RICORSIONE

ricorsione su dati automatici

problemi si dividono in sottoproblemi e

```
....F(......)
{

    .......G(...)

}
```

e se G = F ?   Ricorsione

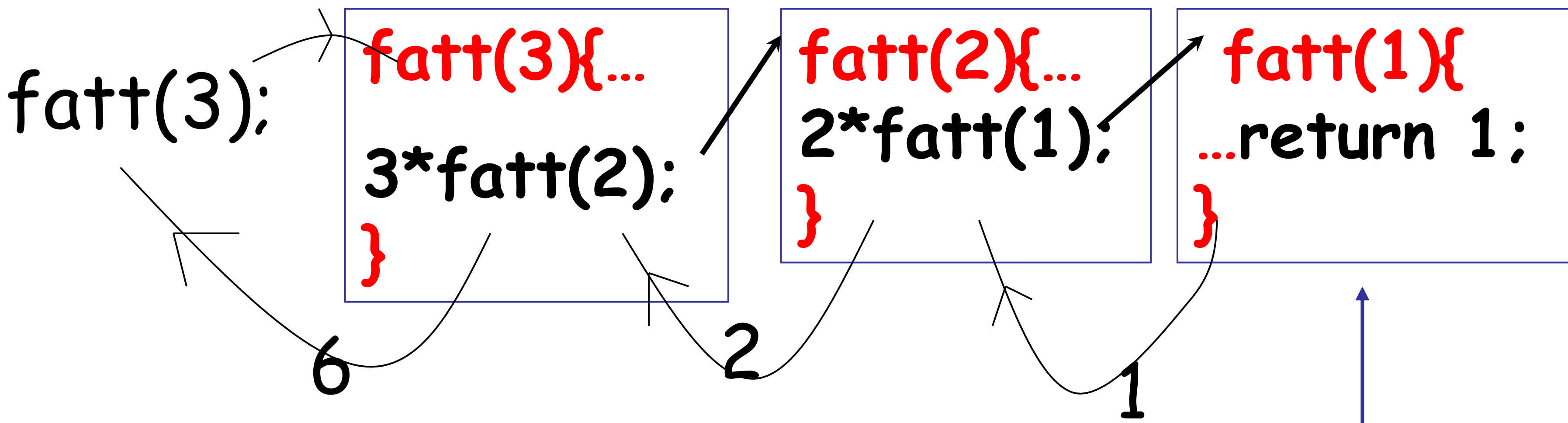
ci sono problemi che si prestano ad una soluzione ricorsiva :

-il fattoriale di 0 e 1 è 1

-il fattoriale di n>1 è

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

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



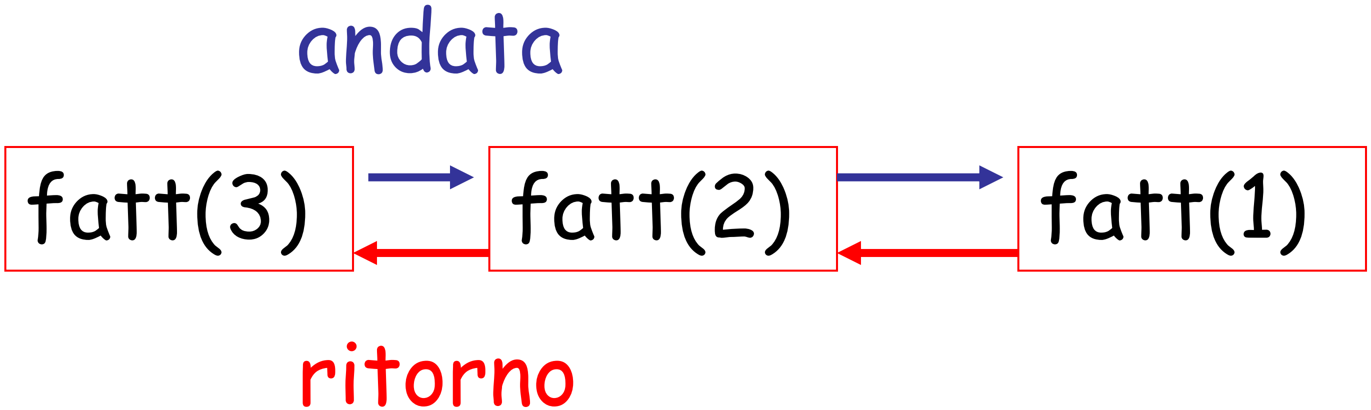

```
int fatt(int n)
{ if(n<=1) return 1;
else
  return n * fatt(n-1);
}
```

condizione di terminazione

se non avessimo la condizione di terminazione

```
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 se tutto va bene:



ma che succede durante questa esecuzione ?

c'è un solo codice di fatt

e diverse copie dei dati

stack dei dati

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);
}
```

x= 6

n=3

fatt(2) 2

n=2

fatt(1) 1

n=1

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=(presente 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
presente(resto)= presente(A+1,dim-1)

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
```

si assume che l'invocazione ricorsiva sia corretta rispetto a PRE e POST

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

è quello che si fa sempre con le invocazioni di funzione !
ma ha senso in caso di invocazione ricorsiva ?

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)

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

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

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

PRE_ric ?

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

POST_ric => POST

POST=(presente 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 ....

passando da dim a dim+1 si richiede la dimostrazione del passo induttivo!!

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

visto che il test ci può permettere di interrompere la ricorsione, facendolo al ritorno, rischiamo di fare invocazioni inutili equivale più o meno a:

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

iterazioni inutili che si eviterebbero con

```
while(dim && !trovato) {.....}
```

altra funzione presente (Cap.8 (5))

PRE=(A[0..top-1] ordinato, 0<=pos <=top,)

bool presente(int* A, int top, int pos, int y, int & start)

POST=
(restituisce true → pos<=start<top, A[start]=y, A[pos..start-1]<y) && (restituisce false → A[pos..top-1] != y)

```
bool presente(int*A, int top, int pos, int y,
int & start)
{if(pos==top) return false;

if(A[pos]==y)
    {start=pos; return true;}

if(A[pos]<y)
   return presente(A,top,pos+1,y, start);

return false;
}
```

correttezza per induzione:

-BASE dell'INDUZIONE
dimostrare che se vale la PRE allora vale la POST nei casi base;

- e poi PASSO INDUTTIVO

```
PRE=(A[0..top-1] ordinato, 0<=pos <=top, )
{if(pos==top) return false;

if(A[pos]==y)
     {start=pos; return true;}

if(A[pos]>y) return false;
} POST=
(restituisce true → pos<=start<top,
A[start]=y, A[pos..start-1]<y)    &&
(restituisce false → A[pos..top-1] != y)
```

-PASSO INDUTTIVO: assumere che le invocazioni ricorsive siano corrette rispetto alla PRE e POST

e poi dimostare che il corpo è tale che prima delle invocazioni ricorsive vale la PRE_ric (dall'ipotesi → vale la POST_ric al ritorno)
e da questo dimostrare che vale la POST alla fine del corpo

```
PRE=(A[0..top-1] ordinato, 0<=pos
<=top, )
{if(pos==top) return false;

if(A[pos]==y)
     {start=pos; return true;}

if(A[pos]>y) return false;
// qui sappiamo che pos<top e A[pos]< y
```

```
// qui sappiamo che pos<top e A[pos]<y

    PRE_ric ?
  return presente(A,top,pos+1,y, start);
    POST_ric=> POST ?
}
```

POST=
(restituisce true → pos<=start<top, A[start]=y, A[pos..start-1]<y) &&
(restituisce false → A[pos..top-1] != y)

calcolo di quante occorrenze di y ci sono

PRE=(A[start..top-1] ordinato &&
0<=start<=top &&
se start <top =>A[start]>=y)

```
int quanti(int*A, int top, int start, int y)
```

POST= restituisce k t.c.
(k>0 => A[start..start+k-1] =y &&
(start+k=top || A[start+k]>y)) &&
(k=0 => A[start..top-1] >y)

```
int quanti(int*A, int start, int top, int y)
{
  if(start==top)
      return 0;

  if(A[start]==y)
      return 1+quanti(A,start+1,top,y);
  else
      return 0;
}
```

verifica di correttezza

PRE= (A[start..top-1] ordinato &&
0<=start<=top && se start <top =>
A[start] >= y)

```
int quanti(int*A,int top,int start, int y)
{  if(start==top)
       return 0;
   if(A[start]!=y)
         return 0;}
```

(k>0 => A[start..start+k-1] =y &&
(start+k=top || A[start+k..top-1]>y)) &&
(k=0 => A[start..top-1] >y)

```
//start<top

  if(A[start]==y)
                        PRE_ric
      return 1+quanti(A,start+1,top,y);
                        POST_ric

(k>0 => A[start..start+k-1] =y &&
(start+k=top || A[start+k..top-1]>y) ) &&
(k=0 => A[start..top-1] >y)
```

eliminazione di una porzione di array

PRE=(A[next..top-1] def, next<=s<=top)

void shift(int*A, int top, int next, int s)

POST=

(A[next..next+(top-s)-1] = vA[s..top-1]

```
void shift(int*A, int top, int next, int s )
{
  if(s<top)
 {
  A[next]=A[s];
  shift(A,top,next+1,s+1);
 }
}
```

```
void tutto(int*A, int top, int y)
{ int start;
  if(presente(A,top,0, y, start))
   {
      int q=quanti(A,top,start,y);
      shift(A,top,start, start+quanti-1);
       top=top-q;
   }
}
```

eliminazione di tutti gli y da *A*

//(*A*[next..top-1] def, next<=curr<=top, *A*=v*A*)

int del(int**A*, int top, int next, int curr, int y)

//(restituisce k=lung(v*A*(-y)[curr..top-1]) &&
(*A*[next..next+k-1]= v*A*(-y)[curr..top-1]

```
if(curr==top) return 0;
if(A[curr]==y)
  return del(A,top, next, curr+1, y);
else
  {
 A[next]=A[curr];
   return 1+ del(A,top, next+1,curr+1,y);
  }
```