RICORSIONE

ricorsione su dati automatici

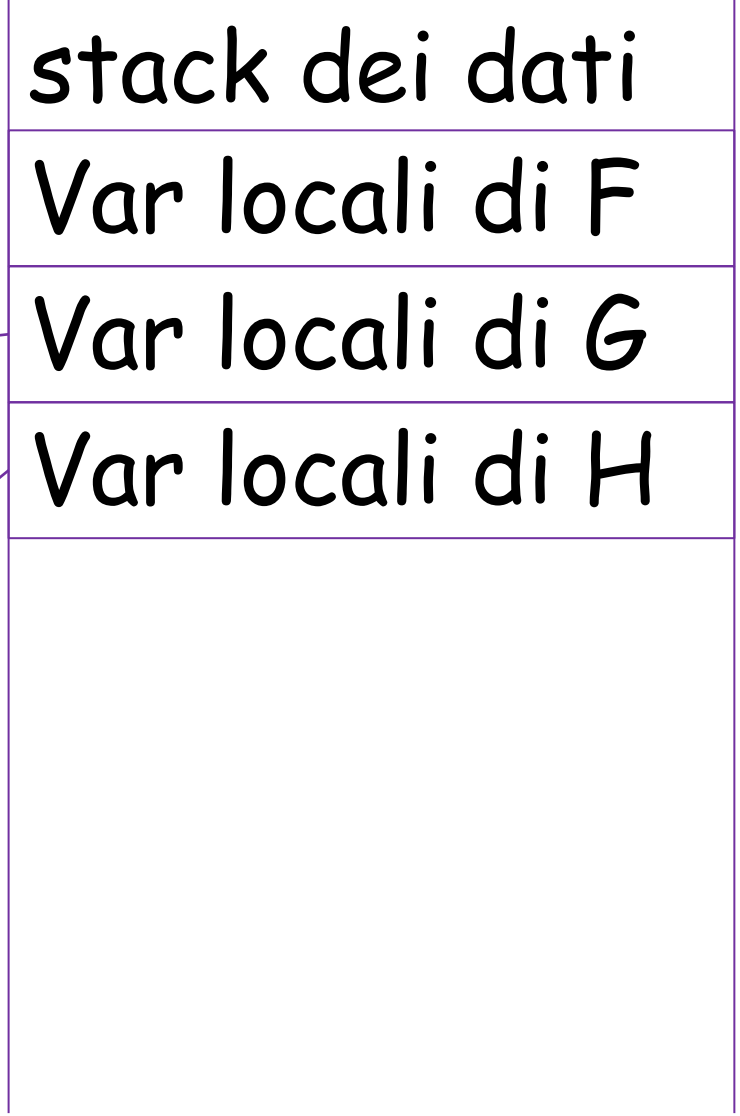
(testo Cap. 10)

problemi si dividono in sottoproblemi e

```
int F(……)
{
                                  void G(….)
                                  {
…….G(…)                             ….H(..) … e così via
                                  }
}
```

e se F = G = H ? Ricorsione

```
inf F(..)
{
 ....G(..)..
}
void G(...)
{
....H(..)...
}
```



```
inf F(..)
{
 ....F(..)..
}
```

una sola funzione, ma tante invocazioni e un RA per le variabili locali di ciascuna invocazione

calcoli ricorsivi :

-il fattoriale di 0 e 1 è 1

-il fattoriale di n>1 è

n * (n-1) * (n-2) * … 1

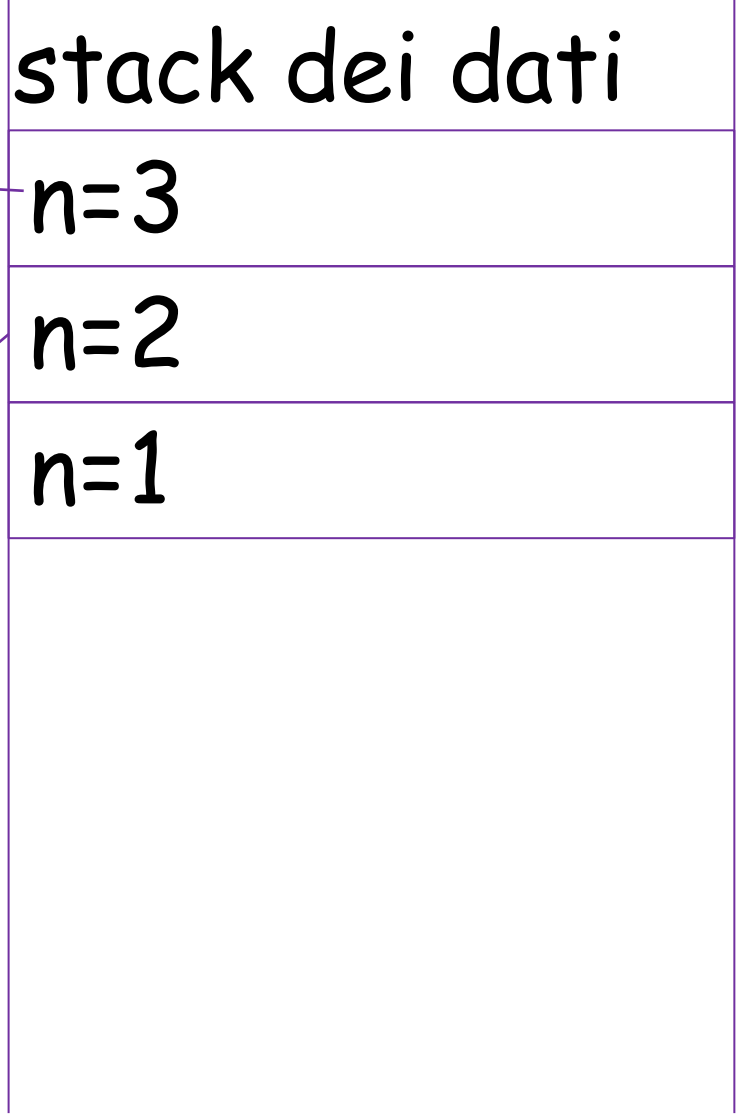
fatt(n-1)

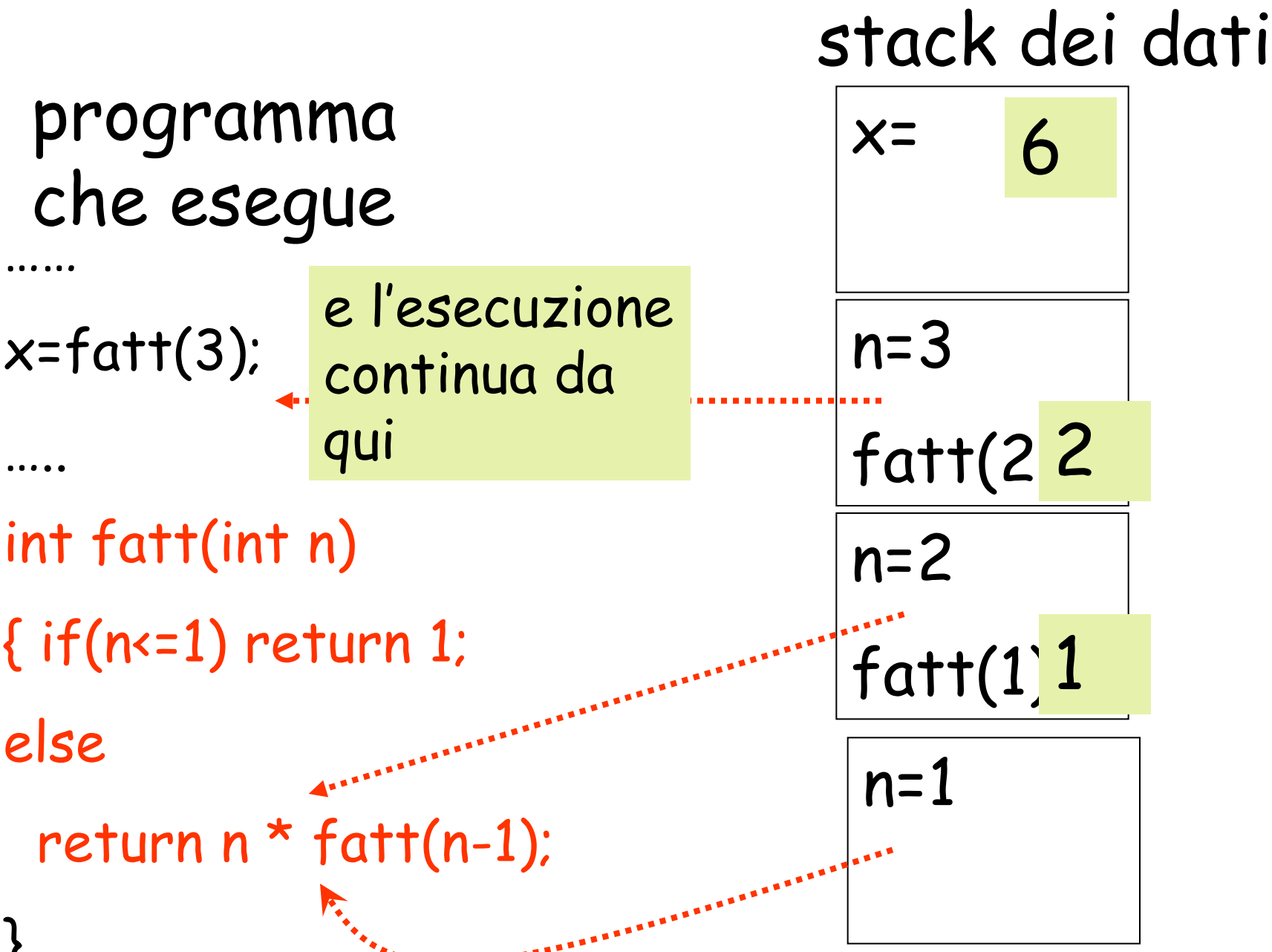
fatt(n)

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

caso base

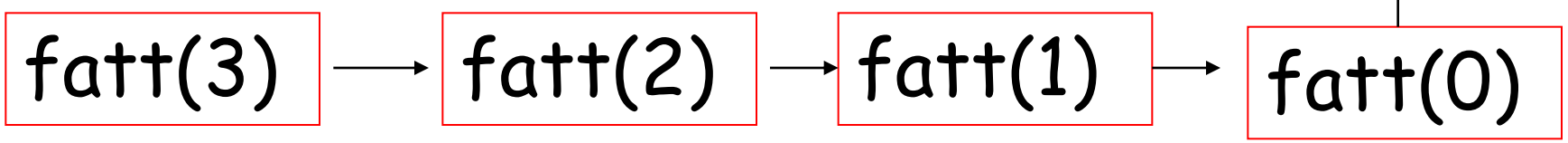
```
...fact(3)....


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



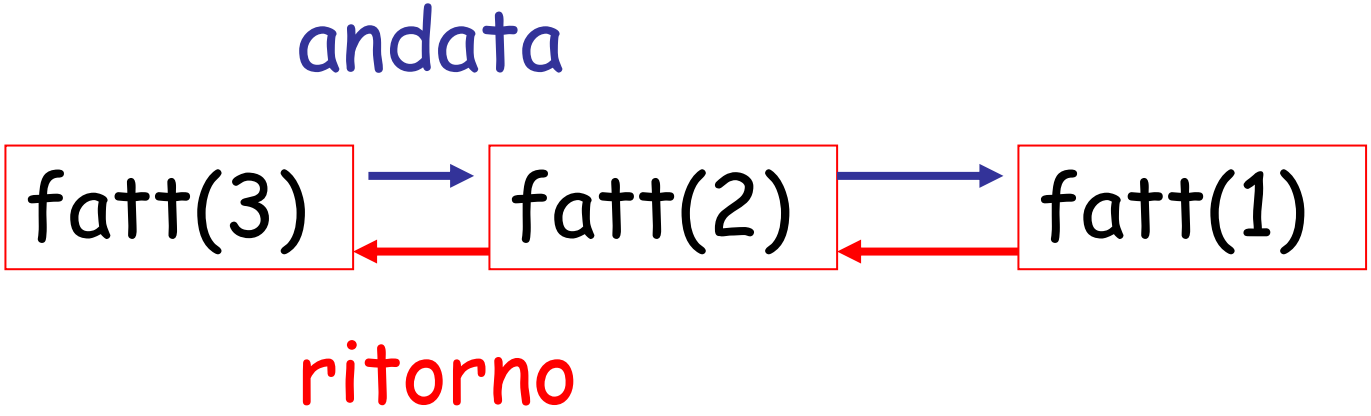

programma
che esegue
.....
x=fatt(3);
e l'esecuzione
continua da
qui
.....
int fatt(int n)
{ if(n<=1) return 1;
else
return n * fatt(n-1);
}
stack dei dati
x=
6
n=3
fatt(2
2
n=2
fatt(1)
1
n=1

il caso base è importante

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



in un calcolo ricorsivo



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

PRE=(dim>=0, A[0..dim-1] def., z def.)

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], allora, se x==z, allora true e altrimenti si deve cercare nel resto => presente(A+1,dim-1,z)

```
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
```

per dimostrare che la funzione è corretta si deve assumere che presente(A+1,dim-1,z); sia corretta rispetto a
PRE_ric= (dim-1>=0, (A+1)[0..dim-2] def., z def.)
e
POST_ric=(restituisce true sse (A+1)[0..dim-2] contiene z)

assumere che l'invocazione ricorsiva sia corretta rispetto a PRE e POST significa

assumere che, se prima dell'invocazione vale PRE allora al ritorno vale POST

è quello che si fa sempre con le invocazioni di funzione !

ma in caso di funzione ricorsiva sembra che stiamo mordendoci la coda !!

prova induttiva (testo 10.2.1):

-caso base:
PRE<caso base> POST

-passo induttivo:
  -ipotesi induttiva: le invocazioni ricorsive sono corrette rispetto a PRE e POST
  -vale PRE <caso non base> POST

caso base:

PRE=(dim>=0, A[0..dim-1] def., z def.)

if (dim==0) return false;

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

altro caso base

PRE=(dim>=0, A[0..dim-1] def., z def.)

```
{ if(dim==0)

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

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

if(A[0]==z) then return true

else (dim>0, A[0] != z)

PRE_ric ??

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

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

ha senso l'ipotesi induttiva?

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

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 riassume tutte le prove in una volta sola

rivediamo la nostra funzione ricorsiva

```
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);
         }

}
```

possiamo fare lo stesso con un while

```
bool trovato=false;

while(dim>0 && !trovato)

{ if(A[0]==z)
        trovato=true;

else

        {A++; dim--; }
}
```

prova difficile perché perdiamo il già fatto

più facile qui

```
bool trovato=false; int i=0;

while(i<dim && !trovato)

{ if(A[i]==z)
      trovato=true;

i++;

}
```

R=(0<=i<=dim)(trovato sse z in A[0..i-1])

invariante parla del passato

post della ricorsione parla di quello che resta da fare

presente 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);
}
```

scambiare le 2 condizioni significa fare chiamate ricorsive potenzialmente inutili (se A[0]==z)

faremmo prima l'invocazione ricorsiva e poi il test su X[0]

insomma il test viene fatto "al ritorno" della ricorsione

vista la valutazione short-cut di queste espressioni, il test ci può permettere di terminare il calcolo,

facendolo dopo l'invocazione, rischieremmo di fare invocazioni inutili

sarebbe come questo

```
bool trovato=false;

while(dim>0)

{ if(A[0]==z)
        trovato=true;
A++; dim--;
}
```

un ciclo + compatto

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

esercizio: scrivete l'invariante di questo:

```
bool trovato=false; int i=0;

while(i<dim && !trovato)

{ if(A[i]==z)
        trovato=true;

else

        i++;
}
```