RICORSIONE

ricorsione su dati automatici

(testo Cap. 10)

problemi si dividono in sottoproblemi e

```
int F(……)
{

…….G(…)

}
```

```
double G(....)
{
  ....H(..) ... e così via
}
```

e se F = G = H ?   Ricorsione

```
int F(..)
{
 ....G(..)..
}
double G(...)
{
....H(..)...
}
```

stack dei dati

Var locali di F
Var locali di G
Var locali di H

Record d’Attivazione (RA) contiene le var locali e anche l’indirizzo di ritorno

con ricorsione

```
int F(..)
{
 ....F(..)..
}
```

una sola funzione, ma tante invocazioni e un RA per le variabili locali di ciascuna invocazione ci sono calcoli naturalmente ricorsivi :

-il fattoriale di 1 è 1

-il fattoriale di n>1 è

$$\underbrace{n * \underbrace{\boxed{(n-1) * (n-2) * \ldots 1}}_{\text{fatt(n-1)}}}_{\text{fatt(n)}}$$

```
int fatt(int n)
{
 if(n==1)
   return 1;
else
  return n * fatt(n-1);
}
```

caso base

caso ricorsivo

```
...fatt(3)....


int fatt(int n)
{
 if (n==1) return 1;
 else
 return n*fatt(n-1);
}
```

stack dei dati

n=3
n=2
n=1

programma
che esegue

```
 ......
 x=fatt(3);
 .....
int fatt(int n)
{ if(n<=1) return 1;
else
  return n * fatt(n-1);
}
```

stack dei dati

x=
n=3 fatt(2)
n=2 fatt(1)
n=1

il caso base è importante

int fatt(int n)

~~{ if(n<=1) return 1;~~

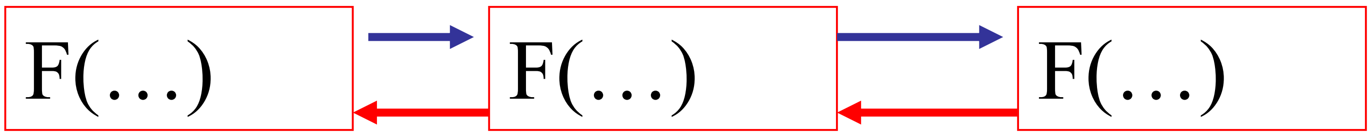
~~else~~

return n * fatt(n-1);

}

fatt(3) → fatt(2) → fatt(1) → fatt(0) → fatt(-1) → fatt(-2) → all'infinito in un calcolo ricorsivo

andata



ritorno

```
...F(..)
{.....
   F();
  .....
return…
;
}
```

andata

ritorno

ESEMPIO:
determinare se in un array c'è z:

PRE=(dim>=0, A[0..dim-1] è definito)

```
bool presente(int* A, int dim, int z)
```

POST=(restituisce true sse A[0..dim-1] contiene z) caso base:

-se dim==0 allora l’array è vuoto e quindi la risposta è false

passo induttivo:

-se A=[x, resto], se x=z, allora true, altrimenti si deve cercare nel resto e questo lo fa l’invocazione ricorsiva: presente(A+1,dim-1,z)

```
//PRE=(dim>=0, A[0..dim-1] è definito)
bool presente(int *A, int dim, int z)
{
    if(dim==0)
          return false;
     else
         if(A[0]==z)
           return true;
         else
           return presente(A+1,dim-1, z);
}
//POST=(restituisce true sse A[0..dim-1] contiene z)
```

come facciamo per dimostrare la correttezza??

PRE(A,dim,z) e POST(A,dim,z)
e per l'invocazione ricorsiva:
PRE(A+1,dim-1,z) e POST(A+1,dim-1,z)

osserva che se

A[0..dim-1] allora

(A+1)[0...dim-2] = A[1..dim-1]

come facciamo per le funzioni normali?

vale PRE_g(par attuali)

```
PRE_f
int f(....)
{
  ..................
.......g(par. attuali)
 .................

}
POST_f
```

abbiamo dimostrato che g è corretta rispetto a PRE_g e POST_g

usiamo questo fatto nella prova che f è corretta rispetto a PRE_f e POST_f

allora vale POST_g(par attuali)

con la ricorsione, al posto della prova della correttezza di g, usiamo la seguente **<u>ipotesi induttiva:</u>**

assumiamo che l'invocazione induttiva:
presente(A+1,dim-1, z);
sia corretta rispetto a PRE(A+1,dim-1,z) e POST(A+1,dim-1,z)

<u>per usare l'ipotesi induttiva, dovremo dimostrare che i parametri attuali soddisfano la PRE(A+1,dim-1,z)</u>

prova induttiva (testo 10.2.1):

1)casi base: uno alla volta,
- PRE<caso base> POST

2)passo induttivo:

-**ipotesi induttiva**: si assume che le invocazioni ricorsive sono corrette rispetto a PRE(par attuali) e POST(par attuali)

- PRE <caso ricorsivo> POST

al posto di PRE(par attuali) usiamo PRE_RIC

e lo stesso per POST_RIC

primo caso base:

PRE=(dim>=0, A[0..dim-1] è definita)

if (dim==0) return false;

POST=(presente restituisce true sse A[0..dim-1] contiene z)

secondo caso base

```
//PRE=(dim>=0, A[0..dim-1] è definito)


  if(dim==0)
     return false;
   else  // (dim>0, A[0..dim-1] definito)
     if(A[0]==z)
        return true;
     else...

//POST=(restituisce true sse A[0..dim-1] contiene z)
```

caso induttivo

if(A[0]==

(dim>0, A[0..dim-1] è definito) =>
PRE_ric= (dim-1>=0, (A+1)[0..dim-2] è definito)

else

return presente(A+1,dim-1,z) ;

POST_ric=(restituisce true sse (A+1)[0..dim-2]=A[1..dim-1] contiene z)

assieme a !(A[0]== z) =>

POST=(restituisce true sse A[0..dim-1] contiene z) è corretto assumere l'ipotesi induttiva?

consideriamo dim=0, dim=1, dim=2,....

la funzione presente è corretta con array vuoto,
con array con 1 elemento,
con array con 2 elementi
e così via ....

quando dimostriamo il caso dim usiamo la correttezza del caso dim-1 che abbiamo dimostrato prima

ma non conta il particolare valore dim che consideriamo !!!!

il passo induttivo non dipende da dim => vale per ogni dim

basta fare il passo induttivo 1 sola volta rivediamo la funzione ricorsiva presente

```
//PRE=(A ha dim>=0 elementi)
bool presente(int *A, int dim, int z)
{
  if(dim==0)
        return false;
  else
        if(A[0]==z)
              return true;
        else
              return presente(A+1,dim-1,z);
}
//POST=(true sse A[0..dim-1] contiene z)
```

possiamo realizzare la funzione presente anche con l’iterazione

supponiamo che l’array da considerare sia

int B[100], * A=B;

in questo modo possiamo cambiare A

PRE=(vA=A, vdim=dim, A ha dim>=0 elementi)

```
bool trovato=false;
while(dim>0 && !trovato)
{
    if(A[0]==z)
        trovato=true;
    A++;
    dim--;
}
```

R= (0<=dim<=vdim)
&& (vA <= A <=
(vA+vdim))
&& (trovato sse z è in
vA[0..vdim-dim-1])

POST=(trovato sse z è in vA[0..vdim-1])

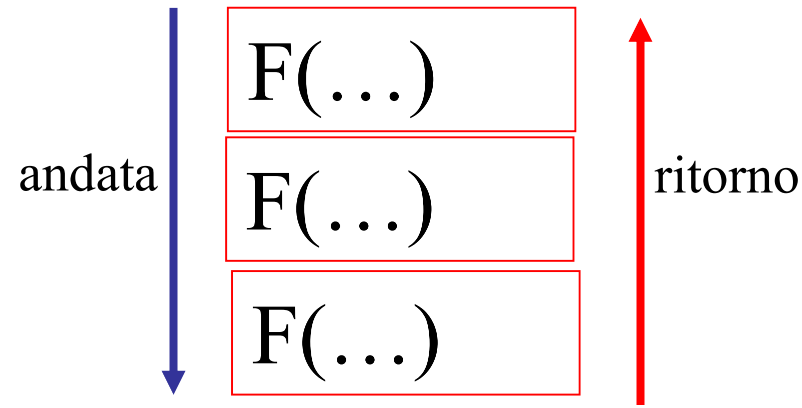
```
bool trovato=false;
while(dim>0 && !trovato)
 {
    if(A[0]==z)
       trovato=true;
    A++;
    dim--;
}
```

R afferma che trovato sse z è nella parte di A che ho visto

```
//PRE
bool presente(int *A, int dim, int z)
{
  if(dim==0)
          return false;
   else
          if(A[0]==z)
           return true;
          else
           return presente(A+1,dim-1,z);
}//POST =(true sse z è in A[0..dim-1])
```

iterazione va in avanti
ricorsione va indietro

in un calcolo ricorsivo



col while è possibile simulare l'andata, ma non il ritorno

una funzione ricorsiva che non calcola nulla al ritorno è detta ricorsiva terminale (tail recursive)

è facile simularla con un while

presente ricorsiva si può scrivere anche così:

```
bool presente(int *A, int dim, int z)
{
  if(dim==0)
     return false;
  else
     return (A[0]==z) || presente(A+1,dim-1,z);
}
```

è ricorsiva terminale
niente da fare dopo
l'invocazione ricorsiva

```
bool trovato=false;
while(dim>0 && !trovato)
 {
    if(A[0]==z)
          trovato=true;
    A++;
    dim--;
}
```

versione che fa cose inutili

```
bool presente_stupid(int *A, int dim, int z)
{
  if(dim==0)
      return false;
   else
      return presente_stupid(A+1,dim-1,z) || (A[0]==z) ;
}
```

non è ricorsiva terminale

A=[2,1,0], dim=3 e z=2; presente_stupid(A,3,2)

[2,1,0] , 3,2
[1,0],2,2
[0],1,2
[ ],0,2

possiamo fare i test

2==A+2[0]=0

2==A+1[0]=1

2==A[0]=2

<u>solo perché abbiamo la pila dei RA</u> che contiene A,A+1 e A+2

col while non abbiamo la pila

con la ricorsione terminale la pila dei RA non serve
ci basta 1 solo RA cambiando le variabili A e dim

1 RA = 1 while

per simulare ricorsione non terminale con l'iterazione dobbiamo simulare la pila dei RA e poi avremo bisogno di almeno 2 cicli
1 per l'andata e 1 per il ritorno

Esercizio: un ciclo più compatto

```
while(dim>0 && A[0]!=z)
{A++; dim--;}
```

trovate l'invariante e POST

R = (0 ≤ dim ≤ vdim) && (vA ≤ dim ≤ vA + vdim)

POST = (esaminati elementi A[~~0~~ vdim ... dim] && A[0]!=z altro ciclo:

```
int i=0;
while(i <dim && A[i]!=z)
 i++;
```

trovate l'invariante e la POST

R = (0 ≤ i ≤ dim) e A[0 .. i-1] esaminato ... dim] infinito

POST = esaminati [dim-1] elementi
sse A[i] != z