

# Corso di Laurea in Medicina e Chirurgia TD

Corso di

Tecniche di Programmazione

Docente: Pierangelo Veltri (credits delle slides Prof. Sergio Greco)

La ricorsione: algoritmi elementari ed introduzione alle tecniche  
ricorsive

# La ricorsione

La **ricorsione** è una tecnica di programmazione basata sulla definizione di metodi e classi (tipi) ricorsivi.

La ricorsione è connessa alla nozione matematica di induzione

La **ricorsione** è una tecnica di programmazione utile ed efficace nell'implementazione di operazioni (metodi) su insiemi e tipi di dato definiti in modo induttivo. **Un metodo ricorsivo è un metodo che, direttamente o indirettamente, può invocare se stesso.**

Una funzione matematica è definita ricorsivamente quando nella sua definizione compare un riferimento (chiamata) a se stessa.

# Definizione induttiva di insiemi

Un insieme definito in modo induttivo è un insieme definito in termini di se stesso

- l'insieme  $\mathbf{N}$  dei numeri naturali
  - $0 \in \mathbf{N}$
  - se  $n \in \mathbf{N}$ , allora anche  $n+1 \in \mathbf{N}$
- l'insieme  $\mathbf{S}_\Sigma$  delle stringhe su un alfabeto  $\Sigma$  di caratteri
  - $"" \in \mathbf{S}_\Sigma$
  - se  $\mathbf{CAR} \in \Sigma$  e  $\mathbf{S} \in \mathbf{S}_\Sigma$ , allora anche  $\mathbf{CAR+S} \in \mathbf{S}_\Sigma$
- l'insieme delle espressioni Java sugli interi
- l'insieme delle istruzioni di Java

Nella definizione induttiva di un insieme è possibile identificare

- uno o più **casi base**
- uno o più **casi induttivi**

# Definizione induttiva di funzioni

Una funzione definita in modo induttivo è una funzione definita in termini di se stessa

- nella definizione induttiva di una funzione è possibile identificare
  - uno o più casi base
  - uno o più casi induttivi
- i diversi casi partizionano il dominio della funzione

# Più semplicemente...

La **ricorsione** (**recursion**) è una tecnica di programmazione molto potente, che sfrutta l'idea di suddividere un problema da risolvere in sottoproblemi simili a quello originale, ma più semplici.

Un **algoritmo ricorsivo** per la risoluzione di un dato problema deve essere definito nel modo seguente:

- prima si definisce come risolvere dei problemi analoghi a quello di partenza, ma che hanno "dimensione piccola" e possono essere risolti in maniera estremamente semplice (detti **casi base**);
- poi si definisce come ottenere la soluzione del problema di partenza combinando la soluzione di uno o più problemi analoghi, ma di "dimensione inferiore".

# Esempio: funzione fattoriale

$$n! = n * (n - 1) * \dots * 3 * 2 * 1 \rightarrow n! = n * (n-1) * \dots * 2 * 1 = n * (n-1)!$$

La definizione induttiva del fattoriale  
è la seguente:

$$0! = 1 \quad (\text{caso base})$$

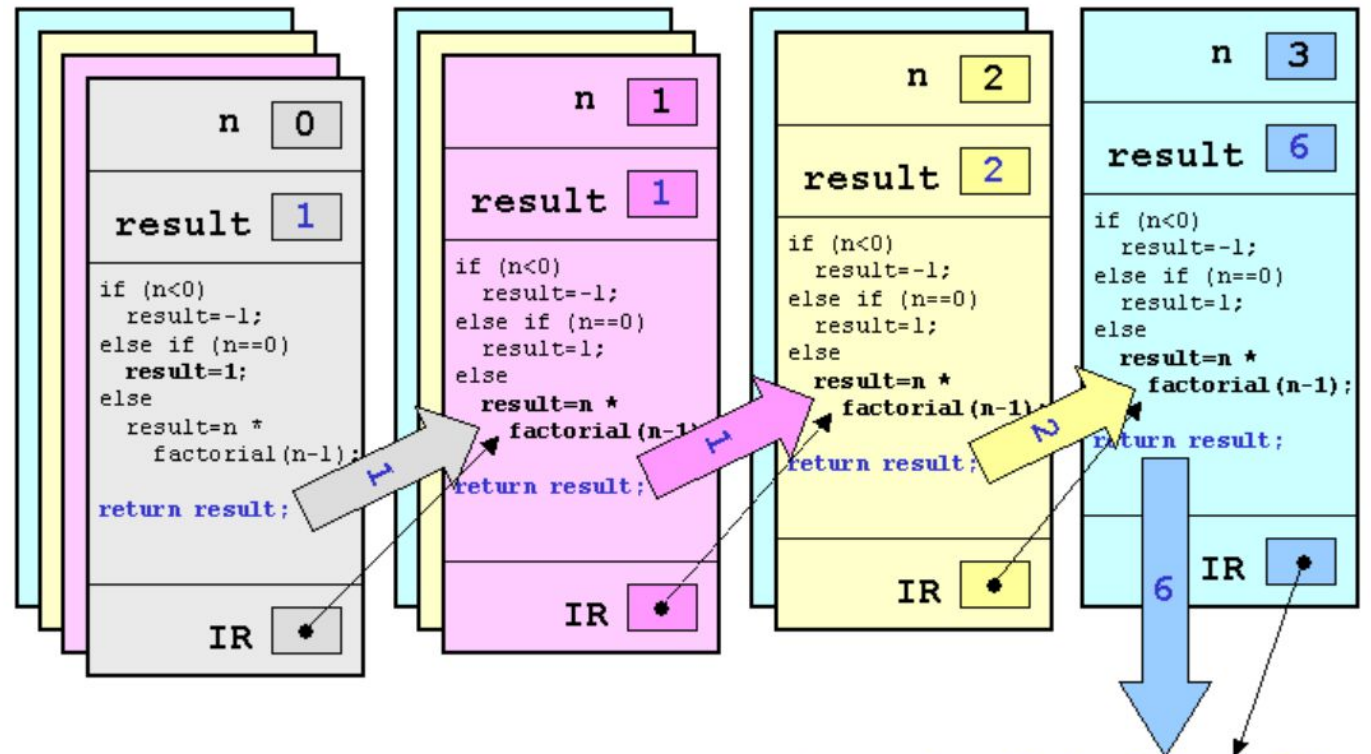
$$n! = n * (n-1)! \quad (\text{se } n > 0)$$

$$3! = 3 * 2! = 3! = 3 * 2 = 6$$

$$2! = 2 * 1! = 2! = 2 * 1 = 2$$

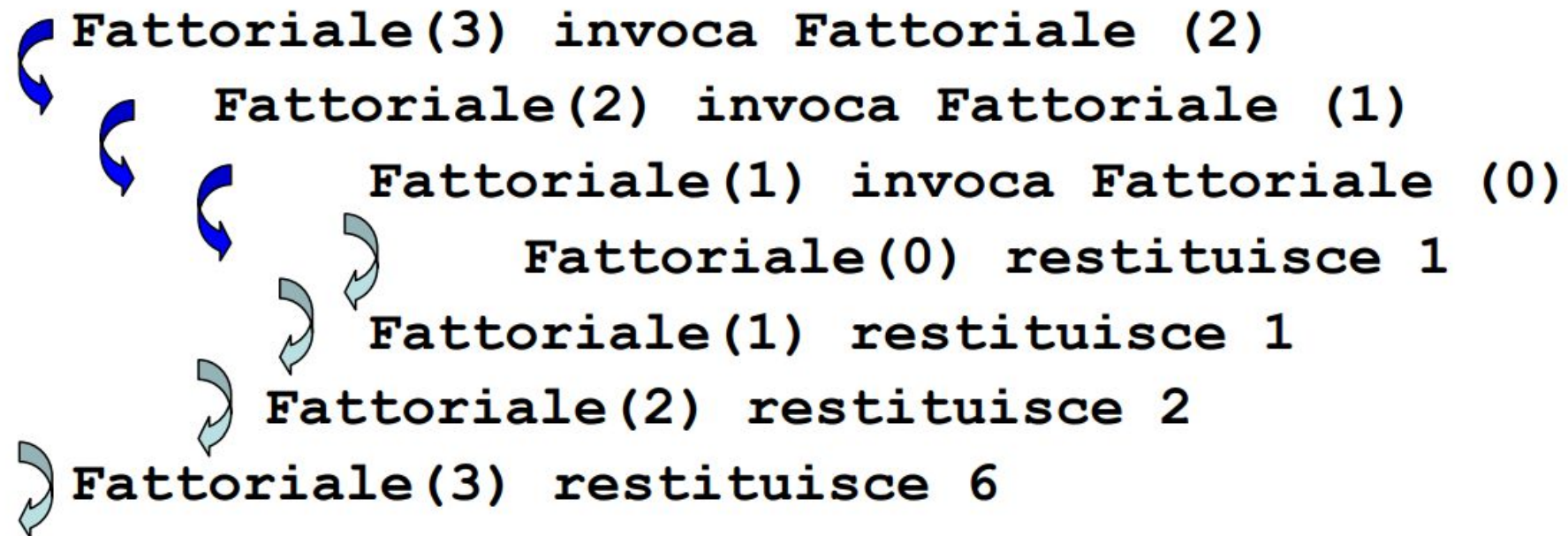
$$1! = 1 * 0! = 1! = 1 * 1 = 1$$

$$0! = 1$$



# Funzione fattoriale

- sequenza usata per calcolare 3!



# **Esecuzione di una funzione (Gestione della Memoria)**



# Gestione della ricorsione

Ogni volta che viene invocata una funzione

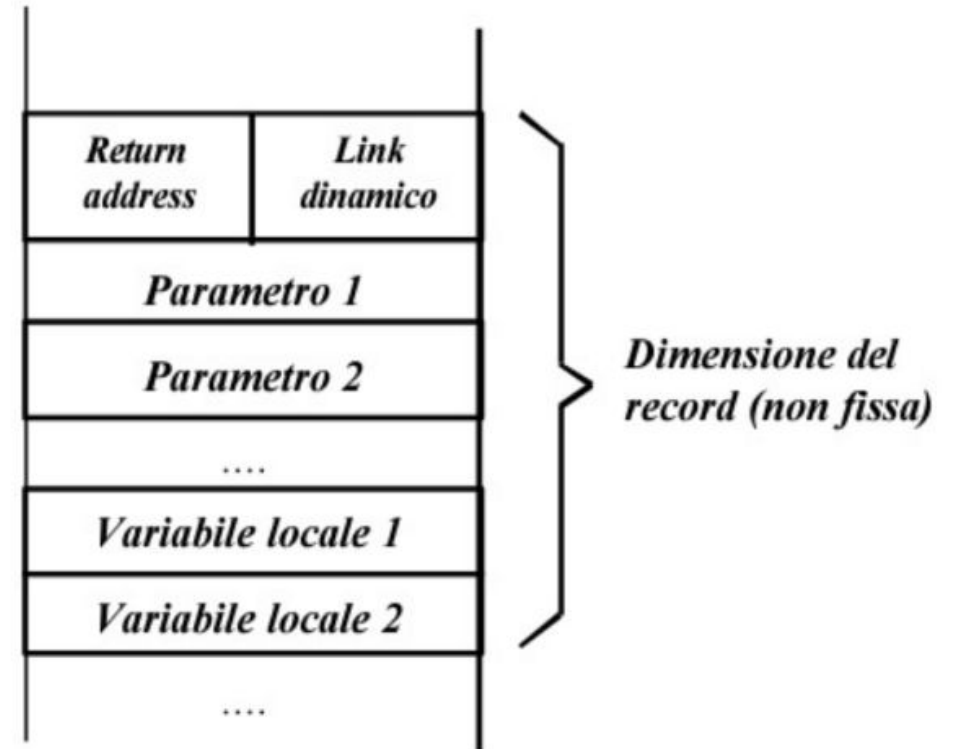
- ✓ si crea una nuova istanza
- ✓ viene allocata memoria per i parametri e le variabili locali
- ✓ si effettua il passaggio dei parametri
- ✓ si trasferisce il controllo alla funzione
- ✓ si esegue il codice

Al momento dell'attivazione viene creata una struttura dati che contiene tutti i dati utili all'esecuzione della funzione (parametri e variabili locali) detta **record di attivazione**. Il record permane per tutto il tempo di esecuzione della funzione ed è distrutto al termine della funzione

# Record di attivazione

Il Record di Attivazione contiene:

- variabili locali
- parametri formali
- indirizzo (del chiamante) a cui tornare al termine dell'esecuzione
- riferimento al record di attivazione del chiamante.

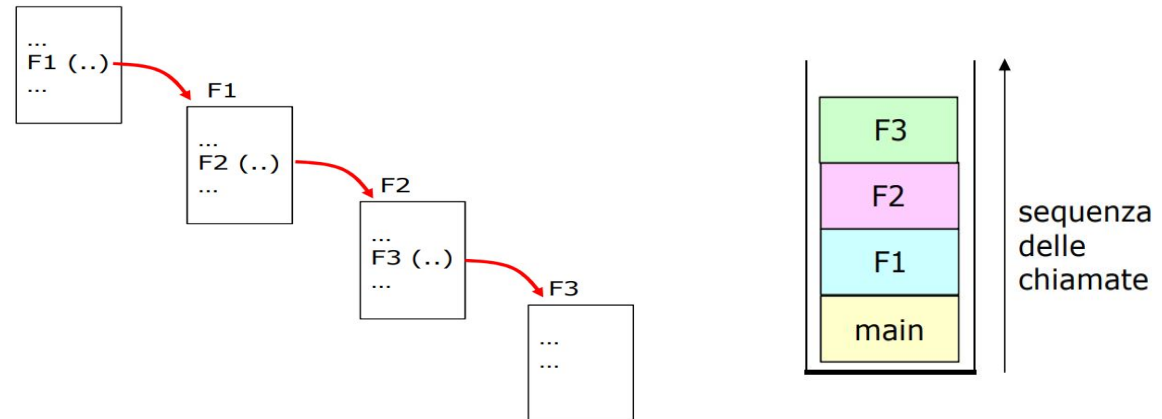


# Gestione della ricorsione

Funzioni che richiamano altre funzioni danno luogo a una sequenza di record di attivazioni

Nel caso di funzioni ricorsive si genera una sequenza di record della stessa dimensione contenenti dati locali diversi

L'area di memoria in cui sono allocati i record viene gestita come una PILA (struttura LIFO - Last In First Out), dove il main si trova nel fondo della pila e l'ultima funzione chiamata si trova al top della pila



# Allocazione della memoria

## STATICA

- Variabili allocate in modo permanente (allocate in anticipo e finché il programma viene eseguito)
- Memoria allocata durante la compilazione o prima dell'esecuzione del programma
- Struttura dati **stack** (variabili memorizzate nella memoria dello stack)
- Memoria allocata non riutilizzabile

## DINAMICA

- Variabili allocate dinamicamente (non permanenti e allocate mentre un programma è in esecuzione)
- Memoria allocata in fase di esecuzione o durante l'esecuzione del programma
- Struttura dati **heap** (variabili memorizzate nella memoria heap)
- Memoria allocata può essere rilasciata e riutilizzata

# Memoria dinamica

## VANTAGGI

- Un vantaggio dell'allocazione dinamica della memoria in Python è che non dobbiamo preoccuparci in anticipo di quanta memoria abbiamo bisogno per il nostro programma.
- La manipolazione della struttura dei dati può essere fatta liberamente senza preoccuparsi della necessità di una maggiore allocazione di memoria se la struttura dei dati si espande.

## SVANTAGGI

- L'allocazione dinamica della memoria fatta durante l'esecuzione del programma implica un consumo maggiore di tempo per il suo completamento.
- La memoria allocata deve essere liberata dopo che è stata utilizzata per evitare il verificarsi di potenziali problemi (es. perdite di memoria).

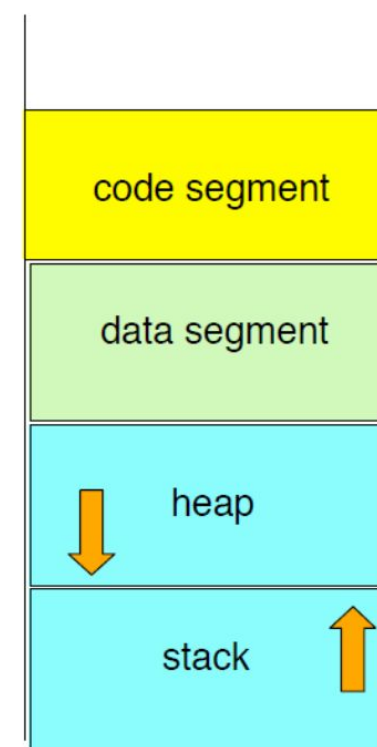
# Allocazione di memoria

La memoria allocata ad ogni processo è divisa in

- code (contiene il codice eseguibile)
- data (contiene le variabili globali, visibili a tutte le funzioni)
- heap (contiene le variabili dinamiche)
- stack (contiene i record di attivazione)

Code e data hanno dimensione fissata staticamente al momento della compilazione

La dimensione stack+heap è fissa (se stack cresce, heap diminuisce e viceversa)



# Un semplice programma in Python (P1)

```
X = 1  
Y = 2  
Z = X + Y  
print(Z)
```

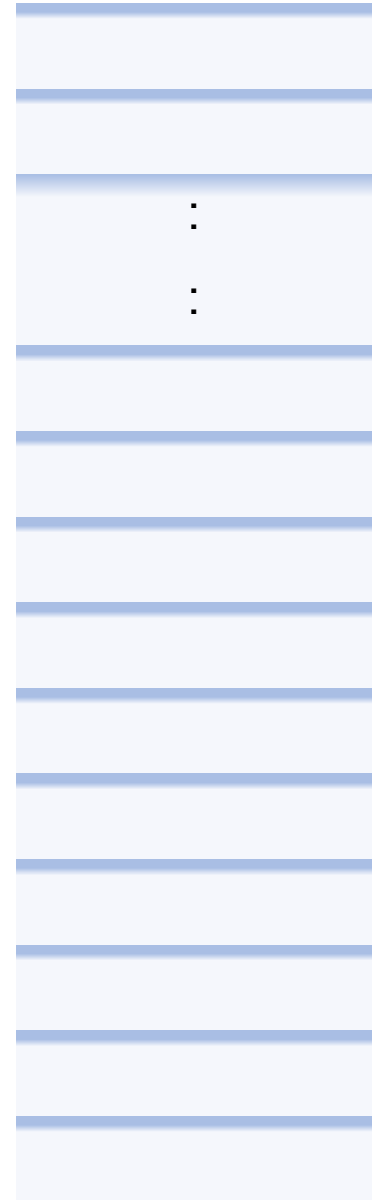
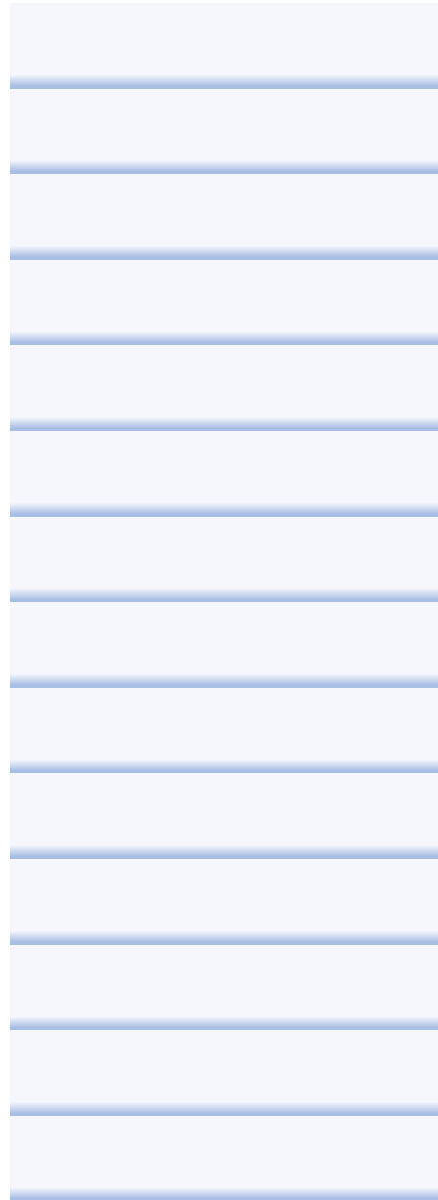


*i1)*

*i2)*

*i3)*

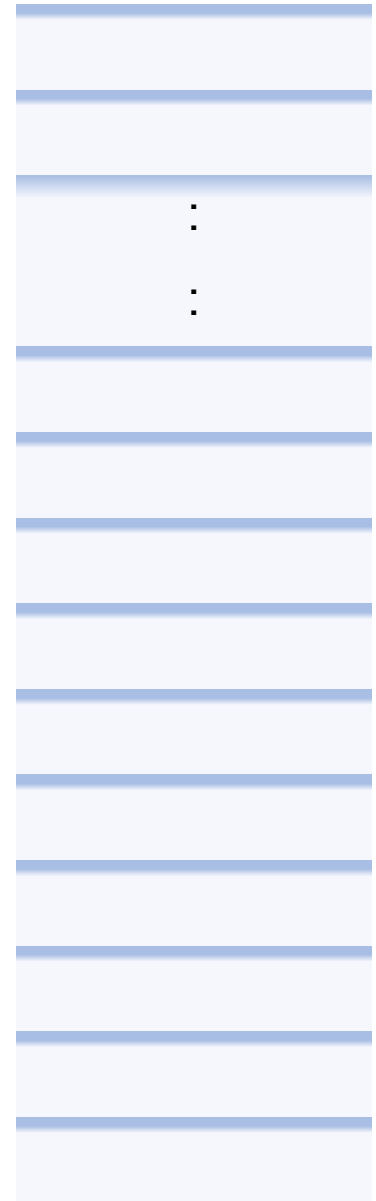
*i4)*



# Un semplice programma in Python (P1)

```
X = 1  
Y = 2  
Z = X + Y  
print(Z)
```

*i1)* **X = 1**  
*i2)* **Y = 2**  
*i3)* **Z = X + Y**  
*i4)* **print(Z)**





# Un semplice programma in Python (P1)

```
X = 1
Y = 2
Z = X + Y
print(Z)
```



*i1)* **X = 1**  
*i2)* **Y = 2**  
*i3)* **Z = X + Y**  
*i4)* **print(Z)**



**X**

**1**

# Un semplice programma in Python (P1)

```
X = 1
Y = 2
Z = X + Y
print(Z)
```



*i1)* **X = 1**  
*i2)* **Y = 2**  
*i3)* **Z = X + Y**  
*i4)* **print(Z)**



**Y**  
**X**

**:**  
**:**

**2**  
**1**

# Un semplice programma in Python (P1)

```
X = 1
Y = 2
Z = X + Y
print(Z)
```



*i1)* **X = 1**  
*i2)* **Y = 2**  
*i3)* **Z = X + Y**  
*i4)* **print(Z)**



	:
	:
<b>Z</b>	<b>3</b>
<b>Y</b>	<b>2</b>
<b>X</b>	<b>1</b>

# Un semplice programma in Python (P1)

```
X = 1
Y = 2
Z = X + Y
print(Z)
```

**3**

*i1)*

*i2)*

*i3)*

*i4)*

:

:

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



*j1)*

*j2)*

*j3)*

*i1)*

*i2)*

*i3)*

*i4)*

*i5)*

:

:

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



*j1)*    **somma(A,B):**

*j2)*    **C = A + B**

*j3)*    **Return C**

*i1)*    **X = 1**

*i2)*    **Y = 2**

*i3)*    **Z = somma(X,Y)**

*i4)*    **Z = Z \* Z**

*i5)*    **print(Z)**

:

:

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



*j1)* **somma(A,B):**

*j2)* **C = A + B**

*j3)* **Return C**

*i1)* **X = 1**

*i2)* **Y = 2**

*i3)* **Z = somma(X,Y)**

*i4)* **Z = Z \* Z**

*i5)* **print(Z)**



**X**

**1**

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



*j1)* **somma(A,B):**

*j2)* **C = A + B**

*j3)* **Return C**

*i1)* **X = 1**

*i2)* **Y = 2**

*i3)* **Z = somma(X,Y)**

*i4)* **Z = Z \* Z**

*i5)* **print(Z)**



**Y**

**2**

**X**

**1**



# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1) **somma(A,B):**

j2) **C = A + B**

j3) **Return C**

i1) **X = 1**

i2) **Y = 2**

i3) **Z = somma(X,Y)**

i4) **Z = Z \* Z**

i5) **print(Z)**



**Z**

**Y**

**X**

:

:

**2**

**1**

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1)	<b>somma(A,B):</b>	←
j2)	<b>C = A + B</b>	
j3)	<b>Return C</b>	
i1)	<b>X = 1</b>	
i2)	<b>Y = 2</b>	
i3)	<b>Z = somma(X,Y)</b>	←
i4)	<b>Z = Z * Z</b>	
i5)	<b>print(Z)</b>	

	:
	:
<b>B</b>	<b>2</b>
<b>A</b>	<b>1</b>
<b>ret</b>	<b>i3</b>
<b>Z</b>	
<b>Y</b>	<b>2</b>
<b>X</b>	<b>1</b>

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1)	somma(A,B):
j2)	C = A + B
j3)	Return C
i1)	X = 1
i2)	Y = 2
i3)	Z = somma(X,Y)
i4)	Z = Z * Z
i5)	print(Z)



	:
	:
C	3
B	2
A	1
ret	i3
Z	
Y	2
X	1

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1)	somma(A,B):
j2)	C = A + B
j3)	Return C
i1)	X = 1
i2)	Y = 2
i3)	Z = somma(X,Y)
i4)	Z = Z * Z
i5)	print(Z)



	:
	:
C	3
B	2
A	1
ret	i3
Z	
Y	2
X	1

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1)	<b>somma(A,B):</b>
j2)	<b>C = A + B</b>
j3)	<b>Return C</b>
i1)	<b>X = 1</b>
i2)	<b>Y = 2</b>
i3)	<b>Z = somma(X,Y)</b>
i4)	<b>Z = Z * Z</b>
i5)	<b>print(Z)</b>



	:
	:
<b>C</b>	<b>3</b>
<b>B</b>	<b>2</b>
<b>A</b>	<b>1</b>
<b>ret</b>	<b>i3</b>
<b>Z</b>	<b>3</b>
<b>Y</b>	<b>2</b>
<b>X</b>	<b>1</b>

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1)	somma(A,B):
j2)	C = A + B
j3)	Return C
i1)	X = 1
i2)	Y = 2
i3)	Z = somma(X,Y)
i4)	Z = Z * Z
i5)	print(Z)



	:
	:
C	3
B	2
A	1
ret	i3
Z	3
Y	2
X	1

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



j1) **somma(A,B):**

j2) **C = A + B**

j3) **Return C**

i1) **X = 1**

i2) **Y = 2**

i3) **Z = somma(X,Y)**

i4) **Z = Z \* Z**

i5) **print(Z)**



**Z**

**3**

**Y**

**2**

**X**

**1**

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```



*j1)* **somma(A,B):**

*j2)* **C = A + B**

*j3)* **Return C**

*i1)* **X = 1**

*i2)* **Y = 2**

*i3)* **Z = somma(X,Y)**

*i4)* **Z = Z \* Z**

*i5)* **print(Z)**



**Z**

**9**

**Y**

**2**

**X**

**1**



# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```

9

j1) **somma(A,B):**

j2) **C = A + B**

j3) **Return C**

i1) **X = 1**

i2) **Y = 2**

i3) **Z = somma(X,Y)**

i4) **Z = Z \* Z**

i5) **print(Z)**



**Z**

**3**

**Y**

**2**

**X**

**1**

# Un semplice programma in Python (P2)

```
def somma (A, B) :  
    C = A + B  
    return C
```

```
X = 1  
Y = 2  
Z = somma (X, Y)  
Z = Z * Z  
print (Z)
```

9

*j1)*

*j2)*

*j3)*

*i1)*

*i2)*

*i3)*

*i4)*

*i5)*

# Un semplice programma in Python (P3)

```
X = [1, 2]
Y = X
Y.append(3)
print(X)
```



*j1)*

*j2)*

*j3)*

*i1)*

*i2)*

*i3)*

*i4)*

*i5)*

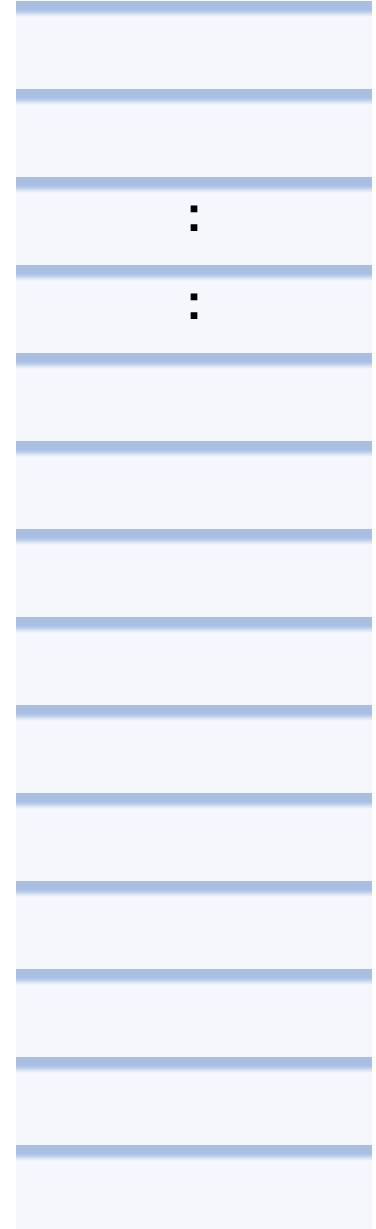
:

:

# Un semplice programma in Python (P3)

```
X = [1, 2]  
Y = X  
Y.append(3)  
print(X)
```

*i1)* **X = [1,2]**  
*i2)* **Y = X**  
*i3)* **Y.append(3)**  
**print(X)**



# Un semplice programma in Python (P3)

```
X = [1, 2]  
Y = X  
Y.append(3)  
print(X)
```



*i1)* **X = [1,2]** ← **k1**  
*i2)* **Y = X**  
*i3)* **Y.append(3)**  
*i4)* **print(X)**



# Un semplice programma in Python (P3)

```
X = [1, 2]
Y = X
Y.append(3)
print(X)
```



*i1)* **X = [1,2]**

*i2)* **Y = X**

*i3)* **Y.append(3)**

*i4)* **print(X)**

**k1**

**[1, 2]**

:

:

**Y**

**k1**

**X**

**k1**



# Un semplice programma in Python (P3)

```
X = [1, 2]
Y = X
Y.append(3)
print(X)
```



i1) **X = [1,2]**

i2) **Y = X**

i3) **Y.append(3)**

i4) **print(X)**

**k1**

**[1, 2, 3]**

:

:

**Y**

**k1**

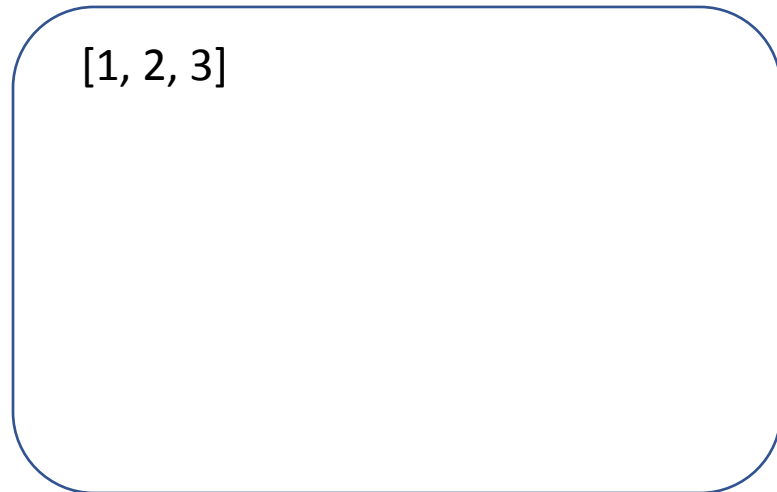
**X**

**k1**



# Un semplice programma in Python (P3)

```
X = [1, 2]
Y = X
Y.append(3)
print(X)
```



i1) **X = [1,2]**  
i2) **Y = X**  
i3) **Y.append(3)**  
i4) **print(X)**

**k1**

**[1, 2, 3]**

:

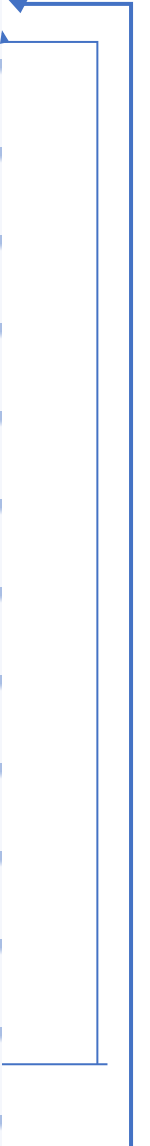
:

**Y**

**k1**

**X**

**k1**





# Un semplice programma in Python (P3)

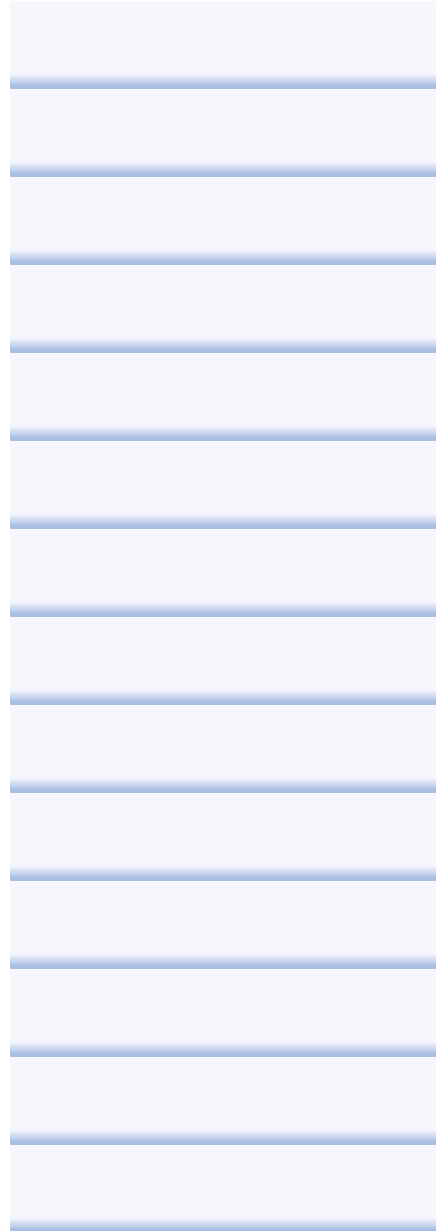
```
X = [1, 2]  
Y = X  
Y.append(3)  
print(X)
```

[1, 2, 3]

*i1)*

*i2)*

*i3)*



# Una semplice funzione: Fibonacci

## Definizione ricorsiva della serie di Fibonacci

La serie di Fibonacci ha una **definizione ricorsiva**, in cui vengono specificati i primi due termini per poi definire un generico elemento della serie come somma dei precedenti:

$$F_n = \begin{cases} 1 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n \geq 3 \end{cases}$$

Storia [ modifica | modifica wikitesto ]

L'intento di **Leonardo Fibonacci** era quello di trovare una legge matematica che descrivesse la crescita di una popolazione di **conigli**.

Assumendo per ipotesi che:

- si disponga di una coppia di conigli appena nati
- questa prima coppia diventi fertile al compimento del primo mese e dia alla luce una nuova coppia al compimento del secondo mese;
- le nuove coppie nate si comportino in modo analogo;
- le coppie fertili dal secondo mese di vita in poi diano alla luce una coppia di figli al mese;

si verifica quanto segue:

- dopo un mese una coppia di conigli sarà fertile,
- dopo due mesi ci saranno due coppie di cui una sola fertile,
- nel mese seguente, terzo mese dal momento iniziale, ci saranno  $2 + 1 = 3$  coppie perché solo la coppia fertile avrà generato; di queste tre, due saranno le coppie fertili, quindi
- nel mese seguente (quarto mese dal momento iniziale) ci saranno  $3 + 2 = 5$  coppie

	In principio	Al termine del primo mese	Al termine del secondo mese	Al termine del terzo mese	Al termine del quarto mese	Al termine del quinto mese	Al termine del sesto mese	Al termine del settimo mese	Al termine dell'ottavo mese	Al termine del nono mese	Al termine del decimo mese	Al termine dell'undicesimo mese	Al termine del dodicesimo mese
Coppie di conigli	1	1	2	3	5	8	13	21	34	55	89	144	233

# La funzione Fibonacci in Python - iterativa

```
def fibIt1(n):  
    F = [0,1]  
    for i in range(2,n+1):  
        F.append(F[i-1]+F[i-2])  
    return F[n]  
  
print(fibIt1(30))
```

# La funzione Fibonacci in Python - iterativa

```
def fibI2(n):  
    if n == 0 or n == 1:  
        return n  
    f2 = 0  
    f1 = 1  
    f = 1  
    for i in range(2, n+1)  
        f2 = f1  
        f1 = f  
        f = f1 + f1  
    return f  
  
print(fibI2(30))
```

# La funzione Fibonacci in Python - ricorsiva

```
def fibRec(n) :  
    if n == 0 or n == 1 :  
        return n  
    else :  
        return fibRec(n-1) + fibRec(n-2)  
  
print(fibRec(5))
```

# Una semplice funzione: fattoriale

Fattoriale(n) definito per n numero naturale

$$\bullet \textit{Fact}(n) = \begin{cases} 1 & \textit{se } n = 0 \\ n \times \textit{Fact}(n - 1) & \textit{se } n > 0 \end{cases}$$

Equivalente a:

$$\bullet \textit{Fact}(n) = \prod_{k=1}^n k$$

$$\begin{aligned} 5! &= 5 \times 4! = \\ 5 \times 4 \times 3! &= \\ 5 \times 4 \times 3 \times 2! &= \\ 5 \times 4 \times 3 \times 2 \times 1! &= \\ 5 \times 4 \times 3 \times 2 \times 1 \times 0! &= \\ 5 \times 4 \times 3 \times 2 \times 1 \times 1 &= \\ 120 \end{aligned}$$

# La funzione fattoriale in Python - iterativa

```
def factIt(n):  
    s = 1  
    for i in range(1, n+1):  
        s = s*i  
    return s
```

```
print(factIt(5))
```

$$Fact(n) = \prod_{i=1}^n i$$



# Un semplice programma in Python (Fattoriale Iter.)

```
def fact(n):  
    f = 1  
    for i in range(2,n):  
        f = f * i  
    return f
```

```
n = 3  
f = fact(n)  
print(f)
```



*j1)*

*j2)*

*j3)*

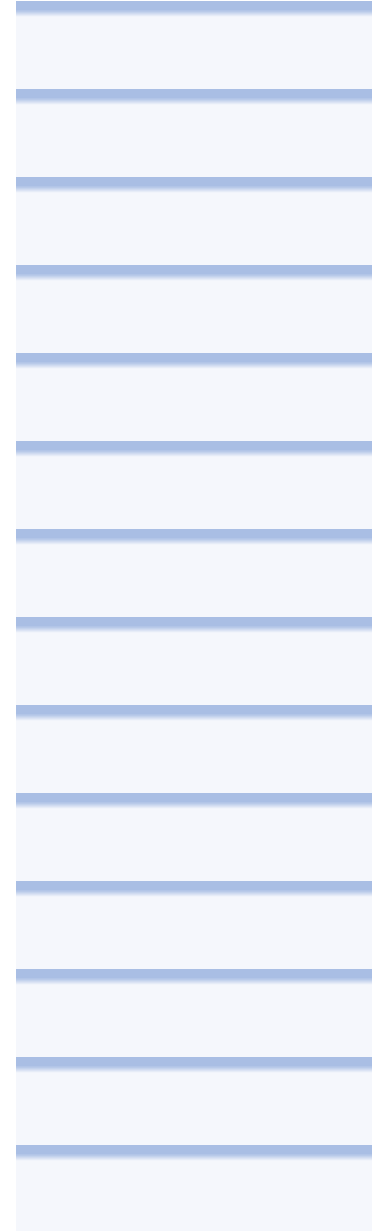
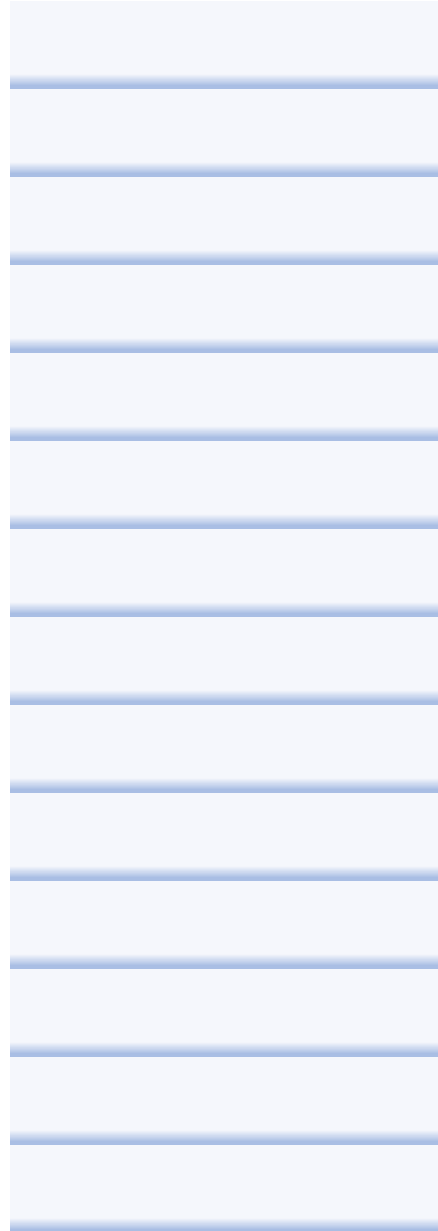
*j4)*

*j5)*

*i1)*

*i2)*

*i3)*



# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1)	<b>factIt(n)</b>
j2)	<b>f = 1</b>
j3)	<b>for i in</b> <b>range(2,n+1)</b>
j4)	<b>f = f * i</b>
j5)	<b>return f</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1)	<b>factIt(n)</b>
j2)	<b>f = 1</b>
j3)	<b>for i in</b> <b>range(2,n+1)</b>
j4)	<b>f = f * i</b>
j5)	<b>return f</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>



n

3

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



*j1)*    **factIt(n)**  
*j2)*    **f = 1**  
*j3)*    **for i in**  
         **range(2,n+1)**  
*j4)*    **f = f \* i**  
*j5)*    **return f**

*i1)*    **n = 3**  
*i2)*    **f = factIt(n)**  
*i3)*    **printf(f)**



**f**

**n**

**3**

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1) **factIt(n)**

j2) **f = 1**

j3) **for i in  
range(2, n+1)**

j4) **f = f \* i**

j5) **return f**

i1) **n = 3**

i2) **f = factIt(n)**

i3) **print(f)**



n

3

ii2

f

n

3

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



*j1)*    **factIt(n)**  
*j2)*    **f = 1**  
*j3)*    **for i in**  
         **range(2,n+1)**  
*j4)*    **f = f \* i**  
*j5)*    **return f**

*i1)*    **n = 3**  
*i2)*    **f = factIt(n)**  
*i3)*    **printf(f)**



**f**

**1**

**n**

**3**

**ii2**

**f**

**n**

**3**

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



*j1)*    **factIt(n)**  
*j2)*    **f = 1**  
*j3)*    **for i in**  
         **range(2,n+1)**  
*j4)*    **f = f \* i**  
*j5)*    **return f**

*i1)*    **n = 3**  
*i2)*    **f = factIt(n)**  
*i3)*    **printf(f)**



i	2
f	1
n	3
	ii2
f	
n	3

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1) **factIt(n)**  
j2) **f = 1**  
j3) **for i in range(2, n+1)**  
j4) **f = f \* i**  
j5) **return f**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **print(f)**



i	2
f	2
n	3
	ii2
f	
n	3



# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



*j1)*    **factIt(n)**  
*j2)*    **f = 1**  
*j3)*    **for i in**  
         **range(2,n+1)**  
*j4)*    **f = f \* i**  
*j5)*    **return f**

*i1)*    **n = 3**  
*i2)*    **f = factIt(n)**  
*i3)*    **printf(f)**



<b>i</b>	<b>3</b>
<b>f</b>	<b>2</b>
<b>n</b>	<b>3</b>
	<b>ii2</b>
<b>f</b>	
<b>n</b>	<b>3</b>

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1) **factIt(n)**  
j2) **f = 1**  
j3) **for i in**  
      **range(2,n+1)**  
j4) **f = f \* i**  
j5) **return f**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **printf(f)**



	:
	:
i	<b>2</b>
f	<b>6</b>
n	<b>3</b>
	ii2
f	
n	<b>3</b>

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1) **factIt(n)**  
j2) **f = 1**  
j3) **for i in range(2, n+1):**  
j4) **f = f \* i**  
j5) **return f**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **print(f)**



	:
	:
i	4
f	1
n	3
	ii2
f	
n	3

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1) **factIt(n)**  
j2) **f = 1**  
j3) **for i in range(2,n+1):**  
j4) **f = f \* i**  
j5) **return f**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **print(f)**



	:
	:
f	6
n	3
	ii2
f	
n	3

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2, n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



j1) **factIt(n)**  
j2) **f = 1**  
j3) **for i in range(2, n+1):**  
j4) **f = f \* i**  
j5) **return f**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **print(f)**



f	6
n	3
	ii2
f	6
n	3

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```



*j1)*    **factIt(n)**  
*j2)*    **f = 1**  
*j3)*    **for i in**  
         **range(2,n+1)**  
*j4)*    **f = f \* i**  
*j5)*    **return f**

*i1)*    **n = 3**  
*i2)*    **f = factIt(n)**  
*i3)*    **printf(f)**



<b>f</b>	<b>6</b>
<b>n</b>	<b>3</b>

# Un semplice programma in Python (Fattoriale)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```

6

j1) **factIt(n)**  
j2) **f = 1**  
j3) **for i in**  
    **range(2,n+1)**  
j4) **f = f \* i**  
j5) **return f**  
  
i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **printf(f)**



**f**

**6**

**n**

**3**

# Un semplice programma in Python (Fattoriale It)

```
def factIt(n):  
    f = 1  
    for i in range(2,n+1):  
        f = f * i  
    return f
```

```
n = 3  
f = factIt(n)  
print(f)
```

**6**

*j1)*

*j2)*

*j3)*

*j4)*

*j5)*

*i1)*

*i2)*

*i3)*



# La funzione fattoriale in Python - ricorsiva

```
def factRec(n) :  
    if n == 0 :  
        return 1  
    else :  
        return n * factR(n-1)  
  
print(factRec(5))
```

$$Fact(n) = \begin{cases} 1 & se\ n = 0 \\ n \times Fact(n-1) & se\ n > 0 \end{cases}$$

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



*j1)*

*j2)*

*j3)*

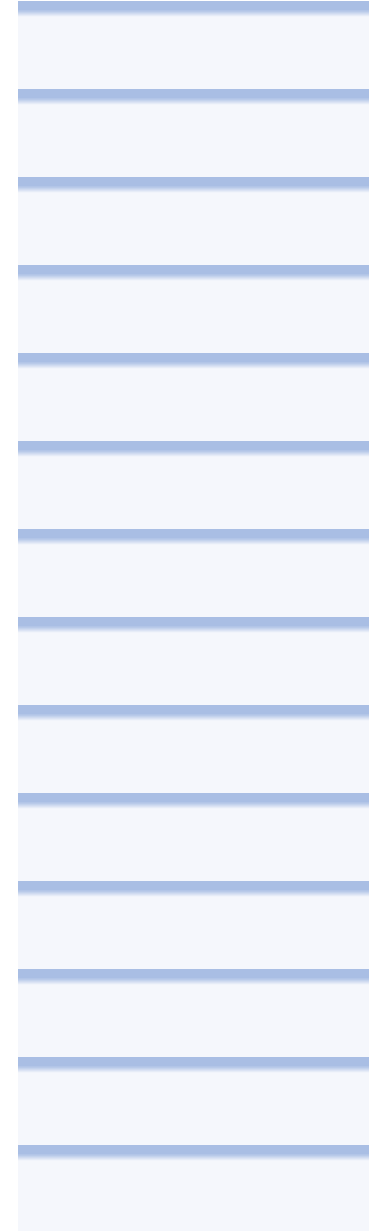
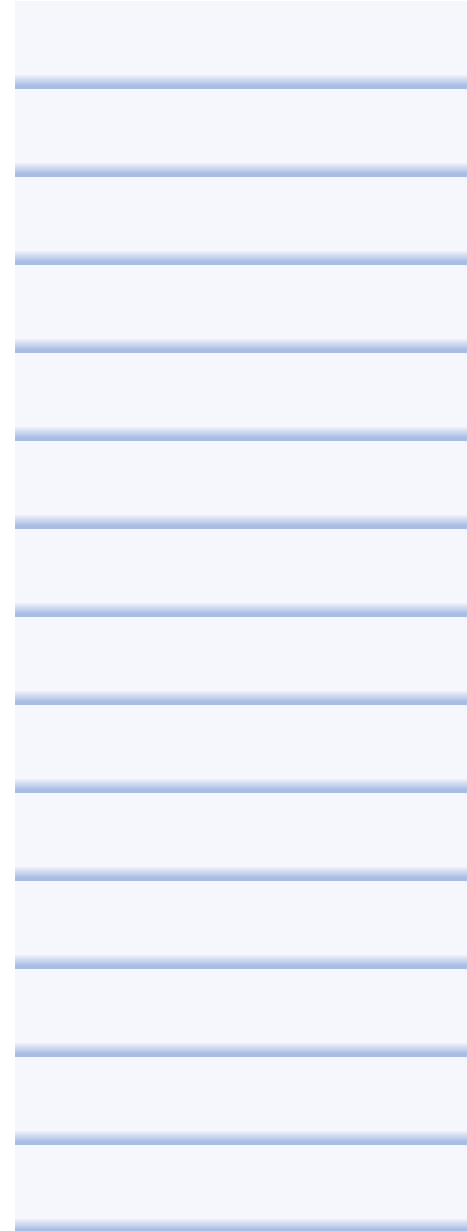
*j4)*

*j5)*

*i1)*

*i2)*

*i3)*



# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g = factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g =</b> <b>factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>



n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g =</b> <b>factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>



n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g =</b> <b>factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>



**f**

**n**

**3**

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1) **factRec(n)**  
j2) **if n==0 or n==1**  
j3) **return 1**  
j4) **else:**  
j5) **g = factRec(n-1)**  
j6) **return n\*g**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **printf(f)**



n

3

i2

f

n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



*j1)* **factRec(n)**  
*j2)* **if n==0 or n==1**  
*j3)* **return 1**  
*j4)* **else:**  
*j5)* **g = factRec(n-1)**  
*j6)* **return n\*g**

*i1)* **n = 3**  
*i2)* **f = factIt(n)**  
*i3)* **printf(f)**



**n**

**3**

**i2**

**f**

**n**

**3**



# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	factRec(n)
j2)	if n==0 or n==1
j3)	return 1
j4)	else:
j5)	g = factRec(n-1)
j6)	return n*g
i1)	n = 3
i2)	f = factI(n)
i3)	printf(f)



g

n

f

n

3

i2

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1) **factRec(n)**  
j2) **if n==0 or n==1**  
j3) **return 1**  
j4) **else:**  
j5) **g = factRec(n-1)**  
j6) **return n\*g**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **prinf(f)**



n

2

j5

g

n

3

i2

f

n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	factRec(n)
j2)	if n==0 or n==1
j3)	return 1
j4)	else:
j5)	g = factRec(n-1)
j6)	return n*g
i1)	n = 3
i2)	f = factI(n)
i3)	printf(f)



n

2

j5

g

n

3

i2

f

n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	factRec(n)
j2)	if n==0 or n==1
j3)	return 1
j4)	else:
j5)	g = factRec(n-1)
j6)	return n*g
i1)	n = 3
i2)	f = factIt(n)
i3)	prinf(f)



g

n

2

j5

g

n

3

i2

f

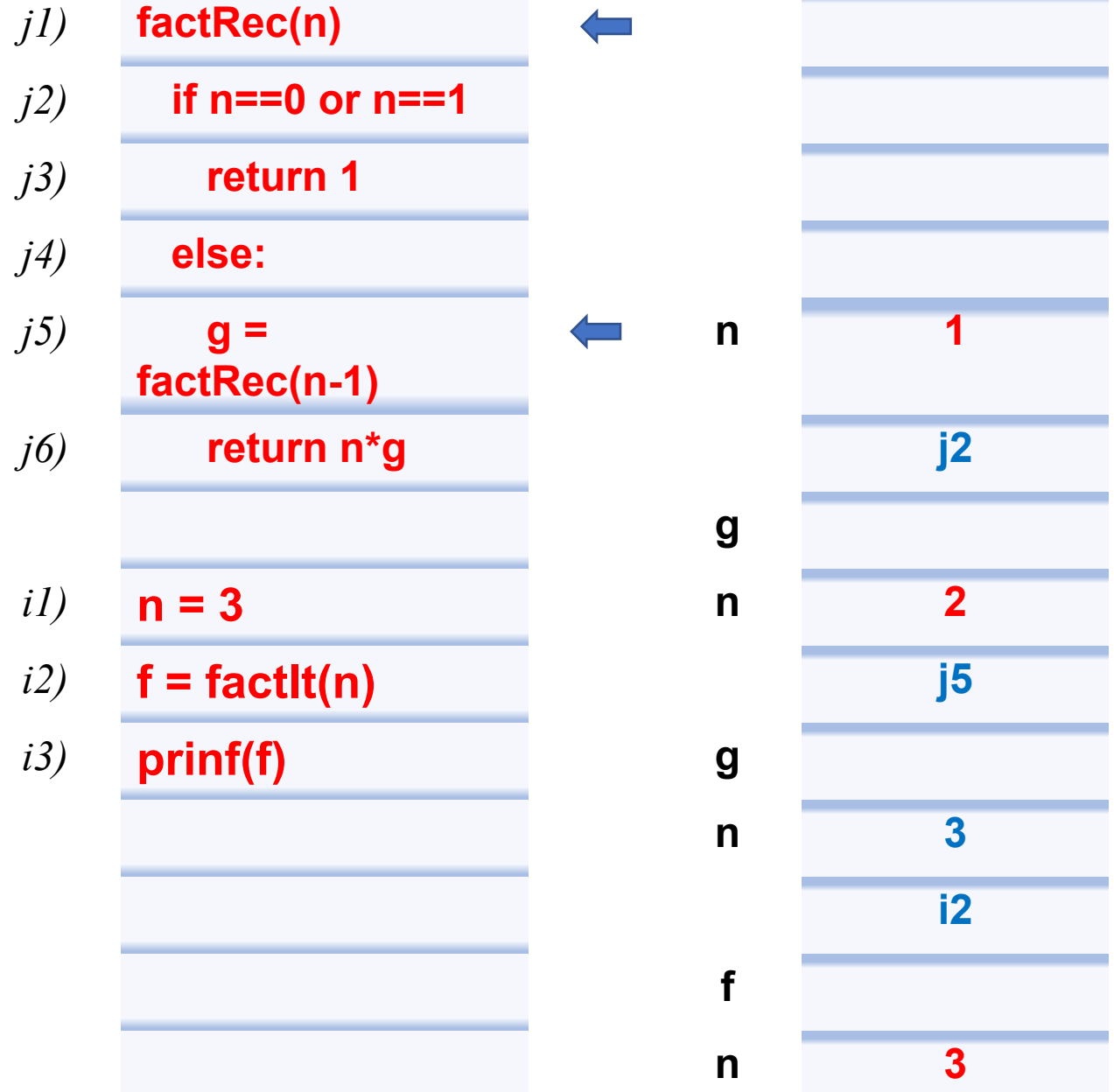
n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



**Rec)**

```
n = 3
f = factIRec(n)
print(f)
```



# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	factRec(n)
j2)	if n==0 or n==1
j3)	return 1
j4)	else:
j5)	g = factRec(n-1)
j6)	return n*g
i1)	n = 3
i2)	f = factIt(n)
i3)	prinf(f)



n	1
ret	j2
g	
n	2
ret	j5
g	
n	3
ret	i2
f	
n	3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	factRec(n)
j2)	if n==0 or n==1
j3)	return 1
j4)	else:
j5)	g = factRec(n-1)
j6)	return n*g
i1)	n = 3
i2)	f = factIRec(n)
i3)	printf(f)



n	1
ret	j2
g	1
n	2
ret	j5
g	
n	3
ret	i2
f	
n	3



# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g =</b> <b>factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>prinf(f)</b>



g	<b>1</b>
n	<b>2</b>
ret	<b>j5</b>
g	
n	<b>3</b>
ret	<b>i2</b>
f	
n	<b>3</b>

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1) **factRec(n)**  
j2) **if n==0 or n==1**  
j3) **return 1**  
j4) **else:**  
j5) **g = factRec(n-1)**  
j6) **return n\*g**

i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **prinf(f)**



g **1**  
n **2**  
ret **j5**  
g **3**  
n **3**  
ret **i2**  
f  
n **3**

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



*j1)*    **factRec(n)**  
*j2)*    **if n==0 or n==1**  
*j3)*    **return 1**  
*j4)*    **else:**  
*j5)*    **g =**  
         **factRec(n-1)**  
*j6)*    **return n\*g**

*i1)*    **n = 3**  
*i2)*    **f = factIt(n)**  
*i3)*    **prinf(f)**



**g**  
**n**  
**ret**  
**f**  
**n**

**3**

**3**

**i2**

**3**

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g =</b> <b>factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>prinf(f)</b>



g	<b>3</b>
n	<b>3</b>
ret	<b>i2</b>
f	
n	<b>3</b>

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```



j1)	<b>factRec(n)</b>
j2)	<b>if n==0 or n==1</b>
j3)	<b>return 1</b>
j4)	<b>else:</b>
j5)	<b>g =</b> <b>factRec(n-1)</b>
j6)	<b>return n*g</b>
i1)	<b>n = 3</b>
i2)	<b>f = factIt(n)</b>
i3)	<b>printf(f)</b>



<b>f</b>	<b>6</b>
<b>n</b>	<b>3</b>

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```

6

j1) **factRec(n)**  
j2) **if n==0 or n==1**  
j3) **return 1**  
j4) **else:**  
j5) **g = factRec(n-1)**  
j6) **return n\*g**  
  
i1) **n = 3**  
i2) **f = factIt(n)**  
i3) **prinf(f)**



f

6

n

3

# Un semplice programma in Python (Fattoriale Rec)

```
def factRec(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        g = factRec(n-1)  
        return n*g
```

```
n = 3  
f = factIRec(n)  
print(f)
```

6

*j1)*

*j2)*

*j3)*

*j4)*

*j5)*

*j6)*

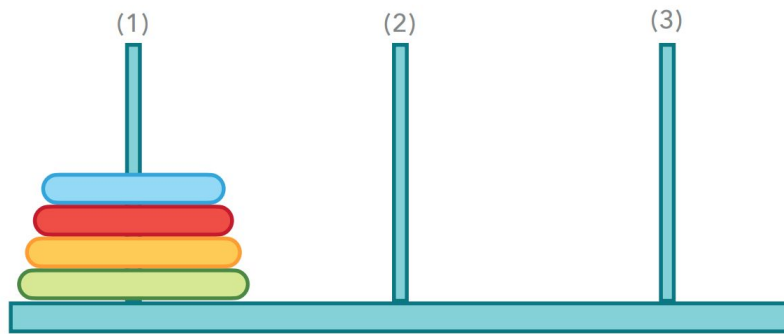
*i1)*

*i2)*

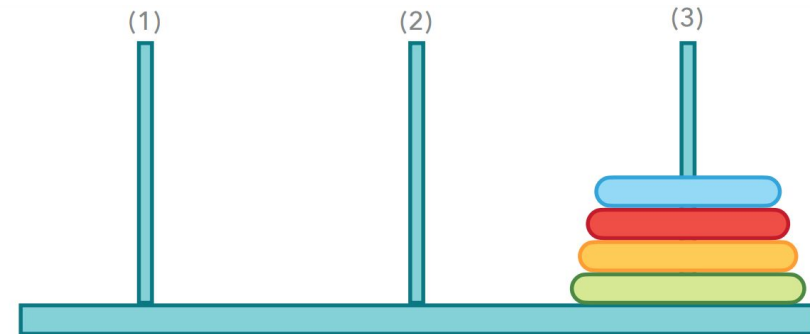
*i3)*

# La Torre di Hanoi

- Tre pile di dischi (“torri”) allineate
- All’inizio tutti i dischi si trovano sulla pila di sinistra
- Alla fine tutti i dischi si devono trovare sulla pila di destra
- I dischi sono tutti di dimensioni diverse e nessun disco può avere sopra di sé dischi più grandi



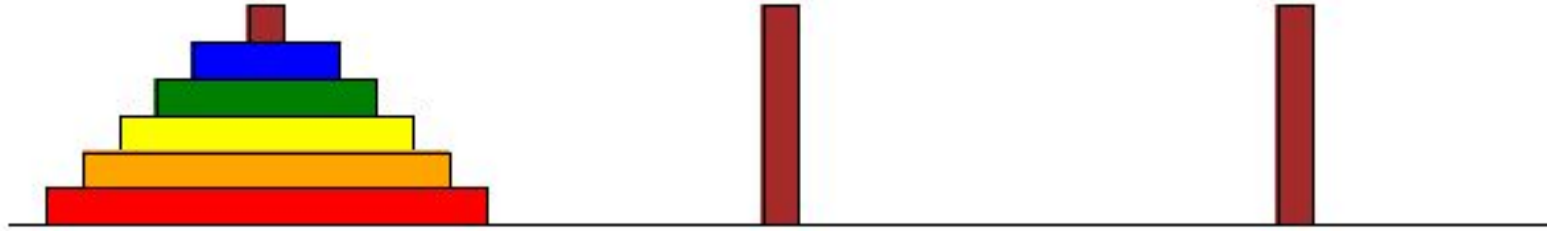
Situazione iniziale



Situazione finale



# Torre di Hanoi



<https://see-math.math.tamu.edu/2010/CounselorMovies/philip-y.html>

<https://www.mathsisfun.com/games/towerofhanoi.html>

# La Torre di Hanoi

**Caso base**  $\square$   $n = 1$ : possiamo spostare liberamente il disco da una torre ad un'altra

Per spostare  $n$  dischi dalla torre 1 alla torre 3:

- 1) gli  $n-1$  dischi in cima alla torre 1 vengono spostati sulla torre 2, usando la torre 3 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 3 rimane vuota)
- 2) il disco rimasto nella torre 1 viene portato nella torre 3
- 3) gli  $n-1$  dischi in cima alla torre 2 vengono spostati sulla torre 3, usando la torre 1 come deposito temporaneo (si usa una chiamata ricorsiva, al termine della quale la torre 1 rimane vuota)

Il numero di mosse necessarie per risolvere il rompicapo con l'algoritmo proposto è pari a  $2^n - 1$

Il tempo necessario alla soluzione è proporzionale al numero di mosse

```
def TorreHanoi(n , A, B, C):
```

```
    if n==1:
```

```
        print("Disk 1 from",A,"to",B)
```

```
        return
```

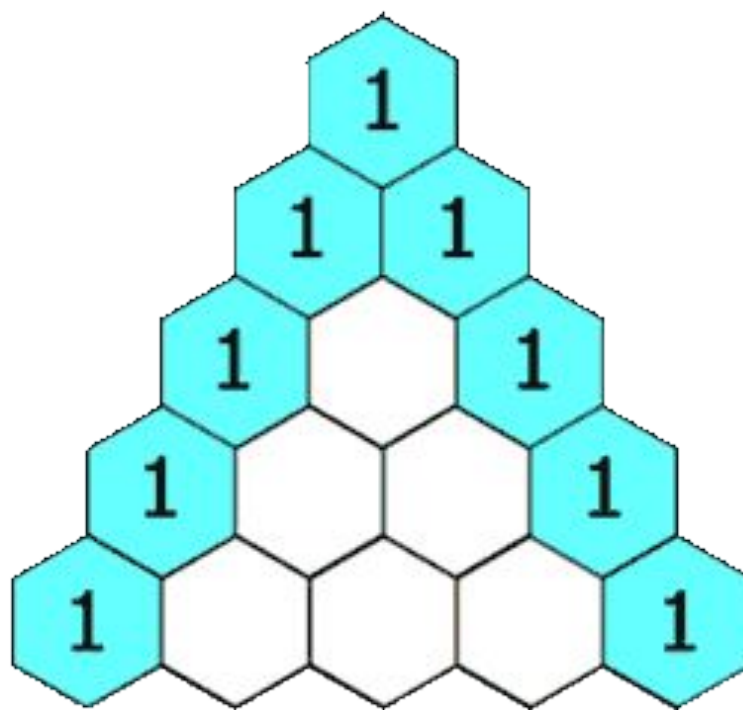
```
    TorreHanoi(n-1, A, C, B)
```

```
    print("Disk",n,"from",A,"to",B)
```

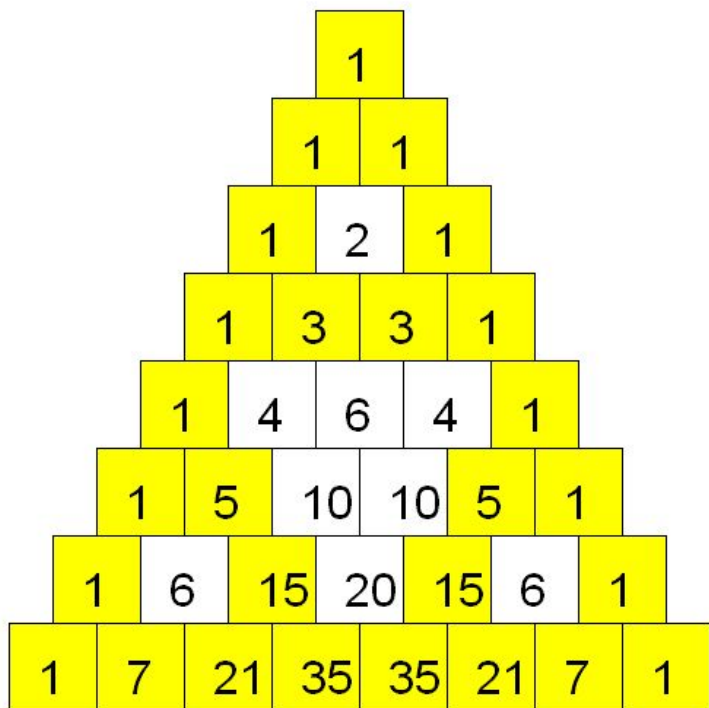
```
    TorreHanoi(n-1, C, B, A)
```

```
TorreHanoi(3, 'A', 'B', 'C')
```

# Triangolo di Tartaglia



# Triangolo di Tartaglia



In matematica, il **triangolo di Tartaglia** è una disposizione geometrica dei coefficienti binomiali, ossia dei coefficienti dello sviluppo del binomio elevato a una qualsiasi potenza, a forma di triangolo.

## Formula ricorsiva

Proprietà dei binomiali per cui 
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Questo porta a una formula ricorsiva per calcolare un numero del triangolo: il numero alla riga  $n$  al posto  $k$  è uguale alla somma di due numeri della fila precedente allo stesso posto e al posto precedente.

# **Operazioni su Vettori**

# Somma elementi di un vettore

```
def Sum(A) :  
    s = 0  
    n = len(A)  
    for i in range(n) :  
        s = s + A[i]  
    return s  
V = [0,1,2,4]  
print(Sum(V))
```

# Somma elementi di un vettore – versione 1 ricorsiva

```
def SumRec1(A,k) :  
    if k >= len(V) :  
        return 0  
    else:  
        return A[k] + SumRec1(A,k+1)  
V = [0,1,2,4]  
print(SumRec1(V,0))
```



# Somma elementi di un vettore – versione 2 ricorsiva

```
def SR2 (A,k) :  
    if k < 0 :  
        return 0  
    else :  
        return SR2 (A,k-1) + A[k]  
V = [0,1,2,4]  
print (SR2 (V, len (V) -1) )
```

# Somma di due vettori

```
def SumVect (A,B) :  
    C = []  
    n = len(V)  
    for i in range(n) :  
        C.append(A[i]+V[i])  
    return C  
  
U = [5,2,1,4]  
V = [0,1,2,4]  
print (SumVect (U,V) )
```

# Somma di due vettori – versione 1 ricorsiva

```
def SumVectRec (A,B,k) :  
    if k < 0 :  
        return []  
    else :  
        C = SumVectRec (A,B,k-1)  
        C.append (A[k] +B[k] )  
    return C  
  
U = [5,2,1,4]  
V = [0,1,2,4]  
print (SumVectRec (U,V,len (U) -1) )
```

# Somma elementi di un vettore – versione 2 ricorsiva

```
def SV(A,B,k) :  
    if k < 0:  
        return []  
    else:  
        C = SV(A,B,k-1)  
        C.append(A[k]+B[k])  
    return C  
U = [5,2,1,4]  
V = [0,1,2,4]  
print(SV(U,V,len(U)-1))
```

# **Algoritmi di ricerca e di ordinamento (vettori)**

# Ricerca in una sequenza di elementi

- Data una sequenza di elementi, verificare se un elemento fa parte della sequenza oppure se l'elemento non è presente nella sequenza stessa.
- Una sequenza di elementi può essere realizzata come un array e la ricerca avviene usando un indice.
- Se la sequenza non è ordinata a priori occorre eseguire una **ricerca lineare o sequenziale**.
- Se la sequenza è ordinata è opportuno eseguire una **ricerca binaria**.

# Ricerca lineare (o sequenziale)

- Algoritmo di ricerca lineare (o sequenziale) in un array:
  - Gli elementi dell'array vengono analizzati in sequenza, confrontandoli con l'elemento da ricercare (detta chiave)
  - La ricerca termina quando si trova un elemento uguale alla chiave
- L'algoritmo prevede che al più tutti gli elementi dell'array vengano confrontati con la chiave.
- Se l'elemento viene trovato prima di raggiungere la fine dell'array non sarà necessario proseguire la ricerca.

# Ricerca lineare (o sequenziale)

Sequential search

steps: 0



1	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



# Ricerca lineare: conteggio delle occorrenze

- L'algoritmo di ricerca lineare può essere usato per verificare quante volte un elemento è presente in una sequenza
  - 1. Gli elementi dell'array vengono analizzati in sequenza, confrontandoli con l'elemento da ricercare per determinare se uno degli elementi è uguale alla chiave.
  - 2. Quando si trova un elemento uguale alla chiave si incrementa un contatore e il passo 1 viene rieseguito fino alla fine della sequenza.

# Esempio di ricerca lineare in Python

Array: lista = [1,8,0,1,1,9,2,8]

Key: k = 9

## Soluzione 1

```
def ricerca(lista, k):  
    n=len(lista)  
    pos=-1 // se k non presente  
    for i in range(n):  
        if(lista[i] == k):  
            pos=i // se k è presente alla  
                posizione i
```

**return pos**  
Se x compare più volte nella lista allora  
la posizione restituita sarà l'ultima

## Soluzione 2

```
def ricerca(lista, k):  
    n=len(lista)  
    for i in range(n):  
        if(lista[i] == k):  
            return i  
  
    return -1
```

Restituisce la prima posizione

# Ricerca binaria (o dicotomica)

- Cerca un elemento in una sequenza ordinata crescente eseguendo una serie di passi finché l'elemento viene trovato o la ricerca è completata senza trovarlo:
  1. La chiave è confrontata con l'elemento centrale della sequenza
  2. Se la chiave è uguale all'elemento centrale, la ricerca termina
  3. Se la chiave è maggiore dell'elemento centrale si effettua la ricerca solo sulla sottosequenza a destra
  4. Se la chiave è minore dell'elemento centrale dello spazio di ricerca, si effettua la ricerca solo sulla sottosequenza a sinistra.
- Inizialmente la ricerca è fatta su  $N$  elementi dove  $N$  indica la lunghezza dell'array. Ad ogni iterazione lo spazio della ricerca si riduce di "circa" la metà (da  $N$  ad  $N/2$  e così via).
- Caso peggiore: l'elemento cercato non si trova nella sequenza e l'iterazione viene eseguita  $\log_2 N$  volte.



# Esempio di ricerca binaria in Python

## Soluzione 1 (non ricorsiva)

```
def binaria(lista, k):  
    left=0  
    right=len(lista)-1  
    while(left <= right) :  
        center=(left+right)/2  
        if k<lista[center]:  
            right=center-1  
        elif k>lista[center]:  
            left=center+1  
        else:  
            return center  
    return -1
```

## Soluzione 2 (ricorsiva)

```
def binariaric(lista, k, left=0, right=None):  
    if right==None: right=len(lista)  
    center=(left+ right)/2  
    if left>right:  
        return -1  
    if k<lista[center]:  
        return binariaric(lista, k, left, center-1)  
    elif k>lista[center]:  
        return binariaric(lista, k, center+1,  
right)  
    else:  
        return center
```

*left* indice inferiore della sequenza e *right* indice superiore della sequenza

# Algoritmi di ordinamento

- selection sort (semplice, intuitivo, poco efficiente)
- bubble sort (semplice, un po' più efficiente)
- insertion sort (intuitivo, abbastanza efficiente)
- merge sort (non intuitivo, molto efficiente)
- quick sort (non intuitivo, alquanto efficiente)

# Selection sort: definizione

Il **selection sort** o **ordinamento per selezione** opera dividendo la sequenza di input in due parti:

- la sottosequenza di elementi già *ordinati* (che occupa le prime posizioni dell'array) e la sottosequenza di elementi *non ordinati* (che occupa il resto dell'array).

Inizialmente, la sottosequenza ordinata è vuota, mentre quella non ordinata rappresenta l'intero input.

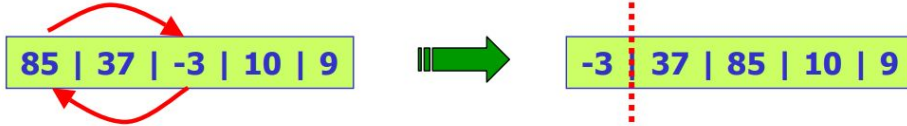
L'algoritmo seleziona di volta in volta il numero minore nella sottosequenza non ordinata e lo sposta in quella ordinata.

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

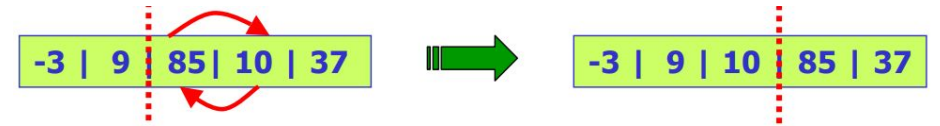
# Selection Sort: esempio

85 | 37 | -3 | 10 | 9

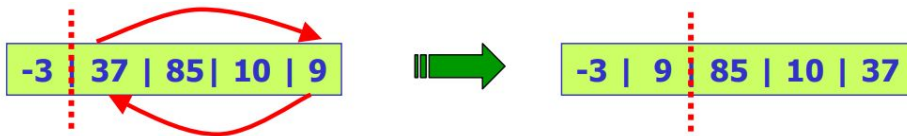
1° operazione



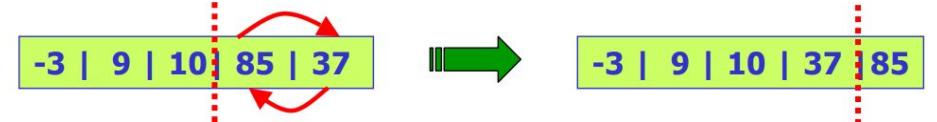
3° operazione



2° operazione



4° operazione





# Selection sort: funzionamento

- E' implementato innestando due cicli for annidati.
- Ogni elemento alla **posizione di indice corrente**, del ciclo più esterno, viene posto in una variabile "*minimo*" e verrà confrontato con i successivi elementi della lista, durante il **ciclo for** più interno, il quale, ogni volta che individua un valore più piccolo dell'elemento considerato, aggiorna la variabile "*minimo*".
- Alla fine dell'iterazione **vengono scambiate** le posizioni dell'elemento a cui punta l'indice del primo for e del valore di **minimo** trovato, iniziando così una nuova iterazione.
- Ad ogni scambio di posizione, è stato individuato il primo valore di minimo della lista, il quale andrà ad occupare la sua posizione definitiva: di conseguenza il secondo for non deve verificare gli elementi già posizionati.

```
SELECTIONSORT( $A, n$ )  
1: per  $i = 1, 2, \dots, n$  ripeti  
2:   per  $j = i + 1, \dots, n$  ripeti  
3:     se  $a_j < a_i$  allora  
4:       scambia  $a_i$  e  $a_j$   
5:     fine-condizione  
6:   fine-ciclo  
7: fine-ciclo
```

# Selection sort: implementazione in Python

```
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        minimo = lista[i] # primo valore di minimo coinciderà con l'elemento in posizione i-esima
        trovato = False
        for j in range(i+1,n): # trova il minimo corrente tra i valori successivi all'elemento in posizione i-esima
            if lista[j] < minimo:
                trovato = True
                minimo = lista[j]
                indice_trovato = j
        if trovato: # se "trovato" è True, nuovo valore di minimo individuato e si esegue scambio
            occ = lista[i]
            lista[i] = lista[indice_trovato]
            lista[indice_trovato] = occ
    return lista

# lista di prova
l=[5,3,4,1,2,-1,6,-9,0]
print(selection_sort(l))

# output:
[-9, -1, 0, 1, 2, 3, 4, 5, 6]
```

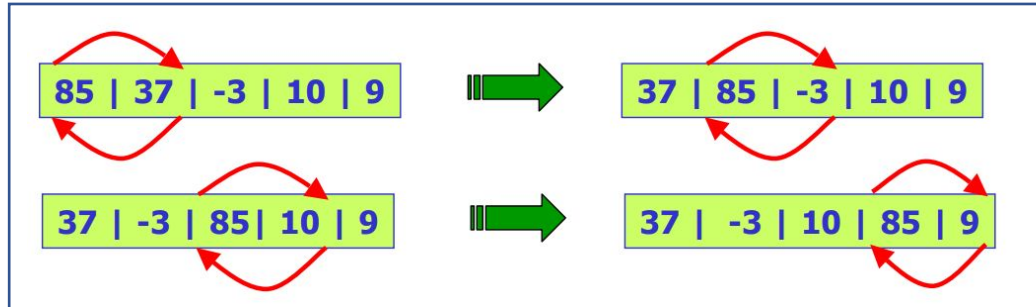
# Bubble sort: definizione

- Scorre ripetutamente l'elenco
- Confronta gli elementi adiacenti
- Prevede lo spostamento, ad ogni iterazione, dell'elemento più grande della lista nell'ultima posizione considerata.
- Scambia gli elementi se sono nell'ordine sbagliato.
- Alla prima iterazione è spostato in ultima posizione l'elemento più grande della lista, alla seconda iterazione è spostato il secondo elemento più grande, alla terza il terzo e così via

6 5 3 1 8 7 2 4

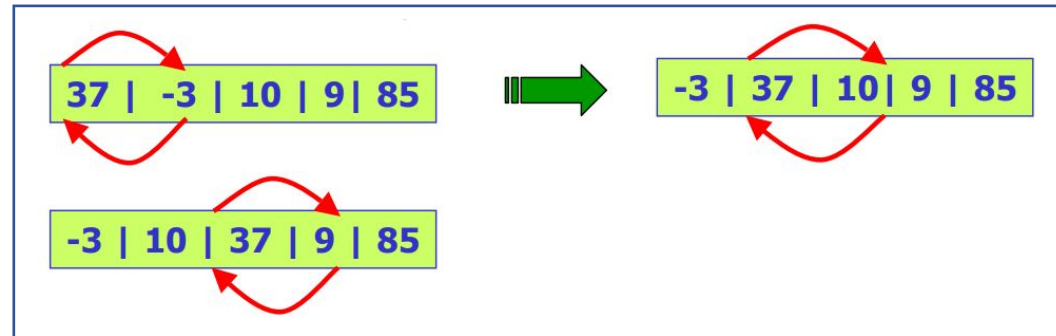
# Bubble sort: esempio

85 | 37 | -3 | 10 | 9

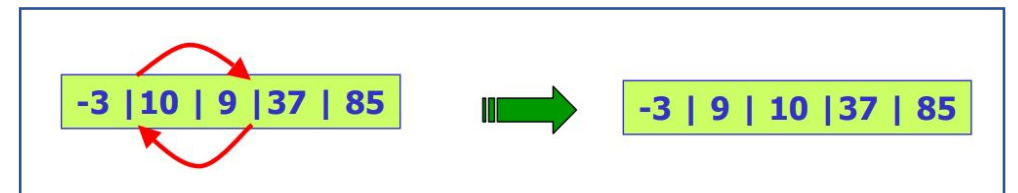


1° Scansione

2° Scansione



3° Scansione



# Bubble sort: funzionamento

- Immaginiamo che ogni elemento di una data lista sia una bolla di sapone: le bolle più pesanti (i numeri più grandi) vanno verso il basso, mentre le bolle più leggere (i numeri più piccoli), salgono verso l'alto.
- E' implementato con due cicli for annidati.
- Il **ciclo più esterno** esegue  $n-1$  iterazioni durante le quali il **ciclo for più interno** confronta ogni elemento alla posizione  $j$ -esima con il successivo: se il primo è maggiore del secondo inverte le posizioni.
- Alla fine di ogni iterazione del ciclo for più interno, sarà individuato il valore più grande che sarà collocato alla posizione definitiva.
- Ad ogni nuova iterazione **diminuisce** il range del *secondo ciclo for* poiché non sarà necessario confrontare gli elementi che si trovano alle posizioni definitive.

---

BUBBLESORT( $A, n$ )

```
1:  $flag = 1$ 
2:  $stop = n - 1$ 
3: fintanto che  $flag = 1$  ripeti
4:    $flag = 0$ 
5:   per  $i = 1, 2, \dots, stop$  ripeti
6:     se  $a_i > a_{i+1}$  allora
7:       scambia  $a_i$  e  $a_{i+1}$ 
8:        $flag = 1$ 
9:     fine-condizione
10:  fine-ciclo
11:   $stop = stop - 1$ 
12: fine-ciclo
```

---

**Algoritmo 3:** Bubble sort

# Bubble sort: implementazione in Python

```
def bubble_sort(lista):  
    n = len(lista)  
    for i in range(n - 1): # ciclo for per confrontare le coppie di valori  
        for j in range(n-1-i):  
            if lista[j] > lista[j+1]:  
                occ = lista[j+1] # variabile usata invertire le posizioni della coppia di valori confrontata  
                lista[j+1] = lista[j]  
                lista[j] = occ  
    return lista  
  
# lista di prova  
l=[5,3,4,1,2,-1,6,-9,0]  
print(bubble_sort(l))  
  
# output:  
[-9, -1, 0, 1, 2, 3, 4, 5, 6]
```

# Insertion sort: definizione

6 5 3 1 8 7 2 4

# Insertion sort: funzionamento

- Considerare ad ogni iterazione una **sottolista ordinata** di lunghezza pari all'indice del ciclo for più esterno (alla prima iterazione sarà formata da un solo elemento) e una **sottolista non ordinata** formata dai restanti elementi della lista.
- Ad ogni iterazione il primo elemento della sottolista non ordinata viene inserito nella sottolista ordinata, scalando ogni elemento maggiore di esso verso destra

```
INSERTIONSORT( $A, n$ )  
1: per  $i = 2, 3, \dots, n$  ripeti  
2:    $k = i - 1$   
3:   fintanto che  $k \geq 1$  e  $a_k > a_{k+1}$  ripeti  
4:     scambia  $a_k$  e  $a_{k+1}$   
5:      $k = k - 1$   
6:   fine-ciclo  
7: fine-ciclo
```

Algoritmo 2: Insertion sort



# Insertion Sort: implementazione in Python

```
def insertion_sort(lista):
    n = len(lista)
    for i in range(1,n): # effettua n-1 iterazioni a partire dal secondo elemento della lista
        valore = lista[i] # valore da inserire salvato nella variabile "valore"
        j = i-1
        while j >= 0 and valore < lista[j]:
            lista[j+1] = lista[j] # si esegue almeno un spostamento verso destra dei valori maggiori di "valore"
            j -= 1
        lista[j+1] = valore
    return lista

# lista di prova
l=[5,4,3,6,-9,0]
print(insertion_sort(l))

# output:
[-9, 0, 3, 4, 5, 6]
```

# Merge sort: definizione

Il **merge sort** è un algoritmo di ordinamento basato su confronti che utilizza un processo di **risoluzione ricorsivo**, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola.

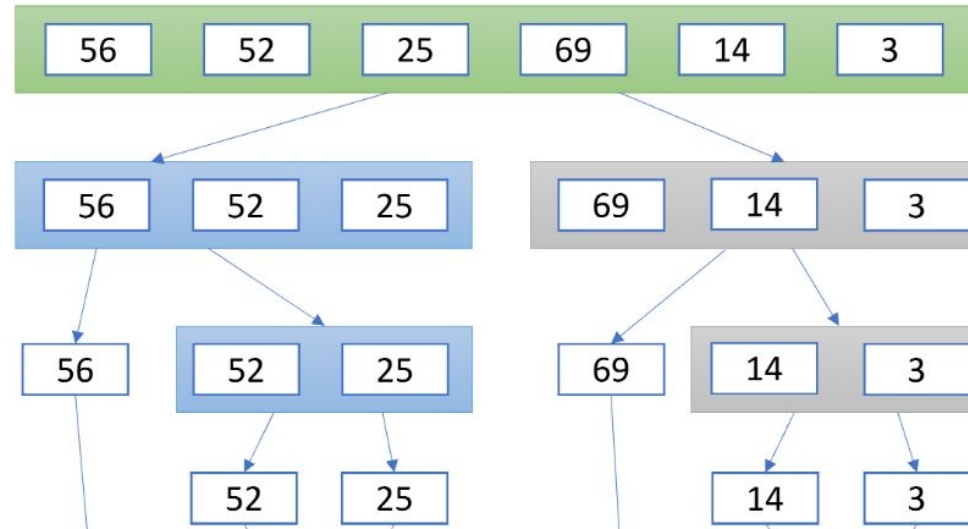
L'algoritmo lavora su due fasi:

1. **splitting**: divide ricorsivamente la lista fino a quando la dimensione dei dati risulta inferiore rispetto ad una certa soglia
2. **merging**: elabora i dati fino a restituire il risultato finale

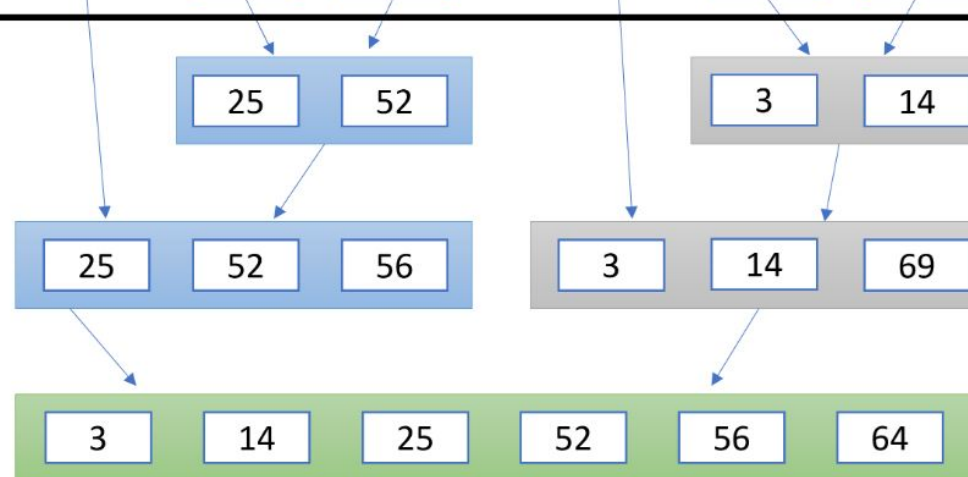
6 5 3 1 8 7 2 4

# Merge sort: esempio

Fase 1: SPLITTING



Fase 2: MERGING




# Merge sort: funzionamento

```
MERGESORT( $A, p, r$ )
1: se  $p < r$  allora
2:    $q = \frac{p+r}{2}$ 
3:   MERGESORT( $A, p, q$ )
4:   MERGESORT( $A, q+1, r$ )
5:   MERGE( $A, p, q, r$ )
6: fine-condizione

MERGE( $A, p, q, r$ )
1: siano  $k = 0, i = p$  e  $j = q+1$ 
2: fintanto che  $i \leq q$  e  $j \leq r$  ripeti
3:   se  $a_i < a_j$  allora
4:      $b_k = a_i, i = i+1, k = k+1$ 
5:   altrimenti
6:      $b_k = a_j, j = j+1, k = k+1$ 
7:   fine-condizione
8: fine-ciclo
9: se  $i \leq q$  allora
10:  copia  $(a_i, \dots, a_q)$  in  $B$ 
11: altrimenti
12:  copia  $(a_j, \dots, a_r)$  in  $B$ 
13: fine-condizione
14: copia  $B$  in  $A$ : per  $i = p, \dots, r, A_i = B_{i-p}$ 
```

Algoritmo 5: Merge sort



# Merge sort: implementazione in Python

```
def merge_sort(elements):
    if len(elements) > 1:
        # fase 1: splitting
        mid = len(elements) // 2
        left = elements[:mid]
        right = elements[mid:]
        merge_sort(left)
        merge_sort(right)

        # fase 2: merging
        a = b = c = 0
        while a < len(left) and b < len(right):
            if left[a] < right[b]:
                elements[c] = left[a]
                a += 1
            else:
                elements[c] = right[b]
                b += 1
            c += 1

        while a < len(left):
            elements[c] = left[a]
            a += 1
            c += 1

        while b < len(right):
            elements[c] = right[b]
            b += 1
            c += 1

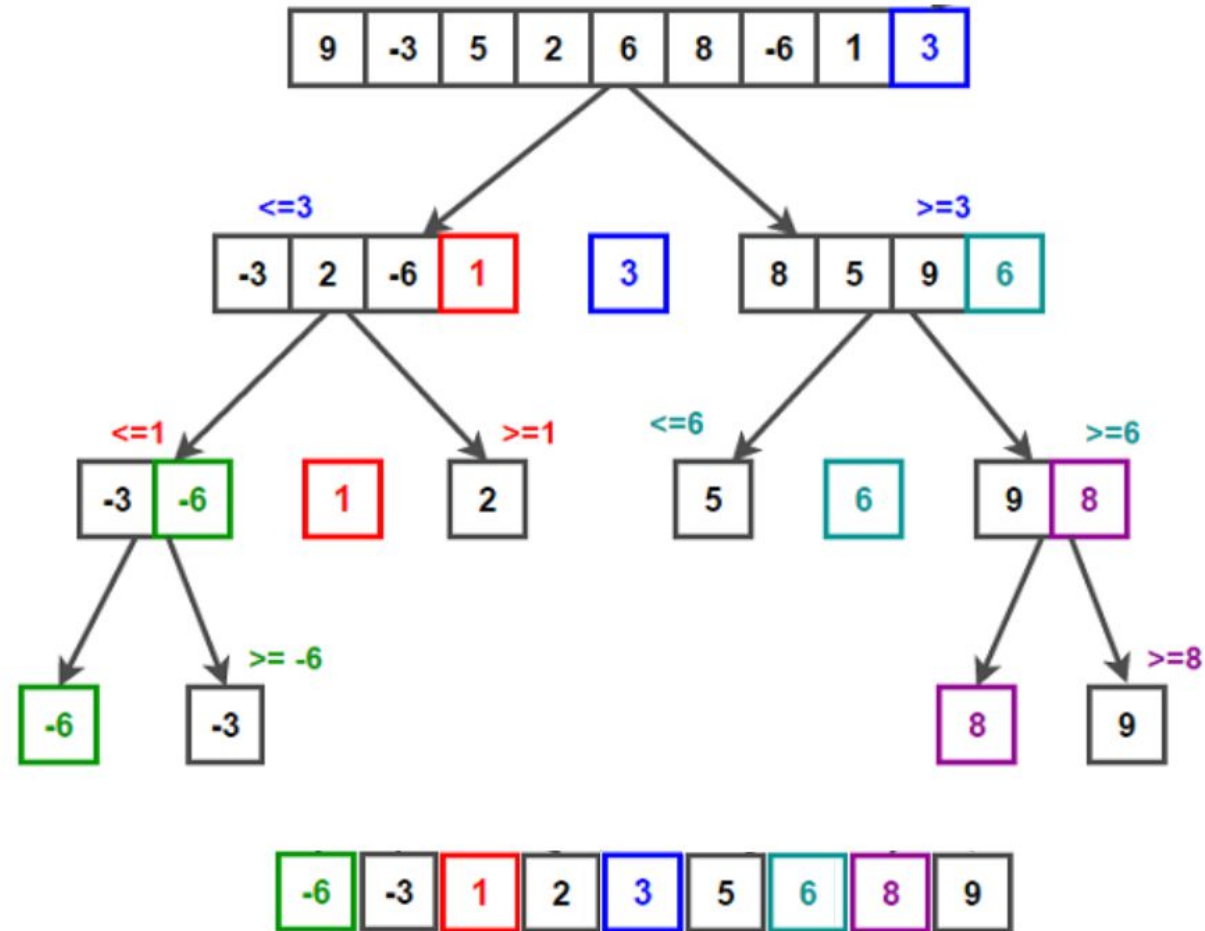
    return elements
```

# Quick sort: definizione

- Funzionamento basato su **pivot**, ovvero elemento che può essere selezionato in vari modi: può essere il primo elemento, l'ultimo, l'elemento di mezzo o un elemento scelto a caso.
- Algoritmo di ordinamento basato sull'approccio **divide et impera**!
- Concetto di partizione, ovvero suddivisione di un array in sottoarray: gli elementi più piccoli del pivot saranno posizionati alla sua sinistra mentre gli elementi più grandi saranno posizionati alla sua destra.

6 5 3 1 8 7 2 4

# Quick sort: esempio



# Quick sort: funzionamento

- **Divide** – Dividendo la lista da ordinare scomponiamo il problema in sotto-problemi.
- **Impera** – Si risolve l'algoritmo e si applica la ricorsione.
- **Combina** – Si combinano gli output precedenti ottenuti dalle chiamate ricorsive.


```
QUICKSORT( $A, p, r$ )
1:  $pivot = \text{FINDPIVOT}(A, p, r)$ 
2: se  $pivot \neq \text{null}$  allora
3:    $q = \text{PARTITION}(A, p, r, pivot)$ 
4:   QUICKSORT( $A, p, q$ )
5:   QUICKSORT( $A, q + 1, r$ )
6: fine-condizione

PARTITION( $A, p, r, pivot$ )
1:  $i = p, j = r$ 
2: ripeti
3:   fintanto che  $a_j \geq pivot$  ripeti
4:      $j = j - 1$ 
5:   fine-ciclo
6:   fintanto che  $a_i < pivot$  ripeti
7:      $i = i + 1$ 
8:   fine-ciclo
9:   se  $i < j$  allora
10:    scambia  $a_i$  e  $a_j$ 
11:   fine-condizione
12: fino a quando  $i < j$ 
13: restituisci  $j$ 

FINDPIVOT( $A, p, r$ )
1: per  $k = p + 1, \dots, r$  ripeti
2:   se  $a_k > a_p$  allora
3:     restituisci  $a_k$ 
4:   altrimenti se  $a_k < a_p$  allora
5:     restituisci  $a_p$ 
6:   fine-condizione
7: fine-ciclo
8: restituisci  $\text{null}$ 
```

Algoritmo 4: Quick sort





# Quick sort: implementazione in Python

```
def quicksort(numbers, p, r):  
    """indichiamo con:  
        p - l'indice della sottolista di sinistra  
        r - l'indice della sottolista di destra  
    """  
    if p < r:  
        q = partition(numbers, p, r)  
        quicksort(numbers, p, q - 1)  
        quicksort(numbers, q + 1, r)  
def partition(numbers, p, r):  
    pivot = numbers[r] #definiamo il pivot assegnando l'ultimo elemento  
    i = p - 1 #inizializziamo l'indice dell'array di sinistra  
    for j in range(p, r):  
        if numbers[j] <= pivot:  
            i += 1  
            numbers[i], numbers[j] = numbers[j], numbers[i] #scambio i valori  
    numbers[i + 1], numbers[r] = numbers[r], numbers[i + 1] #scambio i valori  
    return i + 1  
number_list = [10,5,12,1,9,7]  
quicksort(number_list,0,len(number_list)-1)  
print(number_list)
```

# **Analisi degli Algoritmi e Ordini di Grandezza (Materiale Supplementare)**

# Analisi degli Algoritmi

*Confrontare gli algoritmi principalmente in termini di tempo di esecuzione (**running time**), ma anche in termini di fattori quali spazio di memoria, impegno del programmatore, semplicità della soluzione, ecc.*

- Il tempo di esecuzione (**running time**) è espresso come funzione della dimensione dell'**input** (i.e.,  $f(n)$ ).
- Per confrontare due algoritmi con running time  $f(n)$  e  $g(n)$ , serve una misura (anche approssimata) di quanto velocemente queste funzioni crescono.
- Questa analisi è indipendente dal tempo macchina, dal linguaggio di programmazione, ecc.

# Esempio: algoritmi di Searching

- Problema del searching su una lista ordinata.
  - Given a list  $L$  of  $n$  elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
  - And given a particular element  $x$ ,
  - Determine whether  $x$  appears in the list, and if so, return its index (position) in the list.

# Algoritmo di Searching #1 – Ricerca lineare

**def** *linearSearch*(X):

*i* **←** 0

**while** (*i* < len(X) **and** *x* != X[*i*]):

*i* **←** *i* + 1

**if** *i* < *n*:

*location* **←** *i*

**else:**

*location* **←** -1

**return** *location*

**procedure** *linear search*

(*x*: integer,  $a_1, a_2, \dots, a_n$ : distinct integers)

*i* **←** 1

**while** ( $i \leq n \wedge x \neq a_i$ )

*i* **←** *i* + 1

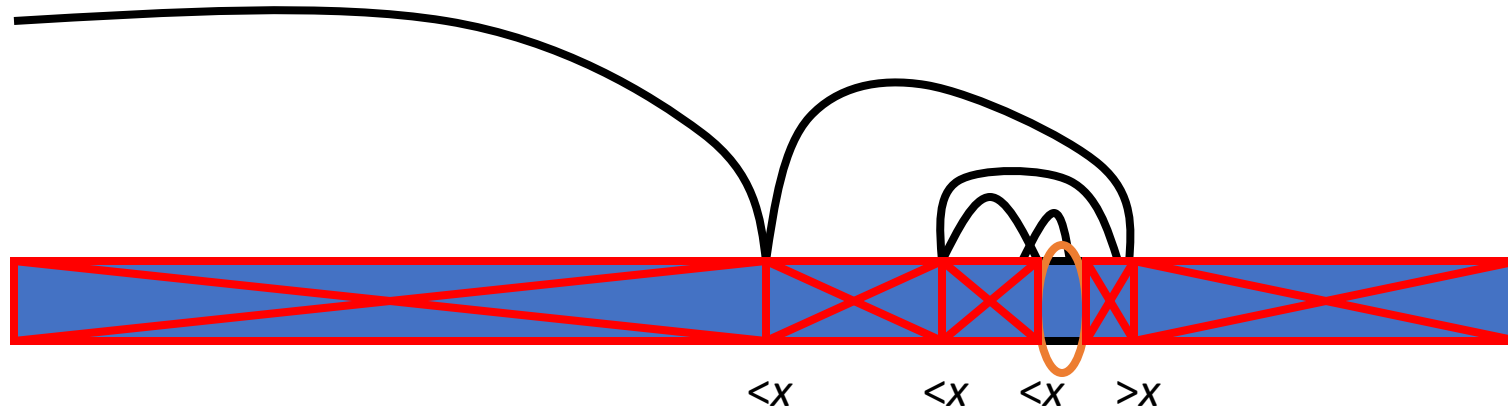
**if**  $i \leq n$  **then** *location* **←** *i*

**else** *location* **←** 0

**return** *location* {index or 0 if not found}

# Algoritmo di Searching #2 – Ricerca binaria

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



# Algoritmo di Searching #2 – Ricerca binaria

**procedure** *binary search*

( $x$ :integer,  $a_1, a_2, \dots, a_n$ : distinct integers)

$i \leftarrow 1$  {left endpoint of search interval}

$j \leftarrow n$  {right endpoint of search interval}

**while**  $i < j$  **begin** {while interval has  $>1$  item}

$m \leftarrow \lfloor (i+j)/2 \rfloor$  {midpoint}

**if**  $x > a_m$  **then**  $i \leftarrow m+1$  **else**  $j \leftarrow m$

**end**

**if**  $x = a_i$  **then**  $location \leftarrow i$  **else**  $location \leftarrow 0$

**return**  $location$

# É più efficiente la ricerca binaria?

- **Numero di iterazioni:**

- Per una lista di  $n$  elementi (lunghezza), la ricerca binaria può eseguire al Massimo  $\log_2 n$  iterazioni, mentre la ricerca lineare può compiere fino a  $n$  iterazioni

- **Numero di calcoli per iterazione:**

- La ricerca binaria esegue più calcoli per iterazione rispetto alla ricerca lineare.

- **Complessivamente:**

- Se il numero di elementi è piccolo ( $<20$ ), la ricerca lineare è più veloce
- Se il numero di elementi è grande, la ricerca binaria è più veloce.



# Come si analizzano gli algoritmi?

(1) Confrontare i tempi di esecuzione?

dipende dal singolo computer!!



(2) Contare il numero di istruzioni eseguite?

dipende dal linguaggio di programmazione



(3) Usiamo il running time come funzione della  
dimensione dell'input

indipendente dal tempo macchina, dal linguaggio di  
programmazione, ecc.



# Esempio

## *Algoritmo 1*

```
arr[0] = 0;  
arr[1] = 0;  
arr[2] = 0;  
...  
arr[N-1] = 0;
```

## *Algoritmo 2*

```
for(i=0; i<N; i++)  
    arr[i] = 0;
```

# Calcolo del running time

- Associare un “costo” a ciascuna istruzione e calcolare il “costo totale” sommando il costo di tutte le istruzioni che vengono eseguite.
- Si esprime il running time in termini della dimensione del problema

## **Algorithm 1**

### **Cost**

arr[0] = 0;       $c1$

arr[1] = 0;       $c1$

arr[2] = 0;       $c1$

...

arr[N-1] = 0;       $c1$

---

$$c1 + c1 + \dots + c1 = c1 \times N$$

## **Algorithm 2**

### **Cost**

for(i=0; i<N; i++)       $c2$

arr[i] = 0;       $c1$

---

$$(N+1) \times c2 + N \times c1 =$$
$$(c2 + c1) \times N + c2$$

# Comparing Functions Using - Rate of Growth

- Consider the example of buying *elephants* and *goldfish*:

**Cost:** cost\_of\_elephants + cost\_of\_goldfish

**Cost**  $\sim$  cost\_of\_elephants (**approximation**)

- The low order terms in a function are relatively insignificant for **large**  $n$

$$n^4 + 100n^2 + 10n + 50 \sim n^4$$

*i.e.*,  $n^4 + 100n^2 + 10n + 50$  and  $n^4$  have the same rate of growth

# Rate of Growth $\equiv$ Analisi Asintotica

- Using *rate of growth* as a measure to compare different functions implies comparing them **asymptotically**.
- If  $f(x)$  is *faster growing* than  $g(x)$ , then  $f(x)$  always eventually becomes larger than  $g(x)$  **in the limit** (for large enough values of  $x$ ).

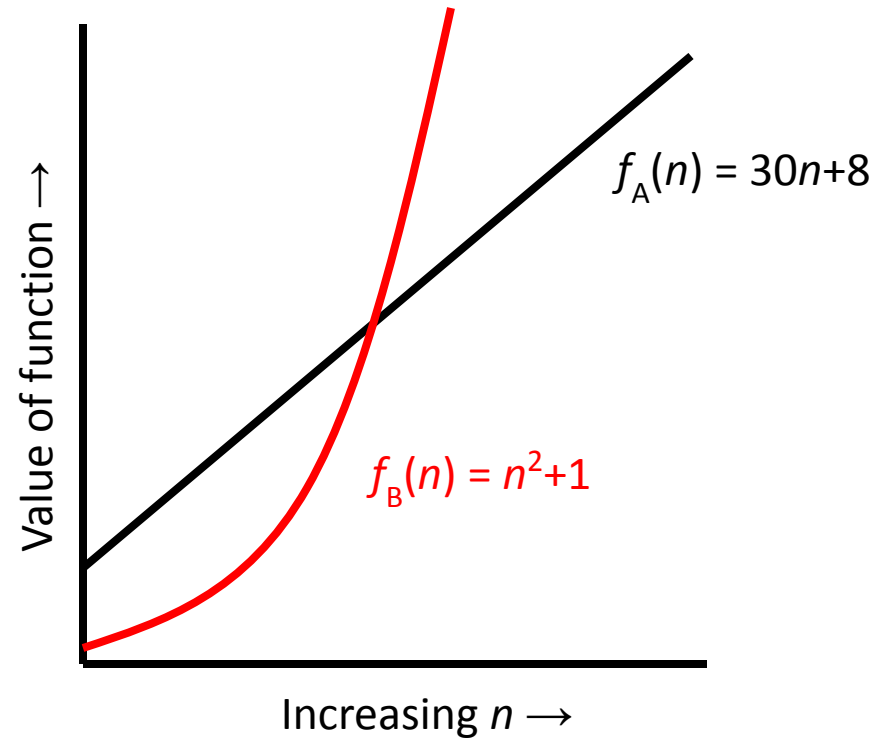
# Example

- Suppose you are designing a web site to process user data (*e.g.*, financial records).
- Suppose program A takes  $f_A(n) = 30n+8$  microseconds to process any  $n$  records, while program B takes  $f_B(n) = n^2+1$  microseconds to process the  $n$  records.
- Which program would you choose, knowing you'll want to support millions of users?

A

# Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



# Notazione O-grande (Big-O)

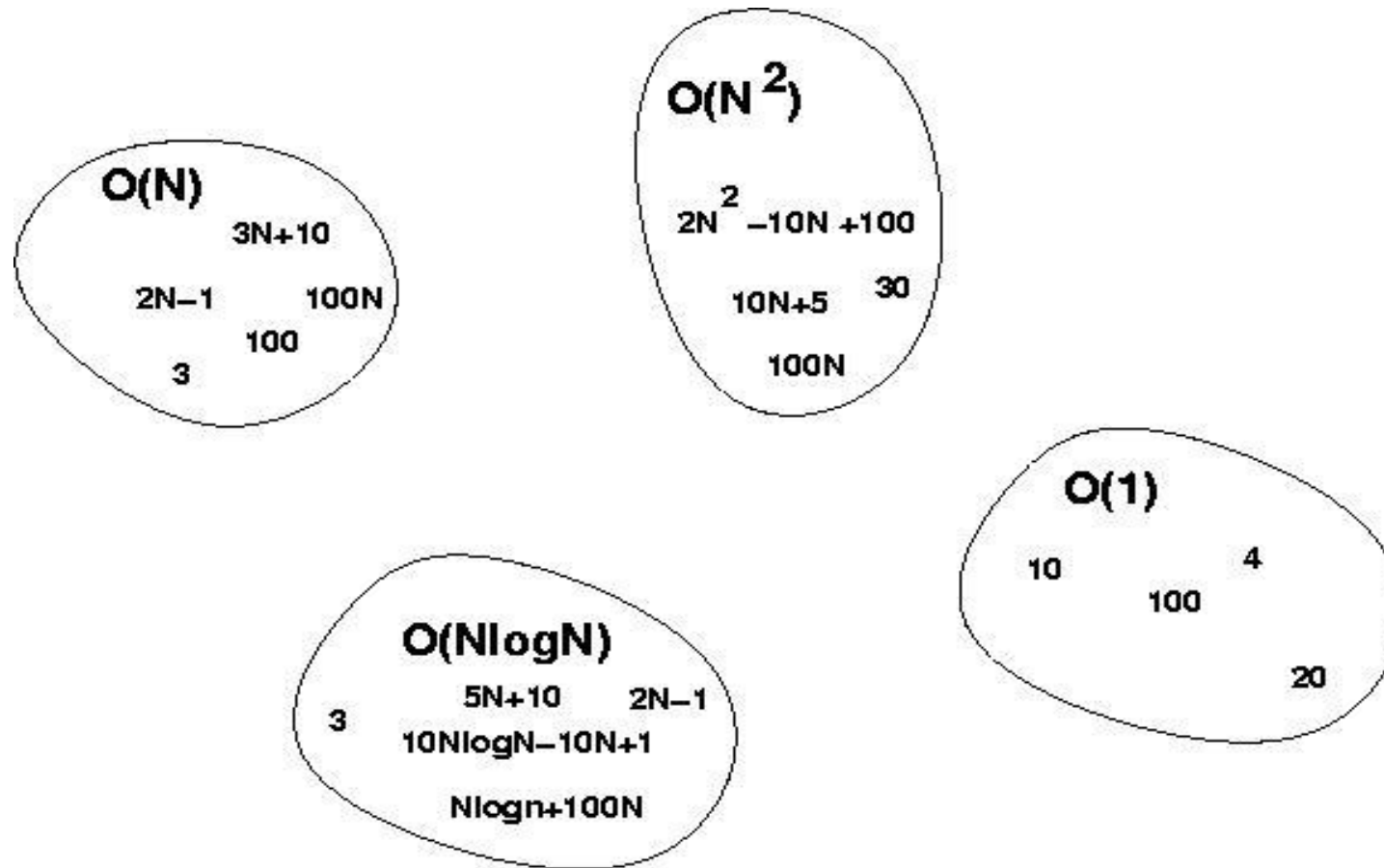
- Diciamo che  $f_A(n) = 30n+8$  è di ordine  $n$ , oppure  $O(n)$ . It is, **al massimo**, approssimativamente *proporzionale a  $n$* .
- $f_B(n) = n^2+1$  è di ordine  $n^2$ , o  $O(n^2)$ . It is, **at most**, roughly proportional to  $n^2$ .
- In generale, un algoritmo  $O(n^2)$  sarà più lento di un algoritmo  $O(n)$ .
- **Warning:** Una funzione  $O(n^2)$  crescerà asintoticamente più velocemente di una funzione  $O(n)$ .



# Ulteriori esempi

- $n^4 + 100n^2 + 10n + 50$  è di ordine  $n^4$  oppure  $O(n^4)$
- $10n^3 + 2n^2$  è di ordine  $O(n^3)$
- $n^3 - n^2$  è di ordine  $O(n^3)$
- 10 è di ordine  $O(1)$ ,
- 1273 è di ordine  $O(1)$

# Big-O Visualization



# Computing running time

## *Algorithm 1*

	<b>Cost</b>
arr[0] = 0;	c1
arr[1] = 0;	c1
arr[2] = 0;	c1
...	
arr[N-1] = 0;	c1

-----  
 $c1 + c1 + \dots + c1 = c1 \times N$

## *Algorithm 2*

	<b>Cost</b>
for(i=0; i<N; i++)	c2
arr[i] = 0;	c1

-----  
 $(N+1) \times c2 + N \times c1 =$   
 $(c2 + c1) \times N + c2$

*$O(n)$*

# Computing running time (cont.)

*Cost*

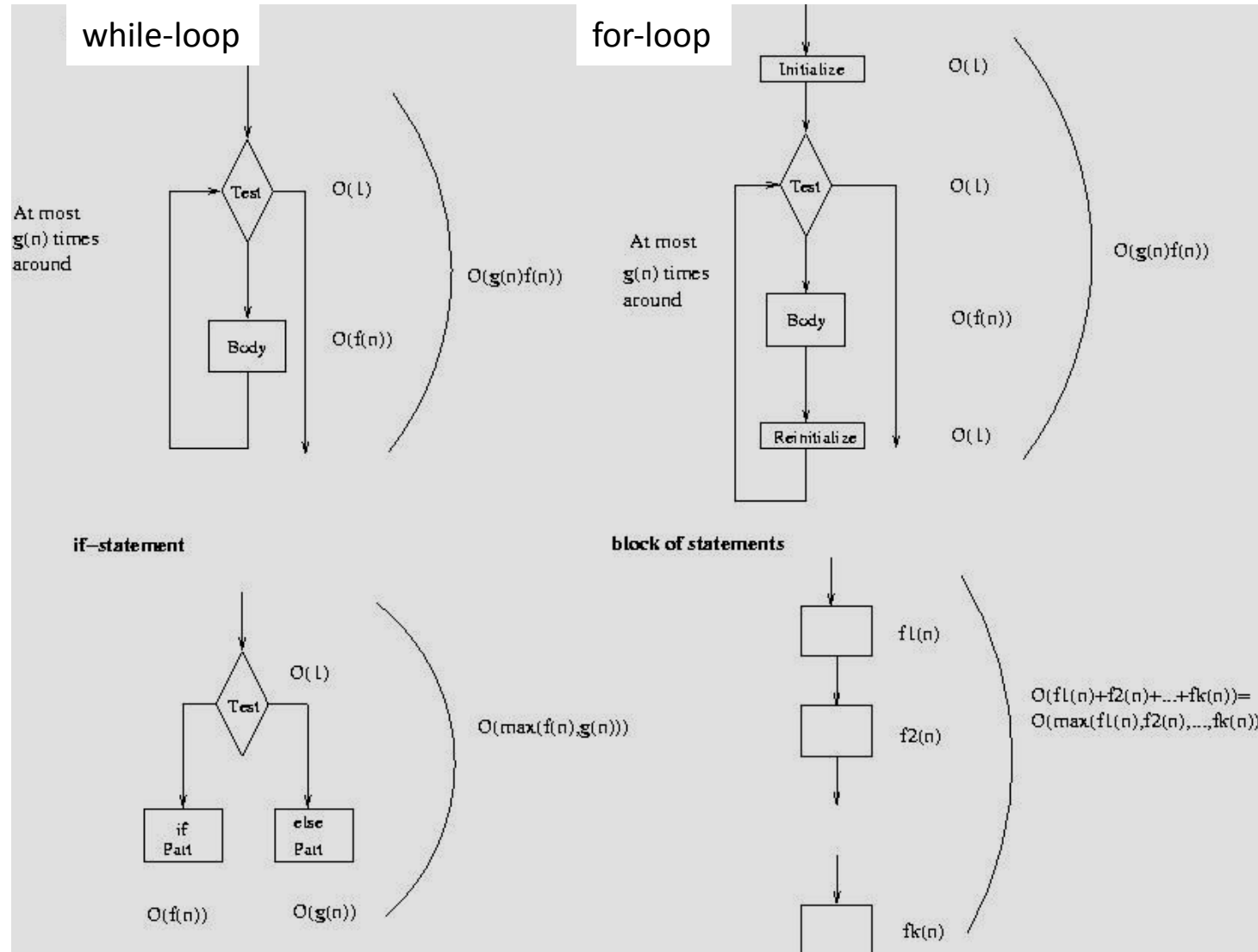
sum = 0;	c1
for(i=0; i<N; i++)	c2
for(j=0; j<N; j++)	c2
sum += arr[i][j];	c3

-----

$$c1 + c2 \times (N+1) + c2 \times N \times (N+1) + c3 \times N \times N$$

$$O(n^2)$$

# Running time of various statements



# Examples

```
i = 0;
```

```
while (i < N) {
```

```
    X = X + Y;           // O(1)
```

```
    result = mystery(X); // O(N), just an example...
```

```
    i++;                // O(1)
```

```
}
```

- The body of the while loop:  $O(N)$
- Loop is executed: N times

$$N \times O(N) = O(N^2)$$

# Examples (cont.'d)

if (i<j)

for ( i=0; i<N; i++ )

X = X+i;

} O(N)

else

X=0;

} O(1)

Max ( O(N), O(1) ) = O (N)

# Notazione asintotica

- Notazione  $O$ : asintoticamente “minore di”:
  - $f(n)=O(g(n))$  implica:  $f(n) \leq g(n)$
- Notazione  $\Omega$ : asintoticamente “maggiore di”:
  - $f(n)=\Omega(g(n))$  implica:  $f(n) \geq g(n)$
- Notazione  $\Theta$ : asintoticamente “uguale”:
  - $f(n)=\Theta(g(n))$  implica:  $f(n) = g(n)$



# Definizione: $O(g)$ , al massimo di ordine $g$

Siano  $f$  e  $g$  due funzioni  $\mathbf{R} \rightarrow \mathbf{R}$ .

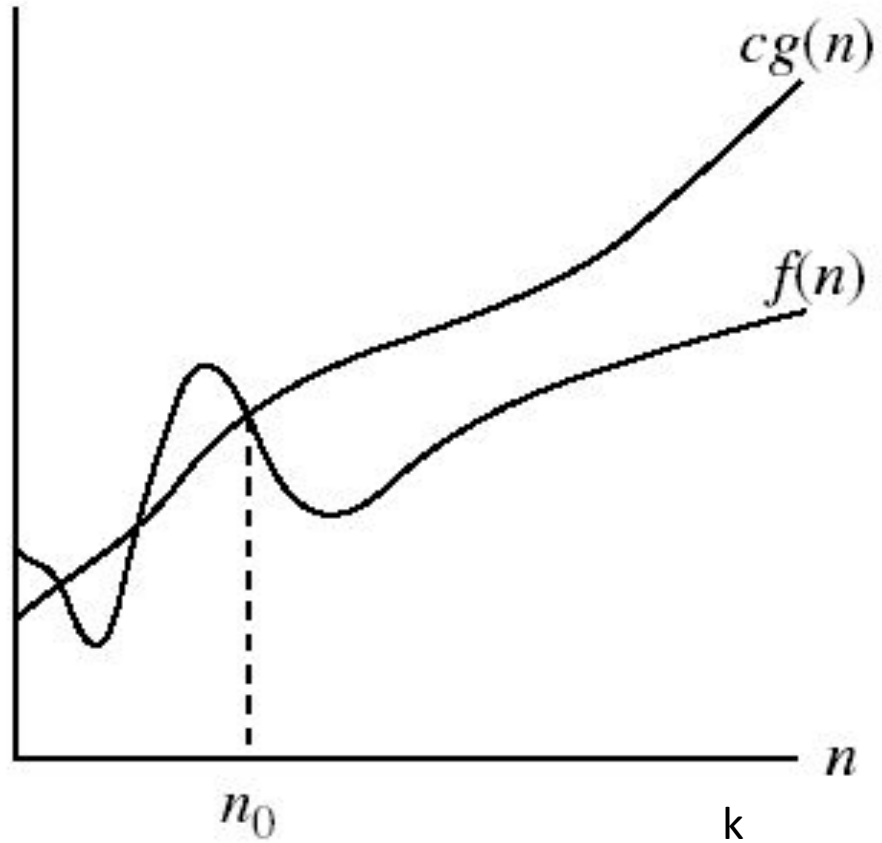
- “ $f$  è al massimo di ordine  $g$ ”, se:

$$\exists c, k: f(x) \leq cg(x), \forall x > k$$

“Superato un punto  $k$ , la funzione  $f$  è almeno  $c$  volte  $g$  ( $c$  è una costante).”

- “ $f$  è al massimo di ordine  $g$ ”, o “ $f$  è  $O(g)$ ”, o “ $f=O(g)$ ” significano che  $f \in O(g)$ .
- Spesso l’espressione “al massimo” è omessa e si dice semplicemente “è di ordine  $g$ ” o è “ $O(g)$ ”.

# Big-O Visualization



$g(n)$  is an *asymptotic upper bound* for  $f(n)$ .

# Complessità algoritmica

- The *algorithmic complexity* of a computation is some measure of how *difficult* it is to perform the computation.
- Measures some aspect of *cost* of computation (in a general sense of cost).

# Problem Complexity

- The complexity of a computational *problem* or *task* is the complexity of the algorithm with the lowest order of growth of complexity for solving that problem or performing that task.
- *E.g.* the problem of searching an ordered list has *at most logarithmic* time complexity. (Complexity is  $O(\log n)$ .)

# Tractable vs. Intractable Problems

- A problem or algorithm with at most polynomial time complexity is considered *tractable* (or *feasible*).  $\mathbf{P}$  is the set of all tractable problems.
- A problem or algorithm that has more than polynomial complexity is considered *intractable* (or *infeasible*).
- **Note**
  - $n^{1,000,000}$  is *technically* tractable, but really impossible.
  - $n^{\log \log \log n}$  is *technically* intractable, but easy.
  - Such cases are rare though.

# Dealing with Intractable Problems

- Many times, a problem is intractable for a small number of input cases that do not arise in practice very often.
  - Average running time is a better measure of problem complexity in this case.
  - Find approximate solutions instead of exact solutions.

# Unsolvable problems

- It can be shown that there exist problems that no algorithm exists for solving them.
- Turing discovered in the 1930's that there are problems unsolvable by *any* algorithm.
- Example: the *halting problem* (see page 176)
  - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an “*infinite loop*?”

# NP and NP-complete

- **NP** is the set of problems for which there exists a tractable algorithm for *checking solutions* to see if they are correct.
- **NP-complete** is a class of problems with the property that if any one of them can be solved by a polynomial worst-case algorithm, then all of them can be solved by polynomial worst-case algorithms.
  - *Satisfiability problem*: find an assignment of truth values that makes a compound proposition true.



# P vs. NP

- We know  $\mathbf{P} \subseteq \mathbf{NP}$ , but the most famous unproven conjecture in computer science is that this inclusion is *proper* (i.e., that  $\mathbf{P} \subset \mathbf{NP}$  rather than  $\mathbf{P} = \mathbf{NP}$ ).
- It is generally accepted that no **NP-complete** problem can be solved in polynomial time.
- Whoever first proves it will be famous!

# Questions

- Find the best big-O notation to describe the complexity of following algorithms:
  - A binary search of  $n$  elements
  - A linear search to find the smallest number in a list of  $n$  numbers
  - An algorithm that lists all ways to put the numbers  $1, 2, 3, \dots, n$  in a row.

# Questions (cont'd)

- An algorithm that prints all bit strings of length  $n$
- An iterative algorithm to compute  $n!$
- An algorithm that finds the average of  $n$  numbers by adding them and dividing by  $n$
- An algorithm that prints all subsets of size three of the set  $\{1, 2, 3, \dots, n\}$
- The best case analysis of a linear search of a list of size  $n$ .

# Questions (cont'd)

- The worst-case analysis of a linear search of a list of size  $n$
- The number of print statements in the following

```
while n>1 {  
    print "hello"  
    n=n/2  
}
```

# Questions (cont'd)

- The number of print statements in the following  
    **for (i=1, i ≤ n; i++)**  
        **for (j=1, j ≤ n; j++)**  
            **print "hello"**
- The number of print statements in the following  
    **for (i=1, i ≤ n; i++)**  
        **for (j=1, j ≤ i; j++)**  
            **print "hello"**

# Points about the definition

- Note that  $f$  is  $O(g)$  as long as **any** values of  $c$  and  $k$  exist that satisfy the definition.
- But: The particular  $c, k$ , values that make the statement true are not unique: **Any larger value of  $c$  and/or  $k$  will also work.**
- You are **not** required to find the smallest  $c$  and  $k$  values that work. (Indeed, in some cases, there may be no smallest values!)

However, you should **prove** that the values you choose do work.

# “Big-O” Proof Examples

- Show that  $30n+8$  is  $O(n)$ .
  - Show  $\exists c, k: 30n+8 \leq cn, \forall n > k$ .
    - Let  $c=31, k=8$ . Assume  $n > k=8$ . Then  $cn = 31n = 30n + n > 30n+8$ , so  $30n+8 < cn$ .
- Show that  $n^2+1$  is  $O(n^2)$ .
  - Show  $\exists c, k: n^2+1 \leq cn^2, \forall n > k$ .
    - Let  $c=2, k=1$ . Assume  $n > 1$ . Then  $cn^2 = 2n^2 = n^2 + n^2 > n^2+1$ , or  $n^2+1 < cn^2$ .

# Big-O example, graphically

- Note  $30n+8$  isn't less than  $n$  *anywhere* ( $n>0$ ).
- It isn't even less than  $31n$  *everywhere*.
- But it *is* less than  $31n$  everywhere to the right of  $n=8$ .

