Correctesa de Programes Iteratius i Programació Recursiva

Assignatura PRO2

Octubre 2017

Índex

Capítol 5

Correctesa de Programes Iteratius

5.1 El principi d'inducció

Per demostrar la correctesa de programes iteratius i recursius utilitzarem el principi d'inducció. Aquest serveix en general per demostrar propietats dels nombres naturals, i és una eina matemàtica bàsica.

La primera versió del principi d'inducció, on $P(x)$ significa que x compleix la propietat P, és

$$[P(0) \wedge \forall x \in N(P(x) \implies P(x+1))] \implies \forall y \in N\ P(y).$$

Aquest principi diu que si volem demostrar que tots els nombres naturals compleixen una certa propietat P, és a dir $\forall y \in N\ P(y)$, cal verificar primer:

1. El nombre zero compleix la propietat P, és a dir $P(0)$.

2. Per qualsevol natural, si aquest compleix la propietat P, llavors el seu successor també la complirà, és a dir $\forall x \in N(P(x) \implies P(x+1))$.

La demostració de que el nombre zero compleix la propietat P s'anomena *base de la inducció*. D'altra banda, demostrar que si un nombre natural compleix la propietat, llavors el seu successor també, s'anomena *pas d'inducció*.

El principi d'inducció és bastant natural. Donat que hi ha un nombre infinit de nombres naturals, no podem comprovar un per un que tots ells tenen la propietat que volem. El que fem és demostrar que el zero la té, i que si un número la té, el següent també. D'aquesta manera, donat que el zero té la propietat, el 1 també la té, això implica que el 2 també la té, i així successivament.

Existeix una altra forma de principi d'inducció que també utilitzarem. És aquesta:

$$[P(0) \wedge \forall x \in N(\forall y < x\ P(y) \implies P(x))] \implies \forall z \in N\ P(z)$$

En aquest cas, si volem demostrar que tots els naturals compleixen una propietat P, això és $\forall z \in N\ P(z)$, cal demostrar:

1. El nombre zero compleix la propietat, això és $P(0)$.

2. Per qualsevol natural x, si tots els seus antecessors compleixen la propietat P, llavors aquest també la compleix, és a dir, $\forall y < x\, P(y) \implies P(x)$.

Ambdós principis són molt utilitzats per a demostrar propietats bàsiques dels naturals, com per exemple $\forall n, 1+2+\cdots+(n-1)+n = n(n+1)/2$, etc. El que no està clar, de moment, és com utilitzar aquests principis per a ajudar-nos a demostrar la correcció d'algoritmes iteratius i recursius, i en aquest cas, de quines propietats P de nombres naturals estem parlant.

5.2 Correctesa de programes iteratius

El format habitual dels programes iteratius és el següent:

```
/* Pre */
inicialitzacions
while (B){
  S
}
/* Post */
```

Per demostrar la correcció d'un bucle, hem de garantir que si a l'inici de l'execució les variables compleixen la precondició, al finalitzar l'execució del bucle compleixen la postcondició. Per això, és essencial trobar l'*invariant*. Aquest permet transmetre la informació de la precondició a la postcondició. Per tant l'invariant és una propietat que s'ha de mantenir en cada iteració. És a dir, que mentre es compleix la condició B, l'invariant serà cert a l'inici i al finalitzar cada iteració del bucle. A més s'han de garantir un parell de coses més. Per una banda les inicialitzacions han de fer cert l'invariant, i d'altre, en la darrera iteració del bucle, quan B ja no sigui cert, l'invariant (que es complirà com al final de cada iteració) ha d'implicar la postcondició.

Els elements a demostrar que hem mencionat són els següents:

1. Demostrar que $\{Pre\}Inicialitzacions\{Invariant\}$. És a dir, si les variables compleixen la precondició, després de realitzar les instruccions d'inicialització abans del bucle, les variables compliran l'invariant.

2. Demostrar que $\{Invariant \wedge B\}S\{Invariant\}$. És a dir, si les variables compleixen l'invariant i la condició d'entrada al bucle, després d'executar les instruccions d'aquest, es tornarà a complir l'invariant.

3. $\{Invariant \wedge \neg B\} \implies \{Post\}$. És a dir, si les variables compleixen l'invariant però no la condició d'entrada al bucle, llavors han de complir la postcondició.

Veiem la relació d'aquests passos amb el principi de inducció que hem introduït en la secció anterior. Suposem que la propietat que volem demostrar és que les variables del programa en qualsevol iteració del bucle compleixen l'invariant, on el nombre natural que compleix aquesta propietat és el nombre d'iteració. En la iteració zero es compleix l'invariant perquè demostrar

el punt 1 és equivalent a això. El punt 2 és el pas d'inducció. En ell diem que si al final d'una iteració es compleix l'invariant, al final de la següent també es complirà. El resultat d'aquest raonament seria concloure que en tota iteració del bucle es complirà l'invariant. Malgrat tot, en aquest cas no volem concloure que sempre es complirà l'invariant, sinó únicament si es compleix la condició d'entrada al bucle. Finalment la condició 3 ens diu que si no es compleix la condició d'entrada al bucle, es a dir $\neg B$, llavors s'ha de complir la postcondició que és el que volíem demostrar.

Una darrera cosa és important per a la correcció de l'algoritme. Hem d'assegurar-nos de que aquest acaba. És a dir, que no estem dins d'un bucle infinit. Per això hem de trobar una variable del bucle o una funció d'aquestes (funció de fita), que tingui un valor natural i que decreixi en cada iteració del bucle. Aquest paràmetre ha d'apropar-nos cada cop més a la condició d'acabament del bucle, és a dir $\neg B$. Així demostrarem que el bucle acaba després d'un nombre finit de passos.

Per tant, si la nostra funció de fita es diu t, haurem de demostrar dues coses. Una, que $Invariant \Rightarrow t \in N$, per assegurar-nos que si estem en la situació determinada per l'invariant, llavors t té un valor natural. L'altre, $\{Invariant \wedge B \wedge t = T\}S\{t < T\}$, per assegurar-nos que t decreix després de cada iteració.

5.3 Disseny inductiu d'algoritmes iteratius

Volem donar una idea de com generar un programa iteratiu a partir de la seva especificació. La idea és obtenir primer l'invariant, i després les instruccions del bucle. Aquests serien els passos:

- A partir fonamentalment de la postcondició, que és l'asserció que ens parla del que ha de assolir el bucle, hem de caracteritzar l'invariant. Per això hem de pensar que l'invariant ha de descriure un punt intermig de l'execució del bucle. Una altra manera de veure-ho és pensant que l'invariant és un afebliment de la postcondició, en el sentit de que és la postcondició respecte a la part tractada en l'execució. Per poder afeblir la postcondició necessitarem parlar de noves variables (seran variables locals en el codi), amb les que expressar l'invariant.

- A partir de l'invariant hem de deduir l'estat en el que l'execució ha acabat. Aquesta serà la condició d'acabament del bucle, i negant aquesta obtindrem la condició B del bucle.

- Veure quines instruccions necessita el cos del bucle per tal que si a l'entrada d'aquest es compleix l'invariant, al final de l'execució del cos del bucle també es complirà. I donat que cada iteració del bucle ens ha d'apropar al final de l'execució d'aquest, hem de trobar una funció que decreixi en cada iteració.

- A més hem de veure quines instruccions o inicialitzacions faran que es compleixi l'invariant abans d'entrar al bucle.

Tingueu en compte que quan diem que una instrucció o conjunt d'instruccions simples (sense iteració ni recursivitat) fan que es compleixi alguna cosa, això comporta una sèrie comprovacions

addicionals: si es tracta d'accesos a vectors, que les posicions visitades estiguin definides, si es tracta de crides a operacions, que es compleixi la seva precondició, si el que tenim és un `if`, que cada branca arribi a l'objectiu per separat, etc.

5.4 Exemples de justificació de programes iteratius

A l'hora de treballar amb els exemples, posarem una serie de petites restriccions als nostres codis, per tal de poder fer les justificacions.

1. Només podrem fer iteracions amb la instrucció while.
2. Les expressions booleanes no s'avaluen amb prioritat.
3. En el cas de funcions, solament podem utilitzar un return, i aquest es trobarà al final de tot.

Aquestes restriccions no són importants, i no suposen restriccions reals en el codi. La primera implica només que en lloc d'utilitzar la instrucció, per exemple, del "for", utilitzarem la del "while". Respecte a la segona restricció, el que vol dir és que en una expressió booleana conjunció o disjunció, s'avaluen les 2 subexpressions. Per exemple, si tenim "p and q", si p es fals, q també s'avalua. Si tenim "p or q", si p es cert, q també s'avalua. En quant a la tercera, el fet de fer el return només al final introdueix un cert ordre en el nostre codi, el fa més simple de verificar, i pot evitar errors de programació. Clarament, qualsevol altre codi que no compleixi alguna de les restriccions, pot ser transformat fàcilment en un d'equivalent que sí que les compleixi.

Els primers exemples que veurem són dues variants de la cerca lineal en un vector. Donat un vector d'enters i un enter, s'ha de veure si l'enter apareix al vector. En els dos algorismes tindrem un índex `i` que marca la posició del següent element a comprovar. La diferència entre els dos algorismes resideix en si incrementem o no l'índex `i` en el cas de trobar l'element cercat. Tanmateix, aquesta petita diferència farà que els respectius invariants suguin diferents, així com les justificacions corresponents. Es important, doncs, per poder fer la justificació correctament, que l'invariant donat sigui completament coherent amb el codi de l'algorisme iteratiu.

A continuació, el primer algorisme i la seva justificació:

```
bool cerca_iter1_vec_int(const vector<int> &v, int x)
/* Pre: cert */
/* Post: El resultat ens diu si x és un element de v o no */
{
  int i = 0;
  bool ret = false;

  /* Inv: 0<=i<=v.size(), ret="l'enter x és un element de v[0..i-1]" */

  while(i<v.size() and not ret){
    ret = (v[i] == x);
    ++i;
```

```
  }
  return ret;
}
```

- *Inicialitzacions*. Inicialment no hem comprovat cap element de `v`, per tant inicialitzem `i` a zero, la posició del primer element del vector, satisfent la primera part de l'invariant. Per satisfer la segona, cal posar el valor `false` a `ret`, donat que no pot existir l'enter `x` en un subvector buit `v[0..-1]`.

- *Condició de sortida*. Es pot sortir del bucle per dues raons:

  - Si `i` arriba a ser `v.size()` vol dir, per l'invariant, que hem explorat tot el vector `v` i que `ret` ens diu si l'element `x` està a `v`, com es pretén a la postcondició.
  - En cas contrari, com que per l'invariant tenim `i<=v.size()`, es compleix que `i<v.size()` i per sortir del bucle s'ha de cumplir que `ret` és cert.

    Però si abans d'arribar al final del vector `v` tenim que `ret` passa a ser `true`, voldrà dir (de nou per l'invariant) que hem trobat `x` al subvector `v[0..i-1]` i que `ret` ja ens diu que `x` està a `v` i també es compleix la postcondició.

- *Cos del bucle*. Abans d'avançar l'índex `i`, hem de satisfer l'invariant canviant `i` per `i+1`. Necessitem que `ret` contingui la informació de si l'element `x` es troba al subvector `v[0..i]`. Com que, si hem entrat al bucle vol dir que `ret==false`, i per l'invariant tenim que `ret` conté la informació sobre si `x` està al subvector `v[0..i-1]`, queda clar que `x` no està al subvector `v[0..i-1]` i només ens cal veure si `x` és igual a `v[i]`, i actualitzar el valor de `ret` depenent de si són iguals o no. Un cop que tenim `ret` modificat, ja podem incrementar `i` assegurant que satisfem l'invariant. Noteu que, per entrar al bucle, `i<v.size()`, per tant, marca una posició vàlida del vector i després d'incrementar `i` es compleix que `i<=v.size()` (tal com demana l'invariant).

- *Acabament*. A cada volta decreix la distància entre `v.size()` i l'índex `i`, perquè incrementem `i` a cadascuna d'elles.

Ara, la segona variant (noteu les diferències en l'invariant i en la manera de justificar la correcció de l'algorisme):

```
bool cerca_iter2_vec_int(const vector<int> &v, int x)
/* Pre: cert */
/* Post: El resultat ens diu si x és un element de v o no */
{
  int i = 0;
  bool ret = false;

  /* Inv: 0<=i<=v.size(), el subvector v[0..i-1] no conté l'element x,
     si ret és true llavors v[i]==x */
```

```
  while(i<v.size() and not ret){
    if (v[i] == x) ret = true;
    else ++i;
  }
  return ret;
}
```

- *Inicialitzacions*. Inicialment no hem comprovat cap element de `v`, per tant inicialitzem `i` a zero, la posició del primer element del vector, satisfent la primera i la segona part de l'invariant, donat que no pot existir l'enter `x` en un subvector buit `v[0..-1]`. Per satisfer la tercera part, només cal posar el valor `false` a `ret`, ja que qualsevol implicació que tingui com a premisa `false` sempre es compleix (per definició).

- *Condició de sortida*. Es pot sortir del bucle per dues raons:

  - Si `i` arriba a ser `v.size()` vol dir, per l'invariant, que hem explorat tot el vector `v` i que `ret==false` (altrement, voldria dir que `v[i]==x`, el que és impossible perquè `v[v.size()]` no existeix), com es pretén a la postcondició.
  - En cas contrari, com que per l'invariant tenim `i<=v.size()`, es compleix que `i<v.size()` i per sortir del bucle s'ha de cumplir que `ret` és cert.

    Però si abans d'arribar al final del vector `v` tenim que `ret` passa a ser `true`, voldrà dir (de nou per l'invariant) que hem trobat `x` a la posició `v[i]` i que `ret` ja ens diu que `x` està a `v` i també es compleix la postcondició.

- *Cos del bucle*. Per l'invariant sabem que `x` no es troba a `v[0..i-1]` i per la condició d'entrada sabem que `i<v.size()` marca una posició vàlida del vector i que `ret==false`. Necessitem comprovar doncs si `v[i]==x` o no. Si no ho és, llavors `x` no es troba a `v[0..i]`, i només cal incrementar `i` perquè es torni a complir l'invariant. Si l'element `x==v[i]` llavors les dues primeres parts de l'invariant es continuen complint i només cal assignar `ret=true` per poder sortir del bucle satisfent al mateix temps la tercera part de l'invariant. Noteu que no podem incrementar `i` si trobem l'element `x`, perquè llavors no es compliria la segona part de l'invariant.

- *Acabament*. A cada volta, o bé decreix la distància entre `v.size()` i l'índex `i`, perquè incrementem `i`, o bé posem el booleà `ret` a `true` i sortim del bucle.

El següent exemple consisteix en, donat un vector d'estudiants, retornar el percentatge d'estudiants presentats del vector.

```
double presentats(const vector<Estudiant> &vest)
/* Pre: vest conté al menys un element */
/* Post: el resultat és el percentatge de presentats de vest */
```

Implementació:

Per aconseguir el percentatge de presentats necessitem saber el nombre de presentats, és a dir, el nombre d'estudiants amb nota.

L'estratègia per resoldre aquest problema és anar comptant el nombre d'estudiants amb nota a mida que recorrem el vector. Per tant, considerem que hem comptat els estudiants amb nota fins a un punt "i" (sense incloure'l) i avancem considerant el següent (l'i-èssim). Així doncs, l'invariant és

```
I: 0<=i<=vest.size(), n="nombre d'estudiants amb nota en vest[0..i-1]"
```

Raonem ara sobre les instruccions del bucle.

- *Inicialitzacions*. Inicialment considerem que no hi ha cap estudiant del vector tractat, això ho representem posant un `0` a la `i`, la qual cosa ens permet satisfer l'invariant assignant un `0` a la `n`, ja que no hi ha cap estudiant fins a `i-1` amb nota.

- *Condició de sortida*. Com que la postcondició diu que `n` és el nombre d'estudiants amb nota de tot el vector, sortirem del bucle quan `i=vest.size()`, ja que aquesta condició juntament amb l'invariant ens dóna el valor desitjat de `n`. Per tant, ens quedem mentre `i < vest.size()` (noteu que hem demanat a l'invariant que `i <= vest.size()`).

- *Cos del bucle*. Com que hem d'avançar, el més natural és incrementar la `i` de manera que s'acosti al final del vector. Però abans d'incrementar la `i`, hem de satisfer l'invariant canviant `i` per `i+1`. Necessitem, per tant, que `n` contingui finalment el "nombre d'estudiants amb nota en `vest[0..i]`". Com que actualment `n` conté el "nombre d'estudiants amb nota en `vest[0..i-1]`", només ens cal preguntar si `vest[i]` té nota (l'accés és correcte ja que `0 <= i < vest.size()`). En cas afirmatiu el nombre d'estudiants amb nota és `n+1`, que és el que hem d'assignar a `n`. En canvi, si no té nota el nombre d'estudiants amb nota segueix sent `n`, i per tant no cal fer res.

  Un cop que `n` conté el "nombre d'estudiants amb nota en `vest[0..i]`", ja podem incrementar la `i` assegurant que satisfem l'invariant. A més la condició del bucle ens assegura que `i+1 <= vest.size()`.

- *Acabament*. A cada volta decreix la distància entre `vest.size()` i la `i`.

- *Instruccions finals*. Quan sortim del bucle tenim a `n` el "nombre d'estudiants amb nota en `vest`", per tant només cal multiplicar-lo per `100` i dividir-lo per `vest.size()`, i obtindrem el percentatge.

El programa anotat queda de la següent manera:

```
double presentats(const vector<Estudiant> &vest)
/* Pre: vest conté almenys un element */
/* Post: el resultat és el percentatge de presentats de vest */
{
```

```
  int numEst = vest.size();
  int n = 0;
  int i=0;

  /* Inv: 0<=i<=numEst, n=nombre d'estudiants amb nota en vest[0..i-1]*/

  while(i < numEst){
    if (vest[i].te_nota()) ++n;
    ++i;
  }
  /* n és el nombre d'estudiants amb nota de vest */
  double pres = n*100./numEst;
  return pres;
}
```

El proper exemple és un programa que, donat un vector d'estudiants, el modifica arrodonint-ne les notes a la dècima més propera (es pot fer com a acció o com a funció).
Especificació:

```
void arrodonir_notes(vector<Estudiant> &vest);
/* Pre: cert */
/* Post: vest té les notes dels estudiants arrodonides respecte
al seu valor inicial*/
```

Implementació:

Per simplificar el disseny hem suposat, per abstracció funcional, que tenim una funció que ens arrodoneix un real. Aquesta funció la dissenyarem al final, i d'aquesta manera hem separat el problema tècnic de com fer l'arrodoniment d'un real, del nostre problema general que és arrodonir les notes d'un vector d'estudiants.

Les justificacions són similars al programa anterior. Considerem com a invariant

```
vest[0..i-1] té les notes dels estudiants arrodonides respecte
al seu valor inicial, 0 =< i <= vest.size()
```

L'invariant és la postcondició aplicada a la part tractada del vector. Justifiquem informalment el programa.

- *Inicialitzacions*. Inicialment considerem que no hi ha cap estudiant del vector tractat, això ho representem posant un `0` a la `i`. Per satisfer l'invariant no cal fer res més ja que no hi ha cap estudiant fins a `i-1`.
- *Condició de sortida*. Com que la postcondició diu que `vest[0..vest.size()-1]` ha de tenir les notes arrodonides respecte al seu valor inicial, sortirem del bucle quan `i=vest.size()`, ja que aquesta condició juntament amb l'invariant ens assegura l'anterior. Per tant, ens quedem mentre `i < vest.size()`.

- *Cos del bucle*. Com que hem d'avançar, el més natural és incrementar la `i` de manera que s'acosti al final del vector. Però abans d'incrementar la `i`, hem de satisfer l'invariant canviant `i` per `i+1`. Necessitem, per tant, que `vest` estigui modificat en `[0..i]`. Com que actualment `vest` està modificat en `[0..i-1]`, només ens cal preguntar si `vest[i]` té nota i en cas afirmatiu modificar-li la nota, un cop arrodonida. Si no té nota no cal fer res. Com a l'exemple anterior, l'accés és correcte ja que `0 <= i < vest.size()`.

  Un cop que tenim `vest` modificat amb la nota arrodonida dels estudiants en `[0..i]`, ja podem incrementar la `i` assegurant que satisfem l'invariant (a més la condició del bucle ens assegura que `i+1 <= vest.size()`).

- *Acabament*. A cada volta decreix la distància entre `vest.size()` i la `i`.

El programa descrit queda com:

```
void arrodonir_notes(vector<Estudiant> &vest)
/* Pre: cert */
/* Post: vest té les notes dels estudiants arrodonides respecte
al seu valor inicial */
{
  int numEst = vest.size();
  int i=0;

  /* Inv: vest[0..i-1] té les notes dels estudiants arrodonides
       respecte al seu valor inicial, 0 =< i <= numEst */

  while (i < numEst) {
    /* mirem si vest[i] té nota */
    if (vest[i].te_nota()) {
      /* obtenim la nota de vest[i] arrodonida */
      double aux = arrodonir(vest[i].consultar_nota());
      /* modifiquem la nota de vest[i] amb la nota arrodonida */
      vest[i].modificar_nota(aux);
    }
    ++i;
  }
}
```

Hem utilitzat la variable `aux` per augmentar la llegibilitat. Noteu que hem d'utilitzar `modificar_nota` perquè l'estudiant ja té nota. Finalment, implementem la funció `arrodonir`, considerant que en el llenguatge hi ha una funció `int(...)` que agafa la part entera d'un real.

```
double arrodonir(double r)
/* Pre: cert */
/* Post: el resultat és el valor original de r arrodonit a
la dècima més propera (i més gran si hi ha empat) */
```

```
{
  return int(10.*(r + 0.05)) / 10.0;
}
```

A continuació mostrem un exemple de justificació sobre piles. Es tracta de, donada una pila d'enters, calcular la seva alçada o nombre d'elements. La següent seria una primera especificació del problema.

```
int alcada_pila_int(stack<int> p)
/* Pre: cert */
/* Post: El resultat és el nombre d'elements de p */
```

El problema és que per saber el nombre d'elements d'una pila, hem d'anar desempilant i comptant els elements. Això suposa que el paràmetre pila es modifica en l'execució, i per tant, al final de la execució el paràmetre *p* serà una pila buida, i no serà cert que el resultat és el seu nombre d'elements. Convé per tant canviar l'especificació. D'aquesta manera podrem fer una bona justificació. Per fer una bona especificació convé distingir entre el paràmetre pila *p* que es va modificant, i la pila inicial (o valor inicial de *p*) que anomenarem *P*. Així doncs, la nova especificació (que ja es descriu així en els apunts de la tercera sessió) és:

```
int alcada_pila_int(stack<int> p)
/* Pre: p=P */
/* Post: El resultat és el nombre d'elements de P */
```

Després de presentar l'algoritme el justificarem.

```
int alcada_pila_int(stack<int> p)
/* Pre: p=P */
/* Post: El resultat és el nombre d'elements de P */
{
  int n=0;

  /* Inv: n="nombre d'elements de la part tractada de P" i
     p="elements de la pila inicial P que queda per tractar". */

  while(not p.empty()){
    ++n;
    p.pop();
  }
  return n;
}
```

Per a la funció `alçada` la justificació informal ha estat la següent:

- *Inicialitzacions*. Inicialment considerem que no hi ha cap element de `P` tractat (`p` conté tots els elements), per tant l'invariant requereix que `n` vagui 0.

- *Condició de sortida*. Com que la postcondició diu que `n` ha de ser el nombre d'elements de `P`, cal que tota la pila `P` hagi estat tractada o, equivalentment, que `p` no tingui cap element, és a dir sortirem quan `p` sigui una pila buida. Per tant, ens quedem mentre `not p.empty()`.

- *Cos del bucle*. Com que hem d'avançar, el més natural és eliminar el cim de la pila `p` (per la condició sabem que no és buida). Però abans d'eliminar-lo, hem de satisfer l'invariant canviant `p` per la pila resultat de fer `p.pop()` (novament, això no dóna error perquè `p` no és buida). Necessitem, per tant, que `n` contingui el nombre d'elements de la part tractada després de desempilar. Com que, actualment, `n` conté el nombre d'elements que no són a `p` abans de desempilar, només ens cal sumar-li `1` a `n`, per tenir el nombre d'elements que no són a `p` després de desempilar. Un cop que tenim `n` modificat, ja podem eliminar el cim de `p` assegurant que satisfem l'invariant.

- *Acabament*. A cada volta decreix la mida de `p`.

Donarem ara un exemple de justificació d'una cerca amb cues. Donada una cua d'enters i un enter, s'ha de veure si l'enter apareix a la cua. Aquí com en l'exemple anterior, donat que la cua es modifica durant l'execució, hem d'incloure a l'especificació un símbol que representi el valor inicial de l'estructura, per a que es pugui fer la justificació.

```
bool cercar_iter_cua_int(queue<int> c, int x)
/* Pre: c=C */
/* Post: El resultat ens diu si x és un element de C o no */
{
  bool ret=false;

  /* Inv: ret="l'element x està a la part tractada de C" i
     c és la part de C que queda per tractar. */

  while(not c.empty() and not ret){
    ret = (c.front() == x);
    c.pop();
  }
  return ret;
}
```

- *Inicialitzacions*. Inicialment no s'ha tractat cap element de `C`, és dir, `c` conté tots els elements, per tant l'invariant requereix posar el valor `false` a `ret`.

- *Condició de sortida*. Si `c` arriba a ser una cua buida vol dir, per l'invariant, que hem tractat tota la cua `C` i que `ret` ens diu si l'element `x` està a `C`, com es pretén a la postcondició. Però si abans de buidar `c` tenim que `ret` passa a ser `true`, voldrà dir (de nou per l'invariant) que hem trobat `x` a la part tractada de `C` i que `ret` ja ens diu que `x` està a `C` i també es compleix la postcondició. Així que sortirem o bé quan `c` no tingui cap element, és a dir sigui una cua buida, o bé quan `ret` sigui cert perquè ja hem trobem l'element `x` . Per tant, ens quedem mentre `not c.empty() and not ret`.

- *Cos del bucle*. Com que hem d'avançar, el més natural és eliminar el primer de la cua `c` (per la condició sabem que no és buida). Però abans d'eliminar-lo, hem de satisfer l'invariant canviant `c` per la cua resultat de fer `c.pop()`(novament, això es pot fer perquè `c` no és buida). Necessitem que `ret` contingui la informació de si l'element `x` es troba a la part tractada de `C` després d'avançar (és a dir, els que no són a `c` després de `c.pop()`). Com que, actualment, `ret` conté la informació sobre si `x` està entre els elements que no són a `c` abans d'avançar, ens cal veure si `x` és igual al primer element de `c`, i actualitzar el valor de `ret` depenent de si són iguals o no. Un cop que tenim `ret` modificat, ja podem eliminar el primer de `c` assegurant que satisfem l'invariant.

- *Acabament*. A cada volta decreix la mida de `c`.

Seguidament farem un exemple amb llistes. Donada una llista i un enter k, transformarem la llista sumant k a cada element de la llista original. A continuació donem l'especificació, l'algorisme i l'invariant.

```
void suma_llista_k(list<int> &l, int k)
/* Pre: l=L */
/* Post: Cada element de l és la suma de k i l'element de L
a la seva mateixa posició */
{
  list<int>::iterator it;
  it = l.begin();

  /* Inv: Cada element de l des de l'inici fins a la posició anterior a la que
  marca it són la suma de k més l'element corresponent d'L. A partir d'it fins
  al final de l, els elements són els mateixos que els seus corresponents a L. */

  while(it != l.end()){
    *it+=k;
    ++it;
  }
}
```

- *Inicialitzacions*. Després d'executar `it = l.begin()` s'ha de complir l'invariant. Això es així donat que en el cas que l'iterador estigui al principi de la llista, l'invariant ens diu que els elements de `l` són els mateixos que els de `L`, la llista inicial. En aquest cas l'invariant coincideix amb la precondició, i com que no hem tractat cap element, és cert.

- *Condició de sortida*. Com que la postcondició diu que tots els elements de `l` siguin la suma del seu valor a `L` més `k`, tindrem aquest resultat (segons l'invariant) quan l'últim element de la llista sigui l'anterior al que marca `it`, i això passarà quan `it` sigui igual a `l.end()`, l'element fictici de final de llista. Per tant, ens quedem mentre `it != l.end()`.

- *Cos del bucle*. Suposarem que a l'inici d'una iteració qualsevol del bucle es compleixen l'invariant i la condició d'entrada en el bucle. Veurem que després d'executar les instruccions del bucle es torna a fer cert l'invariant. Quan executem `*it+=k;` les posicions de l de l'inici fins a la apuntada per it tenen la k sumada als valors de la llista inicial. Després de it, els valors de l continuen sent els de la llista original. Per tant, quan executem `++it;` es torna a complir l'invariant.

- *Acabament*. A cada volta decreix la mida de la subllista entre `it` i `l.end()`.

Finalment, veurem un exemple de justificació on tenim dos bucles, un dins de l'altre. En aquests casos, cal donar un invariant per cadascun d'ells i fer justificacions separades, una per bucle.

```
typedef vector<vector<int> > Matriu;

Matriu sumar_Matriu_int(const Matriu& m1, const Matriu& m2)
/* Pre: m1 i m2 tenen les mateixes dimensions */
/* Post: la matriu resultat té les mateixes dimensions que m1 i m2, i
   cada element de la matriu resultat és la suma dels elements
   de m1 i m2 a la seva mateixa posició */
{
  int fil = m1.size();
  int col = m1[0].size();
  Matriu suma (fil, vector<int>(col));
  int i = 0;

  /* Inv. bucle extern:
     fil és el nombre de files de m1, de m2 i de suma,
     col és el nombre de columnes de m1, de m2 i de suma,
     0<=i<=fil,
     cada element de la matriu suma des de la fila 0 a la fila (i-1)
       és la suma dels elements de m1 i m2 a la seva mateixa posició
  */

  while (i<fil) {
    int j = 0;

    /* Inv. bucle intern:
     col és el nombre de columnes de m1, de m2 i de suma,
     0<=j<=col,
     cada element de la fila i de la matriu suma des de la columna 0 a
       la columna (j-1) és la suma dels elements de m1 i m2 a la seva
       mateixa posició
    */

    while (j<col) {
```

```
      suma[i][j] = m1[i][j] + m2[i][j];
      ++j;
    }
    /* cada element de la fila i de la matriu suma és la suma dels
       elements de m1 i m2 a la seva mateixa posició
    */

    ++i;
  }

  return suma;
}
```

La justificació del bucle extern és:

- *Inicialitzacions*. Per satisfer l'invariant creem la matriu `suma` de les mateixes dimensions que `m1` (iguals també que les de `m2` per la Pre), assignem a les variables `fil` i `col` el nombre de files i el nombre de columnes, respectivament, de les tres matrius, i donem el valor `0` a l'índex de files `i`, ja que compleix la restricció de valors i satisfà l'última part de l'invariant al ser `0..i-1` un interval buit de files.

- *Condició de sortida*. Com que la matriu resultat que retornem al final de la funció és `suma`, per complir la postcondició cal que tots els elements de `suma` siguin la suma dels elements de `m1` i `m2` a la seva mateixa posició. Satisfarem aquesta condició i sortirem del bucle quan s'hagin calculat totes les files de `suma`, és a dir, quan es compleixi l'invariant i la condició `i==fil`. Per tant, ens quedem mentre `i < fil` (ja que la negació d'aquesta condició i l'invariant ens garanteixen `i==fil`).

- *Cos del bucle*. Com que hem d'avançar, el més natural és incrementar la `i` de manera que s'acosti al final de la matriu. Però abans d'incrementar la `i`, hem de satisfer l'invariant canviant `i` per `i+1`. Com que l'invariant ens assegura que les files anteriors a la `i` ja han estat calculades, només necessitem, per tant, que la fila `i` de la matriu `suma` contingui la suma de les files `i` de les matrius `m1` i `m2`. El bucle intern s'ocupa de satisfer això i després ja es pot incrementar `i`.

- *Acabament*. A cada volta decreix la distància entre `fil` i la `i`, perquè incrementem `i` a cada iteració.

A continuació, la justificació del bucle intern:

- *Inicialitzacions*. La variable `col` ja conté el nombre de columnes de les tres matrius, perquè ha estat inicialitzada així en el bucle extern i no es modifica posteriorment. Donem el valor `0` a l'índex de columnes `j`, ja que compleix la restricció de valors i satisfà l'última part de l'invariant al ser `0..j-1` un interval buit de columnes.

- *Condició de sortida*. La postcondició del bucle intern és que tots els elements de la fila `i` de `suma` siguin la suma dels elements de `m1` i `m2` a la seva mateixa posició. Satisfarem aquesta condició i sortirem del bucle quan s'hagin calculat els elements de la fila `i` de `suma` per totes les columnes, és a dir, quan es compleixi l'invariant i la condició `j==col`. Per tant, ens quedem mentre `j < col` (ja que la negació d'aquesta condició i l'invariant ens garanteixen `j==col`).

- *Cos del bucle*. Com que hem d'avançar, el més natural és incrementar la `j` de manera que s'acosti al final de la fila. Però abans d'incrementar la `j`, hem de satisfer l'invariant canviant `j` per `j+1`. Com que l'invariant ens assegura que els elements de la fila `i` anteriors a la columna `j` ja han estat calculats, només necessitem, per tant, assignar a l'element `suma[i][j]` la suma de `m1[i][j]` i `m2[i][j]`. Després ja podem incrementar `j`.

- *Acabament*. A cada volta decreix la distància entre `col` i la `j`, perquè incrementem `j` a cada iteració.

Capítol 6

Programació recursiva

La forma més simple de programa recursiu és un condicional (instrucció `if`) on, si es compleix la propietat booleana de l'`if` ($c(x)$ al següent exemple), executem un conjunt d'instruccions, que representa el cas directe i que anomenem d. Si no es compleix la propietat booleana de l'`if`, llavors executem altres instruccions on una d'elles és una crida a la mateixa funció f, però amb els paràmetres de la crida "més petits", és a dir els apliquem primer una funció que aquí anomenem g. Finalment calculem el valor de sortida aplicant una altra funció, que aquí anomenem h, sobre els valors. El següent algoritme representa aquesta versió simplificada d'algoritme recursiu.

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

  if (c(x)) s= d(x);
  else{
    r= f(g(x));
    s= h(x,r);
  }
  return s;
}
```

En general, podem tenir més d'un cas directe i més d'un cas recursiu. Cada cas directe constarà de la seva condició d'entrada c_i i les instruccions directes d_i. Cada cas recursiu constarà de la seva condició d'entrada c_j i les seves funcions g_j i h_j.

6.1 Disseny de programes recursius: justificació de la seva correctesa

Com al cas dels algoritmes iteratius, per demostrar la correcció d'un algoritme recursiu hem de demostrar que si a l'inici les variables implicades (en aquest cas, els paràmetres) compleixen la

precondició, al final de l'execució compleixen la postcondició. El mateix criteri ens ha de guiar a l'hora de dissenyar aquests algoritmes.

L'estructura del disseny d'un algoritme recursiu és la següent.

- Detectar els casos senzills i resoldre'ls sense crides recursives. En definitiva, mirant l'exemple abstracte de la secció anterior, consistiria en detectar la condició booleana de l'`if` ($c(x)$) i el conjunt d'instruccions que s'han d'executar si es compleix $d(x)$. En aquests casos senzills, cal assegurar-se de que si les variables compleixen la precondició i la condició booleana del `if`, llavors després d'executar les instruccions ($d(x)$ en el exemple) es compleix la postcondició de l'especificació de la funció. Per tant, s'ha de demostrar que $Q(x) \wedge c(x) \Rightarrow R(x, d(x))$.

- Considerar el cas o casos recursius i resoldre'ls. Per poder generar les instruccions necessàries quan no es compleix la condició de l'`if`, hem de fer una crida recursiva a la mateixa funció amb un paràmetre d'entrada *menor* que l'actual. Per a que aquesta crida pugui fer-se, si x és el paràmetre d'entrada, haurem de demostrar que si $Q(x) \wedge \neg c(x)$, llavors $g(x)$ no és erroni i $g(x)$ compleix la precondició de la funció recursiva, es a dir $Q(g(x))$. A partir de aquí, suposem la hipòtesi d'inducció que és l'afirmació: si $g(x)$ compleix la precondició $Q(g(x))$, després d'executar la crida recursiva $f(g(x))$, el resultat r compleix la postcondició $R(g(x), r)$. A partir d'aquí hem d'afegir les línies de codi addicionals ($h(x,r)$ en aquest cas) que siguin necessàries per a que el codi compleixi la postcondició $R(x,s)$. En altres paraules, hem de demostrar que si x compleix la condició d'entrada i la precondició, i si després de la crida recursiva es compleix la postcondició $R(g(x), r)$ llavors també es compleix $R(x, h(x,r))$. Per tant hem de demostrar que $Q(x) \wedge \neg c(x) \wedge R(g(x), r) \Rightarrow R(x, h(x,r))$.

- Finalment, hem de raonar sobre les condiciones d'acabament del programa recursiu. És a dir, indicar quin paràmetre, paràmetres, o funció d'aquests, es va fent més petit en cada crida recursiva. En el nostre algoritme recursiu simplificat, si x és el paràmetre d'entrada, la funció del paràmetre x que decreix l'anomenem $t(x)$. D'aquesta manera verificarem que les crides recursives dintre de crides recursives són un nombre finit. Per tant, haurem de demostrar dues coses. Una, que $Q(x) \Rightarrow t(x) \in N$, per assegurar-nos de que la funció que garanteix l'acabament de la recursivitat retorna un natural. L'altre, $Q(x) \wedge \neg c(x) \Rightarrow t(g(x)) < t(x)$, per assegurar-nos de que en les crides recursives utilitzem un paràmetre menor que el paràmetre d'entrada.

Al procés descrit se l'anomena *inductiu*. Un cop generat el codi, podem demostrar la seva correcció utilitzant el principi d'inducció. El que volem demostrar és que tota crida a la funció recursiva compleix la postcondició. Veiem els passos:

- Demostrar que el codi que se executa en els casos bàsics fa que es compleixi la postcondició. Això és equivalent a demostrar $P(0)$, ja que hem fet zero crides recursives, i demostrem que complim l'especificació. Per tant, $P(0)$ correspon a $R(x, d(x))$.

- Demostrar la correcció del cas recursiu. És a dir hem de demostrar $\forall n \in N(P(n) \implies P(n+1))$ o bé $\forall y < x\ P(y) \implies P(x)$. Hem de suposar que la crida recursiva és correcta,

és a dir que si $g(x)$ és més petit que x i compleix la precondició, llavors $f(g(x))$ compleix la postcondició. A partir d'aquí, només queda demostrar que les instruccions posteriors a la crida arriben a la postcondició original. L'enunciat que consisteix en afirmar que la crida recursiva retorna el resultat desitjat (perquè s'ha cridat correctament i acaba) s'anomena *hipòtesi d'inducció*, i és un element essencial de la demostració.

Ara podem deduir que el codi és correcte utilitzant el principi d'inducció. L'únic que ens manca és demostrar que l'algoritme termina. Per a això cal trobar una funció de fita, depenent dels paràmetres , que retorni un natural i que decreixi en cada crida recursiva. En el cas més general la funció de fita pot dependre de tots els paràmetres conjuntament.

6.2 Exemples de disseny i justificació de programes recursius

Dissenyem ara la funció `piles_iguals`, que resoldrem recursivament.

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

    /* HI: ret="P1 sense l'últim element afegit i P2 sense l'últim
    element afegit són dues piles iguals" */
  }
  return ret;
}
```

La justificació informal d'aquesta funció recursiva és la següent.

- *Casos senzills.*

  1. Si alguna de les dues piles és buida, llavors les piles són iguals si les dues són buides. Per tant assignem a `ret` l'expressió `(p1.empty() and p2.empty())`.
  2. Si les dues piles no són buides, en podem consultar els cims, però si són diferents, llavors les piles no poden ser iguals. Per tant posem `ret` a fals.

- *Cas recursiu.* Si les piles no són buides i els cims són iguals, llavors les piles són iguals si coincideixen en la resta d'elements. Això ho podem obtenir, per hipòtesi d'inducció, cridant recursivament a la funció amb les piles sense l'element del cim (podem fer-ho perquè les piles no són buides). Per això assignem a `ret` el resultat de la crida recursiva.

- *Acabament*. A cada crida recursiva fem més petita la primera pila (i la segona).

Dissenyem ara una funció que, donat un arbre binari, ens calculi la seva mida, és a dir, el nombre d'elements que conté.

```
int mida(const BinTree<int>& a)
/* Pre: cert */
/* Post: El resultat és el nombre de nodes d'a */
{
  int x;
  if (a.empty()) x=0;
  else{
    int y=mida(a.left());
    int z=mida(a.right());

    /* HI: y és el nombre de nodes del fill esquerre d'a i
           z és el nombre de nodes del fill dret d'a */

    x=y+z+1;
  }
  return x;
}
```

La justificació informal d'aquesta funció recursiva és la següent.

- *Cas senzill*.

  Si l'arbre és buit té zero elements i aquest és el valor que s'ha d'assignar a `x`.

- *Cas recursiu*. Si l'arbre no és buit, llavors existeixen els seus subarbres esquerre i dret, i es poden obtenir sense error amb les operacions `left` i `right`. La mida de l'arbre serà, per tant, la suma de les mides dels seus subarbres més 1 (la contribució de l'arrel).

  Les mides dels subarbres s'obtenen per hipòtesi d'inducció, amb les crides recursives, que són correctes perquè els paràmetres s'han obtingut correctment i compleixen trivialment la precondició, que és `cert`. Per últim, només cal fer la suma i el programa queda complet.

- *Acabament*. Cada crida recursiva es fa cada vegada amb un arbre més petit.

Fem ara una funció que cerqui un Estudiant en una cua de Estudiants, donat un dni.

```
bool cerca_rec_cua_Estudiant(queue<Estudiant> &c, int i)
/* Pre: i és un dni vàlid i c=C */
/* Post: El resultat ens diu si hi ha algun estudiant amb dni i a C */
{
  bool ret;
  if(c.empty()) ret=false;
```

```
  else if (c.front().consultar_DNI() == i) ret=true;
  else {
    c.pop();
    ret=cerca_rec_cua_Estudiant(c,i);
        /* HI: ret ens diu si hi ha algun estudiant amb dni i a C
           sense el primer element */
  }
  return ret;
}
```

La justificació informal d'aquesta funció recursiva ha estat la següent.

- *Casos senzills*.
  1. Si la cua és buida no hi ha cap element i el resultat ha de ser `fals`.
  2. Si la cua no és buida, es descomposa en el seu primer element i la resta (i es poden obtenir sense error amb `front` i `pop`). Si el seu primer element té el dni que busquem, el resultat serà `cert`.
- *Cas recursiu*. Si la cua no és buida i el seu primer element no és el que busquem, tot dependrà de si aquest es troba a la resta de la cua. Això s'obté per hipòtesi d'inducció, cridant recursivament a la funció després d'avançar la cua.
- *Acabament*. A cada crida recursiva fem més petita la cua.

6.3 Immersió o generalització d'una funció

La tècnica d'immersió es desenvolupa per la necessitat de generalitzar una funció amb l'objectiu de, o bé poder fer un disseny recursiu, o bé transformar un disseny recursiu ja fet per obtenir solucions recursives alternatives més senzilles o eficients.

Una immersió és una generalització d'una funció on, o bé introduïm paràmetres addicionals, o bé resultats addicionals, o les dues coses. Donat que els paràmetres canvien, l'especificació també canviarà, i parlarem de reforçar la precondició o afeblir la postcondició. Al final, per obtenir la funció que volíem obtenir, o bé es fixen els paràmetres addicionals, o bé es rebutgen els resultats addicionals.

La millor manera d'explicar la tècnica d'immersió és mitjançant exemples. A continuació explicarem un primer tipus d'immersions, la immersió *per dades*, i en mostrarem exemples. Al capítol següent veurem exemples d'immersions *per resultats*.

6.3.1 Immersions d'especificacions per afebliment de la postcondició

Els exemples que veurem a continuació utilitzen immersions amb la finalitat de poder fer un algoritme recursiu (immersions d'especificació). En cadascun d'ells obtenim una funció recursiva

auxiliar amb paràmetres addicionals, i la seva postcondició és més feble que la de la funció que volem resoldre originalment.

A continuació dissenyarem una funció que donat un vector no buit d'enters, ens retorna la suma de tots els seus elements. La seva especificació és la següent.

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

Si volem fer un disseny recursiu d'aquesta funció, haurem de tenir una variable que retornarem com a resultat on anem sumant els valors, i en cada execució de la funció, sumar un valor i cridar recursivament a la mateixa funció. El problema és que quan fem la crida recursiva haurem de tenir un paràmetre que ens afiti la part del vector que hem de sumar. D'aquesta forma apareix un paràmetre extra amb la finalitat de resoldre el problema, i la nova especificació serà la següent.

```
int i_suma_vect_int(const vector<int> &v, int i)
/* Pre: v.size()>0, 0<=i<v.size() */
/* Post: el valor retornat és la suma de v[0..i] */
```

Hem afegit un paràmetre enter addicional `i`, amb la finalitat d'afitar els valors del vector a sumar. En aquest cas ens serveix per indicar que el valor retornat és la suma de tots els elements de `v` fins la posició `i`. Noteu que la nova postcondició és més feble que la original, perquè hi ha `v.size()` parells d'enters que la compleixen (cada valor en `[0.. v.size()-1]` i la corresponent suma parcial), mentre que aquella només es compleix amb un d'aquests parells (`v.size()-1` i la suma total). La implementació de la funció és la següent.

```
int i_suma_vect_int(const vector<int> &v, int i)
/* Pre: v.size()>0, 0<=i<v.size() */
/* Post: el valor retornat és la suma de v[0..i]  */
{
  int suma;
  if(i==0) suma=v[0];
  else {
    suma=i_suma_vect_int(v,i-1);
    /* HI: "suma" és la suma  de v[0..i-1] */
    suma+=v[i];
  }
  return suma;
}
```

Justificarem aquest codi recursiu de la mateixa manera que els exemples anteriors. Noteu que tota funció obtinguda fent una immersió per afebliment de la postcondició tindrà la mateixa estructura que aquesta, per tant la seva justificació s'assemblarà força.

- *Cas senzill.* Si `i=0`, la postcondició requereix obtenir la suma de `v[0..0]`, que òbviament és `v[0]` i serà el que retornarem.

- *Cas recursiu.* Si `i!=0`, la suma de `v[0..i]` que ens demanen a la postcondició equival a la suma de `v[0..i-1]` més `v[i]`. Per hipòtesi d'inducció, la primera part ja l'hem obtingut amb la crida recursiva i val `suma`. En conseqüència, el resultat a retornar serà `suma+v[i]`. La crida és vàlida perquè si `i!=0` i apliquem la precondició ens queda que `0<i` o equivalentment `0<=i-1`. L'accés a `v[i]` és correcte per la precondició.
- *Acabament.* A cada crida recursiva el valor d'`i` es fa més petit.

Finalment hem de resoldre el problema original amb una crida a la funció `i_suma_vect_int`. Ho fem amb el valor `v.size()-1` al lloc d'`i`, que compleix la precondició perquè `v.size()>0`.

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

A continuació presentem un exemple on es realitza una cerca dicotòmica en un vector no buit i ordenat d'enters. L'especificació és la següent:

```
int cerca_vect_int(const vector<int> &v, int n)
/* Pre: v.size()>0, v està ordenat de menor a major */
/* Post: Si n es troba a v, llavors el valor retornat és una posició on
es troba l'element n dins del vector v. Si n no es troba a v, llavors el
valor retornat és -1. */
```

Si volem fer una implementació recursiva d'aquest algoritme, necessitarem introduir dos paràmetres addicionals amb la finalitat d'indicar l'inici i el final de la zona on cercarem l'element. A continuació presentem l'especificació i implementació de la funció d'immersió.

```
int i_cerca_vect_int(const vector<int> &v, int n, int esq, int dre)
/* Pre: v.size()>0, v està ordenat de menor a major,
        0 <= esq <= v.size(), -1 <= dre < v.size(),  esq <= dre+1  */
/* Post: Si n es troba a v[esq...dre], llavors el valor retornat és
una posició de v entre els valors esq i dre on es troba el valor n.
Si n no es troba a v[esq...dre], el valor retornat és -1 */
{
  int posicio;
  if(dre < esq) posicio= -1;
  else
  {
    int mig = (dre+esq)/2;
    if (v[mig]==n) posicio=mig;
    else
    {
```

```
      if (v[mig]< n) posicio= i_cerca_vect_int(v, n, mig+1, dre);
            /* HI: Si n es troba a v[mig+1...dre], llavors el valor
            retornat és una posició de v[mig+1...dre] on es troba n.
            Si n no es troba a v[mig+1...dre], el valor retornat és -1. */
      else posicio=i_cerca_vect_int(v, n, esq, mig-1);
            /* HI: Si n es troba a v[esq...mig-1], llavors el valor
            retornat és una posició de v[esq...mig-1] on es troba n.
            Si n no es troba a v[esq...mig-1], el valor retornat és -1. */
    }
  }
  return posicio;
}
```

A continuació mostrem la justificació informal de la funció `i_cerca_vect_int`.

- *Casos senzills.*

  1. Si `dre < esq` llavors l'interval és buit, `n` no es troba a `v[esq..dre]` i per tant la postcondició ens diu que el valor a retornar ha de ser -1.
  2. Si `v[mig]=n`, llavors la postcondició ens diu que el valor a retornar ha de ser `mig`.

- *Casos recursius.* Si l'interval del vector no és buit i `n` no es troba a `mig`, llavors `n` està al subvector `v[esq..mig-1]` o al subvector `v[mig+1..dre]` o no hi és al subvector `v[esq..dre]`. Donat que `v` està ordenat de menor a major, si `v[mig] < n` hem de cercar al subvector `v[mig+1..dre]`, i sinó al subvector `v[esq..mig-1]`. En qualsevol dels dos casos, la hipòtesi d'inducció ens diu que la crida recursiva corresponent retornarà el valor demanat en la postcondició, que serà una posició on es troba `n` al subvector o `-1` si no hi és.

  Hem de demostrar que l'accès a `v[mig]` és correcte i que es compleixen les precondicions de les crides recursives, en particular, les desigualtats que afecten als paràmetres enters. Hi ha un parell de propietats fonamentals, que són: si Pre i $esq <= dre$ llavors $esq <= mig$ i $mig <= dre$. A partir d'elles, les demés surten fàcilment.

  La demostració de totes dues es basa concretament en que si $0 <= esq <= dre$, llavors

  a) $esq = 2 * esq/2 \leq (esq + dre)/2 = mig$

  b) $mig = (esq + dre)/2 \leq 2 * dre/2 = dre$

- *Acabament.* A cada crida recursiva el subvector a tractar és més petit, és a dir, decreix la distància $(dre + 1 - esq)$. Aquesta expressió és natural perquè $esq \leq dre + 1$ forma part de la precondició. Noteu que $(dre - esq)$ no és una bona funció de fita en aquest cas, perquè no garanteix un valor natural en tots els casos que compleixen la precondició. El decreixement s´obté a partir de les dues propietats anteriors.

- $dre+1-(mig+1) < dre+1-esq \equiv dre-mig < dre+1-esq \equiv esq < mig+1 \equiv esq \leq mig$, que és cert com hem vist abans, per a)
- $(mig-1)+1-esq < dre+1-esq \equiv mig-esq < dre+1-esq \equiv mig < dre+1 \equiv mig \leq dre$, que és cert com hem vist abans, per b)

Finalment presentem el codi de la funció que volíem resoldre, que consisteix en una crida a la funció d'immersió fixant els paràmetres addicionals a la primera i última posició del vector. Noteu com les tres desigualtats de la precondició de la crida es compleixen trivialment.

```
int cerca_vect_int(const vector<int> &v, int n)
/* Pre: v.size()>0, v està ordenat de menor a major */
/* Post: Si n es troba a v, llavors el valor retornat és una posició on
es troba l'element n dins del vector v. Si n no es troba a v, llavors el
valor retornat és -1. */
{
  return i_cerca_vect_int(v, n, 0, v.size()-1);
}
```

6.3.2 Immersions d'especificació per reforçament de la precondició

Aquest mètode consisteix en obtenir una funció auxiliar amb la precondició original reforçada amb una versió feble de la postcondició original, de manera que s'escurci el camí cap a la consecució de l'objectiu que marca la postcondició. Aquesta versió afeblida de la postcondició que s'afegeix a la precondició ha de poder complir-se fàcilment, ja que la primera crida recursiva està obligada a complir-la.

Com exemple en presentarem un de la secció anterior, i donarem una solució diferent. Utilitzarem el problema de sumar els elements d'un vector.

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

Una manera de resoldre el problema consisteix en afegir un paràmetre a la funció que contingui ja una suma parcial dels elements del vector, de forma que cada execució de la nova funció afegeixi un altre element a la suma parcial. D'aquesta manera, a la precondició se li afegeix part de la informació de la postcondició, i es fa més fàcil arribar a aquesta. Per últim, necessitarem un paràmetre extra que ens indiqui fins a quina posició del vector hem calculat la suma parcial. La funció d'immersió és la següent:

```
int ii_suma_vect_int(const vector<int> &v, int i, int suma_parcial)
/* Pre: v.size()>0, 0<=i<=v.size(), suma_parcial = suma de v[0..i-1] */
/* Post: el valor retornat és la suma de tots els elements del vector v */
{
  int suma;
```

```
  if(i==v.size()) suma=suma_parcial;
  else suma= ii_suma_vect_int(v,i+1,suma_parcial+v[i]);
  return suma;
}
```

Aquest codi recursiu també es justifica de la mateixa manera que els exemples anteriors. Noteu que tota funció obtinguda fent una immersió per reforçament de la precondició tindrà la mateixa estructura que aquesta, per tant la seva justificació s'assemblarà força.

- *Cas senzill*. Si `i=v.size()`, per la precondició tenim a `suma_parcial` la suma de tot el vector, que és el que ens demanen a la postcondició i serà el que retornarem.
- *Cas recursiu*. Si `i!=v.size()`, sabem per la precondició de la funció que `i<v.size()`, per tant `i+1<=v.size()`. La resta de la precondició de la crida recursiva queda `suma_parcial = suma dels elements de v[0..i]` o equivalentment `suma_parcial = suma dels elements de v[0..i-1] + v[i]`. Per la precondició de la funció, la primera part la tenim a la `suma_parcial` original, de manera que el nou valor de `suma_parcial` haurà de ser `suma_parcial+v[i]`. L'accés a `v[i]` és correcte ja que la precondició diu que `0<=i` i, com ja hem vist, `i<v.size()`.
- *Acabament*. A cada crida recursiva el valor de `v.size()-i` es fa més petit.

Finalment resolem el problema que ens havíem plantejat de la següent manera:

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

Com veiem, en la crida a la funció `ii_suma_vect_int` des de `suma_vect_int`, es compleix trivialment la precondició, donat que la suma parcial que considerem és la d'un subvector buit.

6.3.3 Exemples d'immersions amb accions

Considerem el següent exercici: donat un enter més gran que zero, s'han d'obtenir tots els seus divisors naturals menors que ell en una pila ordenada en ordre decreixent des del cim. Per exemple, si `n=24`, la pila ha de ser `[Cim] 12 8 6 4 3 2 1 [Últim]`.

Especifiquem el problema (a partir d'ara, quan diem "divisor" volem dir "divisor natural"). Recordeu que les funcions recursives que retornen objectes complexos tenen una ineficiència oculta per culpa de les assignacions i les destruccions, de manera que farem servir accions per aquest cas.

```
void divisors(int n, stack<int> &p)
/* Pre: n>0, p és buida */
/* Post: p conté els divisors d'n menors que ell, en ordre decreixent des del cim */
```

Noteu que hem de recòrrer al disseny per immersió, ja que si intentem un disseny recursiu directe veurem que no hi ha una manera senzilla de trobar una crida recursiva (amb un paràmetre menor que `n`) que ens ajudi a obtenir una part dels divisors que poguem completar amb d'altres instruccions.

En canvi, podem plantejar la següent immersió per afebliment de la postcondició, de manera anàloga als exemples anteriors

```
void i_divisors(int n, stack<int> &p, int m)
/* Pre: n>=m>0,  p és buida */
/* Post: p conté els divisors d'n menors que m, en ordre decreixent des del cim */
```

Comprovem que l'operació original es pot programar mitjantçant una crida a l'auxiliar

```
void divisors(int n, stack<int> &p)
/* Pre: n>0, p és buida */
/* Post: p conté els divisors d'n menors que ell, en ordre decreixent des del cim */
{
  i_divisors(n,p,n);
}
```

La crida a `i_divisors` és correcta perquè es compleix la precondició: com que `n>0`, llavors `n>=n>0`; també es compleix que p és buida. A més, la postcondició de la crida garanteix que `p` "conté els divisors d'n menors que n (és a dir, ell mateix), en ordre decreixent des del cim", que es el mateix que la post original.

Ara falta dissenyar el codi de `i_divisors`. Podem aplicar el mateix esquema que les anteriors immersions per afebliment de la postcondició: un cas directe on es tracta la situació més senzilla (en aquest cas, `m=1`) i un cas recursiu que faci una crida amb decreixement del valor d'`m`:

```
void i_divisors(int n, stack<int> &p, int m)
/* Pre: n>=m>0, p és buida */
/* Post: p conté els divisors d'n menors que m, en ordre decreixent des del cim */
{
  if (m > 1) {
    i_divisors(n,p,m-1);
    /* HI:  p conté els divisors d'n menors que m-1, en ordre decreixent des del cim */
    if (n%(m-1) == 0) p.push(m-1);
  }
}
```

La justificació es similar a la dels exemples anteriors del mateix tipus (com ja havíem dit), amb la particularitat de tractar-se d'una acció.

- *Cas senzill*. Si `m=1`, no hi ha cap divisor d'`n` menor que `m`, per tant no s'ha d'afegir res a `p`.
- *Cas recursiu*. Si `m>1`, la col.lecció de divisors d'`n` menors que `m` està formada pels divisors d'`n` menors que `m-1`, més `m-1` si aquest és divisor d'`n`. Obtenim la primera part a `p` amb la crida, per hipòtesi d'inducció, i li empilem `m-1` en cas necessari.

  La crida és vàlida perquè si `m>1` i apliquem la precondició ens queda que `n>=m-1>0`. La divisió per `m-1` és vàlida per la mateixa raó.
- *Acabament*. A cada crida recursiva el valor d'`m` es fa més petit.

Passem ara a obtenir una immersió per enfortiment de la precondició per aquest mateix problema. La idea és que l'auxiliar rebi una pila amb una part dels divisors ja calculats.

```
void ii_divisors(int n, stack<int>& p, int m)
/* Pre: n>=m>0, p conté els divisors d'n menors que m, en ordre decreixent des del cim */
/* Post: p conté els divisors d'n en ordre decreixent des del cim */
```

Comprovem que realment és tracta d'una immersió de l'original

```
void divisors(int n, stack<int> &p)
/* Pre: n>0, p és buida */
/* Post: p conté els divisors d'n menors que ell, en ordre decreixent des del cim */
{
  ii_divisors(n,p,1);
}
```

La crida a `ii_divisors` és correcta perquè es compleix la precondició: com que `n>0`, llavors `n>=1>0`. A més, no hi ha cap divisor d'`n` més petit que 1, per tant `p` ha de ser buida, però això es compleix també per la pre original.

La postcondició de la crida és la mateixa que la post original (com a tota immersió per enfortiment de la precondició ben definida).

Ara falta dissenyar el codi de `ii_divisors`. Podem aplicar el mateix esquema que les anteriors immersions per enfortiment de la precondició: un cas directe (en aquest cas, `m=n`) on es els paràmetres ja porten calculats els resultats definitius i un cas recursiu que ens apropa al cas directe:

```
void ii_divisors(int n, stack<int>& p, int m)
/* Pre: n>=m>0, p conté els divisors d'n menors que m, en ordre decreixent des del cim */
/* Post: p conté els divisors d'n en ordre decreixent des del cim */
{
  if (m<n){
    if (n%m == 0) p.push(m);
    ii_divisors(n,p,m+1);
  }
}
```

Aquest codi recursiu també es justifica de la mateixa manera que els exemples anteriors del mateix tipus.

- *Cas senzill*. Si `m=n`, per la precondició tenim a `p` tots els divisors de `n` menors que ell, que és el que ens demanen a la postcondició i, per tant, no cal fer res.

- *Cas recursiu*. Si `m<n`, sabem per la precondició de l'operació que `n>m>0`, per tant `n>=m+1>0`. La resta de la precondició de la crida recursiva requereix que `p` contingui la col.lecció de divisors d'`n` menors que `m+1` o, equivalentment, que contingui la col.lecció de divisors d'`n` menors que `m`, més `m` a dalt de tot si és divisor de `n`. Per la precondició de l'operació, la primera part la tenim a la `p` original, de manera que només falta empilar-li `m` en cas necessari. Com que `m>0`, la divisió és vàlida.

- *Acabament*. A cada crida recursiva el valor de `n-m` és un natural que es fa més petit.

6.3.4 Relació entre algoritmes recursius lineals finals i algoritmes iteratius

Diem que un algoritme és recursiu *lineal*, si cada crida recursiva genera de manera directa, és a dir dins de la mateixa activació, solament una (o cap) crida recursiva. Una funció recursiva lineal és *final* si l'última instrucció que s'executa és la crida recursiva i el resultat de la funció és el resultat que s'ha obtingut de la crida recursiva, sense cap modificació.

Per exemple, si mirem les dues maneres de resoldre el problema de sumar els elements d'un vector d'enters de les dues seccions anteriors, les dues funcions recursives amb immersió són lineals, però una d'elles té recursivitat final i l'altre no. En la solució de `i_suma_vect_int` per afebliment de la postcondició el valor retornat no és directament el resultat de la crida recursiva, perquè hem de sumar `v[i]`; per tant, no tenim recursivitat final. En canvi, en la solució per reforçament de la precondició, `ii_suma_vect_int`, el valor que es retorna és directament el que retorna la crida recursiva; en aquest cas sí que tenim recursivitat final.

Una propietat que compleixen moltes vegades els algoritmes amb recursivitat final és que la postcondició de la crida recursiva és la mateixa que la postcondició de la funció. En aquest cas diem que la funció te *postcondició constant*.

Existeix una manera directa i molt simple de transformar un algoritme recursiu lineal final amb postcondició en un algoritme iteratiu. Agafem el següent esquema general de recursivitat lineal final:

```
Tipus2 f(Tipus1 x)
/* Pre: Q(x), x=X */
/* Post: R(X,s) */
{
  Tipus2 s;

  if (c(x)) s= d(x);
  else{
    s= f(g(x));
```

```
  }
  return s;
}
```

La condició del else (la del `if` negada) és la condició d'entrada del `while`. Les instruccions internes del `while` són les del else sense la crida a la funció recursiva; és a dir, $g(x)$. Finalment el valor a retornar de la funció iterativa es calcula utilitzant les instruccions del `if`, és a dir $d(x)$. Per tant l'esquema iteratiu queda:

```
Tipus2 f_iter(Tipus1 x)
/* Pre: Q(x) */
/* Post: R(x,s) */
{
  Tipus2 s;

  while(not c(x)) x=g(x);

  s= d(x);

  return s;
}
```

Tornem ara l'exemple de sumar els elements d'un vector d'enters. El codi recursiu lineal final era:

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

Aquest exemple té algunes diferències respecte al esquema recursiu de partida. A l'esquema, suposàvem que teníem només un paràmetre d'entrada, la `x`. En el nostre cas tenim tres `v`, `i` i `suma_parcial`. Per tant, el conjunt d'instruccions `g(x)` és ara un conjunt d'instruccions equivalents per als tres paràmetres. Si mirem la crida recursiva `ii_suma_vect_int(v, i + 1, suma_parcial + v[i])` veurem que la `g(x)` es correspon a sumar-li `v[i]` a `suma_parcial`, i a sumar-li `1` a la `i`. El paràmetre *v* resta constant. Amb la transformació descrita obtenim:

```
int ii_suma_vect_int_iter(const vector<int> &v, int i, int suma_parcial)
/* Pre: v.size()>0, 0<=i<=v.size(), suma_parcial = suma de v[0..i-1] */
/* Post: el valor retornat és la suma de tots els elements del vector v */
```

```
{
  int suma;

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

  return suma;
}
```

Noteu que la precondició no només es compleix al principi, sinó també al entrar i al sortir de cada volta del bucle. Això vol dir que val com a invariant. També podem aprofitar la demostració del decreiximent.

Si, a més, la funció recursiva vé d'una immersió per reforçament de la precondició d'una altra, com en aquest cas, on `ii_suma_vect_int` és una immersió per reforçament de la precondició de `suma_vect_int`, podem obtenir una versió iterativa de l'original, tan sols transformant els paràmetres d'immersió en variables locals, inicialitzades amb els valors de la crida a la immersió. Com a simplificació final, la variable que guarda els càlculs intermitjos es pot substituir per la del resultat final a tot arreu, donat que al final se l'assignem de totes maneres.

```
int suma_vect_int_iter(const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el valor retornat és la suma de tots els elements del vector v */
{
  int suma=0;
  int i=0;
  /* Inv: 0<=i<=v.size(), suma = suma de v[0..i-1] */
  while(i != v.size()){
    suma += v[i];
    ++i;
  }

  return suma;
}
```