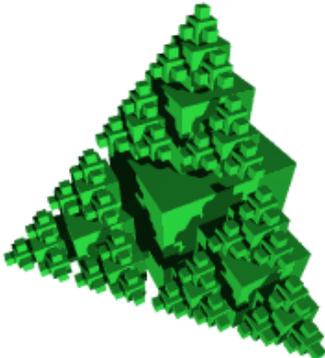


LA RICORSIONE



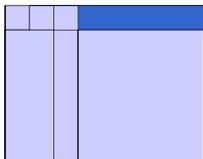
Ricorsione

- 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
 $\langle \text{espressione} \rangle ::= \langle \text{variabile} \rangle \mid \langle \text{costante} \rangle \mid$
 $\langle \text{espressione} \rangle \langle \text{operatore} \rangle \langle \text{espressione} \rangle \mid$
 $\langle \text{operatore_unario} \rangle \langle \text{espressione} \rangle$
- 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;
  ...
}
```

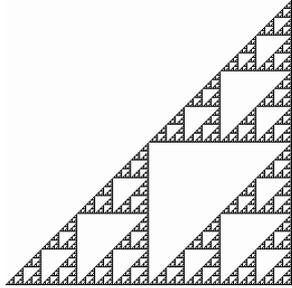
Es: calcolo della derivata

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

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

LA RICORSIONE

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



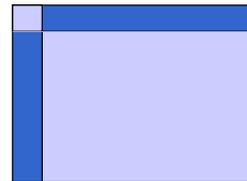
Quando mi fermo?

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
```

PROGRAMMAZIONE TOP-DOWN

- In molti casi, il problema dato è troppo difficile per scrivere il programma "di getto"
- Scomponiamo il problema in raffinamenti successivi:
 1. livello di specifica
 2. livello di specifica
 3. ...
- Le funzioni ci aiutano in questa scomposizione
 1. prima scriviamo il main() usando funzioni che definiremo poi
 2. poi definiamo le funzioni usate dal main() usando altre funzioni,
 3. poi definiamo le funzioni ancora non definite,
 4. ...

ESEMPIO

```
int g(int x)
{ return sin(x)+3;
}
int f(int x)
{ return g(x)+g(x-1);
}
main()
{/* definizione variabili */
    a=f(x);
    ...
}
```

LA RICORSIONE

- In particolare, anche nella *definizione* di una funzione $f()$ può essere utile riutilizzare la funzione $f()$ stessa
- E' lo stesso principio usato in matematica:

Principio di Induzione

- se una proprietà P vale per $n=n_0$ ➡ CASO BASE
- e si può provare che, *assumendola valida per n* , allora vale per $n+1$
allora P vale per ogni $n \geq n_0$

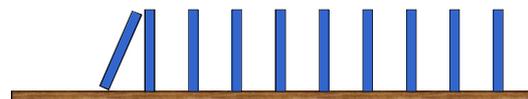
LA RICORSIONE

- 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



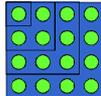
- Es. ho una sequenza infinita di pezzi del domino disposti come in figura
 - il primo cade
 - Il primo fa cadere il secondo
 - il secondo fa cadere il terzo
 - ...
- Come faccio a dimostrare che cadono tutti?
- Col principio di induzione posso farlo, sapendo che
 - il primo cade
 - se cade l' n -esimo, 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 $1+3+5+\dots+(2n-1) = n^2$ e dimostriamo che è vera per $n+1$:
sommiamo $(2n+1)$ ad entrambi i termini:
 $1 + 3 + 5 + \dots + (2n-1) + (2n+1) = n^2 + 2n + 1$



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!: $\mathbb{Z} \rightarrow \mathbb{N}$

n! vale 1 se $n \leq 0$

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

- **Servitore & Cliente:**

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

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

Si valuta l'espressione che costituisce il parametro attuale (nell'environment del main) e si trasmette alla funzione fatt una copia del valore così ottenuto (3).

LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

```
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 funzione fact lega il parametro n a 3. Essendo 3 positivo si passa al ramo else. Per calcolare il risultato della funzione e' necessario effettuare una nuova chiamata di funzione. n-1 nell'environment di fact vale 2 quindi viene chiamata fact(2)

LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

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

Il nuovo servitore lega il parametro n a 2. Essendo 2 positivo si passa al ramo else. Per calcolare il risultato della funzione e' necessario effettuare una nuova chiamata di funzione. n-1 nell'environment di fact vale 1 quindi viene chiamata fact(1)

LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

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

Il nuovo servitore lega il parametro n a 1. Essendo 1 positivo si passa al ramo else. Per calcolare il risultato della funzione e' necessario effettuare una nuova chiamata di funzione. n-1 nell'environment di fact vale 0 quindi viene chiamata fact(0)

LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

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

Il nuovo servitore lega il parametro n a 0. La condizione n <= 0 e' vera e la funzione fact(0) restituisce come risultato 1 e termina.

LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

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

*Il controllo torna al servitore precedente fact(1) che puo' valutare l'espressione n * 1 (valutando n nel suo environment dove vale 1) ottenendo come risultato 1 e terminando.*

LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

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

*Il controllo torna al servitore precedente fact(2) che puo' valutare l'espressione n * 1 (valutando n nel suo environment dove vale 2) ottenendo come risultato 2 e terminando.*

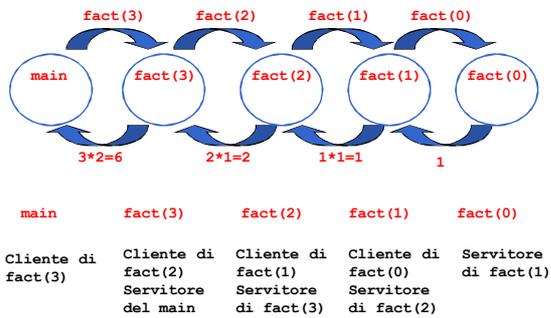
LA RICORSIONE: ESEMPIO

- **Servitore & Cliente:**

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

*Il controllo torna al servitore precedente fact(3) che puo' valutare l'espressione n * 2 (valutando n nel suo environment dove vale 3) ottenendo come risultato 6 e terminando.
IL CONTROLLO PASSA AL MAIN CHE ASSEGNA A fz IL VALORE 6*

LA RICORSIONE: ESEMPIO



Cosa succede nello stack ?

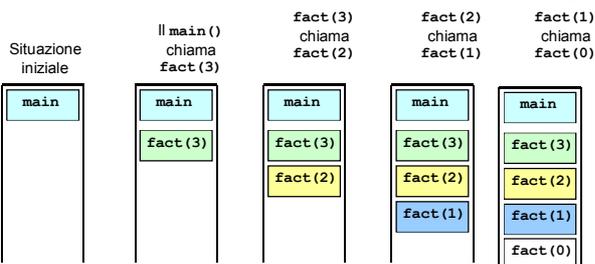
```
int fact(int n)
{ if (n<=0) return 1;
  else return n*fact(n-1);
}

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

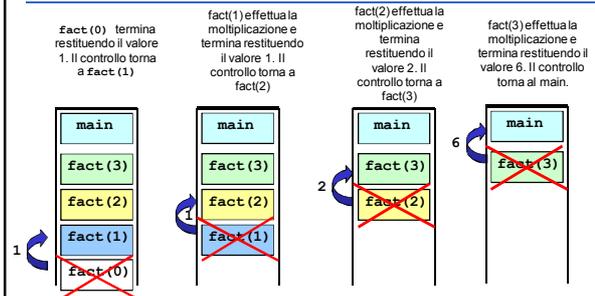
NOTA: Anche il main() è una funzione

Seguiamo l'evoluzione dello stack durante l'esecuzione:

Cosa succede nello stack ?



Cosa succede nello stack ?



Calcolo della somma dei primi N numeri interi positivi

Problema

- Vengono tracciate n rette sul piano
- Non ci sono rette parallele
- Non accade mai che tre (o più) rette si incontrino nello stesso punto
- Si scriva una funzione

```
int regioni(int n)
```

che calcola il numero di regioni in cui viene diviso il piano

Un po' più facile

Ancora più facile

Quando mi fermo?

Problema

- Ora, supponiamo di sapere in quante regioni è diviso il piano con $n-1$ rette
- Aggiungiamo la retta n
- Quante nuove regioni vengono create?



LA RICORSIONE: ESEMPIO

Problema:
calcolare la somma dei primi N interi

Specifica:

Considera la somma $1+2+3+\dots+(N-1)+N$ come composta di due termini:

• $(1+2+3+\dots+(N-1))$ →

• N → Valore noto

Il primo termine non è altro che lo stesso problema in un caso più semplice: calcolare la somma dei primi $N-1$ interi

Esiste un caso banale ovvio: CASO BASE

- la somma fino a 1 vale 1.

LA RICORSIONE: ESEMPIO

Problema:
calcolare la somma dei primi N interi

Algoritmo ricorsivo

Se N vale 1 allora la somma vale 1

altrimenti la somma vale N + il risultato della somma dei primi $N-1$ interi

LA RICORSIONE: ESEMPIO

Problema:
calcolare la somma dei primi N interi

Codifica:

```
int sommaFinoA(int n)
{
    if (n==1) return 1;
    else return sommaFinoA(n-1)+n;
}
```

LA RICORSIONE

- Operativamente un problema con comportamento ricorsivo comporta:
 1. identificare un caso base in cui la 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 $v[i] \leq v[j]$

2	3	5	5	7	8	10	11
---	---	---	---	---	---	----	----
 - Vettore ordinato in senso crescente:**
 - Se $i < j$ si ha $v[i] < v[j]$

2	3	5	6	7	8	10	11
---	---	---	---	---	---	----	----
- In modo analogo si definiscono l'ordinamento in senso **non crescente** e **decrescente**.

RICERCA BINARIA

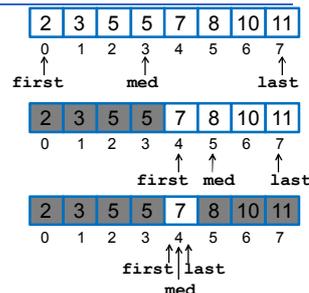
- Ricerca binaria di un elemento in un vettore ordinato in senso non decrescente in cui il primo elemento è **first** e l'ultimo **last**.
- La tecnica di **ricerca binaria** rispetto alla ricerca esaustiva consente di eliminare ad ogni passo metà degli elementi del vettore.

RICERCA BINARIA

- Se $first \leq last$
 - Confronta l'elemento cercato **e1** con quello mediano del vettore, $v[med]$.
 - Se $e1 == v[med]$, fine della ricerca
 - Altrimenti,
 - se $e1 < v[med]$, ripeti la ricerca nella prima metà del vettore (indici da **first** a **med-1**);
 - se $e1 > v[med]$, ripeti la ricerca nella seconda metà del vettore (indici da **med+1** a **last**).

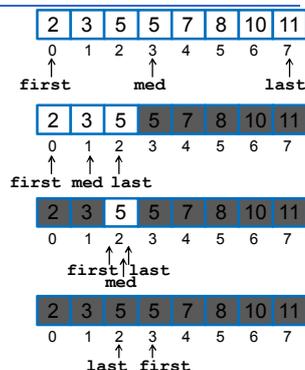
Ricerca Binaria

Si cerca il valore $e1=7$



Ricerca Binaria

Si cerca il valore $e1=4$



RICERCA BINARIA

```
int ricerca_bin(int vet[], int e1, int first,
int last)
{ int med = (first+last)/2;
  if (first>last) return -1;
  if (vet[med]==e1)
    return med;
  if (e1 < vet[med])
    return ricerca_bin(vet,e1,first,med-1);
  else
    return ricerca_bin(vet,e1,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

```
Procedure QuickSort(A:FirstLast);
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;
sp := 1; (2 is 0; longest of a type equivalent to First and Last);
BEGIN
    sp:=0; (The local stack is empty.)
    Push2(firstLast); (The task is to sort this span.)
    while sp > 0 do (While spans yet remain.)
    begin (Get stack in.)
        Pop2(i); (The task of the moment.)
        While i < i do (Outer loop: span is i.)
        begin (Want to split the span.)
            p:=i+1 div 2; (Is would be nice if the median were here.)
            v:=A(p)key; (The Value must not be in the array.)
            repeat (Separate the pieces.)
            until A(i)key < v or A(i)key = v; (The left partition.)
            while A(i)key < v do (Scan the right partition.)
            begin (Wasteful multiple testing.)
                i:=i+1; (to avoid feigning glib; etc.)
            end;
            if i < p then Swap(A(i),A(p)); (An early stop.)
            i:=i+1; r:=i*2-1; (The hiccough has been cleared.)
            end; (Prepare to resume scanning.)
            until i >= r; (Finish here with the desired state attained.)
            if r < i+1 then (Who is the larger piece?)
            begin (The left side piece.)
                if i <= 2 then Push2(i); (Deal with it later.)
                i:=2; (Will shortly attack a new i.)
            end; (And come back to the other piece later.)
            else (But the rth side piece may be larger.)
            begin (So the lth is the smaller.)
                if i < r then Push2(i+1); (Ignoring partitions of length one.)
                r:=2; (Either way, i and r approach each other.)
            end; (One end or the other being adjusted.)
            end; (Until the span has fewer than 2 elements.)
            end; (Playing with sp.)
        END;
    END;
```

Ricorsivo
Iterativo

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:

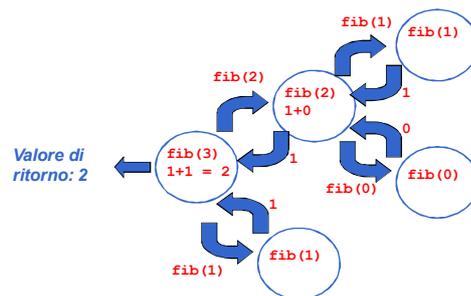
```

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?

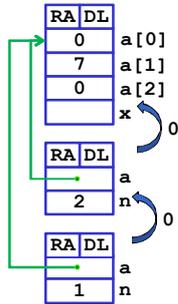
ESERCIZIO

- calcolare se un array è costituito da valori tutti nulli

ESERCIZIO

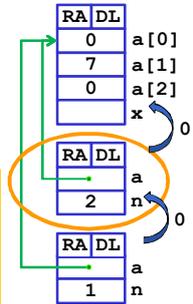
```
int nullo(int a[], int n)
{ if (n<0) return 1;
  if (a[n]==0)
    return nullo(a,n-1);
  else return 0;
}

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;
}
```



Questa invocazione non calcola nulla!
Deve solo restituire lo stesso risultato
che ottiene dall'invocazione
successiva!

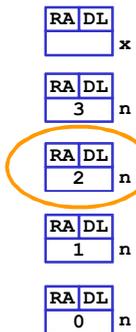
Non userò nessuna delle informazioni
che sono nel record di attivazione!
Ma allora, perché lo memorizzo?

MA È SEMPRE COSÌ?

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

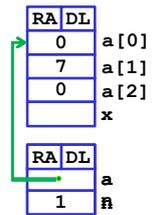
main()
{ int x;
  x = fact(3);
}
```

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;
}
```



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.



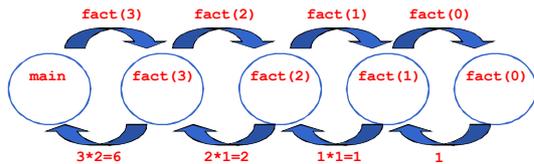
Processo computazionale effettivamente ricorsivo

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`

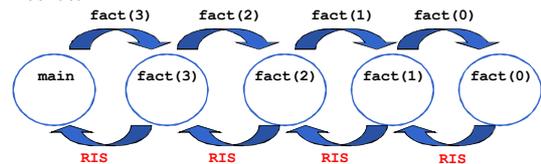
```
int fact(int n)
{ if (n==0)
  return 1;
  else
  return n*fact(n-1);
}
```



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

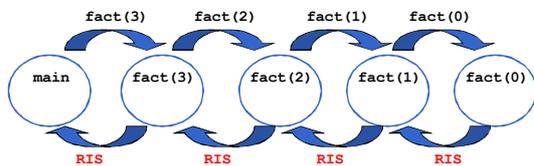
```
int fact(int n)
{ if (n==0)
  return RIS;
  else
  return fact(n-1);
}
```



Rendiamo più efficiente (2)

- Quindi i prodotti vanno calcolati prima della chiamata ricorsiva

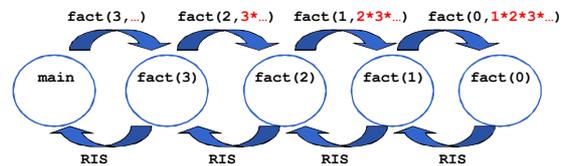
```
int fact(int n)
{ if (n==0)
  return RIS;
  else
  { RIS = n*RIS;
    return fact(n-1);
  }
}
```



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);
  }
}
```

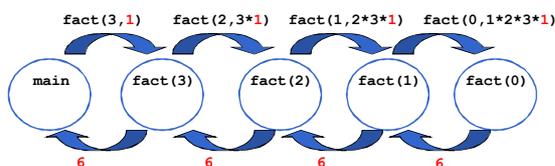


Rendiamo più efficiente (4)

- 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`.

FATTORIALE ITERATIVO

Definizione:

$$n! = 1 * 2 * 3 * \dots * n$$

Detto $v_k = 1 * 2 * 3 * \dots * k$:

$$1! = v_1 = 1$$

$$(k+1)! = v_{k+1} = (k+1) * v_k \quad \text{per } k \geq 1$$

$$n! = v_n \quad \text{per } k=n$$

ITERAZIONE E RICORSIONE TAIL

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

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

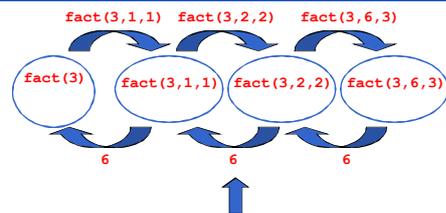
Inizializzazione dell'accumulatore: corrisponde al fattoriale di 1

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")