


***RICORSIONE***

# Agenda

- Record di attivazione
- Ricorsione
- Iterazione
- Ricorsione Tail

# Gestione dell'esecuzione di funzioni mediante record di attivazione



```
int somma(int num1, int num2) {  
    return num1 + num2;  
}  
  
int main() {  
    int numero1, numero2, risultato;  
    numero1 = 10;  
    numero2 = 2;  
    risultato = somma(numero1, numero2);  
    printf("La somma di %d e %d è %d\n", numero1, numero2, risultato);  
    return 0;  
}
```


# Gestione dell'esecuzione di funzioni mediante record di attivazione



```
int somma(int num1, int num2) {  
    return num1 + num2;  
}
```

# Gestione dell'esecuzione di funzioni mediante record di attivazione

Di cosa abbiamo bisogno per invocare questa funzione?



```
int somma(int num1, int num2) {  
    return num1 + num2;  
}
```

# Gestione dell'esecuzione di funzioni mediante record di attivazione



```
int somma(int num1, int num2) {  
    return num1 + num2;  
}
```

# Gestione dell'esecuzione di funzioni mediante record di attivazione




**Non è magia... (Purtroppo)**

```
int somma(int num1, int num2) {  
    return num1 + num2;  
}
```

# Gestione dell'esecuzione di funzioni mediante record di attivazione

Ragioniamoci insieme...


Di cosa avremmo bisogno se dovessimo invocare  
questa funzione?



```
int somma(int num1, int num2) {  
    return num1 + num2;  
}
```




# Passaggio di Parametri



```
void swap( int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

Cosa fa questo codice?

# Passaggio di Parametri




```
void swap( int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

Cosa fa questo codice?

Siamo sicuri che non ci siano effetti collaterali?

# Passaggio di Parametri



```
#include <stdio.h>

void swap(int,int);

void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int a, b;
    a = 12;
    b= 1;
    swap(a,b);
    printf("il valore di a è %d, mentre il valore di b è %d",a,b);
    return 0;
}
```

# Passaggio di Parametri

```
il valore di a è 12, mentre il valore di b è 1  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Ma non li avevamo scambiati?!


# Passaggio di Parametri

```
il valore di a è 12, mentre il valore di b è 1
```

```
...Program finished with exit code 0  
Press ENTER to exit console.
```

Passaggio di parametri per **VALORE!!**

# Passaggio di Parametri



```
#include <stdio.h>

// Dichiarazione della funzione per lo scambio di due variabili
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int a = 12;
    int b = 1;
    printf("Prima dello scambio: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("Dopo lo scambio: a = %d, b = %d\n", a, b);
    return 0;
}
```

# Passaggio di Parametri

```
Prima dello scambio: a = 12, b = 1  
Dopo lo scambio: a = 1, b = 12
```

```
...Program finished with exit code 0  
Press ENTER to exit console.█
```

Lo scambio è avvenuto correttamente

# Passaggio di Parametri

```
Prima dello scambio: a = 12, b = 1  
Dopo lo scambio: a = 1, b = 12
```

Passaggio di parametri per **INDIRIZZO!!**

Lo scambio è avvenuto correttamente



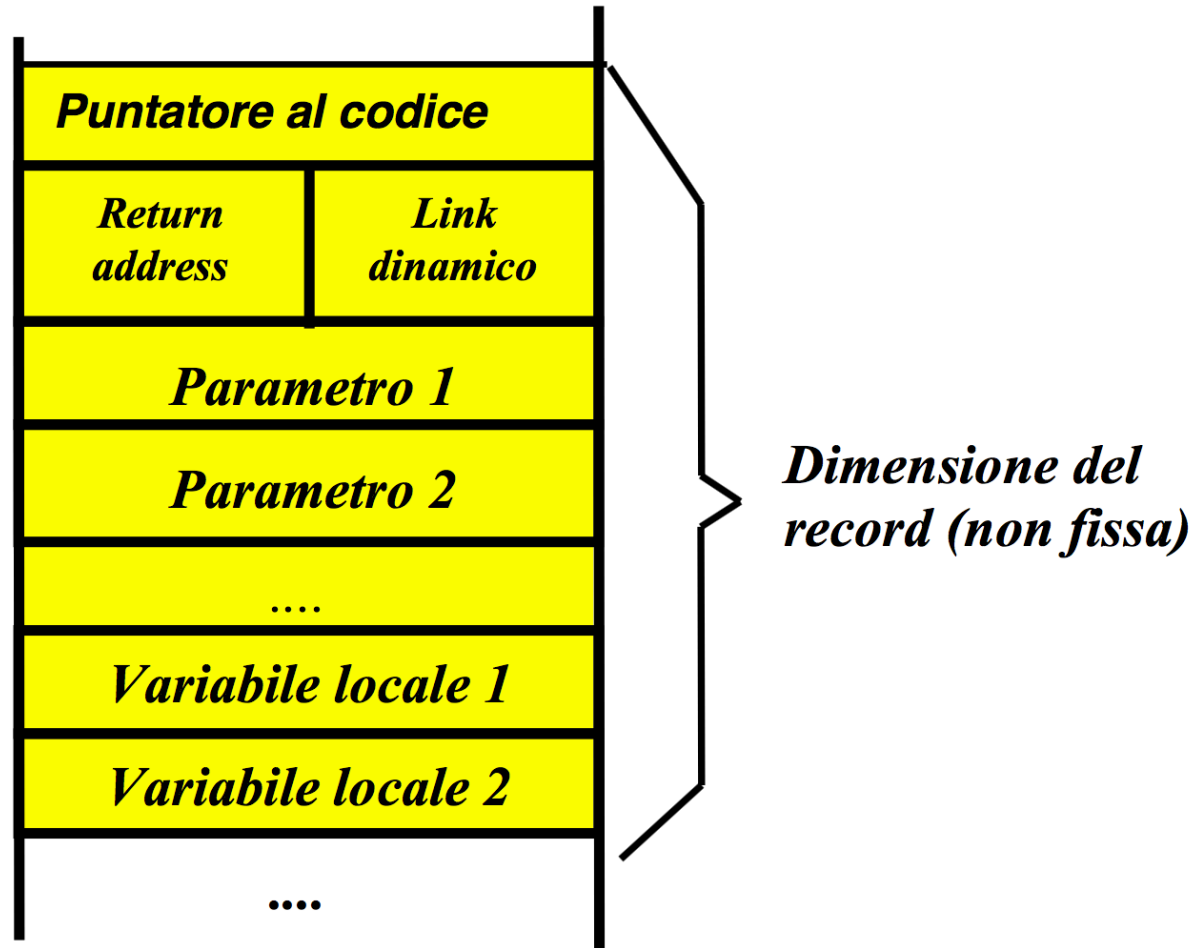
# Gestione dell'esecuzione di funzioni mediante record di attivazione

- Ogni volta che viene invocata una funzione:
  - si crea di una nuova **attivazione** (istanza) del servitore (la funzione chiamata)
  - viene **allocata la memoria** per i parametri e per le variabili locali
  - si effettua il **passaggio dei parametri**
  - si **trasferisce il controllo** al servitore
  - si **esegue il codice** della funzione

# Record di attivazione

- Al momento dell'invocazione:
  - viene creata dinamicamente una struttura dati che contiene il *binding* (legame) dei parametri e degli identificatori definiti localmente alla funzione detta **RECORD DI ATTIVAZIONE**.
- È l'“**environment della funzione**”: contiene tutto ciò che serve per la chiamata alla quale è associato:
  - i **parametri** formali
  - le **variabili locali**
  - l'**indirizzo di ritorno** (Return address RA) che indica il punto a cui tornare (nel codice della funzione chiamante, detta *cliente*) al termine della funzione, per permettere al cliente di proseguire una volta che la funzione termina.
  - **un collegamento al record di attivazione** del cliente (**Link Dinamico DL**)
  - l'**indirizzo del codice** della funzione (**puntatore alla prima istruzione del corpo**)

# Record di attivazione



# Record di attivazione

- Il **record di attivazione** associato a una chiamata di una funzione  $f$ :
  - **creato** al momento della **invocazione** di  $f$
  - **permane** per **tutto il tempo** in cui la funzione  $f$  è in **esecuzione**
  - è **distrutto** (deallocato) al **termine dell'esecuzione** della funzione stessa.
- Ad **ogni chiamata** di funzione viene creato un **nuovo record**, specifico per quella chiamata di quella funzione
- La dimensione del record di attivazione
  - varia da una funzione all'altra
  - per una data funzione, è fissa e calcolabile a priori

# Record di attivazione

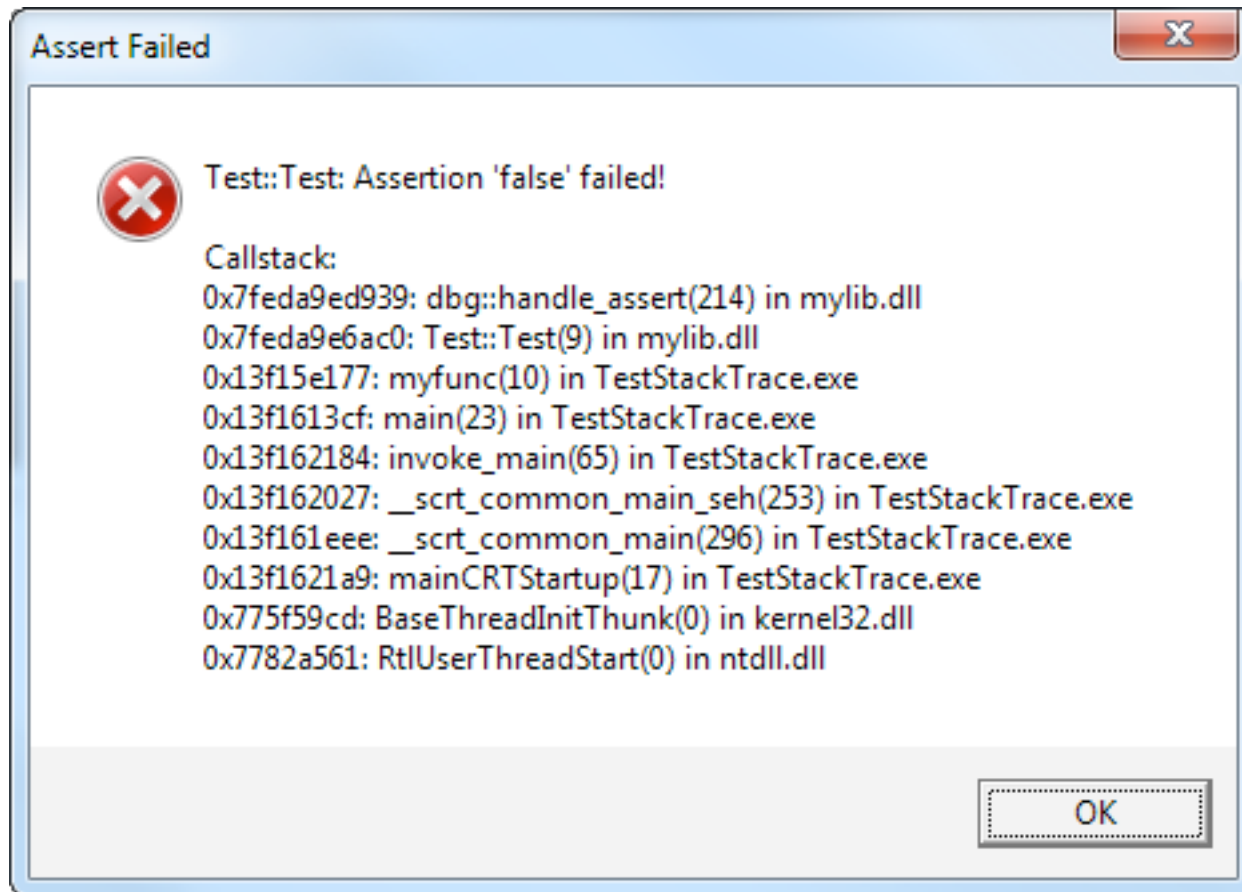
- Il **record di attivazione** associato a una chiamata di una funzione  $f$ :
  - creato al momento della **invocazione** di  $f$

## Ma cosa c'entra con strutture dati?!

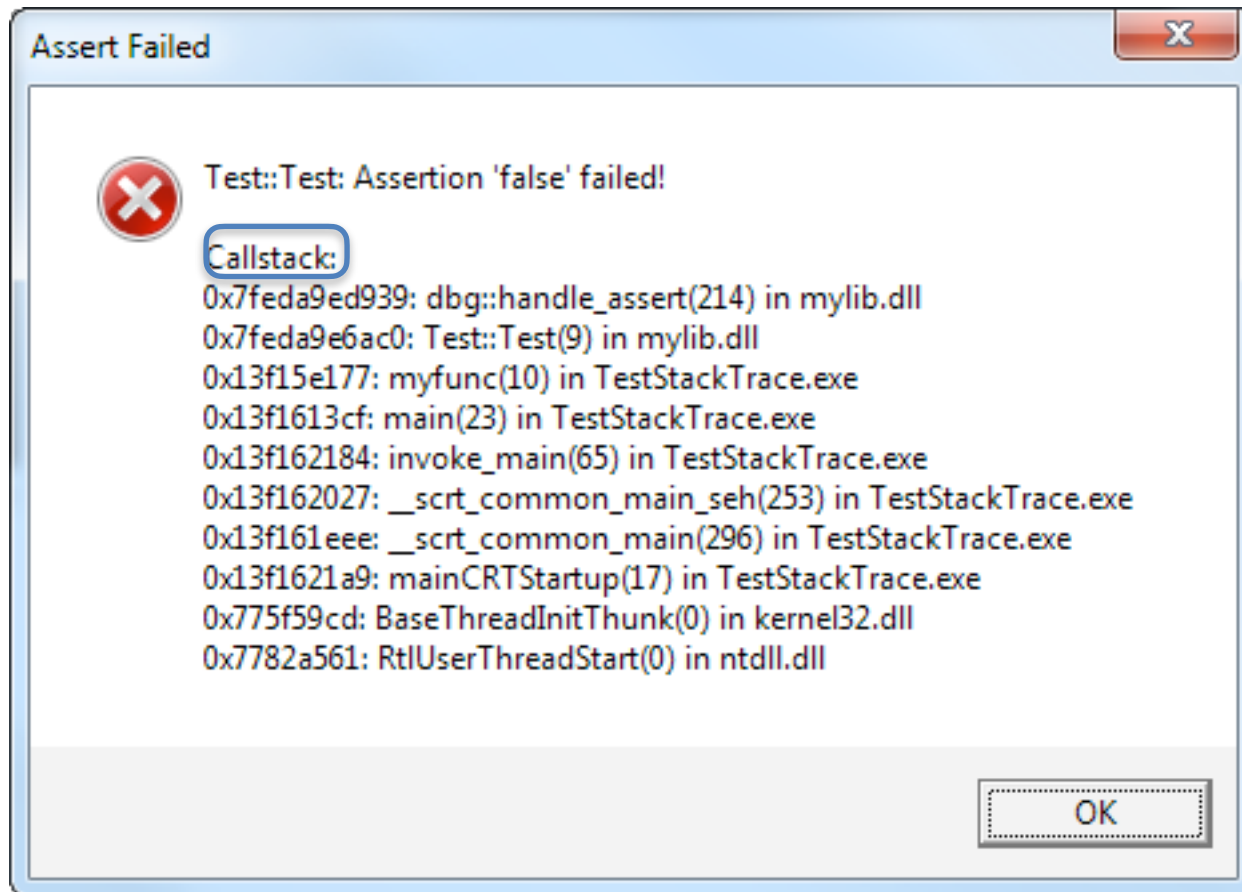
**record**, specifico per quella chiamata di quella funzione

- La dimensione del record di attivazione
  - varia da una funzione all'altra
  - per una data funzione, è fissa e calcolabile a priori

# Record di attivazione



# Record di attivazione



Perché c'è scritto "CallStack"?

# Record di attivazione

- Funzioni che chiamano altre funzioni danno luogo a una sequenza di record di attivazione
  - **allocati secondo l'ordine delle chiamate**
  - **deallocati in ordine inverso**
- La **sequenza dei link dinamici** costituisce la cosiddetta **catena dinamica**, che rappresenta la storia delle attivazioni (“chi ha chiamato chi”)

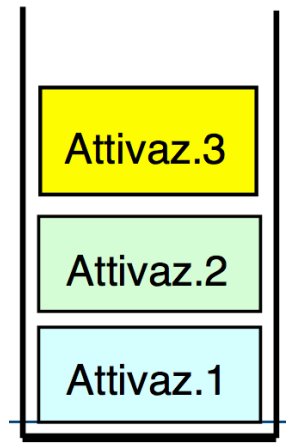


# Stack

- L'area di memoria in cui vengono allocati i record di attivazione viene gestita come una **pila**:

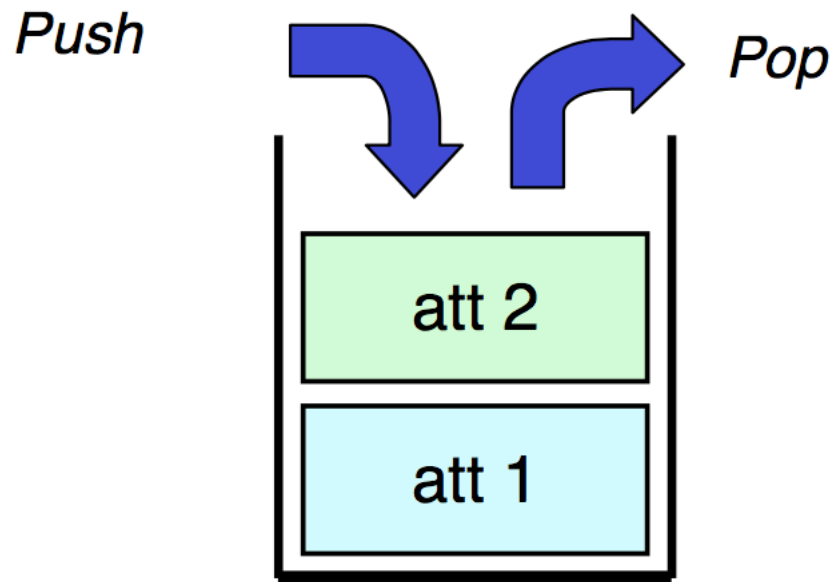
## STACK

- È una struttura dati gestita a tempo di esecuzione con politica LIFO (**Last In, First Out** - l'ultimo a entrare è il primo a uscire) nella quale ogni elemento è un record di attivazione.
- La gestione dello stack avviene mediante due operazioni:
  - **push**: aggiunta di un elemento (in cima alla pila)
  - **pop**: prelievo di un elemento (dalla cima della pila)



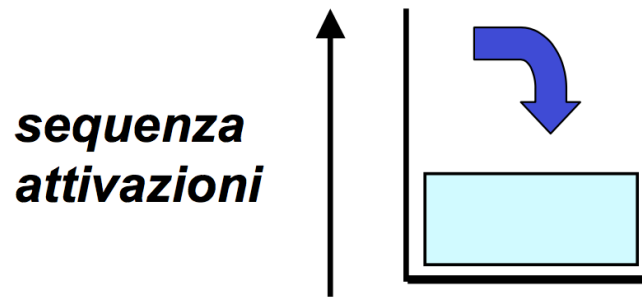
# Stack

L'ordine di collocazione dei record di attivazione nello stack indica la cronologia delle chiamate:

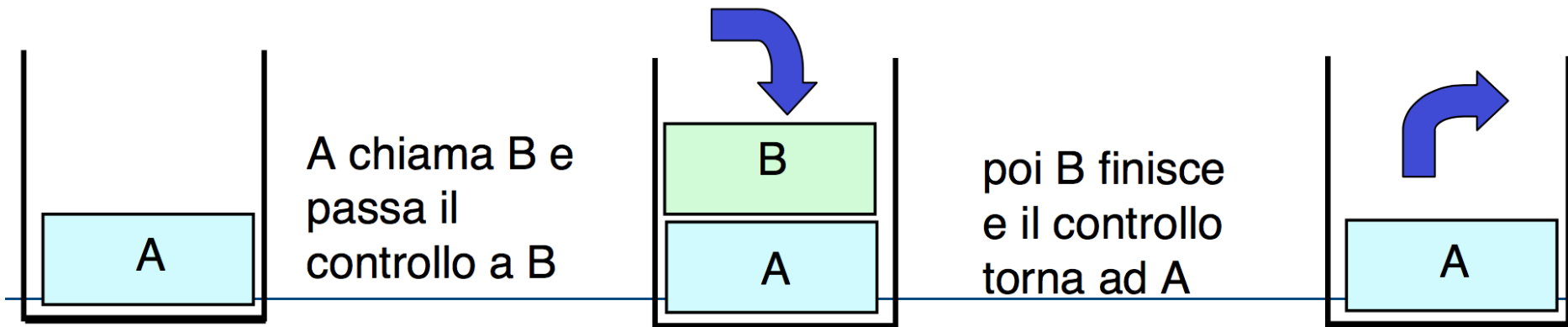


# Record di attivazione

- Normalmente lo STACK dei record di attivazione si disegna nel modo seguente:



- Quindi, se la funzione A chiama la funzione B, lo stack evolve nel modo seguente



# Esempio: chiamate annidate

Programma:

```
int R(int A){  
    return A+1;  
}  
  
int Q(int x){  
    return R(x);  
}  
  
int P(void){  
    int a=10;  
    return Q(a);  
}  
  
main() {  
    int x = P();  
}
```

# Esempio: chiamate annidate

Programma:

```
int R(int A){  
    return A+1;  
}
```

```
int Q(int x){  
    return R(x);  
}
```

```
int P(void){  
    int a=10;  
    return Q(a);  
}
```

```
main() {  
    int x = P();  
}
```

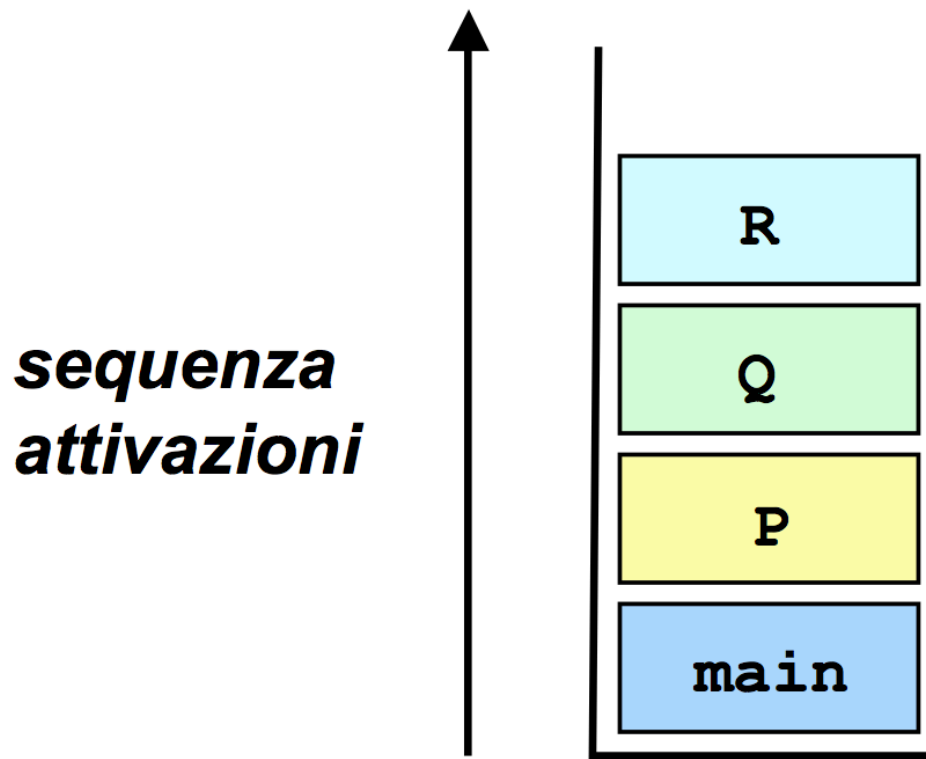
Sequenza chiamate:

S.O.  $\rightarrow$  main  $\rightarrow$  P()  $\rightarrow$  Q()  $\rightarrow$  R()

# Esempio: chiamate annidate

Sequenza chiamate:

- `S.O. → main → P() → Q() → R()`



# Esercizio

Definire i record di attivazione per la seguenti chiamate

```
int main() {
    printf("Main: Inizio\n");
    funzioneA();
    printf("Main: Fine\n");
    return 0;
}

void funzioneA() {
    int a = 1;
    printf("Funzione A: Inizio\n");
    printf("Funzione A: a = %d\n", a);
    funzioneB();
    printf("Funzione A: Fine\n");
}

void funzioneB() {
    int b = 2;
    printf("Funzione B: Inizio\n");
    printf("Funzione B: b = %d\n", b);
    funzioneC();
    printf("Funzione B: Fine\n");
}

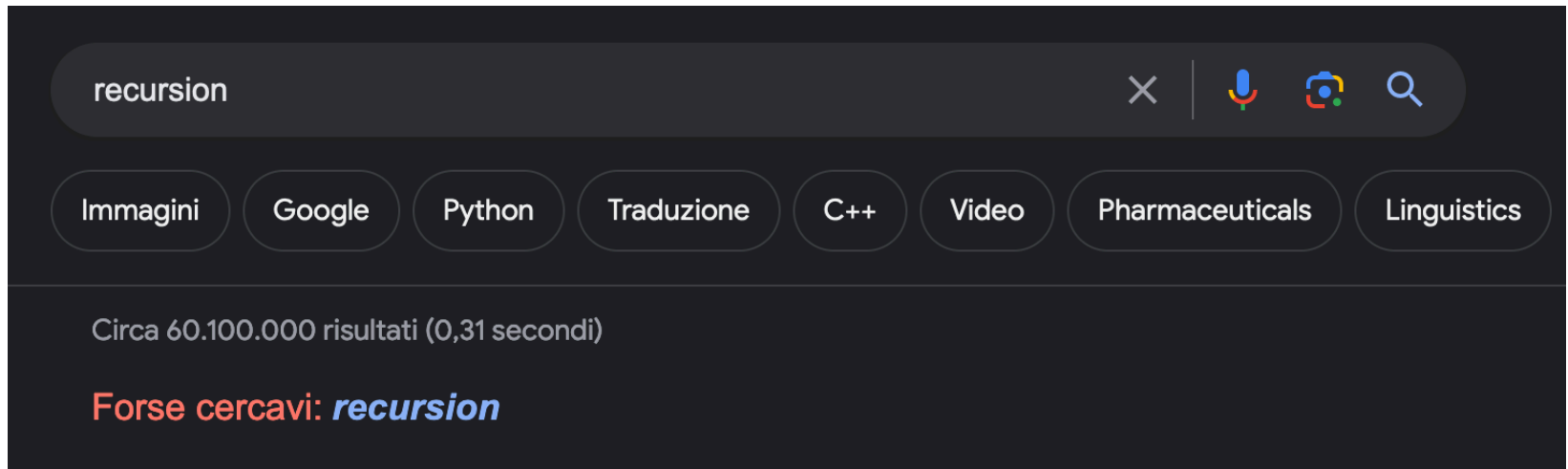
void funzioneC() {
    int c = 3;
    printf("Funzione C: Inizio\n");
    printf("Funzione C: c = %d\n", c);
    printf("Funzione C: Fine\n");
}
```

# Agenda

- Record di attivazione
- Ricorsione
- Iterazione
- Ricorsione Tail



# La Ricorsione



# 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 una proprietà  $P$  vale per  $n=n_0$  (**CASO BASE**)
  - e si può dimostrare che, ***assumendola valida per  $n \geq n_0$*** , allora vale anche per  $n+1$
  - allora  $P$  vale per ogni  $n \geq n_0$

# Esempio di funzione matematica definita ricorsivamente: Il Fattoriale

Esempio: Il fattoriale di un numero naturale  
 $\text{fact}(n) = n!$

$n!: N \rightarrow N$

$$\left\{ \begin{array}{ll} n! \text{ Vale } 1 & \text{Se } n==0 \\ n! \text{ Vale } n*(n-1)! & \text{Se } n \neq 0 \end{array} \right.$$

# La Ricorsione in programmazione

- Operativamente, risolvere un problema con un approccio ricorsivo comporta
  - di identificare un “**caso base**”, con soluzione nota
  - di riuscire a esprimere la soluzione del caso generico  $n$  in termini dello *stesso problema in uno o più casi più semplici* ( $n-1$ ,  $n-2$ , etc.), dove  $n$  è la taglia del problema

# La Ricorsione in programmazione

(cont.)

- Un sottoprogramma ricorsivo è:
  - un sottoprogramma che richiama direttamente o indirettamente se stesso.
- **Non** tutti i linguaggi realizzano il meccanismo della ricorsione. Quelli che lo realizzano, di solito utilizzano la tecnica di **gestione mediante record di attivazione**: ad ogni chiamata è associato un record di attivazione (variabili locali e punto di ritorno).

# La Ricorsione: Il Fattoriale

- In C è possibile realizzare funzioni ricorsive
- Il corpo di ogni funzione ricorsiva contiene almeno una chiamata alla funzione stessa, direttamente o indirettamente.
- **Esempio: definizione in C della funzione ricorsiva fattoriale.**

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

# La Ricorsione: Il Fattoriale

- **Servitore & Cliente:** **fact** è sia servitore che cliente (di se stessa):

```
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: Il Fattoriale (cont.)

## 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 fact() una copia del valore così ottenuto (3)*

*fact(3) effettuerà poi analogamente una nuova chiamata di funzione fact(2)*



# La Ricorsione: Il Fattoriale (cont.)

## 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);  
}
```

*Analogamente, fact(2) effettua una nuova chiamata di funzione. n-1 nell'environment di fact() vale 1 quindi viene chiamata fact(1)*

*E ancora, analogamente, per fact(0)*

# La Ricorsione: Il Fattoriale (cont.)

## 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  $n \leq 0$  è vera e la funzione `fact(0)` torna come risultato 1 e termina*

# La Ricorsione: Il Fattoriale (cont.)

## 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 può valutare l'espressione  $n * 1$  ottenendo come risultato 1 e terminando*

*E analogamente per fact(2) e fact(3)*

# La Ricorsione: Il Fattoriale (cont.)

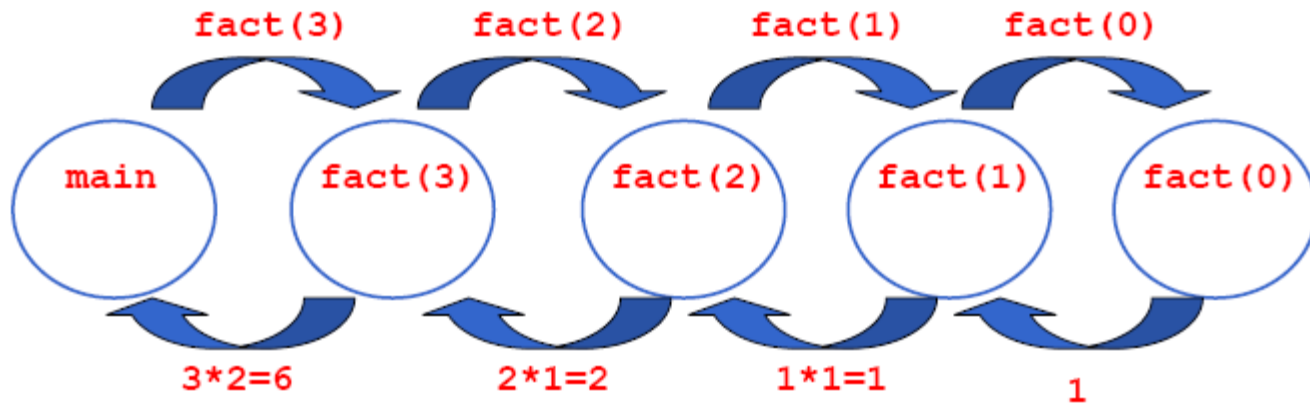
## Servitore & Cliente:

```
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 PASSA INFINE  
AL MAIN CHE ASSEGNA A fz IL  
VALORE 6**

# La Ricorsione: Il Fattoriale (cont.)



**main**      **fact(3) = 3 \* fact(2) = 2 \* fact(1) = 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,f6,z = 5;
    fz = fact(z-2);
}
```

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

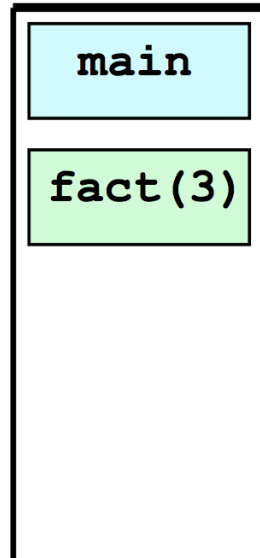
**Seguiamo l'evoluzione dello stack durante l'esecuzione:**

# Cosa succede nello stack ?

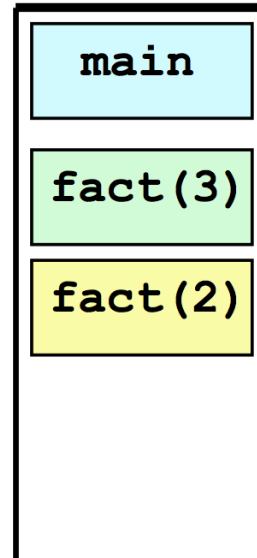
Situazione  
iniziale



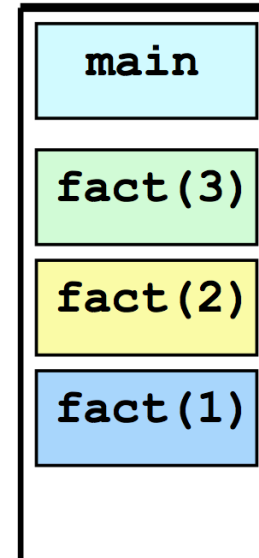
Il `main()`  
chiama  
`fact(3)`



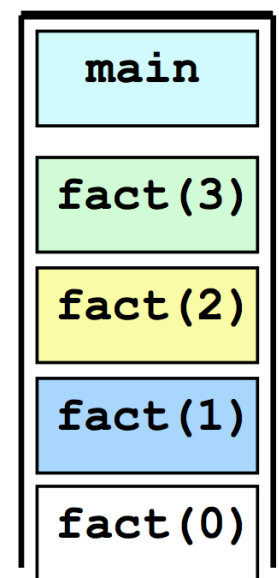
`fact(3)`  
chiama  
`fact(2)`



`fact(2)`  
chiama  
`fact(1)`

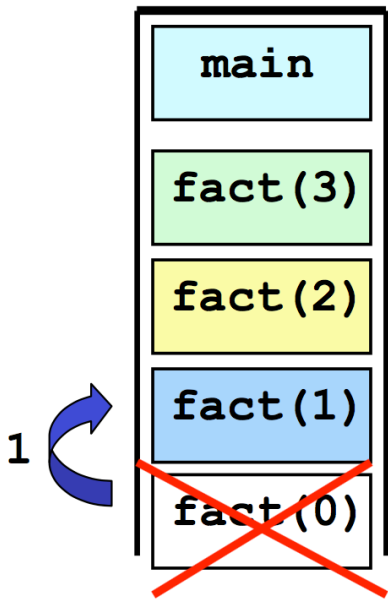


`fact(1)`  
chiama  
`fact(0)`

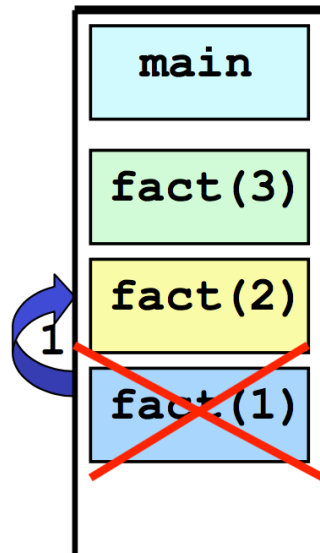


# Cosa succede nello stack ?

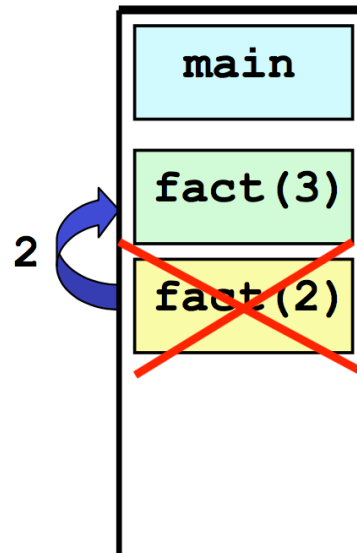
**fact(0)** termina restituendo il valore 1. Il controllo torna a **fact(1)**



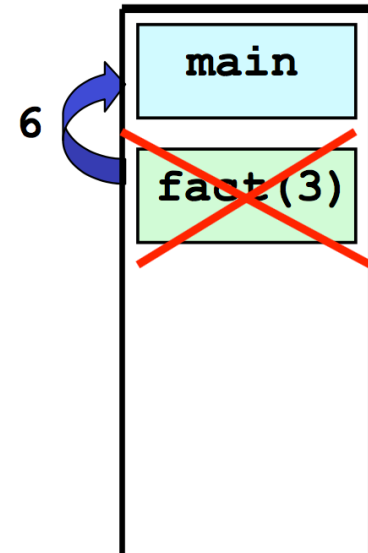
**fact(1)** effettua la moltiplicazione e termina restituendo il valore 1. Il controllo torna a **fact(2)**



**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**.





# La Ricorsione: La Somma dei Primi $n$ Interi

## **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: La Somma dei Primi $n$ Interi (cont.)

**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: La Somma dei Primi $n$ Interi (cont.)

**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: successione di Fibonacci

Piccola curiosità:

## The Fibonacci Sequence

**1,1,2,3,5,8,13,21,34,55,89,144,233,377...**

$$1+1=2$$

$$1+2=3$$

$$2+3=5$$

$$3+5=8$$

$$5+8=13$$

$$8+13=21$$

$$13+21=34$$

$$21+34=55$$

$$34+55=89$$

$$55+89=144$$

$$89+144=233$$

$$144+233=377$$

# La Ricorsione: successione di Fibonacci

## Piccola curiosità:

La sequenza di Fibonacci appare sorprendentemente spesso in natura. La disposizione delle **foglie** su un **fusto**, la **ramificazione** degli **alberi**, la disposizione dei **petali dei fiori**, le **conchiglie** di **lumaca** e molti schemi di riproduzione seguono la sequenza di Fibonacci.

# La Ricorsione: successione di Fibonacci

**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: Fibonacci (cont.)

## 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: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

## **Problema:**

Vogliamo implementare un algoritmo per effettuare la ricerca

## Precondizione

L'array è ordinato in ordine crescente

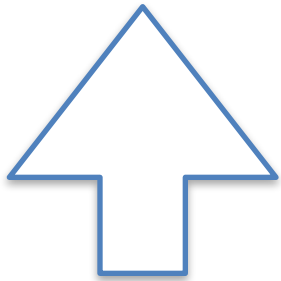
## Input

$x = 1$      $x = 12$      $x = 93$



# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----



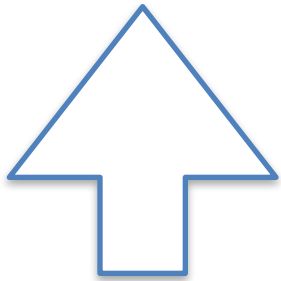
$x = 1$

if  $x == \text{array}[0]$   
trovato = true

Trovato al primo accesso

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

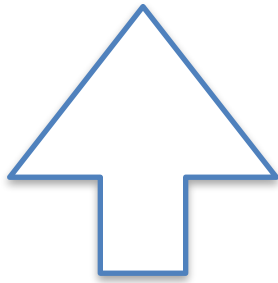


x = 12

```
if x == array[0]  
trovato = False
```

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

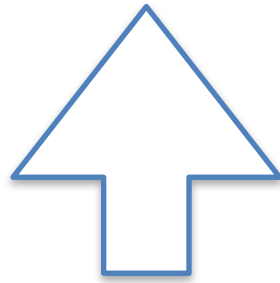


$x = 12$

```
if x == array[0]  
trovato = False
```

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

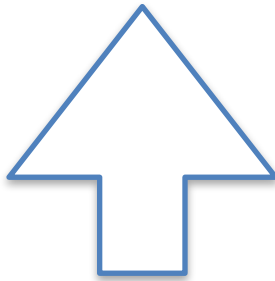


$x = 12$

if  $x == \text{array}[0]$   
trovato = False

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----



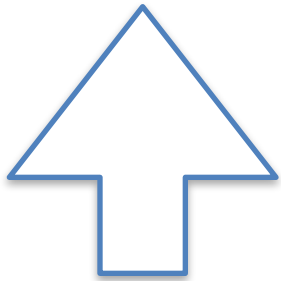
$x = 12$

```
if x == array[0]  
trovato = True
```

Trovato al quarto accesso

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

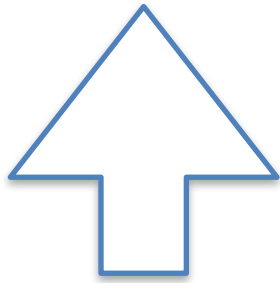


x = 93

```
if x == array[0]  
trovato = False
```

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

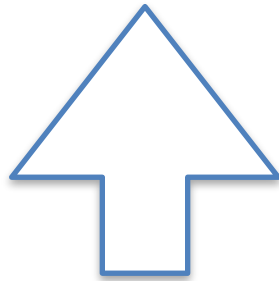


$x = 93$

if  $x == \text{array}[0]$   
trovato = False

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----



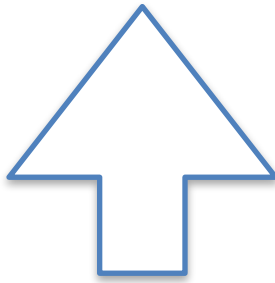
$x = 93$

if  $x == \text{array}[0]$   
trovato = False



# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

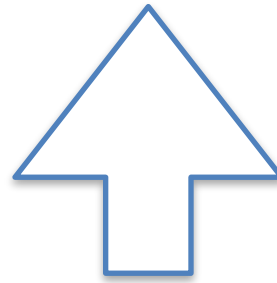


$x = 93$

if  $x == \text{array}[0]$   
trovato = False

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

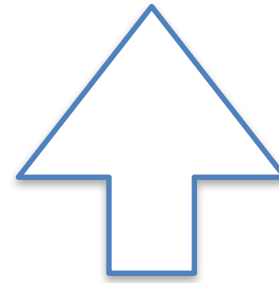


$x = 93$

if  $x == \text{array}[0]$   
trovato = False

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

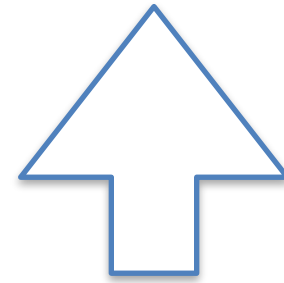


$x = 93$

```
if x == array[0]  
trovato = False
```

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

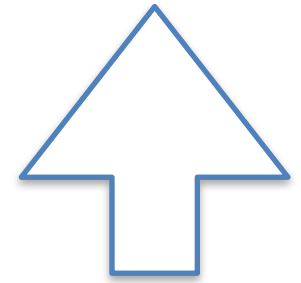


x = 93

```
if x == array[0]  
trovato = False
```

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----



x = 93

Trovato al ottavo accesso

```
if x == array[0]  
trovato = True
```

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

**Caso pessimo: Trovato dopo N iterazioni!**

x = 93

Trovato al ottavo accesso

```
if x == array[0]  
trovato = True
```

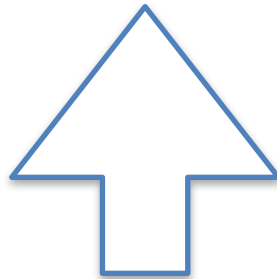
# La Ricorsione: Ricerca Binaria

0	1	2	3	4	5	6	7
1	4	10	12	15	42	47	93

# La Ricorsione: Ricerca Binaria

0	1	2	3	4	5	6	7
1	4	10	12	15	42	47	93

$x = 1$



$x == 12?$

$x < 12$

$x > 12$

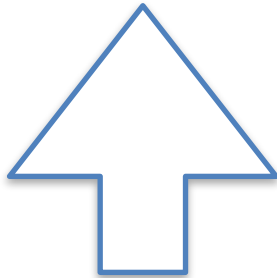
1	4	10	12
---	---	----	----

15	42	47	93
----	----	----	----



# La Ricorsione: Ricerca Binaria

0	1	2	3	4	5	6	7
1	4	10	12				



$x == 4?$

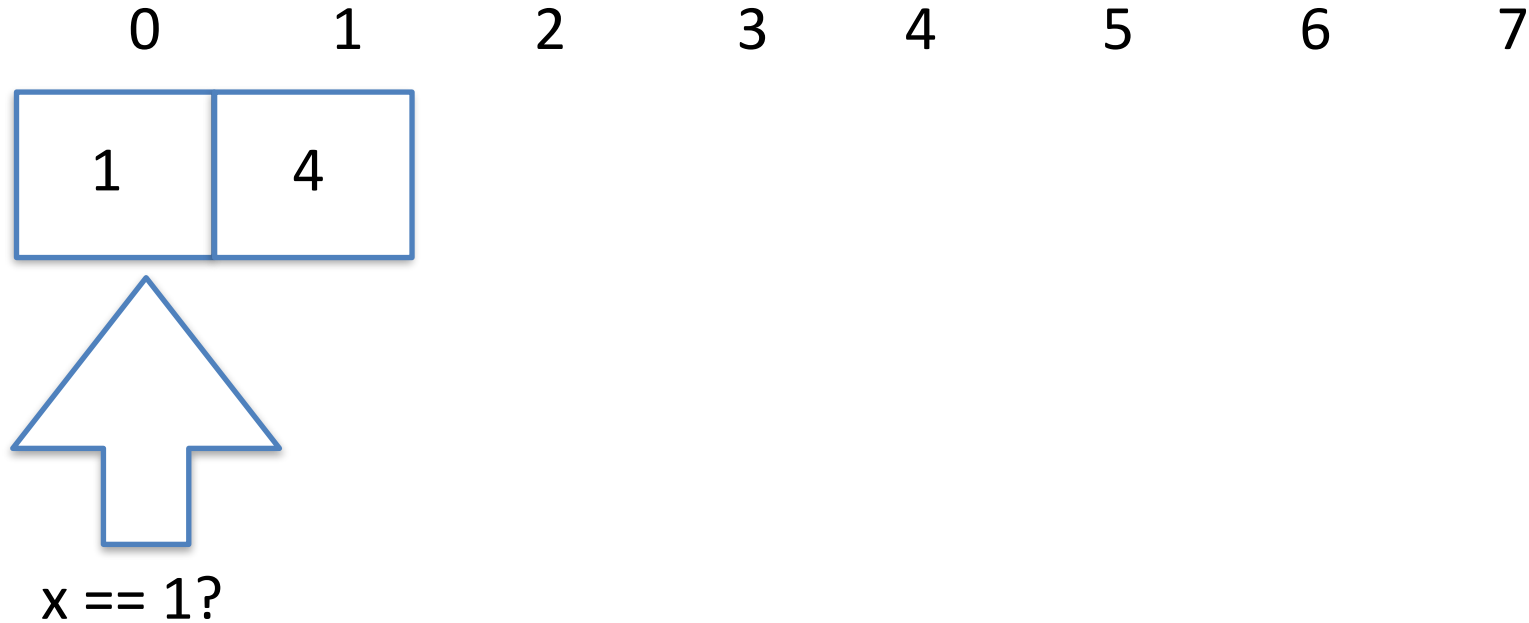
$x < 4$

1	4
---	---

$x > 4$

10	12
----	----

# La Ricorsione: Ricerca Binaria

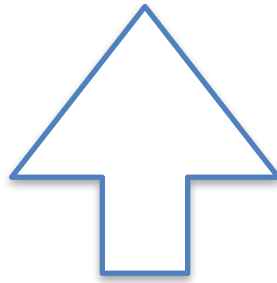


Trovato!



# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----

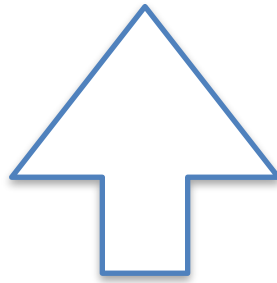


$x = 12$

Trovato al primo accesso

# La Ricorsione: Ricerca Binaria

1	4	10	12	15	42	47	93
---	---	----	----	----	----	----	----



$x == 93?$

$x < 12$

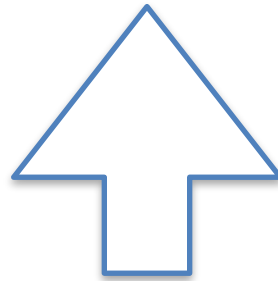
1	4	10	12
---	---	----	----

$x > 12$

15	42	47	93
----	----	----	----

# La Ricorsione: Ricerca Binaria

15	42	47	93
----	----	----	----



$x == 93?$

$x < 42$

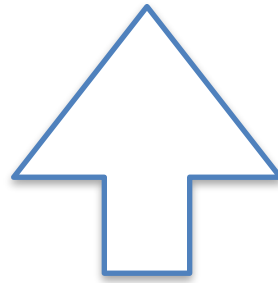
15	42
----	----

$x > 42$

47	93
----	----

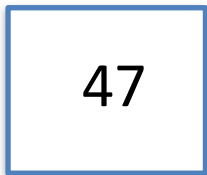
# La Ricorsione: Ricerca Binaria

15	42	47	93
----	----	----	----



$x == 93?$

$x < 47$

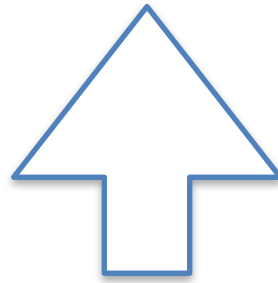


$x > 47$



# La Ricorsione: Ricerca Binaria

15	42	47	93
----	----	----	----



$x == 93?$

$x < 47$



$x > 47$



Trovato!

# La Ricorsione: Ricerca Binaria

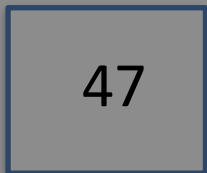
15	42	47	93
----	----	----	----



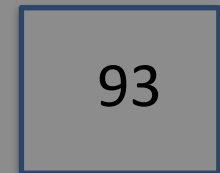
**Caso pessimo:  $\log_2(n)$**

$x == 93?$

$x < 47$



$x > 47$



Trovato!



# La Ricorsione: Riflessioni

Negli esempi visti finora 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: Limitazioni

La ricorsione, nonostante sia uno strumento potente presenta diversi limiti che possono influenzare la scelta di utilizzarla in determinate situazioni

- **Consumo di memoria elevato:** Ogni chiamata ricorsiva aggiunge un **nuovo livello allo stack** di chiamate, consumando memoria. Questo può portare rapidamente a un overflow dello stack, specialmente con profondità di ricorsione
- **Prestazioni:** Le chiamate ricorsive possono essere meno efficienti rispetto ai cicli (iterazione), a causa del tempo e della memoria aggiuntivi necessari per gestire le chiamate e i ritorni di funzione elevate

# La Ricorsione: Limitazioni

- **Complessità di debugging:** Il debugging di funzioni ricorsive può essere più complesso rispetto alle loro controparti iterative, soprattutto per ricorsioni profonde o complesse
- **Rischio di loop infinito:** Se il caso base non è definito correttamente o la condizione di terminazione non viene mai raggiunta, la ricorsione può portare a un **loop infinito**
- **Comprensione del codice:** il codice ricorsivo può essere meno intuitivo e più difficile da seguire rispetto a un approccio iterativo