Correctesa d'Algorismes Recursius
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Professorat de PRO2

Primavera 2024

- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Estratègies de construcció de programes recursius

Transformar la definició rebuda del que volem calcular en una definició recursiva (si es pot). Després traduir a codi.
Exemple:
La funció Factorial. $n! = 1 \cdot 2 \cdot 3 \cdots n$.
Es transforma en:

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

Traducció a codi:

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

Estratègies de construcció de programes recursius

Un altre exemple:
La potència. x^y és x multiplicat per si mateix y vegades.
Definició recursiva:

$$x^y = \begin{cases} 1 & \text{si } y = 0 \\ x \cdot x^{y-1} & \text{si } y > 0 \end{cases}$$

Traducció a codi:

```
// Pre: x > 0 ∧ y ≥ 0
// Post: retorna x^y
int potencia(int x, int y) {
    if (y == 0) return 1;
    else return x*potencia(x,y-1);
}
```

Estratègies de construcció de programes recursius

Cal identificar:

- Un o més casos base: Valors de paràmetres en què podem satisfer la Post amb càlculs directes
- Un o més casos recursius: Valors de paràmetres en què podem satisfer la Post si tinguessim el resultat per a alguns paràmetres x' "més petits" que x.
- La construcció de l'algorisme recursiu va de la mà de la seva demostració de correctesa.

Correctesa d'un algorisme recursiu

A demostrar:

- Per a tot valor dels paràmetres que satisfaci Pre,
- l'algorisme acaba - nombre finit de crides recursives
- i acaba satisfent $Post$

O el que és el mateix:

- $Terminacio$: Si la precondició es compleix abans de la execució, llavors l'algorisme acaba.
- $Correccio$: Si la pre es compleix abans de la execució, llavors la post es compleix després de l'execució.

Correcció de programes recursius

Fonament de la Correcció: el Principi d'Inducció.

Suposem que volem demostrar que tots els nombres naturals $\mathbb{N}$ tenen una certa propietat P.

Pases:

- Demostra que $0 \in P$
- Demostra que per a tot $n \in \mathbb{N}$, si $n \in P$, llavors $n + 1 \in P$.
- Obtens que per a tot $n \in \mathbb{N}$, $n \in P$.

Fonament de la Terminació: Tota seqüència decreixent de nombres enters no negatius és finita.

Acabament: nombre finit de crides recursives

Formalització:

- Triem una funció de mida $|\cdot|$ dels paràmetres que sempre té valor enter
- Demostrem: Si $|x| = 0$ l'algorisme tracta x amb un cas base $\implies$ cap crida recursiva
- Demostrem: Cada crida recursiva fa decrèixer la mida dels paràmetres, i.e., si la funció $\mathtt{F}$ amb paràmetre x fa la crida recursiva $\mathtt{F}(x')$ llavors $|x'| < |x|$

Fonament: tota seqüència decreixent d'enters no negatius és finita

Correctesa d'un algorisme recursiu

- A demostrar: Si el paràmetre x satisfà la precondició llavors el resultat satisfà la postcondició (una funció d'x).
- Quan x és un cas base, no hi ha crida recursiva. És un cas senzill i directe.
- Si no es un cas base hem the trobar la Hipòtesi d'Inducció (HI). Dirà que per paràmetres de tamany menor que x, si es compleix la pre, després de la crida recursiva es complirà la post.
- Suposant la HI, veure que les demés instruccions del cas recursiu fan que es compleixi la post per a x.

Correctesa d'un algorisme recursiu

Cas recursiu:

- Si x no és un cas base, apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició $\texttt{Pre}(x')$ i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició $\texttt{Post}(x')$"

Correctesa d'un algorisme recursiu

Cas recursiu:

- Si x no és un cas base, apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició $\texttt{Pre}(x')$ i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició $\texttt{Post}(x')$"

- Hem de demostrar que qualsevol crida recursiva $\texttt{F}(x')$ quan $|x| > 0$ compleix: 1) $\texttt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors l'H.I.

Correctesa d'un algorisme recursiu

Cas recursiu:

- Si x no és un cas base, apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició $\texttt{Pre}(x')$ i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició $\texttt{Post}(x')$"

- Hem de demostrar que qualsevol crida recursiva $\texttt{F}(x')$ quan $|x| > 0$ compleix: 1) $\texttt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors l'H.I.

- Aplicant l'H.I. deduïm que després d'una crida recursiva $\texttt{Post}(x')$.

Correctesa d'un algorisme recursiu

Cas recursiu:

- Si x no és un cas base, apliquem l'hipòtesi d'inducció:

  H.I. = "Si x' compleix la precondició $\texttt{Pre}(x')$ i $|x'| < |x|$ llavors l'algorisme acaba en temps finit i es compleix la postcondició $\texttt{Post}(x')$"

- Hem de demostrar que qualsevol crida recursiva $\texttt{F}(x')$ quan $|x| > 0$ compleix: 1) $\texttt{Pre}(x')$; 2) $|x'| < |x|$. Podem aplicar llavors l'H.I.

- Aplicant l'H.I. deduïm que després d'una crida recursiva $\texttt{Post}(x')$.

- Finalment cal demostrar que l'estat al qual s'arriba just després o fent alguns càlculs addicionals satisfà $\texttt{Post}(x)$

Factorial

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

- Acabament: mida = $|n| = n$, valor enter.
  $|n| = 0$ és cas base.
  Per $|n| = n > 0$ es fa la crida `fact(n-1)`,
  $|n - 1| = n - 1 < |n| = n$

Factorial

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

- Acabament: mida = $|n| = n$, valor enter.
  $|n| = 0$ és cas base.
  Per $|n| = n > 0$ es fa la crida `fact(n-1)`,
  $|n-1| = n-1 < |n| = n$
- Correcció: si $n = 0$ retornem 1 ($0! = 1$).
  Si $n > 0$ es fa crida amb `fact(n-1)`. $n - 1 \geq 0$,
  $|n-1| < |n|$ i podem aplicar H.I.= fact(n-1)=(n-1)!.

Factorial

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

- Acabament: mida = $|n| = n$, valor enter.
  $|n| = 0$ és cas base.
  Per $|n| = n > 0$ es fa la crida `fact(n-1)`,
  $|n-1| = n-1 < |n| = n$
- Correcció: si $n = 0$ retornem 1 ($0! = 1$).
  Si $n > 0$ es fa crida amb `fact(n-1)`. $n - 1 \geq 0$,
  $|n-1| < |n|$ i podem aplicar H.I.= fact(n-1)=(n-1)!.
- Correcció: asumint la H.I., `fact(n)` retorna
  $n \cdot (n-1)! = n!$

Potència ràpida

```
// Pre: x > 0 ∧ y ≥ 0
// Post: retorna x^y
int potencia(int x, int y);
```

Observem que

$$x^y = \begin{cases} 1 & \text{si } y = 0 \\ (x^2)^{\lfloor y/2 \rfloor} & \text{si } y \text{ és parell} \\ x \cdot (x^2)^{\lfloor y/2 \rfloor} & \text{si } y \text{ és senar} \end{cases}$$

Potència ràpida

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x*x,y/2);
    /* HI1: el resultat és (x^2)^⌊y/2⌋ */
    else return potencia(x*x,y/2);
    /* HI2: el resultat és (x^2)^⌊y/2⌋ */
}
```

- Acabament: podem agafar $|y| = y$.
- Acabament: també $|y| = \lceil 1 + \log_2(y) \rceil$, la longitud de la representació binària de y més 1. Compleix que $\lceil 1 + \log_2(y/2) \rceil = \lceil \log_2(y) \rceil < \lceil 1 + \log_2(y) \rceil$.

Potència ràpida

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else if (y%2 == 1) return x*potencia(x*x,y/2);
    else return potencia(x*x,y/2);
}
```

- Correcció: si $y = 0$ llavors retornem $x^0 = 1$.
- Si $y > 0$, es fa la crida recursiva `potencia(x*x,y/2)`. Com $x > 0$, tenim $x^2 > 0$. Com $y > 0$, $y/2 \geq 0$. $|y/2| < |y|$. Es pot aplicar $H.I.$: `potencia(x*x,y/2)` $= (x^2)^{\lfloor y/2 \rfloor}$.
- Si y és parell, per H.I. `potencia(x*x,y/2)` retorna $(x^2)^{\lfloor y/2 \rfloor} = x^{2 \times y/2} = x^y$. Resultat correcte.
- Si $y =$ és senar, per H.I. `potencia(x*x,y/2)` retorna $(x^2)^{\lfloor y/2 \rfloor} = x^{2 \times \lfloor y/2 \rfloor} = x^{y-1}$; llavors la funció retorna $x \cdot x^{y-1} = x^y$. Resultat correcte.

Potència ràpida

Exercici: Demostreu la correcció de la següent implementació alternativa:

```
int potencia(int x, int y) {
    if (y == 0) return 1;
    else {
      int p = potencia(x, y/2);
      if (y%2 == 0) return p * p;
      else return x * p * p;
    }
}
```

Funció d'immersió: funció auxiliar

La funció original crida la funció d'immersió

- Fixant els paràmetres addicionals
- Ignorant alguns dels resultats retornats

Sobre la funció d'immersió

- Afegeix paràmetre(s).
- La nova funció d'immersió d'ha d'especificar.

Suma dels elements d'un vector: afebliment de la Post

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

Suma dels elements d'un vector: afebliment de la Post

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

Funció d'immersió

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

Suma dels elements d'un vector: afebliment de la Post

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

Funció d'immersió

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

Crida inicial

```
// Pre: cert
// Post: retorna la suma de v */
int suma(const vector<int>& v) {
  return i_suma(v, v.size()-1);
}
```

Implementació de la funció d'immersió

```
// Pre: −1 ≤ i < v.size()
// Post: retorna la suma de v[0..i]
int i_suma(const vector<int>& v, int i) {
  if (i < 0)
     return 0;
  else
     return i_suma(v, i-1) + v[i];
     /* HI: retorna la suma de v[0..i − 1] */
}
```

Cerca en un vector ordenat

```
// Pre: v.size() > 0 i v està ordenat creixentment
// Post: Retorna una posicio on es troba l'element x dins
//       el vector v, si x ∈ v. Si x ∉ v, retorna -1.

int cerca(const vector<int>& v, int x);
```

Afebliment de la post

Canviar v per $v[i..j]$

```
// Pre: 0 ≤ i ≤ v.size(), −1 ≤ j < v.size(), i ≤ j + 1
//      i v està ordenat creixentment
// Post: Retorna una posicio on és troba l'element x dins
//       el subvector v[i..j], si x ∈ v[i..j]. Si x ∉ v[i..j],
//       retorna -1.

int i_cerca(const vector<int>& v, int x, int i, int j);
```

Cerca dicotòmica

```
// Pre: 0 ≤ i ≤ v.size(), −1 ≤ j < v.size(), i ≤ j + 1
//      i v està ordenat creixentment
// Post: Retorna una posicio on és troba l'element x dins
//       el subvector v[i..j], si x ∈ v[i..j]. Si x ∉ v[i..j], retorna -1.

int i_cerca(const vector<int>& v, int x, int i, int j){
  int posicio;
  if(j < i) posicio= -1;
  else{
    int mig = (j+i)/2;
    if (v[mig]==x) posicio=mig;
    else{
      if (v[mig]< x) posicio= i_cerca(v, x, mig+1, j);
      /* HI: Si x es troba a v[mig+1...j], llavors el valor
         retornat és una posició de v[mig+1...j] on es troba x.
         Si x no es troba a v[mig+1...j], es retorna -1 */
      else posicio=i_cerca(v, x, i, mig-1);
      /* HI: Si x es troba a v[i...mig-1], llavors el valor
         retornat és una posició de v[i...mig-1] on es troba x.
         Si x no es troba a v[i...mig-1], es retorna -1 */
    }
  }
  return posicio;
}
```

Cerca dicotòmica: casos senzills

Justificació informal de la funció `i_cerca`

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

Cerca dicotòmica: casos senzills

Justificació informal de la funció `i_cerca`

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

Cerca dicotòmica: casos recursius

- Es comprova que les crides recursives compleixen la pre

Cerca dicotòmica: casos recursius

- Es comprova que les crides recursives compleixen la pre

- Si l'interval del vector no és buit i `x` no es troba a `mig`, llavors `x` està al subvector `v[i..mig-1]` o al subvector `v[mig+1..j]` o no hi és al subvector `v[i..j]`.

Cerca dicotòmica: casos recursius

- Es comprova que les crides recursives compleixen la pre
- Si l'interval del vector no és buit i `x` no es troba a `mig`, llavors `x` està al subvector `v[i..mig-1]` o al subvector `v[mig+1..j]` o no hi és al subvector `v[i..j]`.
- Donat que `v` està ordenat de menor a major, si `v[mig] < x` hem de cercar al subvector `v[mig+1..j]`, i sinó al subvector `v[i..mig-1]`.

Cerca dicotòmica: casos recursius

- Es comprova que les crides recursives compleixen la pre
- Si l'interval del vector no és buit i `x` no es troba a `mig`, llavors `x` està al subvector `v[i..mig-1]` o al subvector `v[mig+1..j]` o no hi és al subvector `v[i..j]`.
- Donat que `v` està ordenat de menor a major, si `v[mig] < x` hem de cercar al subvector `v[mig+1..j]`, i sinó al subvector `v[i..mig-1]`.
- Si `v[mig] < x`, la hipòtesi d'inducció diu que la crida recursiva retornarà la posició de v[mig+1...j] on es troba x, o -1, que es el que demana la postcondició.

Cerca dicotòmica: casos recursius

- Es comprova que les crides recursives compleixen la pre
- Si l'interval del vector no és buit i `x` no es troba a `mig`, llavors `x` està al subvector `v[i..mig-1]` o al subvector `v[mig+1..j]` o no hi és al subvector `v[i..j]`.
- Donat que `v` està ordenat de menor a major, si `v[mig] < x` hem de cercar al subvector `v[mig+1..j]`, i sinó al subvector `v[i..mig-1]`.
- Si `v[mig] < x`, la hipòtesi d'inducció diu que la crida recursiva retornarà la posició de v[mig+1...j] on es troba x, o -1, que es el que demana la postcondició.
- Si `x < v[mig]`, per HI la crida recursiva retornarà la posició de v[i...mig-1] on es troba x, o -1.

Funció de mida

Cal comprovar l'acabament de la funció recursiva:

Funció de mida

Cal comprovar l'acabament de la funció recursiva:

- La funció de mida seria la mida del segment de `v` no explorat, es a dir `|v[i..j]|`.

Funció de mida

Cal comprovar l'acabament de la funció recursiva:

- La funció de mida seria la mida del segment de `v` no explorat, es a dir `|v[i..j]|`.
- `j - i` no és una funció correcta ja que pot arribar a ser -1.

Funció de mida

Cal comprovar l'acabament de la funció recursiva:

- La funció de mida seria la mida del segment de `v` no explorat, es a dir `|v[i..j]|`.
- `j - i` no és una funció correcta ja que pot arribar a ser -1.
- `j - i + 1` és la funció correcta, donat que sempre és més gran o igual a 0.

Funció de mida

Cal comprovar l'acabament de la funció recursiva:

- La funció de mida seria la mida del segment de `v` no explorat, es a dir `|v[i..j]|`.
- `j - i` no és una funció correcta ja que pot arribar a ser -1.
- `j - i + 1` és la funció correcta, donat que sempre és més gran o igual a 0.
- i a totes les crides recursives la funció decreix.

Cerca dicotòmica

Finalment el codi de la funció que volíem resoldre, una crida a la funció d'immersió fixant els paràmetres addicionals a la primera i última posició del vector.

Les tres desigualtats de la precondició de la crida es compleixen trivialment.

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

Mergesort

Volem ordenar creixentment un vector recursivament.

```
// Pre: v = V, v.size()>0
// Post: v està ordenat creixentment i v és una permutació de V

void ordenar(const vector<int> & v)
```

Mergesort

Per simplificar, direm n a $v.size()$

```
// Pre: 0 ≤ e ≤ d < n ∧ v = V
// Post: v[0..e − 1] = V[0..e − 1] ∧ v[d + 1..n − 1] = V[d + 1..n − 1]∧
//       v[e..d] ordenat creixentment i és una permutació de V[e..d]

void mergesort(vector<int>& v, int e, int d) {
    if (e < d) {
        int m = (e + d)/2;
        mergesort(v, e, m);
        /* HI1: v[0..e − 1] = V[0..e − 1] ∧ v[m + 1..n − 1] = V[m + 1..n − 1] i
                v[e..m] ordenat creixentment i
                v[e..m] és una permutació de V[e..m] */
        mergesort(v, m + 1, d);
        /* HI2: v[0..m] = V[0..m] ∧ v[d + 1..n − 1] = V[d + 1..n − 1] i
                v[m + 1..d] ordenat creixentment i
                v[m + 1..d] és una permutació de V[m + 1..d] */
        fusiona(v, e, m, d);
    }
}
```

Mergesort

Ens cal una operació per fusionar ordenadament dos segments consecutius el `v` cadascú dels quals està ordenat.
Aquesta operació es iterativa i no la implementarem aquí.

```
// Pre: 0 ≤ e ≤ m < d < v.size() ∧ v = V ∧
///      v[e..m] i v[m+1..d] estàn ordenats creixentment
// Post: v[0..e−1] = V[0..e−1] ∧ v[d+1..n−1] = V[d+1..n−1] ∧
//       v[e..d] ordenat creixentment i és una permutació de V[e..d]

void fusiona(vector<int>& v, int e, int m, int d);
```

Mergesort

- Cas base: un segment de vector de mida 1 està trivialment ordenat (cas e=d).

Mergesort

- Cas base: un segment de vector de mida 1 està trivialment ordenat (cas e=d).

- Les crides recursives compleixen les pres:
  $0 \leq e \leq m < n$ i $0 \leq m+1 \leq d < n$, donat que si $d > e$ llavors $e \leq (e+d)/2$ i $(e+d)/2 + 1 \leq d$

Mergesort

- Cas base: un segment de vector de mida 1 està trivialment ordenat (cas e=d).

- Les crides recursives compleixen les pres: $0 \leq e \leq m < n$ i $0 \leq m+1 \leq d < n$, donat que si $d > e$ llavors $e \leq (e+d)/2$ i $(e+d)/2 + 1 \leq d$

- Podem aplicar les H.I. que ens asseguren (entre altres coses) que $v[e..m]$ està ordenat creixentment i $v[m+1..d]$ també. Per tant si l'algorisme "fusiona" es correcte, $v[e..d]$ està ordenat.

Mergesort

- Cas base: un segment de vector de mida 1 està trivialment ordenat (cas e=d).

- Les crides recursives compleixen les pres:
  $0 \leq e \leq m < n$ i $0 \leq m+1 \leq d < n$, donat que si $d > e$ llavors $e \leq (e+d)/2$ i $(e+d)/2+1 \leq d$

- Podem aplicar les H.I. que ens asseguren (entre altres coses) que $v[e..m]$ està ordenat creixentment i $v[m+1..d]$ també. Per tant si l'algorisme "fusiona" es correcte, $v[e..d]$ està ordenat.

- La funció de mida és $d-e$, quan sigui 0 és perque e i d són iguals i per tant el segment `v[e..d]` és `v[e]` (o `v[d]`).

Mergesort

- Cas base: un segment de vector de mida 1 està trivialment ordenat (cas e=d).

- Les crides recursives compleixen les pres:
  $0 \leq e \leq m < n$ i $0 \leq m+1 \leq d < n$, donat que si $d > e$ llavors $e \leq (e+d)/2$ i $(e+d)/2+1 \leq d$

- Podem aplicar les H.I. que ens asseguren (entre altres coses) que $v[e..m]$ està ordenat creixentment i $v[m+1..d]$ també. Per tant si l'algorisme "fusiona" es correcte, $v[e..d]$ està ordenat.

- La funció de mida és $d-e$, quan sigui 0 és perque e i d són iguals i per tant el segment `v[e..d]` és `v[e]` (o `v[d]`).

- Com $m = (e+d)/2$, $e \leq m$, $e < m+1$ i $m < d$. Llavors $m-e < d-e$ i $d-(m+1) < d-e$.
  Per tant després d'un nombre finit de crides recursives

Mergesort

Per últim cal donar el codi de l'operació `ordenar`

```
// Pre: v = V, v.size()>0
// Post: v està ordenat creixentment i v és una permutació de V

void ordenar(const vector<int> & v)
{
  mergesort(v,0,v.size()-1);
}
```

Recursivitat lineal final

- Algorisme recursiu lineal: algorisme recursiu que a cada crida recursiva genera solament una crida recursiva
- Funció recursiva lineal final (*tail recursion*):
  - La darrera instrucció que s'executa (cas recursiu) és la crida recursiva
  - El resultat de la funció (cas recursiu) és el resultat que s'ha obtingut de la crida recursiva, sense cap modificació