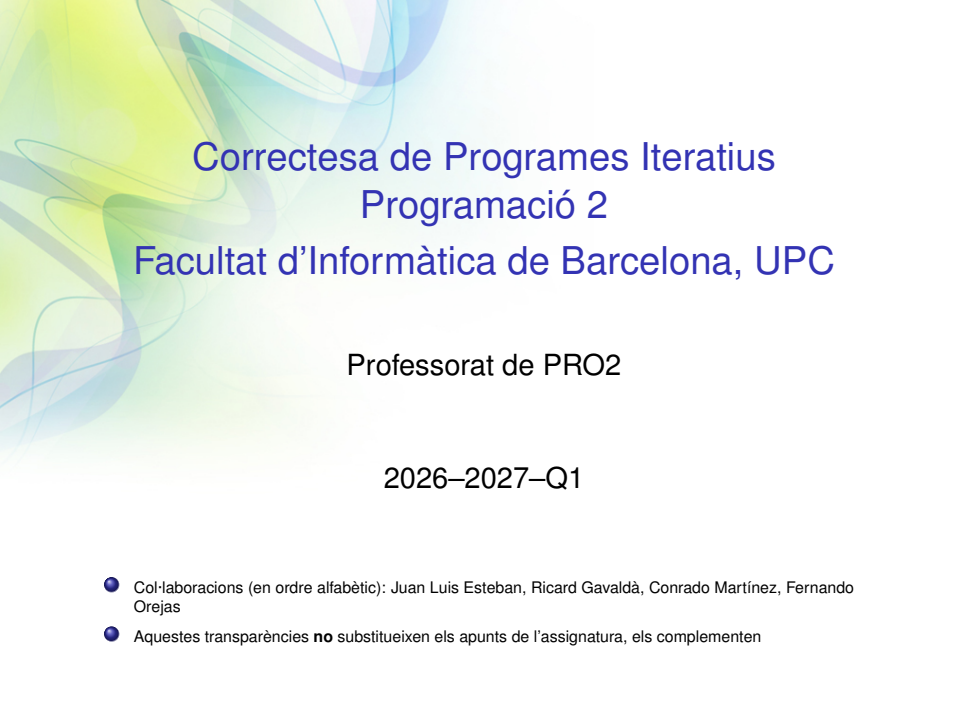




Correctesa de Programes Iteratius

Programació 2

Facultat d'Informàtica de Barcelona, UPC

Professorat de PRO2

2026–2027–Q1

- Col·laboracions (en ordre alfabètic): Juan Luis Esteban, Ricard Gavaldà, Conrado Martínez, Fernando Orejas
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

- 
- 1 Correctesa de programes
 - 2 Estats i assercions
 - 3 Correctesa de programes iteratius
 - 4 Disseny inductiu

Correctesa d'un programa

Definició:

L' **estat** d'un programa en un punt determinat de la execució ve donat pel **valor** de totes les **variables actives** en aquell punt.

```
// Estat = ( x = 3, y = 7, ... )  
++x;  
// Estat = ( x = 4, y = 7, ... )
```

Correctesa d'un programa

Definició: Correcció d'un programa

Si l'estat inicial del programa o funció satisfà la **Precondició**, llavors el programa acaba en un nombre finit de passos i l'estat final satisfà la **Postcondició**

Correctesa d'un programa

Definició: Correcció d'un programa

Si l'estat inicial del programa o funció satisfà la **Precondició**, llavors el programa acaba en un nombre finit de passos i l'estat final satisfà la **Postcondició**

- Com sabem que un programa és correcte?
- Només podem fer un nombre finit (i petit) de proves
- Raonament genèric sobre els estats del programa

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Demostració de correctesa?

Ho he provat i ...

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

Demostració de correctesa?

Ho he provat i ...
... amb 5 i 3 dona 125

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

Ho he provat i ...

... amb 5 i 3 dona 125

... amb 0 i 100 dona 0

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Ho he provat i ...

... amb 5 i 3 dona 125

... amb 0 i 100 dona 0

... amb -4 i 2 dona 16

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Ho he provat i ...

... amb 5 i 3 dona 125

... amb 0 i 100 dona 0

... amb -4 i 2 dona 16

... amb 4 i -2 no cal provar (no
es compleix la Pre)

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Ho he provat i ...

... amb 5 i 3 dona 125

... amb 0 i 100 dona 0

... amb -4 i 2 dona 16

... amb 4 i -2 no cal provar (no
es compleix la Pre)

... per tant, és correcte!

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Ho he provat i ...

... amb 5 i 3 dona 125

... amb 0 i 100 dona 0

... amb -4 i 2 dona 16

... amb 4 i -2 no cal provar (no
es compleix la Pre)

... per tant, és correcte!

Nombre finit (petit) de casos

$\neq$

Tots els casos

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

Demostració de correctesa?

“Inicialitzem p a 1 (el producte de 0 factors).

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

“Inicialitzem p a 1 (el producte de 0 factors).

Lavors, anem multiplicant p per x i decrementant y en cada pas.

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

“Inicialitzem p a 1 (el producte de 0 factors).

Lavors, anem multiplicant p per x i decrementant y en cada pas.

Repetim fins que $y = 0$, i llavors ja hem acabat.

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

“Inicialitzem p a 1 (el producte de 0 factors).

Lavors, anem multiplicant p per x i decrementant y en cada pas.

Repetim fins que $y = 0$, i llavors ja hem acabat.

Ja es veu que a p tindrem X^Y .”

Demostració de correctesa?

```
// Pre:  $x = X$  and  $y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    p = p * x;
    y = y - 1;
}
// Post:  $p = X^Y$ 
```

“Inicialitzem p a 1 (el producte de 0 factors).

Lavors, anem multiplicant p per x i decrementant y en cada pas.

Repetim fins que $y = 0$, i llavors ja hem acabat.

Ja es veu que a p tindrem X^Y .”

Llegir el programa

≠

Dir per què satisfà la seva
espec

Com ho fem, doncs?

- Raonament genèric sobre tots els estats possibles
- L'eina principal és la **inducció**
- En programes **recursius**, aplicada directament
- En programes **iteratius**, amagada en els **invariants**

- 
- 1 Correctesa de programes
 - 2 Estats i assercions**
 - 3 Correctesa de programes iteratius
 - 4 Disseny inductiu

Com raonar sobre programes

- Recordem: estat d'un programa = valors de totes les variables

`(x = 10, y = -5, b = true)`

`(x = 10, y = -15, b = false)`

Com raonar sobre programes

- Recordem: estat d'un programa = valors de totes les variables

`(x = 10, y = -5, b = true)`

`(x = 10, y = -15, b = false)`

- Asserció: Descripció d'un conjunt d'estats

$P(x, y, b) = "b == (x + y > 0)"$

Com raonar sobre programes

- Recordem: estat d'un programa = valors de totes les variables

`(x = 10, y = -5, b = true)`

`(x = 10, y = -15, b = false)`

- Asserció: Descripció d'un conjunt d'estats

$P(x, y, b) = "b == (x + y > 0)"$

- El comentari `// P` o `/* P */` en un programa vol dir
“en aquest punt l'estat del programa compleix P”

Com raonar sobre programes

- Recordem: estat d'un programa = valors de totes les variables

$(x = 10, y = -5, b = \text{true})$

$(x = 10, y = -15, b = \text{false})$

- Asserció: Descripció d'un conjunt d'estats

$P(x, y, b) = "b == (x + y > 0)"$

- El comentari `// P` o `/* P */` en un programa vol dir “en aquest punt l'estat del programa compleix P”
- La Precondició (Pre) és l'asserció que l'estat inicial ha de satisfer
- La Postcondició (Post) és l'asserció que ha de ser certa per l'estat final; altrament el programa **no satisfà l'especificació**

Com raonar sobre programes

- **Mètode:** Anotarem el programa amb assercions que descriuen els estats en diferents punts, i argumentarem que cada anotació està ben feta

Com raonar sobre programes

- **Mètode:** Anotarem el programa amb assercions que descriuen els estats en diferents punts, i argumentarem que cada anotació està ben feta
- Un programa és correcte si és cert que

```
/* Pre */ programa /* Post */
```

Com raonar sobre programes

Donada una assertió P , $P(x \leftarrow E)$ és l'assertió resultant de **reemplaçar simultàniament** les aparicions d' x en l'assertió P per l'expressió E , e.g., $P = "x \geq 5"$,
 $P(x \leftarrow y + 3) = "y + 3 \geq 5"$

- Assignació:

$$/* \ P(x \leftarrow E) \ */ \ x = E \ /* \ P \ */$$

- Composició seqüencial:

Si $/* \ P_1 \ */ \ S1 \ /* \ Q_1 \ */$ és correcte, $/* \ P_2 \ */ \ S2$
 $/* \ Q_2 \ */$ és correcte i $Q_1 \implies P_2$ llavors

$$/* \ P_1 \ */ \ S1; \ S2 \ /* \ Q_2 \ */$$

és correcte.

Com raonar sobre programes

- Composició alternativa/condicional:

Si $/* P \wedge B */$ S1 $/* Q */$ és correcte,

i $/* P \wedge \neg B */$ S2 $/* Q */$ és correcte llavors

```
/* P */  
if (B) S1  
else S2  
/* Q */
```

és correcte.

- 
- 1 Correctesa de programes
 - 2 Estats i assercions
 - 3 Correctesa de programes iteratius**
 - 4 Disseny inductiu

Correctesa d'un bucle

Esquema bàsic

```
// Pre:  $P$ 
inicialitzacions;
// Pre (del bucle):  $P'$ 
while (B) {
    COS
}
// Post (del bucle):  $Q'$ 
tractament final;
// Post:  $Q$ 
```

L'invariant: Concepte i ús

- Invariant: Una assertió I que és certa després de qualsevol nombre d'iteracions (inclòs 0); per tant cal que

$$P' \implies I$$

- A més, quan el bucle acaba, implica la Post:

$$I \wedge \neg B \implies Q'$$

L'invariant: Concepte i ús

- Invariant: Una assertió I que és certa després de qualsevol nombre d'iteracions (inclòs 0); per tant cal que $P' \implies I$
- A més, quan el bucle acaba, implica la Post:
 $I \wedge \neg B \implies Q'$
- Que l'assertió I és un invariant es demostra per inducció sobre el nombre d'iteracions i : s'ha de complir

Esquema bàsic

```
//  $I \wedge B$   
cos del bucle  
//  $I$ 
```

L'invariant: Concepte i ús

- Invariant: Una assertió I que és certa després de qualsevol nombre d'iteracions (inclòs 0); per tant cal que $P' \implies I$
- A més, quan el bucle acaba, implica la Post:
 $I \wedge \neg B \implies Q'$
- Que l'assertió I és un invariant es demostra per inducció sobre el nombre d'iteracions i : s'ha de complir

Esquema bàsic

```
//  $I \wedge B$   
cos del bucle  
//  $I$ 
```

- Finalment, cal demostrar (potser usant l'invariant I) que el bucle segur que acaba
- Trobar i explicitar l'invariant d'un bucle és molt bona documentació d'un bucle: explica per què funciona!

Demostració d'acabament

- **Funció de fita:** Una funció f sobre les variables que diuen quantes iteracions queden com a molt
- Ha de tenir valor enter no negatiu: per a qualsevol estat del programa $f \geq 0$
- Cal que decreixi (al menys en 1) a cada iteració
- Si fem una iteració més, segur que $f > 0$

Passos

0 Inventar un invariant I i una funció de fita f

Demostrar que:

1 Les inicialitzacions del bucle estableixen l'invariant:

$$P' \Rightarrow I$$

2 Si es compleix l'invariant i s'entra en el bucle, al final d'una iteració torna a complir-se l'invariant: $/* I \wedge B */ \text{ cos } /* I */$

$$/* I */$$

3 L'invariant i la *negació* de la condició d'entrada al bucle impliquen la Postcondició: $I \wedge \neg B \Rightarrow Q'$

4 La funció de fita decreix a cada iteració:

$$/* I \wedge B \wedge f = F */ \text{ cos } /* I \wedge f < F */$$

5 Si entrem un cop més al bucle, la funció de fita és estrictament positiva: $I \wedge B \Rightarrow f > 0$

Exemple: Exponenciació

```
// Pre:  $x = X \wedge y = Y \geq 0$   
int p = 1;  
while (y > 0) {  
    p = p * x;  
    y = y - 1;  
}  
// Post:  $p = X^Y$ 
```

- Invariant:

$$x = X \wedge y \geq 0 \wedge p \cdot X^y = X^Y$$

- Fita: y

Exemple: Exponenciació

```
// Pre:  $x = X \wedge y = Y \geq 0$ 
int p = 1;
while (y > 0) {
    if (y % 2 == 0) { x = x*x; y = y/2; }
    else { p = p * x; y = y - 1; }
}
// Post:  $p = X^Y$ 
```

- Invariant:

$$y \geq 0 \wedge p \cdot x^y = X^Y$$

- Fita: y

Una mica de notació

- Donat un vector v de talla n i dos enters i, j amb $0 \leq i, j < n$, $v[i..j]$ denota el subvector entre les components i i j ; si $i > j$ llavors $v[i..j]$ és un subvector buit
- Donada una llista L i dos iteradors $it1$ i $it2$ tals que $it2$ apunta a un element posterior a l'apuntat per $it1$ (o $it2 == it1$) llavors $L[it1 : it2)$ denota la subllista de L el primer element de la qual és l'apuntat per $it1$ i l'últim element és el predecessor de l'element apuntat per $it2$
- $L[: it) \equiv L[L.begin(), it)$
- $L[it :) \equiv L[it, L.end())$
- $L[:] \equiv L$

Exemple: Suma d'un vector

```
// Pre: cert
double suma(const vector<double>& v) {
    int i = 0;
    double s = 0;
    while (i < v.size()) {
        s += v[i];
        ++i;
    }
    return s;
}
// Post: el resultat es la suma de tots els elements de v
```

- Invariant:

$$0 \leq i \leq v.size() \wedge s = \text{suma de } v[0..i - 1]$$

- Fita: $v.size() - i$

Exemple: Cerca un element en una llista

```
// Pre: cert
bool pertany(const list<double>& l, double x) {
    list<double>::const_iterator it = l.begin();
    bool trobat = false;
    while (it != l.end() and not trobat) {
        if (*it == x) trobat = true;
        ++it;
    }
    return trobat;
}
// Post: el resultat indica si x apareix en l
```

Exemple: Cerca un element en una llista

```
// Pre: cert
bool pertany(const list<double>& l, double x) {
    list<double>::const_iterator it = l.begin();
    bool trobat = false;
    while (it != l.end() and not trobat) {
        if (*it == x) trobat = true;
        ++it;
    }
    return trobat;
}
// Post: el resultat indica si x apareix en l
```

- Invariant:

it apunta a un element de *l* o *it* = *l.end()*, i
trobat = “*x* pertany a *l*[: *it*)”

- Funció de fita: nombre d'elements de la subllista *l*[*it* :)

Exemple: variació de cerca lineal

```
// Pre: cert
// Post: retorna la posició en v de'un string x,
// o bé -1 si cap string de v és igual a x
int posicio(string x, const vector<string>& v) {
    int i = 0;
    bool trobat = false;
    while (i < v.size() and not trobat) {
        if (v[i] == x) trobat = true;
        else ++i;
    }
    if (trobat) return i;
    else return -1;
}
```

- Invariant:

$$0 \leq i \leq v.size() \wedge x \notin v[0..i-1]$$

i a més

$$trobat \implies "i < v.size() \wedge v[i] = x"$$

- Funció de fita:

$$v.size() - i - \llbracket trobat \rrbracket, \quad \llbracket P \rrbracket = 1 \text{ si } P \text{ és cert, i } \llbracket P \rrbracket = 0 \text{ si } P \text{ és fals}$$

Exemple: Sumar k a una llista

Problema: donada una llista i un enter k , transformar-la en una altra resultant de sumar k a cada element de la llista original.

```
// Pre:  $l = [a_1, \dots, a_n]$ 
void suma_k(list<int>& l, int k) {
    list<int>::iterator it;
    it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
// Post:  $l = [a_1 + k, \dots, a_n + k]$ 
```

- Invariant: $\exists i : 1 \leq i \leq n + 1 : l[it:] = [a_i, \dots, a_n] \wedge l[:it] = [a_1 + k, \dots, a_{i-1} + k] \quad (*)$
- Funció de fita: nombre d'elements de $l[it:]$

(*) Quan $i = n + 1$ entendrem que $l = [a_i, \dots, a_n] = []$; de manera semblant quan $i = 1$ llavors

$l_{aux} = [a_1 + k, \dots, a_{i-1} + k] = []$

Exemple: Revessar una llista

```
// Pre:  $l = [a_1, \dots, a_n]$ 
void revessa(list<int>& l) {
    list<int> laux;
    while (not l.empty()) {
        laux.insert(laux.begin(), *(l.begin()));
        l.erase(l.begin());
    }
    //  $laux = [a_n, \dots, a_1]$ 
    l = laux;
}
// Post:  $l = [a_n, \dots, a_1]$ 
```

- Invariant:

$$\exists i : 1 \leq i \leq n + 1 : l = [a_i, \dots, a_n] \wedge laux = [a_{i-1}, \dots, a_1]$$

- Funció de fita: `l.size()`

Exercici: Directament sobre `l`, evitant la llista auxiliar

Exemple: cerca dicotòmica

```
// Pre:  $0 \leq esq = E \wedge D = dre < v.size() \wedge esq \leq dre + 1$   
//        $\wedge v$  està ordenat creixentment  
// Post:  $x$  és a  $v[E..D]$  si i només si  
//        $0 \leq esq < v.size() \wedge v[esq] = x$   
int posicio(double x, const vector<double>& v,  
            int esq, int dre) {  
    while (esq < dre) {  
        int pos = (esq + dre)/2;  
        if (v[pos] < x) esq = pos + 1;  
        else dre = pos;  
    }  
    return esq;  
}
```

- Invariant: $x \in v[E..D] \Leftrightarrow x \in v[esq..dre] \wedge \dots$
- Fita: $dre - esq$. Millor encara: $f = \log_2(dre - esq + 1)$

Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: retorna el nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
    int n = 0;
    int i = 0;
    while (i < v.size()) {
        int j = i-1;
        while (j >= 0 and v[j] != v[i]) --j;
        if (j < 0) ++n;
        ++i;
    }
    return n;
}
```

Exemple: comptar nombre d'elements diferents

```
//  $n = N \wedge i < v.size()$   
int j = i-1;  
while (j >= 0 and v[j] != v[i]) --j;  
if (j < 0) ++n;  
//  $n = N + 1$  si  $v[i] \notin v[0..i-1] \wedge$   
//  $n = N$  si  $v[i] \in v[0..i-1]$ 
```

- Invariant (del bucle intern sobre j):

$$v[i] \notin v[j+1..i-1] \wedge j \geq -1$$

- Fita: $j + 1$

Exemple: comptar nombre d'elements diferents

```
// Pre: cert
// Post: diferents(v) = nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
    int n = 0;
    int i = 0;
    while (i < v.size()) {
        int j = i-1; ...; if (j < 0) ++n;
        // n = nombre d'elements diferents a v[0..i]
        ++i;
    }
    return n;
}
```

- Invariant (del bucle extern sobre i):

$$0 \leq i \leq v.size() \wedge n = \text{nombre d'elements diferents a } v[0..i-1]$$

- Fita: $v.size() - i$

Exemple: comptar nombre d'elements diferents (2)

Amb una funció separada:

```
// Pre:  $[a..b] \subseteq [0..v.size() - 1]$ 
// Post: retorna cert sii  $x \in v[a..b]$ 
template <typename T>
bool apareix(const T& x, const vector<T>& v, int a, int b);

// Pre: cert
// Post: retorna el nombre d'elements diferents a v
int diferents(const vector<elem>& v) {
    int n = 0;
    int i = 0;
    while (i < v.size()) {
        if (not apareix(v[i], v, 0, i-1)) ++n;
        ++i;
    }
    return n;
}
```

Exemple: comptar nombre d'elements diferents (2)

Invariant:

$$0 \leq i \leq v.size() \wedge n = \text{nombre d'elements diferents en } v[0..i - 1]$$

- Es fa servir l'especificació d'`apareix` per verificar
- Podem verificar independentment la correcció de `diferents` i d'`apareix` $\Rightarrow$ **Modularitat!**

Invariants “gràfics”

Sovint podem donar una representació gràfica esquemàtica d'un invariant (i en general d'una asserció), molt més intuïtiva i senzilla d'entendre.

Exemple: Donat un vector v d'enters, escriu un procediment que reorganitzi els seus continguts de manera que els elements parells apareguin abans que els elements senars.

```
// Pre: cert  
// Post: ???  
void reorganitza_parells_senars(vector<int>& v);
```

Invariants “gràfics”

```
// Pre:  $v = V$   
// Post: ???  
void reorganitza_parells_senars(vector<int>& v);
```

Invariants “gràfics”

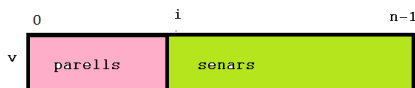
- Postcondició formal: v és una permutació de V ,
 $n = v.size() \geq 0$, existeix una posició i tal que totes les posicions precedents estan ocupades per números parells

$$\exists i : 0 \leq i < n : \left(\forall j : 0 \leq j < i : v[j] \bmod 2 = 0, \right.$$

i tal que la posició i totes les que vénen al seu darrera estan ocupades per números senars

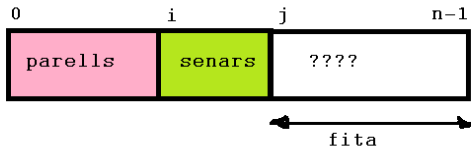
$$\forall j : i \leq j < n : v[j] \bmod 2 = 1)$$

- Postcondició “gràfica”:



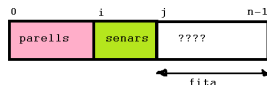
Invariants “gràfics”

```
// Pre:  $v = V$   
// Post: ???  
void reorganitza_parells_senars(vector<int>& v);
```



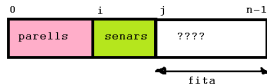
Invariants “gràfics”

```
// Pre:  $v = V$   
// Post: ...  
void reorganitza_parells_senars(vector<int>& v) {  
    int i = 0; int j = 0;  
    while (j < v.size()) {  
        if (v[j] % 2 == 0) {  
            swap(v[i], v[j]); ++i;  
        }  
        ++j;  
    }  
}
```



Invariants “gràfics”

```
// Pre:  $v = V$   
// Post: ...  
void reorganitza_parells_senars(vector<int>& v) {  
    int i = 0;  
    for (int j = 0; j < v.size(); ++j)  
        if (v[j] % 2 == 0) {  
            swap(v[i], v[j]); ++i;  
        }  
}
```



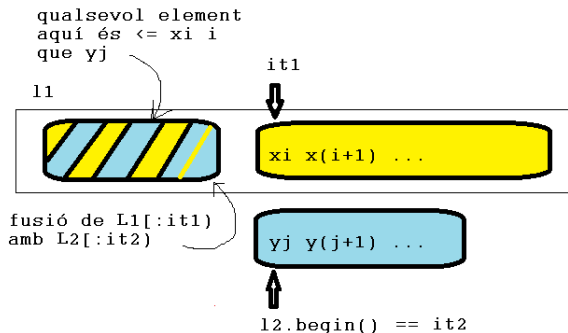
Invariants “gràfics”

```
// Pre:  $l1 = L1 = [x_1, \dots, x_m] \wedge l2 = L2 = [y_1, \dots, y_n] \wedge$   
//        $x_1 \leq x_2 \leq \dots \leq x_m \wedge y_1 \leq y_2 \leq \dots \leq y_n$   
// Post:  $l1 = [z_1, \dots, z_{m+n}] \wedge l2 = []$   
//        $\wedge \{z_1, \dots, z_{m+n}\} = \{x_1, \dots, x_m\} \cup \{y_1, \dots, y_n\}$   
//        $\wedge z_1 \leq z_2 \leq \dots \leq z_{m+n}$   
template <typename T>  
void fusionar(list<T>& l1, list<T>& l2);
```

N.B. Suposem que hi ha un ordre $\leq$ definit entre elements de tipus T

Invariants “gràfics”

- Fita: $\text{mida de } l1[it1:] + l2.size()$
- Invariant:



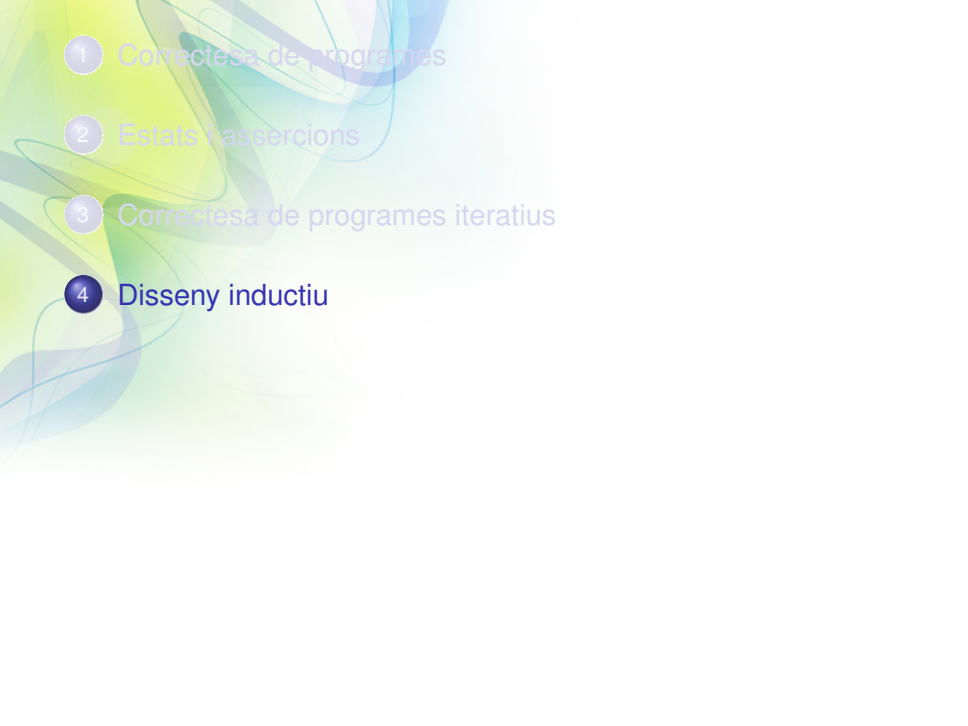
Invariants “gràfics”

```
template <typename T>
void fusionar(list<T>& l1, list<T>& l2) {
    list<T>::iterator it1 = l1.begin();
    list<T>::iterator it2 = l2.begin();
    while (it1 != l1.end() and it2 != l2.end()) {
        if (*it1 <= *it2) ++it1;
        else {
            l1.insert(it1,*it2);
            it2 = l2.erase(it2);
        }
    }
    ...
}
```

Invariants “gràfics”

```
template <typename T>
void fusionar(list<T>& l1, list<T>& l2) {
    list<T>::iterator it1 = l1.begin();
    list<T>::iterator it2 = l2.begin();
    while (it1 != l1.end() and it2 != l2.end()) { ... }
    // it1 = l1.end() i l1[:it1) és la fusió de L1 amb L2[:it2)
    // o it2 = l2.end() i l1[:it1) és la fusió de L1[:it1) amb L2

    // it1 = l1.end() i l1 és la fusió de L1 amb L2[:it2)
    // o it2 = l2.end() i l1 és la fusió de L1 amb L2
    l1.splice(l1.end(), l2);
}
```

- 
- 1 Correctesa de programes
 - 2 Estats i assercions
 - 3 Correctesa de programes iteratius
 - 4 Disseny inductiu**

Disseny inductiu o derivació

Invertim el procés: de la “justificació” a l'algorisme

Donades Pre i Post, proposar:

- un invariant que les generalitzi les dues
- deduïm les inicialitzacions que, amb la Pre, estableixin l'invariant
- un cos del bucle que mantingui l'invariant
- una condició del bucle que, negada, i junt amb l'invariant impliqui la Post

Exemple: Part entera de l'arrel quadrada

```
// Pre:  $x = X \geq 0$   
      ????  
// Post:  $a = \lfloor \sqrt{X} \rfloor$  (part entera per defecte  
//           de l'arrel quadrada de  $X$ )
```

Exemple: Part entera de l'arrel quadrada

```
// Pre:  $x = X \geq 0$   
???  
// Post:  $a^2 \leq X < (a+1)^2$   
// Invariant:  $I = (x = X \geq 0) \wedge (a^2 \leq x < b^2) \wedge (0 \leq a < b)$ 
```

Exemple: Part entera de l'arrel quadrada

La nostra idea és utilitzar *bisecció* per anar reduint l'interval $[a, b)$ que conté l'arrel d' X , buscant el punt mig de l'interval i veient quin subinterval ha de contenir $\lfloor \sqrt{X} \rfloor$

```
// Pre:  $x = X \geq 0$ 
int a = ?;
int b = ?;
while (B) {
    int c = (a+b)/2;
    if (?) a = c;
    else b = c;
}
// Post:  $a^2 \leq X < (a+1)^2$ 
```

Exemple: Part entera de l'arrel quadrada

```
// Pre:  $x = X \geq 0$ 
int a = ?; int b = ?;
while (B) {
    int c = (a+b)/2;
    if (?) a = c;
    else b = c;
}
// Post:  $a^2 \leq X < (a+1)^2$ 
```

- $I \wedge b \leq a + 1 \implies a^2 \leq x < b^2 = (a + 1)^2$, és a dir, $a = \lfloor \sqrt{x} \rfloor$ si $b \leq a + 1$ (ja que $a < b$ sempre, llavors tindrem $b = a + 1$); per tant la condició del bucle ha de ser la negació: $b > a + 1$
- Com $0 \leq x = X < (X + 1)^2$, fent $a = 0$; i $b = x + 1$; establim l'invariant
- Si entrem al bucle tindrem $a < c < b$, i si $x < c^2$ llavors fent $b = c$; es torna a satisfer l'invariant; de manera similar, si $c^2 \leq x$ llavors fer $a = c$; reestablirà l'invariant.

Exemple: Part entera de l'arrel quadrada

```
// Pre:  $x = X \geq 0$ 
int a = 0;
int b = x+1;
//  $I \equiv x = X \geq 0 \wedge a^2 \leq x < b^2 \wedge 0 \leq a < b$ 
while (b > a+1) {
  //  $I \wedge b > a+1$ 
  // Fita =  $b - a - 1 = F > 0$ 
    int c = (a+b)/2;
    if (c * c <= x) a = c;
    else b = c;
  //  $I \wedge \text{Fita} = b - a - 1 < F$ 
}
//  $I \wedge \neg B \implies a^2 \leq X < b^2 = (a+1)^2$ 
// Post:  $a^2 \leq X < (a+1)^2 \equiv a = \lfloor \sqrt{X} \rfloor$ 
```

El nostre algorisme és el conegut mètode de la bisecció per a trobar arrels de funcions contínues

Exemple: Part entera de l'arrel quadrada

L'algorisme acaba sempre i ho podem demostrar amb la funció de fita $b - a - 1$.

- 1 $I \implies b - a - 1 \geq 0$; l'invariant diu que $0 \leq a < b$ sempre i com a i b són enters $b - a - 1 \geq 0$
- 2 Quan la fita és 0 $b - a - 1 = 0$ tenim que $b = a + 1$ i no entrarem en cap nova iteració del bucle; si la condició del bucle és certa llavors la fita és > 0 , per tant,
 $I \wedge B \implies b - a - 1 > 0$
- 3 Si $I \wedge B \wedge b - a - 1 = F$ llavors després de l'execució del cos del bucle tenim que $b' - a' - 1 = (a + b)/2 - a - 1$ o $b' - a' - 1 = b - (a + b)/2 - 1$, on a' i b' són els valors de les variables després de completar una iteració. Així, en acabar una iteració la fita és $b' - a' - 1 = b/2 - a/2 - 1 = (b - a - 1)/2 - 1/2 < (b - a - 1)/2 < b - a - 1 = F$

Nombre de paires ordonnées

Donat un vector v , comptar quantes paires $(v[i], v[i + 1])$ conté tals que $v[i] < v[i + 1]$.

```
int paires_ordenades(const vector<int>& v);
```

Nombre de parelles ordenades

Invariant:

$$I = \left(v.size() = 0 \wedge p = 0 \right) \vee \left(1 \leq i \leq v.size() \wedge \right. \\ \left. p = \text{nombre de parelles ordenades en } v[0..i-1] \right)$$

- La variable i recorre el vector
- La variable p compta les parelles ordenades vistes fins al moment durant el recorregut

Nombre de parelles ordenades

1. Com establim l'invariant al principi?

```
int p = 0;  
int i = 1; // v[0..0] no conté cap parella
```

Nombre de parelles ordenades

2. Quan acabem? I acabem satisfent la Post? La darrera parella que cal comprovar és $(v[n-2], v[n-1])$, amb $n = v.size()$. Si la nostra condició de sortida és $i = n$, hem provat totes les parelles $(v[j-1], v[j])$ amb $j < n$, inclòs el cas $j = n-1$, que és la $(v[n-2], v[n-1])$ i ja hem acabat. Podem posar de condició del bucle $i < v.size()$, que cobreix bé els casos $n = 0$ i $n = 1$, i que implica sortir quan $i = v.size()$.

Nombre de parelles ordenades

3. Com avancem mantenim l'invariant?

Volem avançar fent $i = i + 1$. Posem que ho fem com a darrera instrucció del cos del bucle.

Per tant just abans d'incrementar s'hauria de complir que hem provat totes les parelles $(v[j - 1], v[j])$ amb $j \leq i$.

La que falta doncs és la parella amb $j = i$, que és $(v[i - 1], v[i])$

```
//  $I \wedge i < v.size()$   
if ( $v[i-1] < v[i]$ )  $p = p + 1$ ;  
 $i = i + 1$ ;  
//  $I$ 
```

Nombre de parelles ordenades

```
int parelles_ordenades(const vector<int>& v) {  
    int p = 0;  
    for (int i = 1; i < v.size(); ++i)  
        // Inv: ( $v.size() = 0$  i  $p = 0$ ) ó  
        //      ( $p$  = nombre de parelles ordenades en  $v[0..i-1]$   
        //      i  $1 \leq i \leq v.size()$ )  
        // Fita: 0 si  $v.size() = 0$ ,  $v.size() - i$  altrament  
        if (v[i-1] < v[i]) ++p;  
    return p;  
}
```

Ordenació

- 1 Ordenar un vector v : Deixar-lo de manera que
“per a tot i , $0 \leq i < v.size() - 1$, $v[i] \leq v[i + 1]$ ”
- 2 Invariant:

“ $v[0..j]$ està ordenat” ...

Fixem-nos que quan $j = v.size() - 1$ ja tenim tot el vector ordenat.

Ordenació

“ $v[0..j]$ està ordenat i tots els elements de $v[0..j]$ són més petits o iguals que els de $v[j + 1..v.size() - 1]$ ”

Si incrementem j :

- $v[0..j]$ segueix ordenat! Per què?
- Però no és cert que “ $v[0..j]$ és més petit que $v[j + 1..v.size() - 1]$ ”
- Només és cert si $v[j + 1]$ era un element mínim de $v[j + 1..v.size() - 1]$
- Que hem de fer?
 - Buscar un valor mínim de $v[j + 1..v.size() - 1]$
 - Intercanviar-lo amb $v[j + 1]$
 - Incrementant j es reestableix l'invariant

Aquest és l'algorisme d'**ordenació per selecció**.

Exercici: Si no posem la segona part de l'invariant, deriveu la **ordenació per inserció**

Exemple: Prefix de suma màxima d'un vector

```
// Pre:  $v.size() > 0$   
// Post: el resultat és  $i$  tal que  $v[0] + \dots + v[i]$  és màxima  
//       $i - 1 \leq i < v.size()$   
int psm(const vector<double>& v);
```

Exemple: Prefix de suma màxima d'un vector

```
// Post: el resultat és  $i$  tal que  $v[0] + \dots + v[i]$  és màxima  
//       $-1 \leq i < v.size()$ 
```

Què vol dir “és màxima”? Sigui $n = v.size()$ i definim $S_i = \sum_{k=0}^i v[k]$. Per conveni, $S_{-1} = 0$. Llavors estem dient que el resultat és el valor i , $-1 \leq i < n$, tal que

$$S_i = \max\{S_k \mid -1 \leq k < n\}$$

Exemple: Prefix de suma màxima d'un vector

Això suggereix que la nostra solució faci un bucle sobre j amb l'invariant:

```
// Inv:  $S_i = \max\{S_k \mid -1 \leq k < j\} \wedge -1 \leq i < j \leq n$ 
```

Exemple: Prefix de suma màxima d'un vector

```
// Inv:  $-1 \leq i < j \leq v.size()$ ,  
//       $v[0] + \dots + v[i] \geq v[0] + \dots + v[k]$  per a tot  $k \in [-1..j-1]$ ,  
//       $sum = v[0] + \dots + v[j-1]$ ,  
//       $sum_i = v[0] + \dots + v[i]$ 
```

Exemple: Prefix de suma màxima d'un vector

```
// Pre: v.size() > 0
int psm(const vector<double>& v) {
    int i = -1; int j = 0;
    double sum = 0; double sumi = 0;
    while (j < v.size()) {
        sum += v[j];
        if (sum > sumi) {
            sumi = sum;
            i = j;
        }
        ++j;
    }
    return i;
}
// Post: el resultat és i tal que  $v[0] + \dots + v[i]$  és màxima
//        $i - 1 \leq i < v.size()$ 
```

Exemple: Events acabats

- Donat un `Calendari` retornem el número d'`Events` que han acabat abans o en una `Data` donada
- Especificació:

```
// Pre: cert  
// Post: el resultat és el número  
// d'Events en C que han acabat abans o  
// en una Data donada d  
int events_acabats(const Calendari& C, const Data& d);
```

Exemple: Events acabats

- Tindrem una potscondició P' després del bucle:
“`nr_ev_acabats` és el número d'Events del calendari C que acaben abans o en la Data d ”.
- Per obtenir P' recorrerem el calendari C comptant els events amb data final $\leq d$; id serà l'EventID d'un cert event del calendari (parlarem simplement de “l'event id ”, en comptes de “l'event amb EventID id ”)
- Tindrem el conveni que l'event amb EventID $id = \text{Calendari::event_final}()$ és estrictament major que qualsevol altre event del calendari en el ordre cronològic d'events, és a dir, la seva data d'inici és major que qualsevol altra data
- En altres paraules, l'event `event_final()` és l'(n+1)-èssim event del calendari, sent
 $n = C.\text{num_events}()$

Exemple: Events acabats

Invariant:

```
// Inv: nr_ev_acabats = nombre d'events començats abans o en  
// la data d entre els events que precedeixen a l'event amb  
// EventID id, en ordre cronològic;  
// si id = Calendari::event_final() llavors nr_events_acabats és el  
// nombre d'events del calendari C acabats abans  
// o en la data d
```

Fita: $n + 1 -$

$|\{\text{events de } C \text{ que precedeixen a l'event } id\}|$

Exemple: Events acabats

```
// Pre: cert
int events_acabats(const Calendari& C, const Data& d) {
    int nr_ev_acabats = 0;
    Calendari::EventID id = C.primer_event();
    // Inv: nr_ev_acabats = nombre d'Events acabats abans o en
    // la data d entre els events precedint a l'event id
    while (id != Calendari::event_final()) {
        if (C.consulta_event(id).data_final() <= d)
            ++nr_ev_acabats;
        // nr_ev_acabats = nombre d'Events acabats abans o en la
        // data d entre els events precedint id i el propi event id
        id = C.seguint_event(id);
    }
    // Inv  $\wedge id = Calendari::event\_final() \implies P'$ 
    // Fita:  $n + 1 - |\{\text{events de } C \text{ que precedeixen a l'event } id\}|$ 
    return nr_ev_acabats;
}
// Post: retorna el número d'Events del calendari C acabats
// abans o en la Data d
```

Exemple: Events acabats

- Suposem una certa iteració en la qual $id \neq \text{Calendari}::\text{event_final}()$, i hauriem d'examinar l'event id per veure si ja està acabat o no en la data d .
- Si la data d'inici d_{ini} de l'Event id és $> d$ llavors aquest Event no pot estar acabat abans de la data d , i $nr_ev_acabats$ no ha de canviar
- I com que el calendari C està ordenat cronològicament **cap** dels Events següents a l'Event id pot estar acabat ja que tots s'inicien a la data d_{ini} o més tard

Exemple: Events acabats

- Tenim un nou invariant Inv' ; a l'antic invariant Inv li podem afegir que tots els events que precedeixen a l'event id tenen data d'inici $\leq d$
- Per tant

$$Inv' \wedge id \neq \text{Calendari}::\text{event_final}() \\ \wedge C.\text{consulta_event}(id).\text{data_inici}() > d \implies P'$$

i el bucle pot acabar

- Si n_d és el número d'Events que **comencen** abans de o en la Data d en el calendari C , llavors $0 \leq n_d \leq n$ i podem utilitzar la fita $n_d + 1 -$
 $|\{\text{events de } C \text{ que precedeixen l'event } id\}|$

Exemple: Events acabats

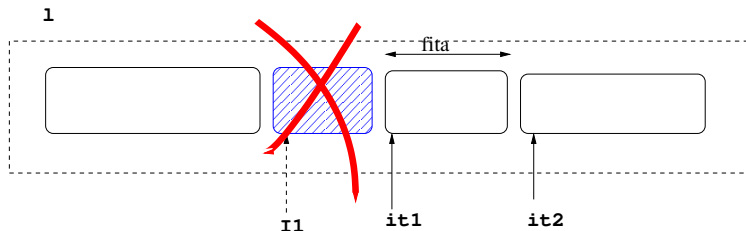
```
// Pre: cert
int events_acabats(const Calendari& C, const Data& d) {
    int nr_ev_acabats = 0;
    Calendari::EventID id = C.primer_event();
    // Inv'
    while (id != Calendari::event_final()
           and C.consulta_event(id).data_inici() <= d) {
        // Inv'  $\wedge id \neq \text{event\_final}()$ 
        //  $\wedge$  la data d'inici de l'Event és  $\leq d$ 
        if (C.consulta_event(id).data_final() <= d)
            ++nr_ev_acabats;
        id = C.seguint_event(id);
        // Inv'
    }
    // Inv'  $\wedge (id = \text{event\_final}() \vee \text{la data d'inici de l'event } id \text{ és } > d) \implies P'$ 
    // Fita:  $n_d + 1 - |\{\text{events de } C \text{ que precedeixen a l'event } id\}|$ 
    return nr_ev_acabats;
}
// Post: retorna el número d'Events del calendari  $C$  acabats
// abans o en la Data  $d$ 
```

Exemple: Eliminació d'una subllista

```
// Pre:  $it1 = I_1$  i  $it2 = I_2$  apunten elements de la llista  $L$  i
//  $it2$  apunta a un element igual o posterior a l'element
// apuntat per  $it1$ ;  $l = L$ 
// Post: La llista  $l$  conté tots els elements d' $L$ , excepte
// els que hi havia entre l'element originalment apuntat per  $it1$  i
// el predecessor de l'element originalment apuntat per  $it2$ , i.e.,
//  $l = L[:I_1] \cdot L[I_2:]$ ;  $\cdot$  denota la concatenació
// de llistes
template <class T>
void elimina_subllista(list<T>& l,
    list<T>::iterator it1, list<T>::iterator it2);
```

Exemple: Eliminació d'una subllista

Com a invariant proposem que la subllista entre el valor original I_1 d' $it1$ i el predecessor de l'element al qual apunta $it1$ ha sigut eliminada; quan $it1 = it2$ tindrem la postcondició:



Exemple: Eliminació d'una subllista

Invariant “formal”:

```
// Inv:  $it1$  i  $it2 = I_2$  apunten elements de la llista  $L$  i  
//       $it2$  apunta a un element igual o posterior a l'element  
//      apuntat per  $it1$ ,  $l = L[I_1) \cdot L[it1 :)$ 
```

- L'invariant és compleix des del primer moment
- Si $it1 = it2$, llavors l'invariant implica la postcondició
- Si l'invariant és cert i $it1 \neq it2$, llavors eliminant l'element apuntat per $it1$ i avançant $it1$ reestablim l'invariant
- La funció de fita és la talla de $L[it1 : it2)$; la precondition (i l'invariant) garanteixen que és ≥ 0 , i amb cada iteració disminuirà en una unitat

Exemple: Eliminació d'una subllista

```
// Pre:  $it1 = I_1$  i  $it2 = I_2$  apunten elements de la llista  $L$  i
//  $it2$  apunta a un element igual posterior a l'element apuntat
// per  $it1$ ;  $l = L$ 
template <class T>
void elimina_subllista(list<T>& l,
    list<T>::iterator it1, list<T>::iterator it2) {

    while (it1 != it2)
        // Inv:  $it1$  i  $it2 = I_2$  apunten elements de la
        // llista  $L$  i  $it2$  apunta a un element
        // igual o posterior a l'element apuntat
        // per  $it1$ ,  $l = L[: I_1) \cdot L[it1 :)$ 

        it1 = l.erase(it1);
    }
// Post: La llista  $l$  conté tots els elements d' $L$ , excepte
// els que hi havia entre  $I_1$  i el predecessor de  $I_2$ , i.e.,
//  $l = L[: I_1) \cdot L[I_2 :)$ 
```