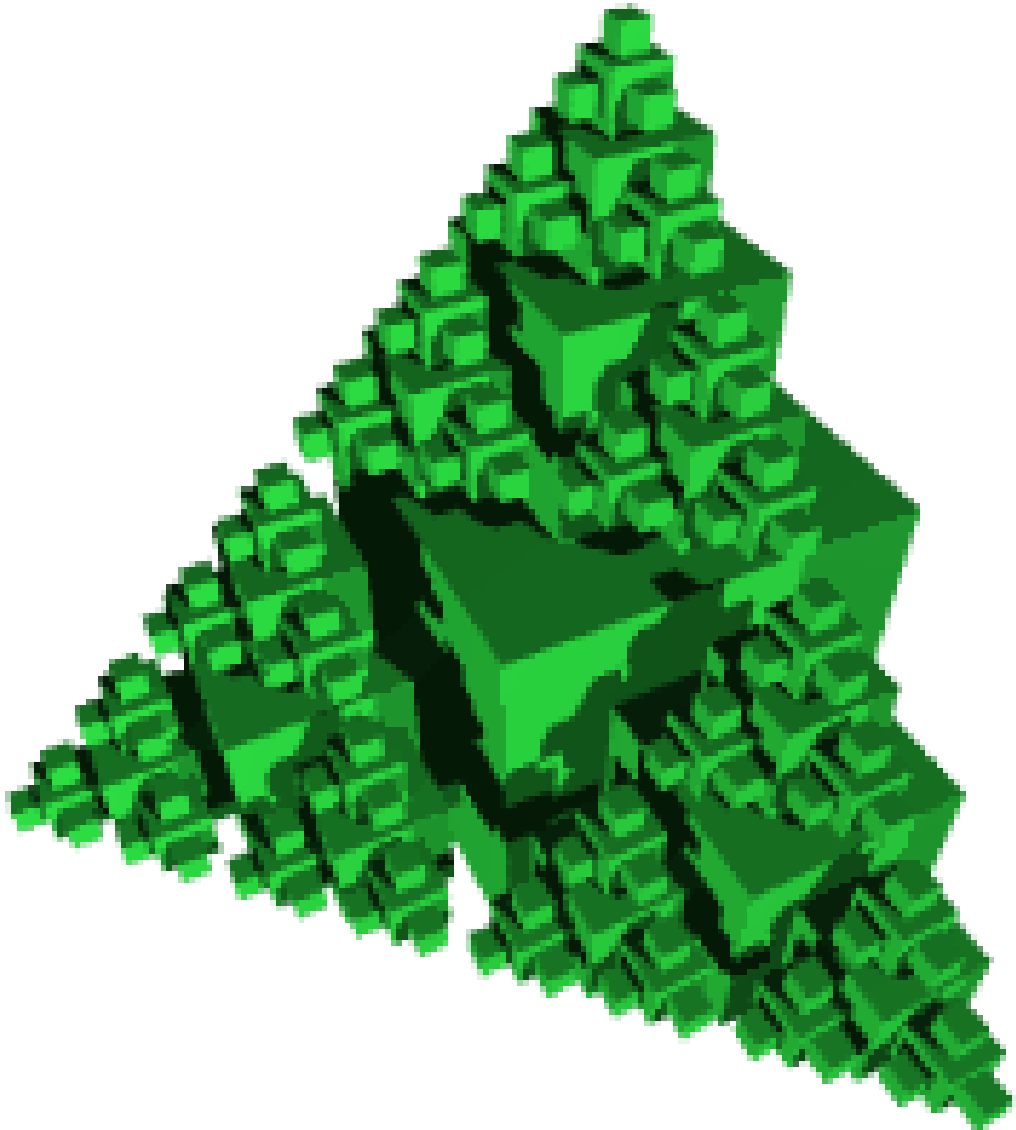
LA RICORSIONE



Ricorsione

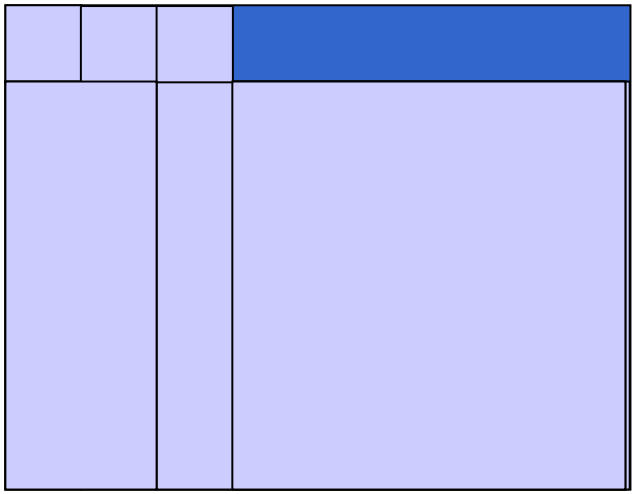
- Una funzione può invocare tutte le funzioni dichiarate prima di lei.
- In particolare, può richiamare anche se stessa.

- … ma …

serve a qualcosa???

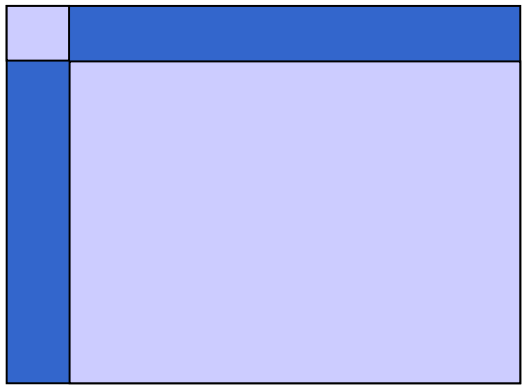
Esempio: determinante

- Il determinante di una matrice è definito in base al determinante di matrici più piccole



Esempio: determinante

- Il determinante di una matrice è definito in base al determinante di matrici più piccole
- Come fare a calcolarlo?
- Sarebbe bello scrivere



```
int det(matrice)
{   scegli una riga
    per tutti gli elementi
    {     calcola det(sottomatr)
          moltiplica per l'elemento
          (eventualm cambia di segno)
          aggiungi il risultato
    }
    return risultato
}
```

Es: Espressioni

- (Come visto in precedenza), un'espressione è

  <espressione> ::= *<variabile>* | *<costante>* |
  <espressione><operatore><espressione>|
  <operatore_unario><espressione>

- Quindi se voglio scrivere un programma che valuta delle espressioni, devo scrivere qualcosa del tipo:

```
int valuta(esp)
{  scomponi espressione in esp1 op esp2
   ris1=valuta(esp1);
   ris2=valuta(esp2);
   se (op=='+') return ris1+ris2;
   se (op=='*') return ris1*ris2;
   …
}
```

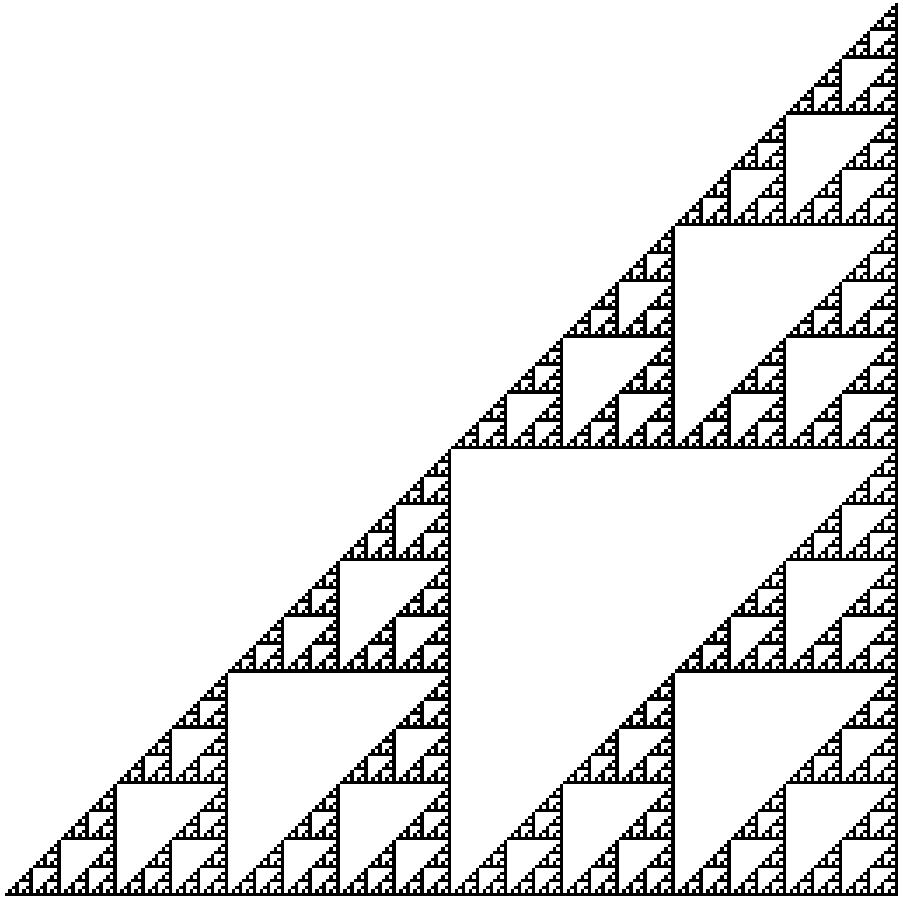
Es: calcolo della derivata

$$\frac{d(f(x)+g(x))}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}$$

$$\frac{d(f(x)\cdot g(x))}{dx} = \frac{df(x)}{dx} g(x) + f(x) \frac{dg(x)}{dx}$$

LA RICORSIONE

- Pensate a come si potrebbe disegnare la figura a lato (triangolo di Sierpinski)



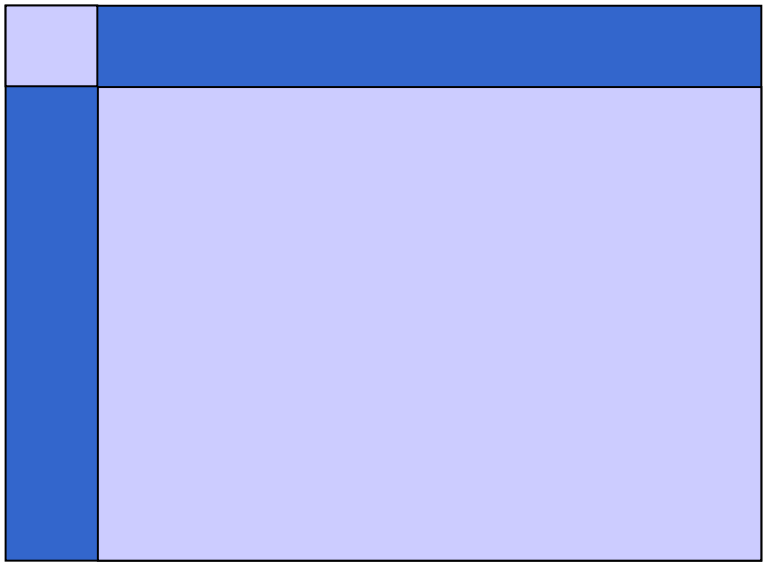
Quando mi fermo?

Se non stabilisco un criterio per terminare, l’elaborazione continua in eterno!

Terminazione

- Tutte le funzioni ricorsive devono avere un caso *“base”* in cui riesco a calcolare il risultato senza invocare la funzione ricorsivamente

Determinante



Espressioni

<*espressione*> ::=

<*variabile*> |

<*costante*> |

<*espressione*><*operatore*><*espressione*>|

<*operatore_unario*><*espressione*>

Es: calcolo della derivata

$$\frac{d(f(x)+g(x))}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}$$

$$\frac{d(f(x)\cdot g(x))}{dx} = \frac{df(x)}{dx} g(x) + f(x) \frac{dg(x)}{dx}$$

In pratica ...

- TUTTE le procedure e funzioni che richiamano se stesse devono avere (almeno) un caso base

- In pratica, cominciano sempre con una selezione

```
if (condizione di uscita)
  risultato caso base
else
  chiamata ricorsiva
```

LA RICORSIONE

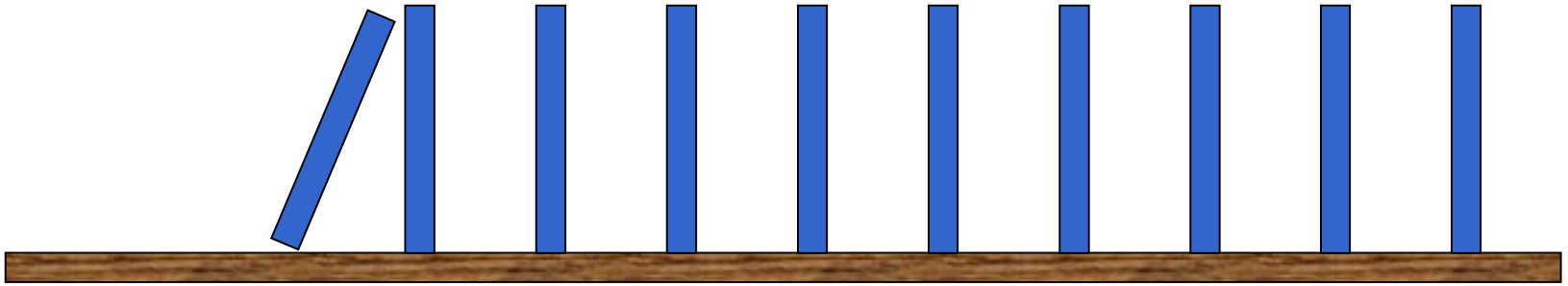
- Una funzione matematica è definita ***ricorsivamente*** quando nella sua definizione compare un riferimento a se stessa
- La ricorsione consiste nella possibilità di *definire una funzione in termini di se stessa.*

- È basata sul *principio di induzione matematica*: se si può provare che una proprietà *P*
  - vale per $n=n_0$ (CASO BASE)
  - e, *assumendola valida per n*, allora vale per $n+1$

  allora *P* vale per ogni $n \geq n_0$

Giuseppe Peano

IL PRINCIPIO DI INDUZIONE



- Es. ho una sequenza infinita di pezzi del domino disposti come in figura
  - il primo cade
  - Il primo fa cadere il secondo
  - il secondo fa cadere il terzo
  - …
- Come faccio a dimostrare che cadono tutti?
- Col principio di induzione posso farlo, sapendo che
  - il primo cade
  - se cade l'*n*-esimo, allora cade l'(*n*+1)-esimo

IL PRINCIPIO DI INDUZIONE

Teorema: *la somma dei primi x numeri dispari è uguale ad x^2.*

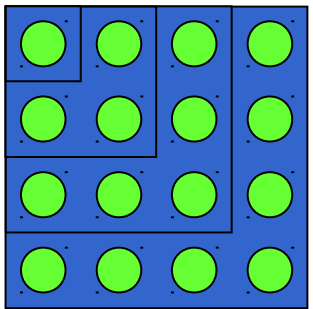
Dimostrazione:

- è vera per x=1, infatti 1^2=1
- supponiamo che per n sia vera, cioè

$$1+3+5+\ldots+(2n-1) = n^2$$

e dimostriamo che è vera per n+1:
sommiamo (2n+1) ad entrambi i termini:

$$1 + 3 + 5 + \ldots + (2n-1) + (2n + 1) = n^2 + 2n + 1$$



LA RICORSIONE

- Possiamo usare il principio di induzione anche per *definire* degli oggetti, non solo per dimostrare proprietà
- In matematica, la funzione fattoriale viene definita così:
  - $n! = 1$ se $n=0$
  - $n! = n*(n-1)!$ altrimenti
- Scomponiamo il problema:
  - troviamo un caso semplice, in cui la soluzione è immediata
  - ci riconduciamo a casi precedenti quando la soluzione non è immediata
- Nei linguaggi di programmazione moderni, viene fornita la stessa possibilità
- Questo ci permette di scrivere semplicemente algoritmi che sarebbero molto difficili

LA RICORSIONE: ESEMPIO

Esempio: il fattoriale di un numero

```
fact(n) = n!
n!: Z → N
   n! vale 1    se n ≤ 0
   n! vale n*(n-1)! se n > 0
```

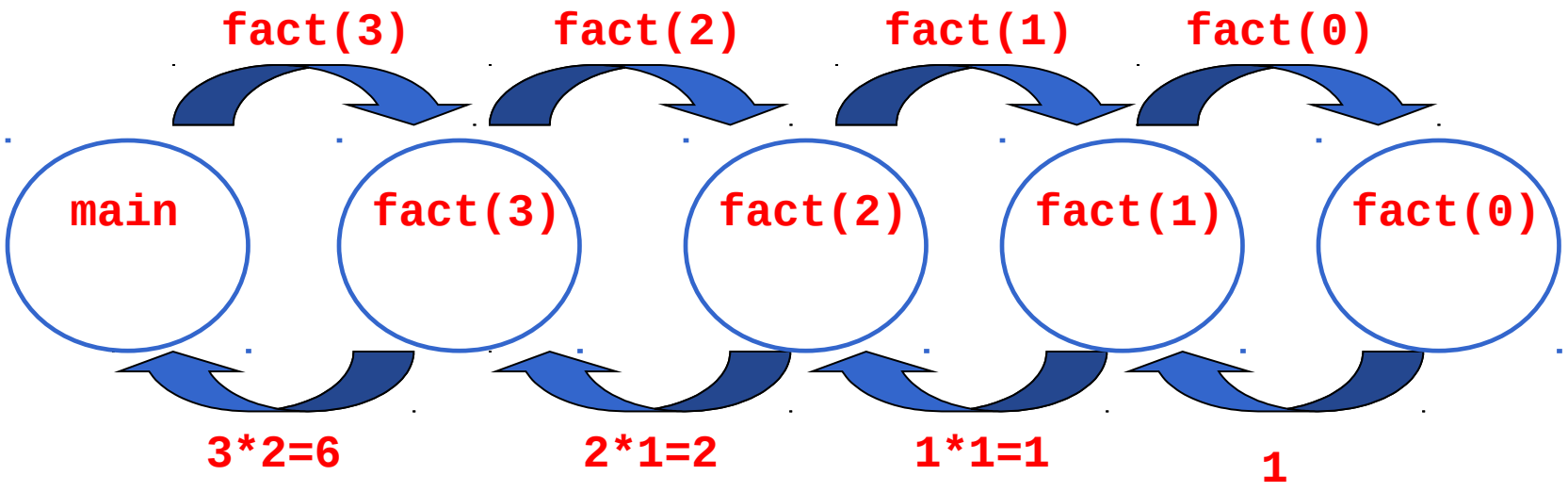
Codifica:

```
int fact(int n)
{ if (n<=0) return 1;
  else return n*fact(n-1);
}
```

LA RICORSIONE: ESEMPIO

```
int fact(int n)
{   if (n<=0) return 1;
    else return n*fact(n-1);
}
main()
{   int fz, z = 5;
    fz = fact(z-2);
}
```

LA RICORSIONE: ESEMPIO



main	fact(3)	fact(2)	fact(1)	fact(0)
Cliente di fact(3)	Cliente di fact(2) Servitore del main	Cliente di fact(1) Servitore di fact(3)	Cliente di fact(0) Servitore di fact(2)	Servitore di fact(1)

Cosa succede nello stack ?

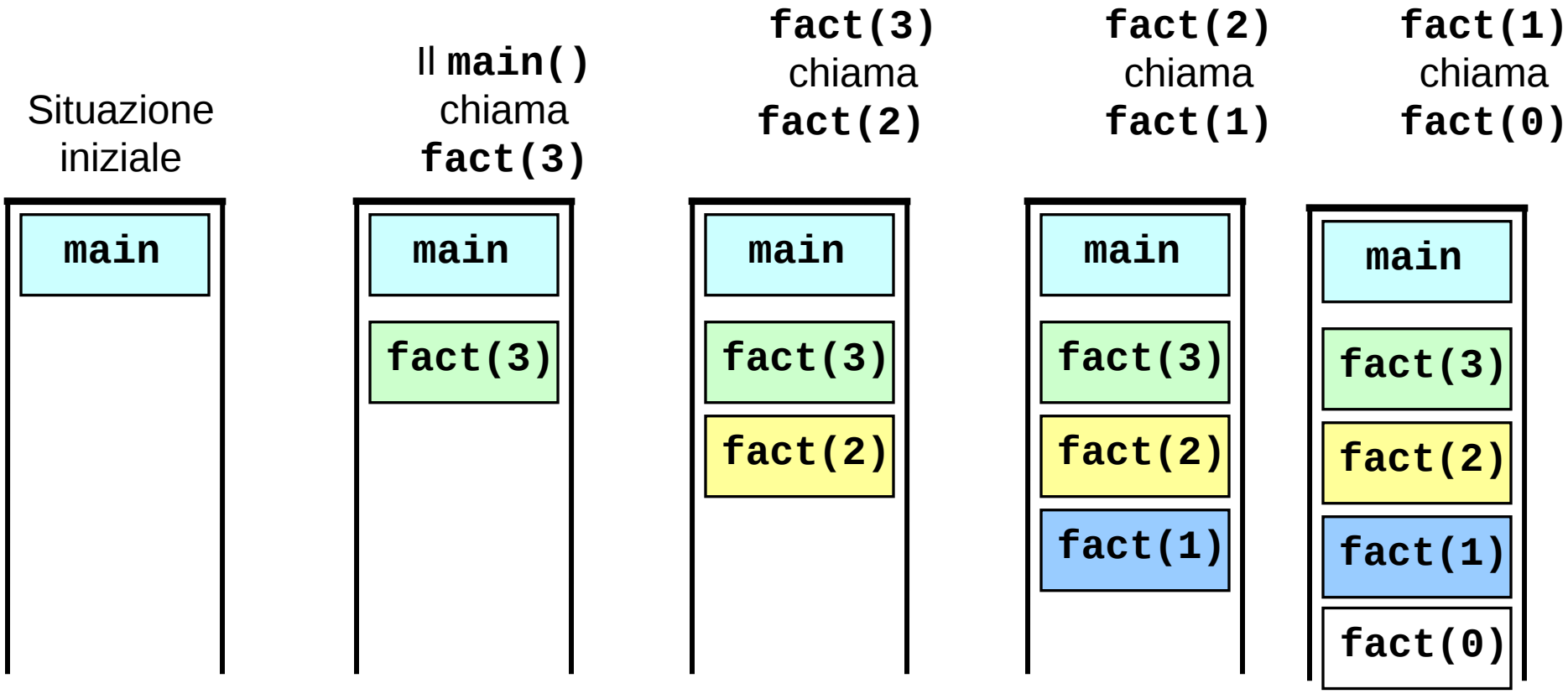
```
int fact(int n)
{  if (n<=0) return 1;
   else return n*fact(n-1);
}

main()
{  int fz,z = 5;
   fz = fact(z-2);
}
```

NOTA: Anche il **`main()`** è una funzione

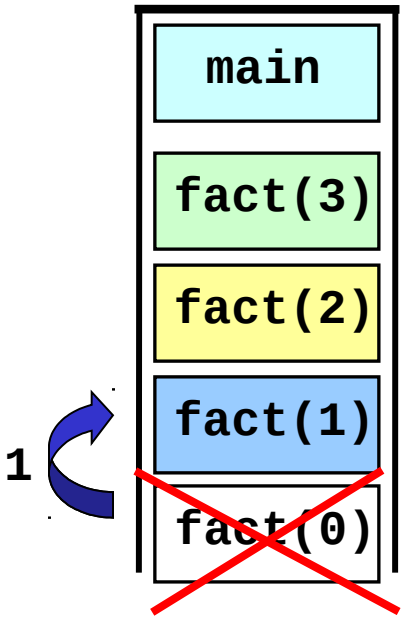
Seguiamo l'evoluzione dello stack durante l'esecuzione:

Cosa succede nello stack ?

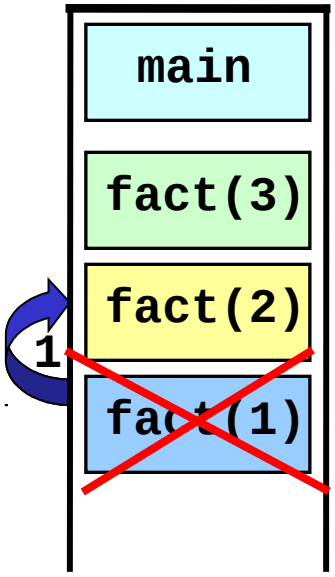


Cosa succede nello stack ?

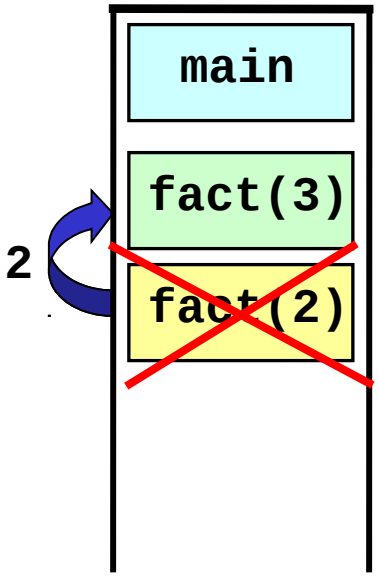
`fact(0)` termina restituendo il valore 1. Il controllo torna a **`fact(1)`**



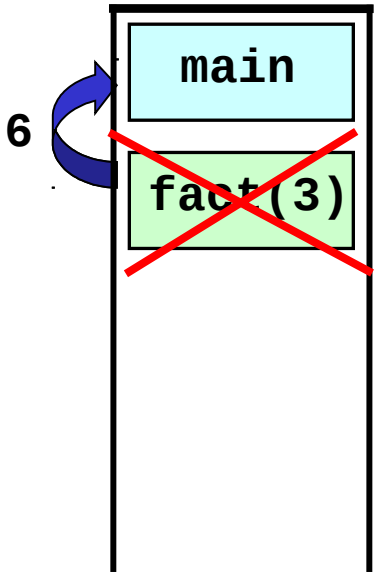
fact(1) effettua la moltiplicazione e termina restituendo il valore 1. Il controllo torna a fact(2)



fact(2) effettua la moltiplicazione e termina restituendo il valore 2. Il controllo torna a fact(3)



fact(3) effettua la moltiplicazione e termina restituendo il valore 6. Il controllo torna al main.



Calcolo della somma dei primi N numeri interi positivi

LA RICORSIONE

- Operativamente, risolvere un problema con un approccio ricorsivo comporta
  1. identificare un "caso base" la cui soluzione sia nota
  2. riuscire a esprimere la soluzione al caso generico *n* in termini dello *stesso problema* *in uno o più casi più semplici* (n-1, n-2, etc).

In un certo senso, può essere visto come un'implementazione del principio del "divide et impera"

Somma elementi di un array

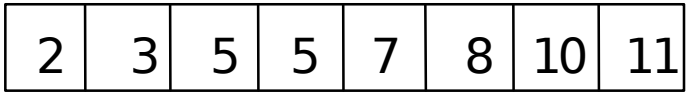
- Si scriva una funzione ricorsiva

  `somma(int a[], int n);`

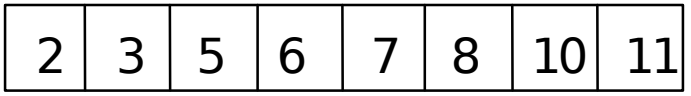
  che calcola la somma degli elementi dell'array **`a`** dall'indice 0 fino all'indice **`n`**

Ricerca in array ordinato

- Sapendo che il vettore è ***ordinato***, la ricerca può essere ottimizzata.
  - ***Vettore ordinato in senso non decrescente:***
    - Esiste una relazione d'ordine totale sul dominio degli elementi del vettore e:
    - Se **`i<j`** si ha **`V[i]<=V[j]`**



  - ***Vettore ordinato in senso crescente:***
    - Se **`i<j`** si ha **`V[i]<V[j]`**



- In modo analogo si definiscono l'ordinamento in senso ***non crescente*** e ***decrescente***.

RICERCA BINARIA

- Ricerca binaria di un elemento in un vettore ordinato in senso non decrescente in cui il primo elemento è **`first`** e l'ultimo **`last`**.

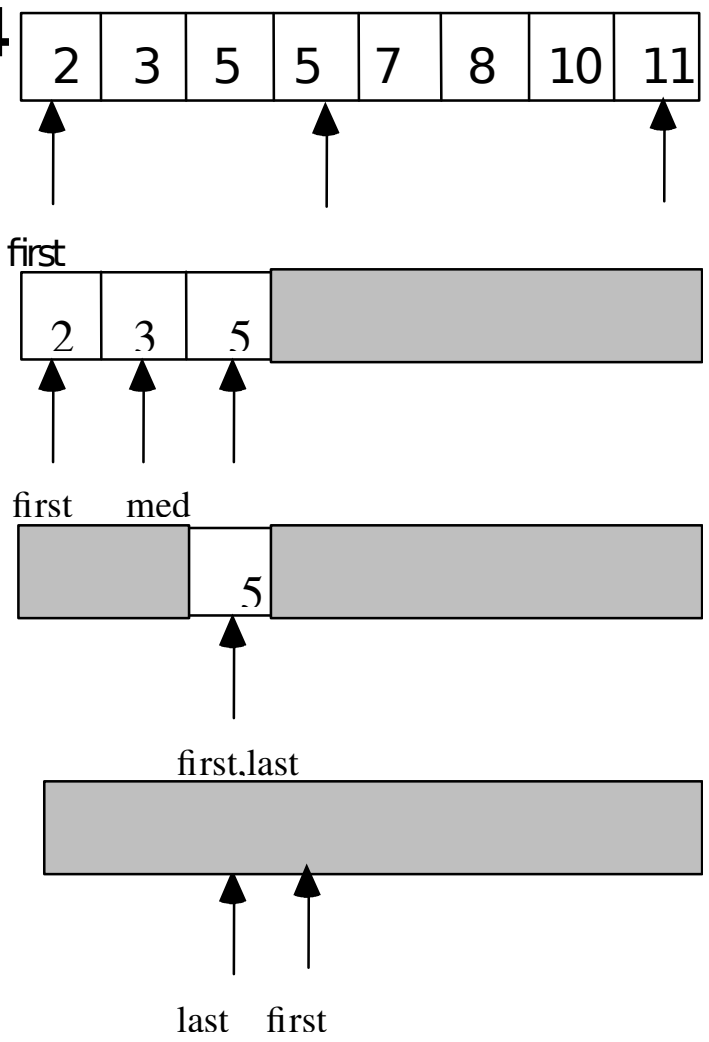
- La tecnica di ***ricerca binaria*** rispetto alla ricerca esaustiva consente di eliminare ad ogni passo metà degli elementi del vettore.

RICERCA BINARIA

- Se first<=last
  - Confronta l'elemento cercato **`el`** con quello mediano del vettore, **`V[`*`med`*`]`**.
  - Se **`el ==V[med]`**, fine della ricerca
  - Altrimenti,
    - se **`el<V[med]`**, ripeti la ricerca nella prima metà del vettore (indici da **`first`** a **`med-1`**);
    - se **`el>V[med]`**, ripeti la ricerca nella seconda metà del vettore (indici da **`med+1`** a **`last`**).

RICERCA BINARIA

- Esempio: si cerca il valore **el=4**
- **med = (first+last)/2**
- **el < V[med]**
- **el > V[med]**
- **el < V[med]**
- **fine**



RICERCA BINARIA

```
int ricerca_bin(int vet[], int el, int first,
  int last)
{ int med = (first+last)/2;
  if (first>last) return -1;
  if (vet[med]==el)
     return med;
  if (el < vet[med])
     return ricerca_bin(vet,el,first,med-1);
  else
     return ricerca_bin(vet,el,med+1,last);
}
```

ESERCIZIO: Ricerca di un elemento

```
#include <stdio.h>
#define N 15
int ricerca_bin (int[],int,int,int);

main ()
{int i,n,pos;
 int a[N];
 printf("Inserisci valori ordinati\n");
 n=leggi_vettore(a);
 printf ("Valore da cercare: ");
 scanf ("%d",&i);
 pos=ricerca_bin(a,i,0,n);
 if (pos<0) printf("\nNon Trovato\n");
    else printf("\nTrovato in pos %d\n",pos);
}
```

RICERCA BINARIA

- Un array contiene
  - *nome* (stringa di 20 caratteri)
  - *via* (stringa di 20 caratteri)
  - *numero civico* (intero)

  per un insieme di persone.
- l'array è ordinato per il campo *nome*
- Si scriva una funzione ricorsiva che effettua la ricerca binaria sull'array

Ricerca binaria su file

Esercizio:

- Un file binario INDIRIZZI.IND contiene
  - *nome* (stringa di 20 caratteri)
  - *via* (stringa di 20 caratteri)
  - *numero civico* (intero)

  per un insieme di persone.
- Il file è ordinato per il campo *nome*
- Si scriva una funzione ricorsiva che effettua la ricerca binaria sul file
- Suggerimento: Si utilizzi la **`fseek`** per posizionarsi sull'elemento corretto nel file

Esercizio

- Si scriva una funzione ricorsiva che calcola la potenza n^m usando il seguente algoritmo:
- se $m = 0$, allora $n^m = 1$
- se m è dispari, allora $n^m = n^{m-1}\, n$
- se m è pari, allora $n^m = (n^{m/2})^2$

QUICKSORT

Iterativo

```
Procedure QuickSort(A,First,Last);
 A is an array to be sorted for elements First to Last inclusive.
 v is a variable of type corresponding to the sort key of array A.
 sp is a stack pointer to a small local data structure used by Push2 and Pop2.
 var l,l2,p,r,r2: longint; of a type equivalent to First and Last.
 BEGIN
  sp:=0;                          {The local stack is empty.}
  Push2(first,last);                 {The task is to sort this span.}
  while sp > 0 do                    {While spans yet remain,}
   begin                          {Get stuck in.}
    Pop2(l,r);                     {The task of the moment.}
    While l < r do                   {Outer jaws: span is l:r}
     begin                         {Want to split the span.}
      p:=(l + r) div 2;              {It would be nice if the median value were here.}
      v:=A(p).key;                   {The Value must not stay in the array!}
      l2:=l; r2:=r;                 {Prepare my inner teeth.}
      repeat                         {Start the inner jaws.}
       while A(l2).key < v do l2:=l2 + 1;  {Scan the left partition.}
       while A(r2).key > v do r2:=r2 - 1;  {Scan the right partition.}
       if l2 <= r2 then                {Wasteful multiple testing }
        begin                         { to avoid feared goto, etc.}
         if l2 <> r2 then Swap (A(l2),A(r2)); {An early stop.}
         l2:=l2 + 1; r2:=r2 - 1;          {The hiccough has been cleared.}
        end;                          {Prepare to resume scanning.}
      until l2 >= r2;                 {Finish here with the desired state attained.}
      if r2 - l > r - l2 then         {Which is the larger piece?}
       begin                         {The left side piece.}
        if l < r2 then Push2(l,r2);     {Deal with it later.}
        l:=l2;                       {Will shortly attack a new l:r.}
       end                            {And come back to the other piece later.}
       else                           {But the rh. side piece may be larger.}
        begin                         {So the lh. is the smaller.}
         if l2 < r then Push2(l2,r);    {Ignoring partitions of length one.}
         r:=r2;                       {Either way, l and r approach each other.}
        end;                          {One end or the other being adjusted.}
     end; {While l < r.}                {Until the span has fewer than 2 elements}
   end; {Playing with sp.}              {In which case we can unstack a span.}
 END;
```

Ricorsivo

```
function quicksort(a, left, right)
   while right > left
      select a pivot value a[pivotIndex]
      pivotNewIndex := partition(a, left, right, pivotIndex)
      leftSize := (pivotNewIndex-1) - left + 1
      rightSize := right - (pivotNewIndex + 1) + 1
      if leftSize < rightSize
         quicksort(a, left, pivotNewIndex-1)
         left  := pivotNewIndex+1
      else
         quicksort(a, pivotNewIndex+1, right)
         right := pivotNewIndex-1
```

UN ESEMPIO PIU` COMPLESSO

- La ricorsione è utile per rendere semplici problemi complessi
- Esempio: Scrivere un programma che stampa tutte gli anagrammi di una stringa data
- Es **`lia → ial ail ali lai ila lia`**
- Caso base: una stringa di 1 solo carattere ha solo un anagramma (se stesso)
- Condizione di ricorsione:
  - ho già la funzione che stampa le stringhe in cui sono permutati solo i primi *n*-1 caratteri
  - metto in fondo (al posto *n*) il primo carattere e stampo le permutazioni dei primi *n*-1
  - poi metto in fondo il secondo e stampo le permutazioni dei primi *n*-1
  - ...
  - metto in fondo l'*n*-esimo e stampo le permutazioni dei primi *n*-1 caratteri

SOLUZIONE

```
void anagrammi(char s[], int n)
{   char temp;
    int i;
    if (n==0)
        printf("%s\n",s);
    else
    {
        for (i=0; i<=n; i++)
        {
             temp=s[i]; s[i]=s[n]; s[n]=temp;
             anagrammi(s,n-1);
             /* s[n]=temp; */ s[n]=s[i]; s[i]=temp;
        }
    }
}

main()
{   char s[30];
    int n;
    printf("Inserisci la stringa ");
    scanf("%s",s);
    n=strlen(s)-1;
    anagrammi(s,n);
}
```

Se devo permutare un solo carattere, stampo tutta la stringa

scambio l'elemento di indice *i* con l'ultimo

rimetto a posto per la prossima invocazione

LA RICORSIONE: ESEMPIO

Problema:
calcolare l'N-esimo numero di Fibonacci

$$\text{fib}(n) = \begin{cases} 0, & \text{se } n=0 \\ 1, & \text{se } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{altrimenti} \end{cases}$$

LA RICORSIONE: ESEMPIO

Problema:

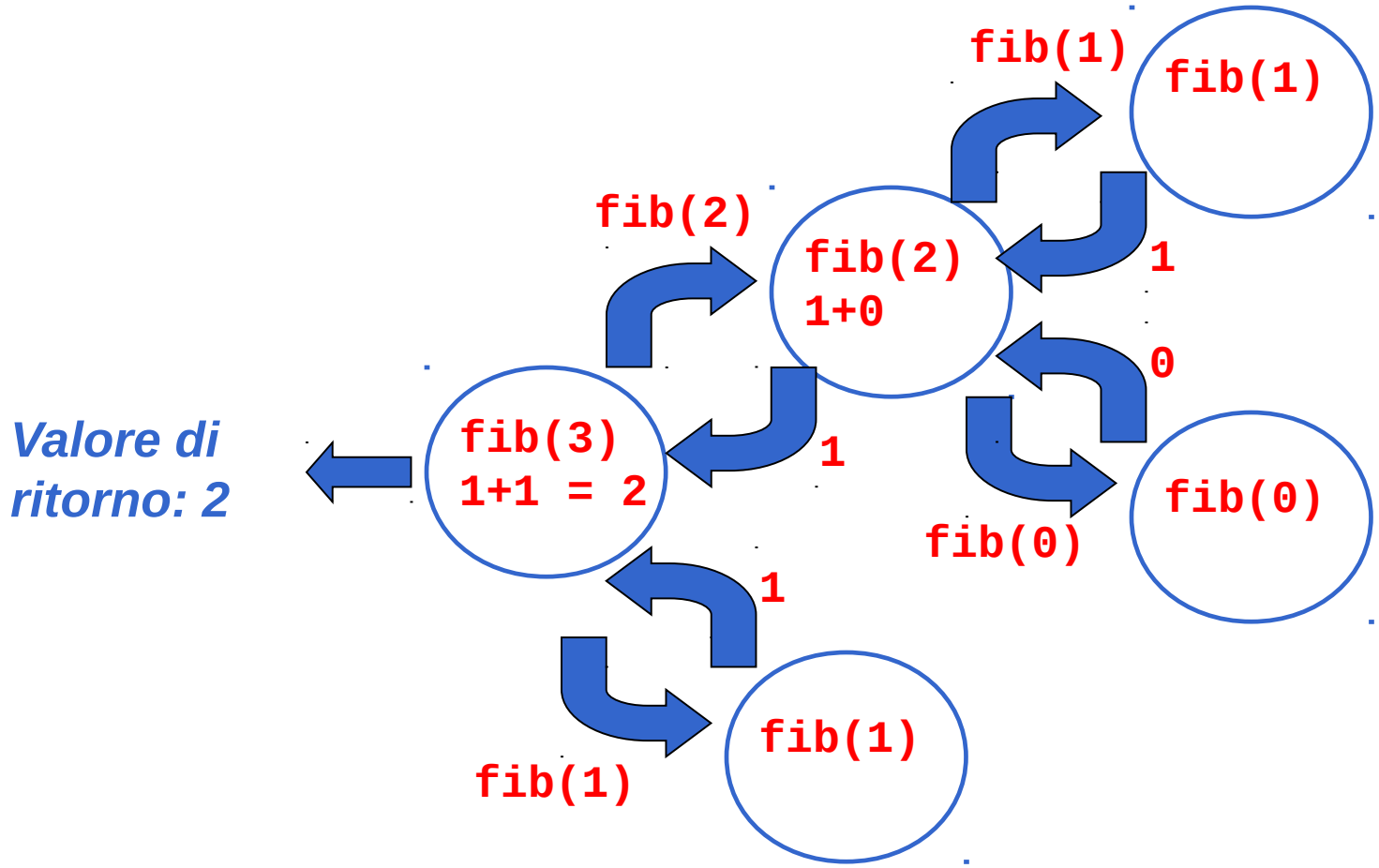
calcolare l'N-esimo numero di Fibonacci

Codifica:

```
unsigned fibonacci(unsigned n)
{ if (n<2) return n;
  else return fibonacci(n-1)+fibonacci(n-2);
}
```

Ricorsione non lineare: ogni invocazione del servitore causa due nuove chiamate al servitore medesimo.

LA RICORSIONE: ESEMPIO



Una riflessione

- La ricorsione permette di affrontare problemi complessi
- però introduce alcune inefficienze
  - memoria (catene di record di attivazione)
  - tempo (creazione del nuovo record)

- Ci possono essere casi in cui *non è necessario* creare un nuovo record di attivazione?

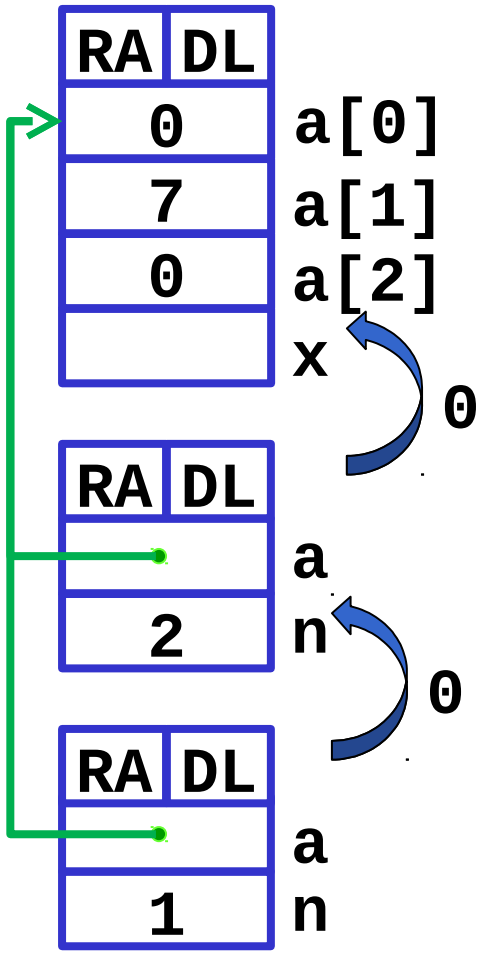
ESERCIZIO

- calcolare se un array è costituito da valori tutti nulli

ESERCIZIO

```
int nullo(int a[], int n)
{ if (n<0) return 1;
  if (a[n]==0)
     return nullo(a,n-1);
     else return 0;
}

main()
{ int a[3]={0,7,0};
  int x;
  x = nullo(a,2);
}
```

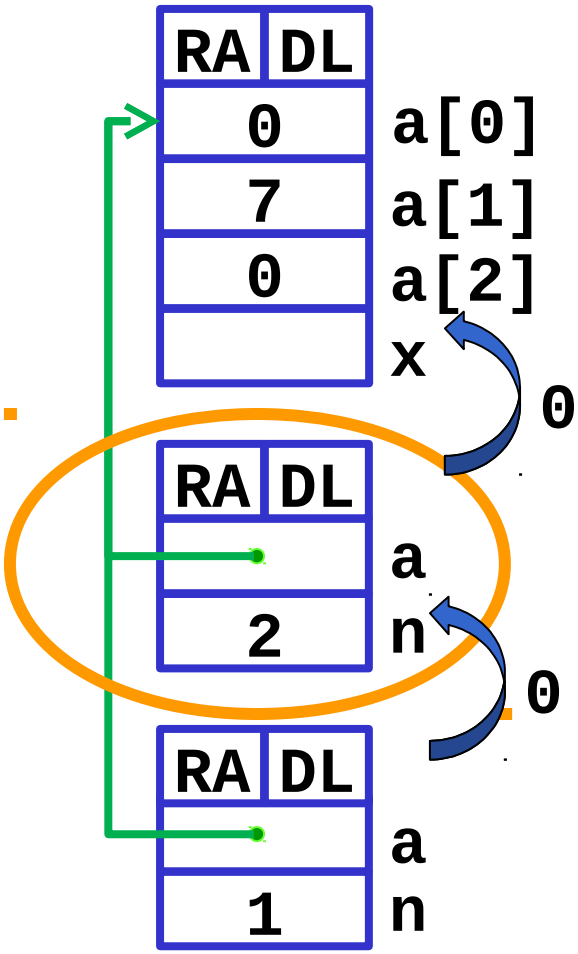


ESERCIZIO

```
int nullo(int a[], int n)
{ if (n<0) return 1;
  if (a[n]==0)
     return nullo(a,n-1);
     else return 0;
}

m
{
}
```

Questa invocazione non calcola nulla!
Deve solo restituire lo stesso risultato che ottiene dall'invocazione successiva!
Non userò nessuna delle informazioni che sono nel record di attivazione!
Ma allora, perché lo memorizzo?

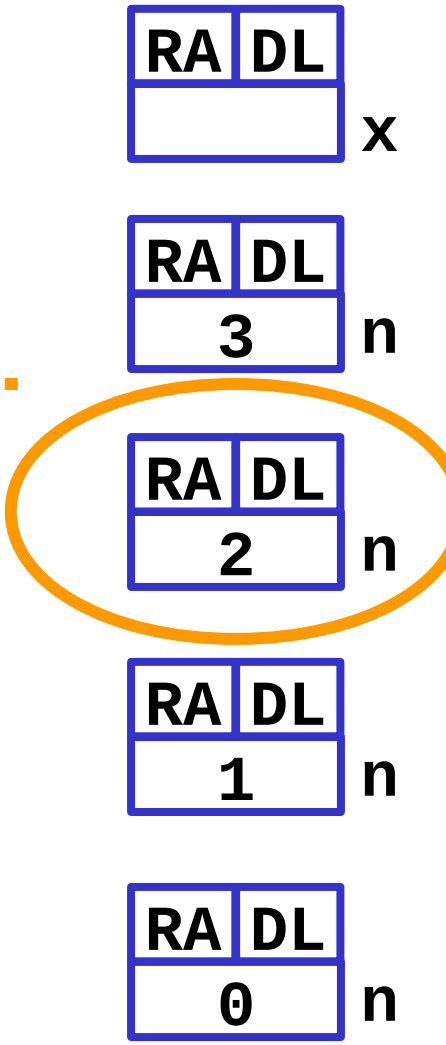


MA È SEMPRE COSÌ?

```
int fact(int n)
{ if (n==0) return 1;
  else return fact(n-1)*n;
}

main()
{ int x;
  x = fact(3);
}
```

Questa deve ancora calcolare il prodotto 2*1

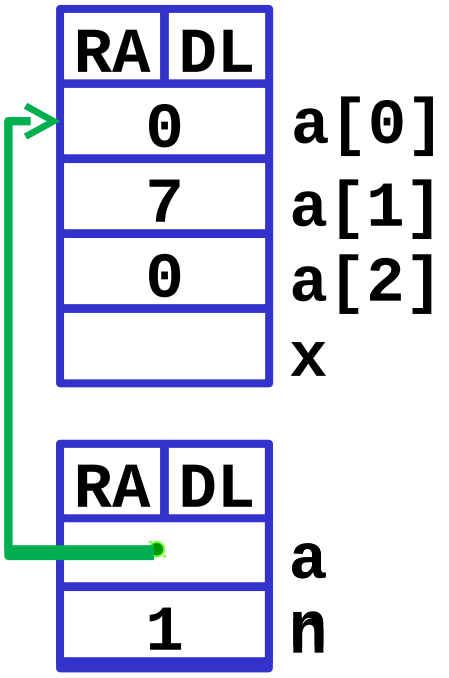


ESERCIZIO

```
int nullo(int a[], int n)
{ if (n<0) return 1;
  if (a[n]==0)
     return nullo(a,n-1);
     else return 0;
}

m
{
  x = nullo(a,2);
}
```

Un compilatore ottimizzante potrebbe accorgersi che dopo l'invocazione non c'è niente da fare e riutilizzare il record di attivazione dell'invocazione precedente



Sfruttiamo le ottimizzazioni!

- Se ho un compilatore ottimizzante, vale la pena usarlo!
- Riusciamo a riscrivere la funzione fattoriale in modo da riuscire a sfruttare le ottimizzazioni del compilatore?
- Dobbiamo fare in modo che l'ultima operazione sia la chiamata ricorsiva

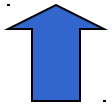
RICORSIONE TAIL

UNA RIFLESSIONE

- Nel caso del fattoriale si inizia a sintetizzare il risultato solo dopo che si sono aperte tutte le chiamate, *"a ritroso"*, mentre le chiamate si chiudono.

> *Le chiamate ricorsive decompongono via via il problema, ma non calcolano nulla*

- Il risultato viene sintetizzato *a partire dalla fine, perché prima occorre arrivare al caso "banale"*:
  - il caso "banale" fornisce il valore di partenza
  - poi si sintetizzano, "a ritroso", i successivi risultati parziali.

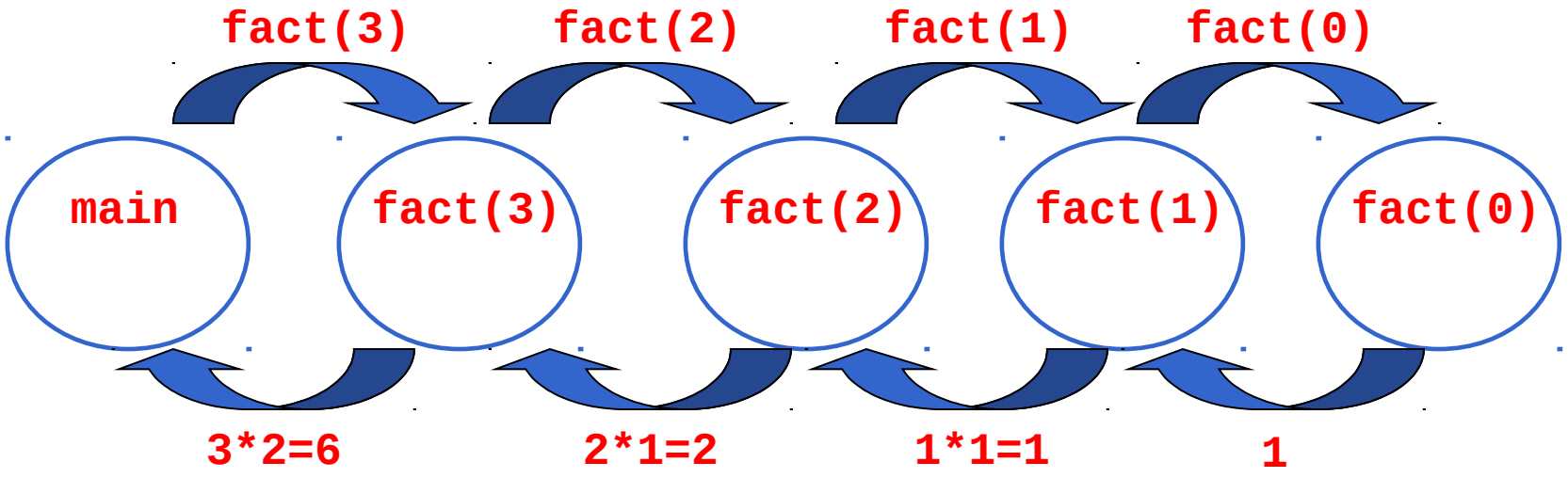


Processo computazionale effettivamente ricorsivo

LA RICORSIONE

PASSI:
1) **`fact(3)`** chiama **`fact(2)`** passandogli il controllo,
2) **`fact(2)`** calcola il fattoriale di 2 e termina restituendo 2
3) **`fact(3)`** riprende il controllo ed effettua la moltiplicazione 3*2
4) termina anche **`fact(3)`** e il controllo torna al main

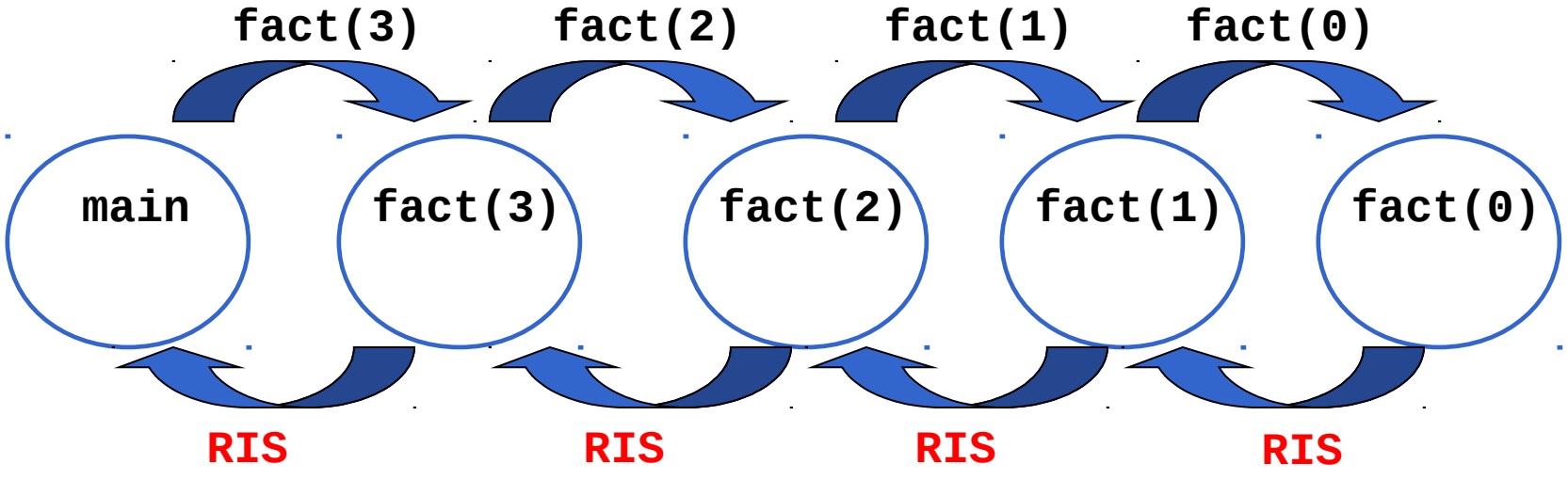
```
int fact(int n)
{ if (n==0)
    return 1;
  else
    return n*fact(n-1);
}
```



Rendiamo più efficiente (1)

- Per sfruttare l'ottimizzazione, ogni invocazione dopo la chiamata ricorsiva deve solo restituire il risultato
- quindi restituisce lo stesso valore che ha ricevuto

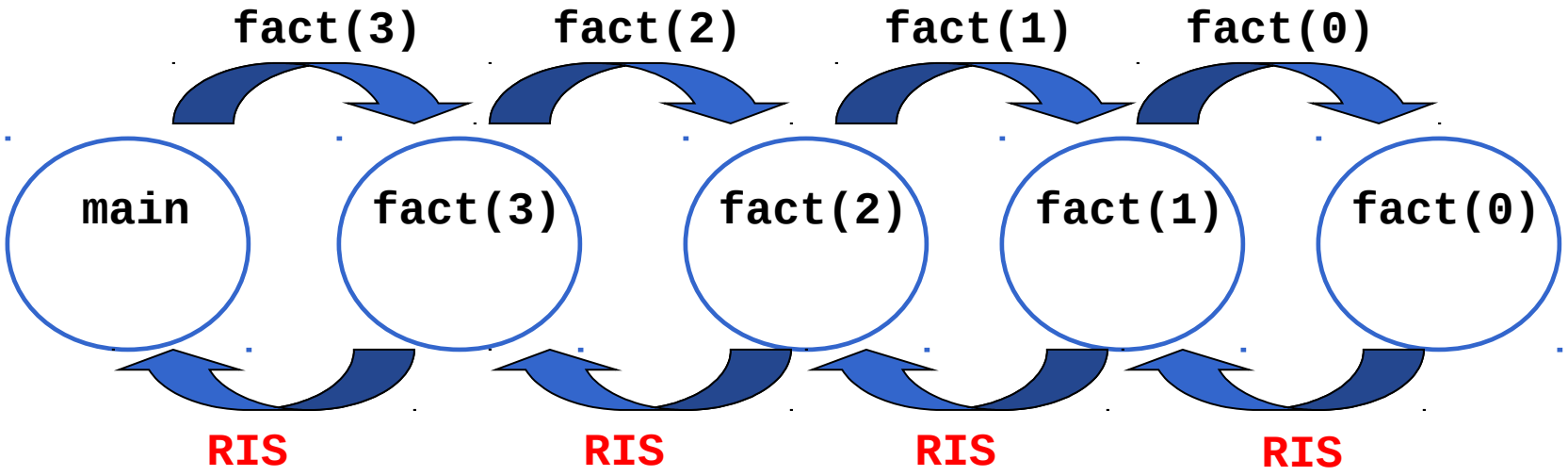
```
int fact(int n)
{ if (n==0)
    return RIS;
  else
    return fact(n-1);
}
```



Rendiamo più efficiente (2)

- Quindi i prodotti vanno calcolati prima della chiamata ricorsiva

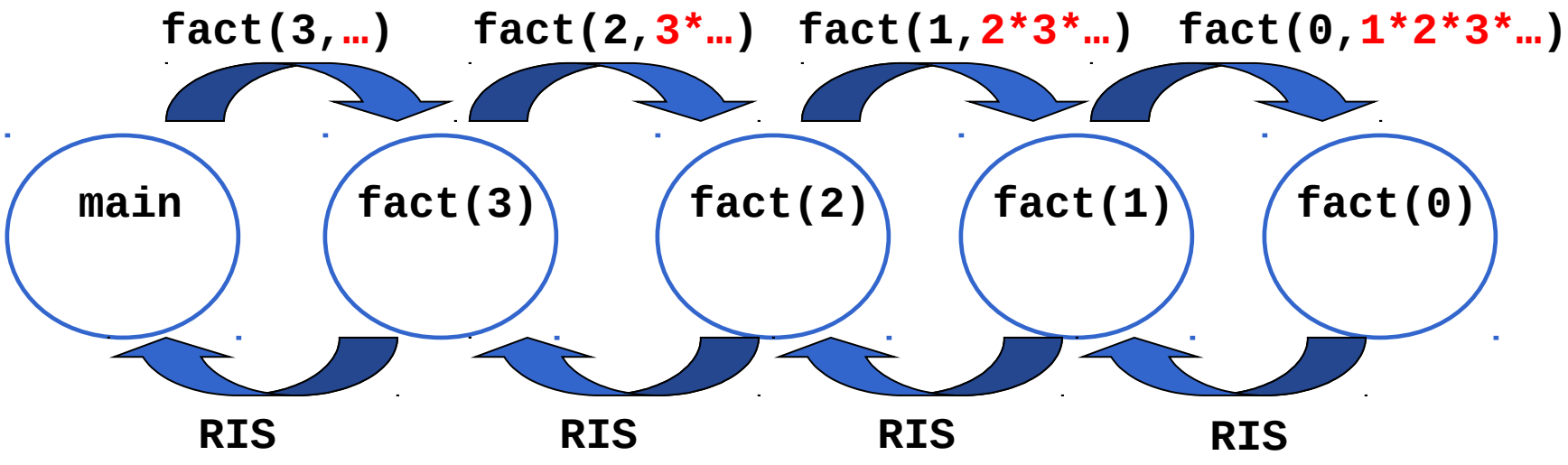
```
int fact(int n)
{ if (n==0)
    return RIS;
  else
  { RIS = n*RIS;
    return fact(n-1);
  }
}
```



Rendiamo più efficiente (3)

- Quindi devo comunicare "*in avanti*" il valore del prodotto parziale
- Quindi devo aggiungere un parametro

```
int fact(int n, int RIS)
{ if (n==0)
    return RIS;
  else
  { RIS = n*RIS;
    return fact(n-1,RIS);
  }
}
```

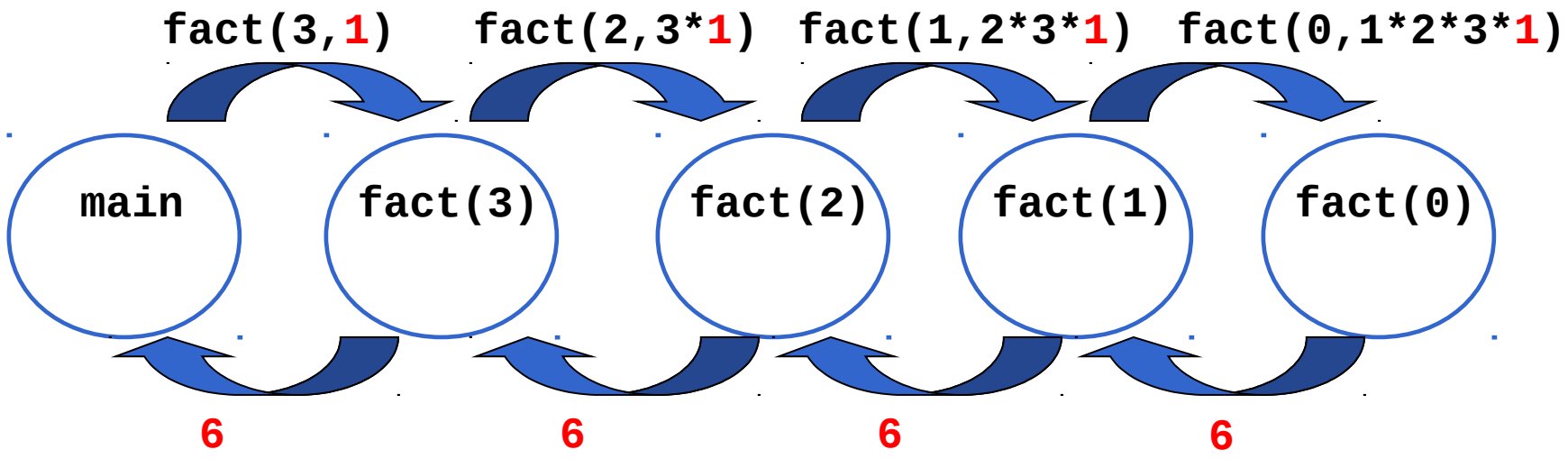


Rendiamo più efficiente (4)

- Quindi il **main** deve dare il valore iniziale di questo parametro

```
main()
{  int f,x=3;
   f=fact(x,1);
}
```

```
int fact(int n, int RIS)
{ if (n==0)
    return RIS;
  else
  { RIS = n*RIS;
    return fact(n-1,RIS);
  }
}
```



ESERCIZIO

- Sia data la seguente funzione che calcola la potenza di un numero

```
double rec_pow(double base, long int esp)
{  double p;
   if (esp==0)
      return 1;
   else
      return rec_pow(base,esp-1)*base;
}
```

- si modifichi la funzione in modo che la chiamata ricorsiva sia l'ultima operazione effettuata

SOLUZIONE

```
double tail_pow(double base, long int esp, double acc)
{
   if (esp==0)
     return acc;
   else
     return tail_pow(base,esp-1,acc*base);
}
```

gcc

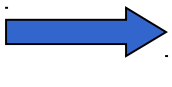
- Il gcc con l'opzione –O2 fa varie ottimizzazioni, fra cui l'ottimizzazione tail
- **`gcc -O2 power.c`**
- calcolo della potenza, con esponente 350.000:

	rec_pow	tail_pow
gcc power.c	Segmentation fault	Segmentation fault
gcc -O2 power.c	Segmentation fault	*Ok!*

FATTORIALE ITERATIVO

- Per rendere tail-ricorsiva una funzione, si può partire dalla versione iterativa
- Il fattoriale si può anche calcolare con un algoritmo iterativo

```
int fact(int n)
{  int i;
   int F=1; /*inizializzazione del fattoriale*/
   for (i=2;i <= n; i++)
      F=F*i;
   return F;
}
```



DIFFERENZA CON LA VERSIONE RICORSIVA: ad ogni passo viene accumulato un risultato intermedio

FATTORIALE ITERATIVO

```
int fact(int n)
{  int i;
      int F=1; /*inizializzazione del fattoriale*/
      for (i=2;i <= n; i++)
          F=F*i;
      return F;
}
```

La variabile F accumula risultati intermedi: se n = 3 inizialmente **F=1** *poi al primo ciclo for* **i=2 F** *assume il valore 2. Infine all'ultimo ciclo for* **i=3 F** *assume il valore 6.*

- **Al primo passo F accumula il fattoriale di 1**
- **Al secondo passo F accumula il fattoriale di 2**
- **Al i-esimo passo F accumula il fattoriale di i**

PROCESSO COMPUTAZIONALE ITERATIVO

- In questo caso il risultato viene sintetizzato *“in avanti”*
- Ogni processo computazionale che computi “in avanti”, per accumulo, costituisce una *ITERAZIONE* ossia è un *processo computazionale iterativo.*
- La caratteristica fondamentale di un processo computazionale *ITERATIVO* è che *a ogni passo è disponibile un risultato parziale*
  - dopo k passi, si ha a disposizione il risultato parziale relativo al caso k
  - questo *non è vero* nei processi computazionali *ricorsivi,* in cui nulla è disponibile finché non si è giunti fino al caso elementare.

Processo computazionale iterativo

- Un processo computazionale iterativo si può realizzare anche tramite funzioni ricorsive
- Si basa sulla disponibilità di una variabile, detta *accumulatore*, destinata a esprimere *in ogni istante* la soluzione corrente
- Si imposta identificando quell'operazione di *modifica dell'accumulatore* che lo porta a esprimere, dal valore relativo al passo k, il valore relativo al passo k+1.

ITERAZIONE E RICORSIONE TAIL

- **il corpo del ciclo rimane *immutato***
- **il ciclo diventa un `if` con, in fondo, la chiamata tail-ricorsiva.**

`while (condizione)`
`{`
`  <corpo del ciclo>`
`}`
	`if (condizione)`
	`{ <corpo del ciclo>`
	`  <chiamata ricorsiva>`
	`}`

Naturalmente, può essere necessario *aggiungere nuovi parametri* nell'intestazione della funzione tail-ricorsiva, per "portare avanti" le variabili di stato.

FATTORIALE ITERATIVO

```
int fact(int n){
  return factIter(n,1,1);
}
```

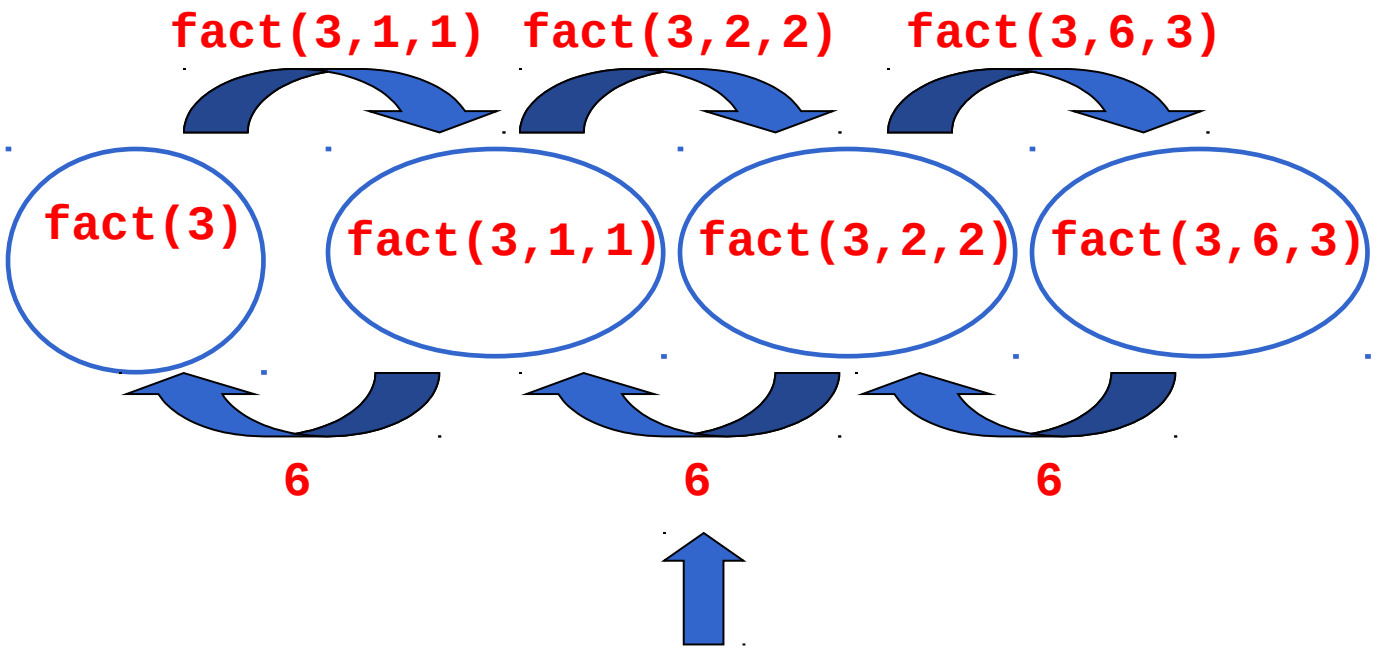
Inizializzazione dell'accumulatore: corrisponde al fattoriale di 1

Contatore del passo

```
int factIter(int n, int F, int i)
{
    if (i < n)
     {F = (i+1)*F;
      i = i+1;
      return factIter(n,F,i);
     }
    else
      return F;
}
```

Accumulatore del risultato parziale

LA RICORSIONE



Al passo i-esimo viene calcolato il fattoriale di i. Quando i = n l'attivazione della funzione corrispondente calcola il fattoriale di n.
NOTA: ciascuna funzione che effettua una chiamata ricorsiva si sospende, aspetta la terminazione del servitore e poi termina, cioè **NON EFFETTUA ALTRE OPERAZIONI DOPO** come succedeva nel caso del fattoriale ricorsivo vero e proprio che dopo la fine del servitore si doveva effettuare una moltiplicazione

RIASSUMENDO….

- La soluzione ricorsiva individuata per il fattoriale è *sintatticamente ricorsiva* ma dà luogo a un *processo computazionale ITERATIVO*

Ricorsione apparente detta RICORSIONE TAIL

- Il risultato viene sintetizzato *in avanti*
  - ogni passo *decompone e calcola*
  - *e porta in avanti il nuovo risultato parziale* quando le chiamate si chiudono non si fa altro che riportare indietro, fino al cliente, il risultato ottenuto.

RICORSIONE TAIL

- Una ricorsione che realizza un processo computazionale *ITERATIVO* *è una ricorsione apparente*
- la chiamata ricorsiva è *sempre l'ultima istruzione*
  - *i calcoli sono fatti prima*
  - *la chiamata serve solo, dopo averli fatti, per proseguire la computazione*
- questa forma di ricorsione si chiama *RICORSIONE TAIL* ("ricorsione in coda")