
            si ¬trobat llavors
                trobar_camí (L, x_o, y_o + 1, x_d, y_d, trobat);
                si ¬trobat llavors
                    trobar_camí (L, x_o, y_o − 1, x_d, y_d, trobat);
                fsi;
            fsi;
        fsi;
        {trobat ⇔ ha trobat un camí}
        si ¬trobat llavors L[x_o, y_o] := 0; fsi; {No hem trobat camí. Desmarquem la casella.}
    fsi;
facció
```

13.4 Les vuit reines

El problema a resoldre té el següent enunciat:

> *Cal posar vuit reines en un tauler d'escacs sense que cap parella d'elles es pugui capturar mútuament.*

En realitat, el problema més general que volem resoldre en aquesta secció és el de posar n reines en un tauler de $n \times n$ caselles. Representarem el nostre tauler amb una matriu $T[1..n, 1..n]$ de booleans, on voldrem que l'algorisme ens "marqui" els llocs on es poden posar les reines en el cas que es trobi una solució. Suposarem que la casella $T[1, 1]$ correspon a la superior esquerra en les nostres figures. A continuació es presenta l'especificació de l'algorisme.

funció *reines (n: nat)* ***retorna*** *(hi_ha:booleà; T:* ***taula****[1..n,1..n]* ***de*** *booleà)*
{`Pre:` $n > 0$}
{`Post:` *hi_ha* $\Leftrightarrow$ *exiteix una ubicació legal de n reines,*
hi_ha $\Rightarrow$ *T conté la ubicació de les n reines*}

Abans de resoldre el problema farem un conjunt d'observacions:

- Les reines es poden moure en horitzontal, vertical i diagonal, qualsevol nombre de caselles.
- No hi podrà haver dues reines a la mateixa fila, altrament es podrien capturar entre elles. Donat que tenim n reines i n files, hi haurà d'haver una reina a cada fila.
- El mateix raonament el podem aplicar a les columnes.
- Podem distingir dos tipus de diagonals, les que van cap amunt i a la dreta ($\nearrow$) i les que van cap amunt i a l'esquerra ($\nwarrow$). A cada diagonal només hi podrà haver una reina.
- Si dues caselles (i_1, j_1) i (i_2, j_2) pertanyen a la mateixa diagonal en sentit $\nearrow$, llavors es compleix que $i_1 + j_1 = i_2 + j_2$. D'aquesta manera, podem identificar totes les diagonals $\nearrow$ amb un índex del conjunt $\{2, .., 2n\}$, que s'obté de sumar $i + j$ per a qualsevol element de la diagonal.
- Si dues caselles (i_1, j_1) i (i_2, j_2) pertanyen a la mateixa diagonal en sentit $\nwarrow$, llavors es compleix que $i_1 - j_1 = i_2 - j_2$. D'aquesta manera, podem identificar totes les diagonals $\nwarrow$ amb un índex del conjunt

$\{1-n, .., n-1\}$, que s'obté de restar $i-j$ per a qualsevol element de la diagonal.

La recursivitat ens ajuda força a fer la resolució d'aquest problema més senzilla. En la solució que es proposa, es fa inducció amb el nombre de files del tauler. L'estratègia que se seguirà s'explica informalment a continuació:

> *Suposem que hem posat $i-1$ reines a les files $1..i-1$, sense que cap parella d'elles es pugui capturar. Per posar la resta de reines farem:*
>
> - *Posar una reina a la posició $T[i,1]$. Si no pot capturar a cap de les $i-1$ anteriors, provarem de posar les reines a les files $i+1..n$ (crida recursiva). Si té èxit, ja hem trobat una solució i la podem retornar.*
> - *Si no s'ha tingut èxit, provarem el mateix per a la posició $T[i,2]$.*
> - *...*
> - *Si no s'ha tingut èxit, provarem el mateix per a la posició $T[i,n]$.*
> - *Si no ha tingut èxit, vol dir que no hi ha una solució vàlida per a les $i-1$ reines posades a les files $1..i-1$.*

Cada vegada que cal posar una reina en una casella, cal comprovar que no captura cap reina de les files anteriors. Això podria realitzar-se mitjançant una funció que explorés totes les caselles dominades per la nova reina i comprovés que no hi ha cap reina en elles. Per evitar fer aquesta exploració cada vegada que volem posar una reina, utilitzarem unes taules que ens "recolliran" tota aquesta informació. En particular tindrem les següents taules, totes elles de tipus booleà (veure Figura 13.4, on els • indiquen les columnes i diagonals dominades):

- $C[1..n]$ que indicarà quines columnes ja estan dominades per les reines posades a les files $1..i-1$.
- $D_d[2..2n]$ que indicarà quines diagonals en sentit $\nearrow$ ja estan dominades.
- $D_e[2..2n]$ que indicarà quines diagonals en sentit $\nwarrow$ ja estan dominades.

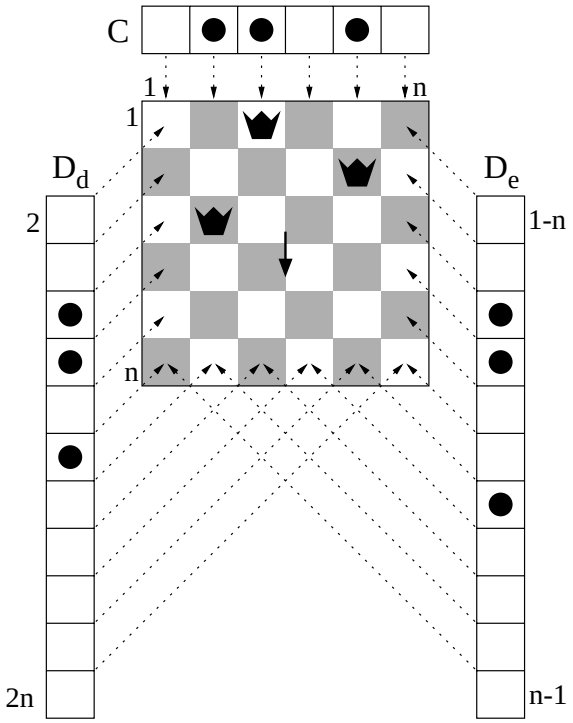


Figura 13.4: Estat dels paràmetres en la crida a *reines_rec* (6,T,4,C,D_d,D_e,*trobat*).

Ara podem plantejar l'especificació de l'acció recursiva que resoldrà el problema de les reines.

acció *reines_rec* (***ent*** *n: nat,* ***e/s*** *T:* ***taula****[1..n,1..n]* ***de*** *booleà;* ***ent*** *i: nat;*
e/s *C:* ***taula*** *[1..n]* ***de*** *booleà;*
e/s D_d*:* ***taula*** *[2..*$2 * n$*]* ***de*** *booleà;*
e/s D_e*:* ***taula*** *[1-n..*$1 + n$*]* ***de*** *booleà;*
sort *trobat: booleà)*

{`Pre:` $n > 0$, T *conté una posició legal de reines a les* $i - 1$ *primeres files,* C, D_d *i* D_e *indiquen quines columnes, diagonals* $\nearrow$ *i diagonals* $\nwarrow$, *respectivament, estan dominades per les* $i - 1$ *reines*}

{`Post:` *trobat* $\Leftrightarrow$ *exiteix una ubicació legal amb les* $i - 1$ *reines de l'entrada,* *trobat* $\Rightarrow$ T *conté la ubicació de totes les reines*}

Fem el raonament recursiu per a dissenyar *reines_rec*.

- Cas senzill: $i > n$, totes les reines han estat posades amb èxit. No cal fer res.

- Cas recursiu: $i \leq n$, s'han posat reines a les files $1..i-1$. Ara cal fer una *cerca* de la columna on es pot posar una reina a la columna i. Això s'implementa amb un bucle de cerca on, després d'haver comprovat que la casella (i, j) no domina a cap reina anterior, es marquen les caselles dominades per aquesta i es prova de trobar una ubicació per a les reines de les files $i + 1..n$. La cerca acaba tant aviat s'ha trobat una columna que permeti obtenir una solució.

- Demostració d'acabament. A cada crida recursiva s'incrementa el valor de i. En algun moment s'arribarà al cas senzill ($i > n$). Això garanteix que mai hi podrà haver una cadena infinita de crides recursives.

```
acció reines_rec (ent n: nat, e/s T: taula[1..n,1..n] de booleà; ent i: nat;
                  e/s C: taula [1..n] de booleà;
                  e/s D_d: taula [2..2*n] de booleà;
                  e/s D_e: taula [1-n..1+n] de booleà;
                  sort trobat: booleà)
{Pre:  n > 0, T conté una posició legal de reines a les i-1 primeres files,
       C, D_d i D_e indiquen quines columnes, diagonals ↗ i diagonals ↖,
       respectivament, estan dominades per les i-1 reines}
{Post: trobat ⇔ existeix una ubicació legal amb les i-1 reines de l'entrada,
       trobat ⇒ T conté la ubicació de totes les reines}
    si i > n llavors {Cas senzill}
        trobat := cert;
    sino {Cas recursiu}
        trobat := fals; j := 1;
        {Inv: No s'ha trobat una posició legal a les caselles T[i,1..j-1],
              trobat ⇒ ha trobat una posició legal a la casella T[i,j], 1 ≤ j ≤ n+1}
        mentre ¬trobat ∧ j ≤ n fer
            si ¬c[j] ∧ ¬D_d[i+j] ∧ ¬D_e[i-j] llavors
                {Marquem tauler i caselles dominades per la posició (i,j)}
                T[i,j] := cert; C[j] := cert; D_d[i+j] := cert; D_e[i-j] := cert;
                {cerquem una solució}
                reines_rec (n, T, i+1, C, D_d, D_e, trobat);
                si ¬trobat llavors
                    {Desmarquem tauler i caselles dominades per la posició (i,j)}
                    T[i,j] := fals; C[j] := fals; D_d[i+j] := fals; D_e[i-j] := fals;
                fsi;
            fsi;
            si ¬trobat llavors j := j+1; fsi; {Provem la següent columna}
        fmentre;
    fsi;
facció
```

Es deixa pel lector la justificació del bucle que implementa el cas recursiu. El raonament correspon al d'una cerca. Finalment, només cal dissenyar la funció inicial que resol el problema de les reines. L'únic que cal fer es declarar les taules utilitzades durant les crides recursives i inicialitzar-les adequadament (cap casella dominada). La crida inicial a *reines_rec* cal fer-la amb el paràmetre $i = 1$ (cap reina posada).

```
función reines (n: nat) retorna (hi_ha:booleà; T: taula[1..n,1..n] de booleà)
{Pre:  n > 0}
{Post: hi_ha ⇔ exiteix una ubicació legal de n reines,
       hi_ha ⇒ T conté la ubicació de les n reines}
    var C: taula [1..n] de booleà;
        D_d: taula [2..2 * n] de booleà;
        D_e: taula [1 − n..1 + n] de booleà;
        i, j: nat;

    {Inicialitzem les taules de columnes i diagonals dominades}
    per i := 1 fins n fer
        C[i] := fals;
        per j := 1 fins n fer T[i, j] := fals; fper;
    fper;
    per i := 2 fins 2 * n fer D_d[i] := fals; fper;
    per i := 1-n fins 1 + n fer D_e[i] := fals; fper;

    {I ara resolem el problema de les reines}
    reines_rec (n, T, 1, C, D_d, D_e, hi_ha);
    retorna (hi_ha, T);
ffunció
```

Com a curiositat direm que el problema de les n reines té solució per a qualsevol valor de n que sigui diferent de 2 i 3.