

# **Algoritmi e Strutture Dati**

Schifano S. Fabio `schifano@fe.infn.it`

# Algoritmi e Strutture Dati

Fabio Schifano

INFN, Palazzina B stanza 208

[schifano@fe.infn.it](mailto:schifano@fe.infn.it)

Scopo: definire procedure di calcolo **efficienti**

Prerequisiti: sapere programmare in C, ovvero aver sostenuto l'esame di programmazione

Libri di riferimento:

- ▷ *Algoritmi e Strutture Dati*, Alan Bertossi, UTET
- ▷ *Introduzione agli algoritmi e strutture dati*, Cormen, Leiserson, Rivest, Stein, McGraw-Hill
- ▷ *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein, MIT-press
- ▷ *The C Programming Language*, Kernighan Ritchie, Prentice Hall
- ▷ *La Struttura degli Algoritmi*, Fabrizio Luccio, Bollati Boringhieri
- ▷ C e Java laboratorio di programmazione, G. Fiorentino, M. R. Lagná, F. Romani, F. Turini, McGraw-Hill
- ▷ *Introduzione al Pascal*, Welsh Elder

<http://www.fe.infn.it/~schifano/asd.html>

<http://df.unife.it/u/schifano>

## Lezioni:

- ▷ Lun 14:00 - 16:00 Aula Informatica
- ▷ Mar 14:00 - 16:00 Aula Informatica
- ▷ mer 14:00 - 16:00 Aula Informatica

## Esercitazioni:

- ▷ realizzazione di strutture dati ed algoritmi
- ▷ si fanno in C usando Linux (non e' previsto l'uso di **Windows**)

## Modalità di Esame:

1. 2 parziali scritti + orale
2. 1 esame scritto + orale

## Esame scritto:

- ▷ definizione di strutture dati ed algoritmi
- ▷ implementazione (realizzazione) di strutture dati ed algoritmi in C
- ▷ analisi di complessità di algoritmi.

# Algoritmi

Algorithm:

A detailed sequence of actions to perform to accomplish some task.

Named after an Iranian mathematician, Al-Khawarizmi.

Technically, an algorithm must reach a result after a **finite number of steps**.

Il termine **Algoritmo** indica la sequenza di azioni che un esecutore automatico deve compiere per giungere alla soluzione di un qualsiasi problema computazionale.

Un algoritmo é uno strumento per risolvere un **problema computazionale** specificato dall'**enunciato** del problema, ovvero dalla relazione che deve intercorrere tra i dati di **input** e quelli di **output**.

**Definizione:**

Un Algoritmo é una procedura computazionale **ben definita** che prende valori in **input** e produce valori in **output** tale che tra i due insiemi sia rispettata la relazione espressa dall'enunciato

Un algoritmo A che risolve un problema computazionale P é **corretto** se per ogni istanza I del problema l'algoritmo **termina** con output O tale che tra I ed O vale la relazione specificata dall'ununciato di P.

# Algoritmi

Esempio:

**Enunciato:** ordinare una sequenza di numeri in modo non decrescente

**Input:** una sequenza di numeri  $\langle a_0, a_1, \dots, a_n \rangle$

**Output:** una sequenza di numeri  $\langle a'_0, a'_1, \dots, a'_n \rangle$  tale che vale la seguente proprietà

$$\forall i, j \mid i < j, a'_i \leq a'_j$$

**Algoritmo:**

1.  $\forall 0 \leq i \leq n,$
2.  $\forall 0 \leq j \leq n,$
3. se  $i \neq j$  e  $a_i < a_j$ , **scambia** la posizione  $a_i$  con quella di  $a_j$

L'algoritmo sopra esegue esattamente  $n^2$  operazioni di confronto per giungere alla soluzione, ove  $n$  é il numero dei dati che si vuole ordinare.

Una **istanza del problema** é definita da un insieme di dati di input, esempio:  $\langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$

Una **soluzione di una istanza** é definita da un insieme di dati di output, esempio:

$\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$

## Descrizione di un Algoritmo

Un algoritmo per un calcolatore elettronico viene descritto in termini di istruzioni elementari utilizzando un opportuno linguaggio di programmazione:

1. linguaggio macchina
2. linguaggio assembly
3. linguaggio ad alto livello: Pascal, C, Java, . . .

Molto spesso per studiare il comportamento di un algoritmo non è necessario avere una descrizione dettagliata dell'algoritmo stesso.

- ▷ il libro usa il Pascal (**NON** è un problema)
- ▷ a lezione utilizzeremo un linguaggio **astratto** ad alto livello o pseudo codice:
  - ▷ per evitare dettagli inutili
  - ▷ per sottolineare il fatto che un algoritmo é **completamente indipendente** dal linguaggio
- ▷ durante le esercitazioni, per implementare gli algoritmi e le strutture dati viste a lezione, useremo il linguaggio **C**

## Esempio

**Enunciato:** calcolare il minimo di un insieme di numeri interi, maggiori o uguali a zero

**Input:** un **vettore** di numeri interi  $\langle a_0, a_1, a_2, \dots, a_n \rangle$

**Output:** un numero intero  $m$  tale che vale la seguente relazione

$$\forall 1 \leq i \leq n, \quad m \leq a_i$$

**Algoritmo:**

```
function min ( vector v )  
  begin  
    min = v[0]  
    foreach i ( 1 .. length v )  
      if ( v[i] < min ) then  
        min = v[i]  
      end  
    end  
  end  
end
```

**istanza:**  $\langle 23, 5, 7, 9, 12, 44, 2, 55, 66 \rangle$

**soluzione:**  $m = 2$

## Esempio: versione C

**Enunciato:** calcolare il minimo di un insieme di numeri interi, maggiori o uguali a zero

**Input:** un **vettore** di numeri interi  $\langle a_0, a_1, a_2, \dots, a_n \rangle$

**Output:** un numero intero  $m$  tale che vale la seguente relazione

$$\forall 1 \leq i \leq n, \quad m \leq a_i$$

**Algoritmo:**

```
// v = vettore da ordinare l = lunghezza del vettore v
int min ( int v[], int l ) {
    int i    = 0;
    int min = v[0];

    for ( i = 1; i < l-1; i++ ) {
        if ( v[i] < min )
            min = v[i];
    }

    return min;
}
```



# I Puntatori in C

## Definizione:

- ▷ Un puntatore é una variabile che contiene un indirizzo di memoria (ad esempio di un'altra variabile)
- ▷ Una variabile può essere usata come puntatore se dichiarata come

```
tipo * nome_variabile
```

- **tipo** indica il nome del tipo a cui la variabile può puntare ( gli indirizzi di memoria sono **tipati** )
- **nome\_variabile** é il nome della variabile puntatore

## Operazione sui puntatori:

- ▷ l'operatore **&** restituisce l'indirizzo di memoria di una variabile

```
pippo_pointer = & pippo;
```

- ▷ l'operatore **\*** restituisce il valore della variabile memorizzato all'indirizzo indicato dal valore della variabile puntatore che lo segue

```
pippo_value = * pippo_pointer
```

- ▷ **NULL** é il puntatore nullo: un particolare valore che si può assegnare ad una variabile puntatore

## Allocazione statica della memoria

La memoria viene allocata staticamente a tempo di compilazione dal compilatore in base alle dichiarazioni fatte dall'utente nel programma.

```
int i;           // integer
int v[10];       // vettore di 10 interi

int * ip1 = &i;  // dichiarazione ed inizializzazione di un puntatore ad un intero
int * ip2 = v;   // dichiarazione ed inizializzazione di un puntatore ad un vettore
                // di interi

struct s {       // dichiarazione di una struct
    int a;
    float b;
};
```

## Allocazione dinamica della memoria

Per gestire la memoria durante l'esecuzione del programma é possibile utilizzare alcune funzioni di sistema che permettono di **allocare** e **deallocare** la memoria

```
int * ip; // integer pointer
int * ap; // integer array pointer

ip      = malloc ( sizeof(int) ); // alloca memoria per contenere un integer
*ip     = 1719;

ap      = malloc ( 100 * sizeof(int) ); // alloca memoria per contenere 100 integer
ap[0]   = 2720;
ap[99]  = 1313;
ap[100] = 7777777; // ← che succede ? in generale SEGMENTATION FAULT

free(ip); // dealloca la memoria
free(ap); // dealloca la memoria
```

### ATTENZIONE:

- ▷ la gestione della memoria é a carico del programmatore (no **garbage collection**)
- ▷ l'uso di puntatori rende i programmi incomprensibili
- ▷ l'operazione **free** se non usata correttamente può causare la perdita di dati

## Tipici errori

### ▷ inizializzazione assente:

```
int main () {  
    int i, *pi;  
  
    i = 1;  
    *pi = i; // <— ERRORE: *pi e' dichiarato ma NON inizializzato (eg: malloc)  
}
```

### ▷ inizializzazione errata:

```
int main () {  
    int i, *pi;  
  
    i = 1;  
  
    pi = i; // pi viene inizializzato con il valore della variabile i !!!!  
  
    printf ("%d\n", *pi); // SEGMENTATION FAULT: pi punta ad un'area di memoria  
                        // non appartenente al processo  
}
```

## Passaggio dei parametri tramite puntatori

Il passaggio dei parametri tramite puntatori può causare la **NON** volontaria modifica di variabili visibili in altre parti del programma.

Esempio:

```
void aggiusta ( int * p ) {  
    *p = 2222; // ← modifica il valore di i  
}  
  
int main () {  
    int i, *pi;  
  
    i = 10;  
  
    pi = &i;           // ← inizializza pi  
  
    aggiusta(pi);  
  
    printf ("i: %d\n", i);  
}
```

## Passaggio dei Parametri

Il passaggio dei parametri può essere effettuato per **valore** o per **variabile**.

- ▷ per **valore**: le modifiche effettuate dalla procedura chiamata sui parametri attuali passati per valore **NON** sono visibili dal chiamante
- ▷ per **variabile**: le modifiche effettuate dalla procedura chiamata sui parametri attuali passati per variabile sono visibili dal chiamante

In C il passaggio dei parametri avviene sempre per **valore**:

- ▷ nel caso degli array si passa per valore l'indirizzo di memoria iniziale,
- ▷ nel caso delle **struct** si copia l'intera struttura

Il passaggio dei parametri per variabile si ottiene utilizzando i puntatori.

```
void setval ( int * pi ) { // procedure
    *pi = 100;
}
```

Le routine i cui parametri attuali sono passati per **variabile** sono dette **procedure**.

Le routine i cui parametri attuali sono passati per **valore** sono dette **funzioni**.

In C le procedure e le funzioni sono chiamate entrambe **function**.

## Ricorsione

La ricorsione é un meccanismo di chiamata di funzione per cui una funzione  $F$  può chiamare se stessa nella propria definizione.

Una definizione di funzione ricorsiva é **ben definita** se al suo interno compare una condizione detta di **chiusura** che interrompe l'esecuzione della procedura stessa.

Esempio: calcolo del fattoriale di un numero naturale

$$F(n) = \begin{cases} 1 & \text{se } n = 0 \\ n * F(n - 1) & \text{se } n > 0 \end{cases}$$

Nella precedente definizione la clausola di chiusura è definita dalla condizione  $n = 0$ , per cui la definizione della funzione  $F(n)$  non è definita in termini di se stessa.

Il calcolo della funzione fattoriale evolve secondo il seguente schema:

$$F(n) = n * F(n - 1) = n * ((n - 1) * F(n - 2)) = n * ((n - 1) * ((n - 2) * (\dots 1 * F(0) \dots)))$$

Il programma C che implementa la precedente definizione di funzione è il seguente:

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

Il comportamento del precedente programma, per  $n \leq 100$ , é equivalente al seguente programma:

```
#define N 100  
  
int ifact (int n) {  
    int stack[N], i, fact;  
  
    while ( n != 0 ) {  
        stack[i] = n;  n--;  i++;  
    }  
  
    i--;  
    fact = 1;  
  
    while ( i >= 0 ) {  
        fact *= stack[i];  i--;  
    }  
  
    return fact;  
}
```



## Calcolo del minimo di un vettore

Problema: dato un vettore  $v[0], \dots, v[n-1]$  calcolare l'elemento minimo usando una definizione di funzione ricorsiva:

$$M(v[0], \dots, v[n-1]) = \begin{cases} v[0] & \text{se } n = 1 \\ \min(v[0], M(v[1], \dots, v[n-1])) & \text{se } n > 1 \end{cases}$$

Il corrispondente programma C che implementa la definizione della funzione  $M$  è il seguente:

```
// v=vettore da ordina i=indice di inizio l=lunghezza del vettore
int M (int v[], int i, int l) {
    int m;

    if ( i == l-1 )
        m = v[i];
    else {
        m = M(v, i+1, l);
        m = (m > v[i]) ? v[i] : m; // calcolo il minimo
    }

    return m;
}
```

## Calcolo della somma degli elementi di un vettore

Problema: dato un vettore  $v[0], \dots, v[n-1]$  calcolare la somma degli elementi del vettore usando prima una definizione di funzione iterativa e poi una definizione di funzione ricorsiva.

La funzione iterativa `sumiter` è definita nel seguente modo:

$$\text{sumiter}(v[0], \dots, v[n-1]) = \sum_{i=0}^{n-1} v[i]$$

Il corrispondente programma C che la implementa è il seguente:

```
// v=vettore da ordina l=lunghezza del vettore
int sumiter (int v[], int i, int l) {
    int i, sum;

    for (i=0; i<n; i++)
        sum = sum + v[i];

    return sum;
}
```

La funzione ricorsiva `sumric` è definita nel seguente modo:

$$\text{sumric}(v[0], \dots, v[n-1]) = \begin{cases} v[0] & \text{se } n = 1 \\ v[0] + \text{sumric}(v[1], \dots, v[n-1]) & \text{se } n > 1 \end{cases}$$

- ▷ se il vettore é composto da un solo elemento allora la somma é uguale al valore di tale elemento
- ▷ altrimenti, la funzione suddivide il vettore in due porzioni, di cui una contiene un solo elemento, richiama se stessa sulla seconda porzione, e successivamente calcola la somma tra le due porzioni.

Il programma C che la implementa è il seguente:

```
// v=vettore da ordina l=lunghezza del vettore i=indice di inizio
int sumric (int v[], int i, int l) {
    int sum;

    if ( i == l-1 )
        return v[i];
    else {
        sum = sumric(v, i+1, l);
        sum = v[i] + sum;
    }

    return sum;
}
```

## Complessità Computazionale

Un **buon algoritmo** deve utilizzare in modo **efficiente** (ovvero no deve sprecare) le risorse hardware di una macchina sia in termini di **spazio** che di **tempo**

Nella definizione di procedura di calcolo é quindi importate saper determinare (in alcuni casi stimare) in modo analitico:

- ▷ il **tempo**  $T$  impiegato dall'algoritmo per terminare la computazione
- ▷ lo **spazio**  $S$  impiegato dall'algoritmo durante la computazione

in modo da poter confrontare algoritmi diversi e progettare algoritmi efficienti.

- ▷ tra spazio e tempo, generalmente il tempo è la risorsa più critica e più costosa.
- ▷ siamo interessati a valutare  $T$  nel caso **peggiore** in modo da garantire che l'algoritmo non richiederà mai, per nessuna istanza, un tempo maggiore

Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, è naturale esprimere la funzione di costo  $T(n)$  in funzione della dimensione  $n$  dei dati di ingresso.

Il costo computazionale in termini di tempo  $T(n)$  si calcola sommando il costo computazionale di ciascuna istruzione che appare nella definizione dell'algoritmo.

Consideriamo ad esempio l'algoritmo iterativo per il calcolo del minimo di un vettore:

	COSTO	VOLTE
<code>int Min ( int v[] , int n ) {</code>		
<code>int min;</code>		
<code>min = v[0];</code>	$c_0$	1
<code>for ( i = 1; i &lt; n; i++ )</code>	$c_1$	$n$
<code>if ( v[i] &lt; min )</code>	$c_2$	$n-1$
<code>min = v[i];</code>	$c_3$	$n-1$
<code>return min;</code>		
<code>}</code>		

L'istruzione `for` viene eseguita  $n$  volte poiché la condizione  $i < n$  viene verificata una volta in più rispetto al **body** del `for` prima di uscire dal ciclo.

Il costo totale di esecuzione della funzione **Min** è quindi:

$$T(n) = c_0 + n * c_1 + (n - 1) * c_2 + (n - 1) * c_3 = (c_1 + c_2 + c_3) * n + (c_0 - c_2 - c_3) = a * n + b$$

Pertanto il tempo di calcolo di `min` può essere espresso come

$$T(n) = an + b$$

con  $a$ ,  $b$  costanti.

Consideriamo adesso la definizione ricorsiva della funzione Min e calcoliamo il tempo di calcolo  $T(n)$ .

	COSTO	VOLTE
<code>// v=vettore da ordina i=indice di inizio l=lunghezza del vettore</code>		
<code>int M (int v[], int i, int l) {</code>		
<code>int m;</code>		
<code>if ( i == l-1 )</code>	$c_1$	1
<code>m = v[i];</code>	$c_2$	1
<code>else {</code>		
<code>m = M(v, i+1, l);</code>	$c_3 + T(n-1)$	1
<code>m = (m &gt; v[i]) ? v[i] : m; /* calcolo il minimo */</code>	$c_4$	1
<code>}</code>		
<code>return m;</code>		
<code>}</code>		

Quindi:

- ▷  $T(n) = c_1 + c_2 = c$  se  $n = 1$
- ▷  $T(n) = c_1 + c_3 + T(n-1) + c_4 = T(n-1) + d$  se  $n > 1$

Il tempo di calcolo deve quindi essere ricavato dalla seguente relazione detta **relazione di ricorrenza**:

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ T(n-1) + d & \text{se } n > 1 \end{cases}$$

Una tecnica per risolvere una relazione di ricorrenza consiste nel riscrivere l'espressione  $T(n)$  per  $n, n - 1, n - 2, \dots$ :

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ T(n - 1) + d & \text{se } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= \\ &= T(n - 1) + d \\ &= T(n - 2) + d + d = T(n - 2) + 2d \\ &= T(n - 3) + d + 2d = T(n - 3) + 3d \\ &\vdots \\ &= T(1) + (n - 1)d \\ &= c + (n - 1)d \\ &= dn + (c - d) \end{aligned}$$

Quindi la funzione  $T(n)$  che esprime il tempo di calcolo é una funzione **lineare**.

## Esempio

Determinare il costo computazione  $T(n)$  del seguente algoritmo:

<code>// v=vettore da ordina i=indice di inizio j=indice di fine</code>		
<code>int sumric (int v[], int i, int j) {</code>	<b>COSTO</b>	<b>VOLTE</b>
<code>  int sum, k;</code>		
 <code>  if ( i == j )</code>	 <code>c0</code>	 <code>1</code>
<code>    return v[i];</code>	<code>c1</code>	<code>1</code>
<code>  else {</code>		
<code>    k = (i+j)/2;</code>	<code>c2</code>	<code>1</code>
<code>    sum = ( sumric(v, i, k) + sumric(v, k+1, j) );</code>	<code>c3+2T(n/2)</code>	<code>1</code>
<code>  }</code>		
 <code>  return sum;</code>		
<code>}</code>		

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + d & \text{se } n > 1 \end{cases}$$

Per risolvere l'equazione di ricorrenza  $T(n)$  supponiamo per semplicità che il numero degli elementi del vettore siano una potenza di 2.



$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + d & \text{se } n > 1 \end{cases}$$

$$\begin{aligned}
 T(n) &= \\
 &= 2T(n/2) + d \\
 &= 2(2T(n/4) + d) + d = 4T(n/4) + 2d + d \\
 &= 4(2T(n/8) + d) + 2d + d = 8T(n/8) + 4d + 2d + 1d \\
 &\vdots \\
 &= 2^h T(n/2^h) + 2^{h-1}d + 2^{h-2}d + \dots 2d + d \\
 &\vdots \\
 &= 2^h T(1) + 2^{h-1}d + 2^{h-2}d + \dots 2d + d \\
 &= 2^h c + \sum_{i=0}^{h-1} 2^i d \\
 &= 2^h c + d + \sum_{i=1}^{h-1} 2^i d \\
 &= 2^h c + d + 2 \frac{1-2^{h-1}}{1-2} d \\
 &= 2^h c + d + (2^h - 2)d \\
 &= (c + d)n + d - 2
 \end{aligned}$$

**N.B.:** somma dei primi  $n$  termini di una serie geometrica di ragione  $q$ :  $\sum_n = a_1 \cdot \frac{1-q^n}{1-q}$

## Complessità Computazionale Asintotica

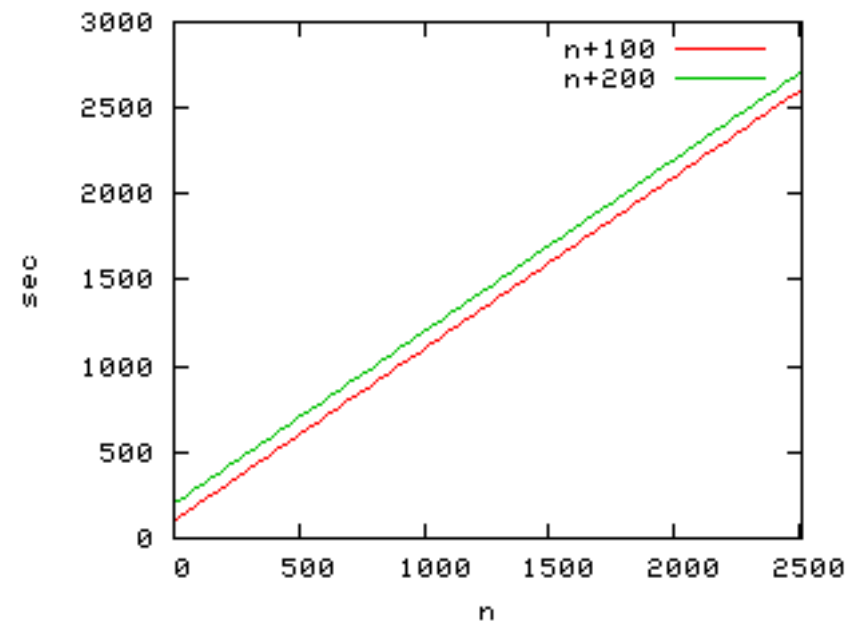
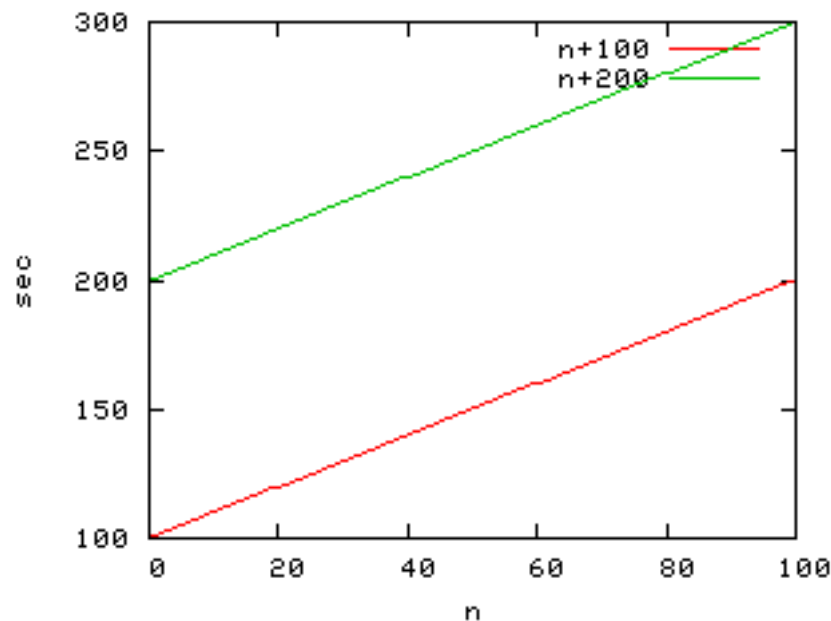
Nel valutare il tempo di calcolo di una procedura é difficile quantificare con esattezza il numero delle operazioni elementari e quindi si quantifica il tempo di calcolo a meno di costanti **moltiplicative** ed **additive**.

Esempio: le seguenti funzioni, al crescere di  $n$ ,

▷  $T_1(n) = n + 100$

▷  $T_2(n) = n + 200$

sono **asintoticamente** equivalenti, cioè sono entrambi funzioni crescenti in modo lineare.



## Complessità Computazionale: Ordine di grandezza

Per definire il comportamento asintotico di una funzione di costo definiamo i seguenti insiemi:

- ▶  $\mathcal{O}(f(n))$  (**omicron** di  $f(n)$ ) é l'insieme di tutte le funzioni  $g(n)$  tali che esistano due costanti positive  $c$  ed  $m$  per cui

$$g(n) \leq c \cdot f(n), \quad \forall n \geq m$$

- ▶  $\Omega(f(n))$  (**omega** di  $f(n)$ ) é l'insieme di tutte le funzioni  $g(n)$  tali che esistano due costanti positive  $c$  ed  $m$  per cui

$$c \cdot f(n) \leq g(n) \quad \forall n \geq m$$

- ▶  $\Theta(f(n))$  (**theta** di  $f(n)$ ) é l'insieme di tutte le funzioni  $g(n)$  tali che esistano tre costanti positive  $c$ ,  $m$  e  $d$  per cui

$$c \cdot f(n) \leq g(n) \leq d \cdot f(n), \quad \forall n \geq m$$

Data una funzione di costo  $T(n)$  diremo:

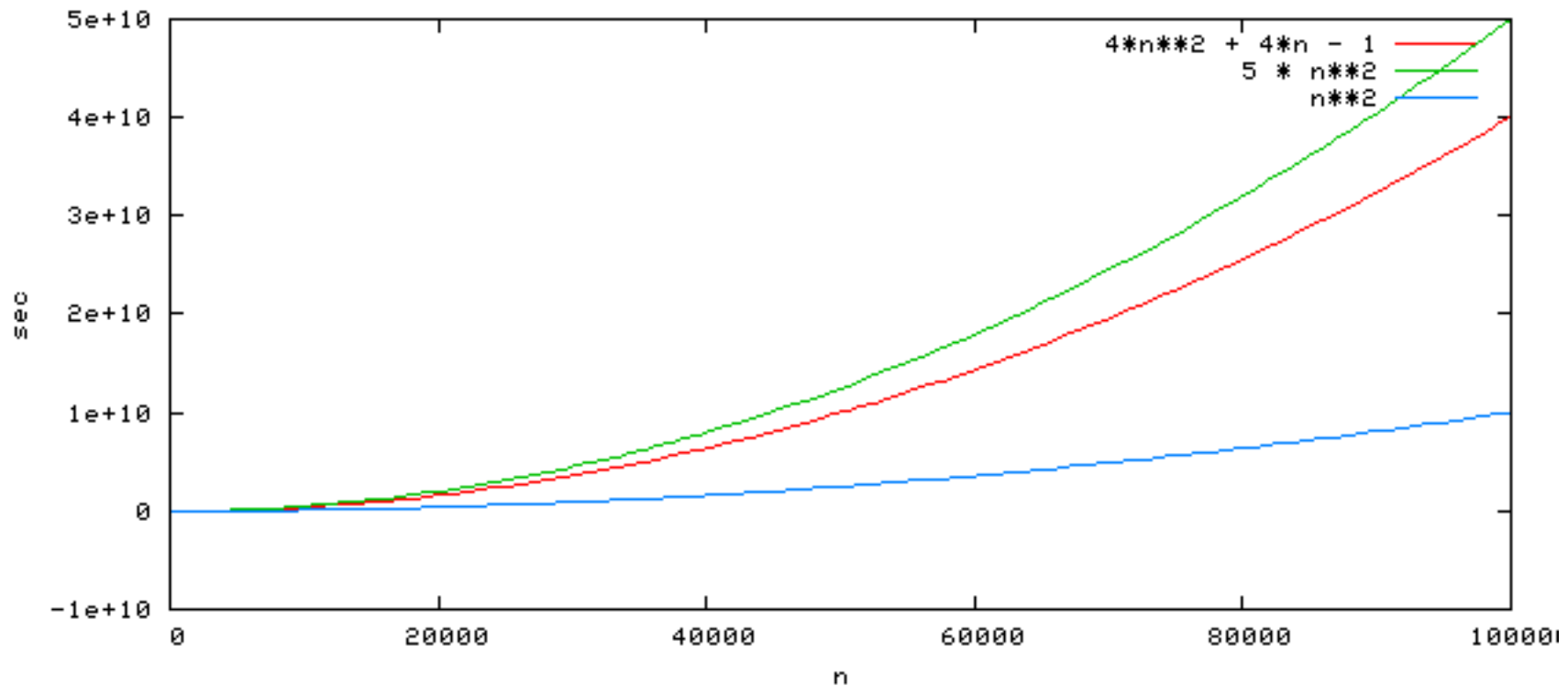
- ▶  $T(n) \in \mathcal{O}(f(n))$  per indicare che il comportamento **asintotico** di  $T(n)$  é limitato **superiormente** dalla funzione  $f(n)$
- ▶  $T(n) \in \Omega(f(n))$  per indicare che il comportamento **asintotico** di  $T(n)$  é limitato **inferiormente** dalla funzione  $f(n)$
- ▶  $T(n) \in \Theta(f(n))$  per indicare che il comportamento  $T(n)$  é limitato **superiormente** ed **inferiormente** dalla funzione  $f(n)$

## Esempio

$g(n) = 4n^2 + 4n - 1$  é  $\mathcal{O}(n^2)$  perché esistono due costanti  $c = 5, m = 4$ , tali che

$$g(n) \leq 5n^2, \quad \forall n \geq 4$$

$g(n)$  é anche  $\Omega(n^2)$  poiché esistono due costanti  $c = 1, m = 1$  tali che  $g(n) \geq n^2, \quad \forall n \geq 1$ .



Quindi l'ordine di complessità di  $g(n) \in \Theta(n^2)$ .

## Valutazione Complessità Computazionale

Nel valutare la complessità di una procedura di calcolo o di una operazione si possono usare le seguenti regole:

1. se  $T(n) = c$ ,  $c$  costante, allora  $T(n) \in \mathcal{O}(1)$ ,  $T(n) \in \Omega(1)$  e  $T(n) \in \Theta(1)$
2. per ogni costante  $c$  ed ogni funzione  $f(n)$ , se  $T(n) = c \cdot f(n)$ , allora  $T(n) \in \mathcal{O}(f(n))$ ,  $T(n) \in \Omega(f(n))$  e  $T(n) \in \Theta(f(n))$
3. se  $g(n) \in \mathcal{O}(f(n))$  e  $f(n) \in \mathcal{O}(h(n))$ , allora  $g(n) \in \mathcal{O}(h(n))$ , lo stesso vale per  $\Omega$  e  $\Theta$
4. la funzione  $f(n) + g(n)$  ha complessità  $\mathcal{O}(\max\{f(n), g(n)\})$ , lo stesso vale per  $\Omega$  e  $\Theta$
5. se  $g(n) \in \mathcal{O}(f(n))$  e  $h(n) \in \mathcal{O}(q(n))$ , allora la funzione  $g(n) \cdot h(n) \in \mathcal{O}(f(n)q(n))$ , lo stesso vale per  $\Omega$  e  $\Theta$

Esempio:

la funzione  $g(n) = 4n^2 + 4n - 1$ , per la regola 4, é di ordine

$$\mathcal{O}(\max\{(n^2), (n), (1)\}) = \mathcal{O}(n^2)$$

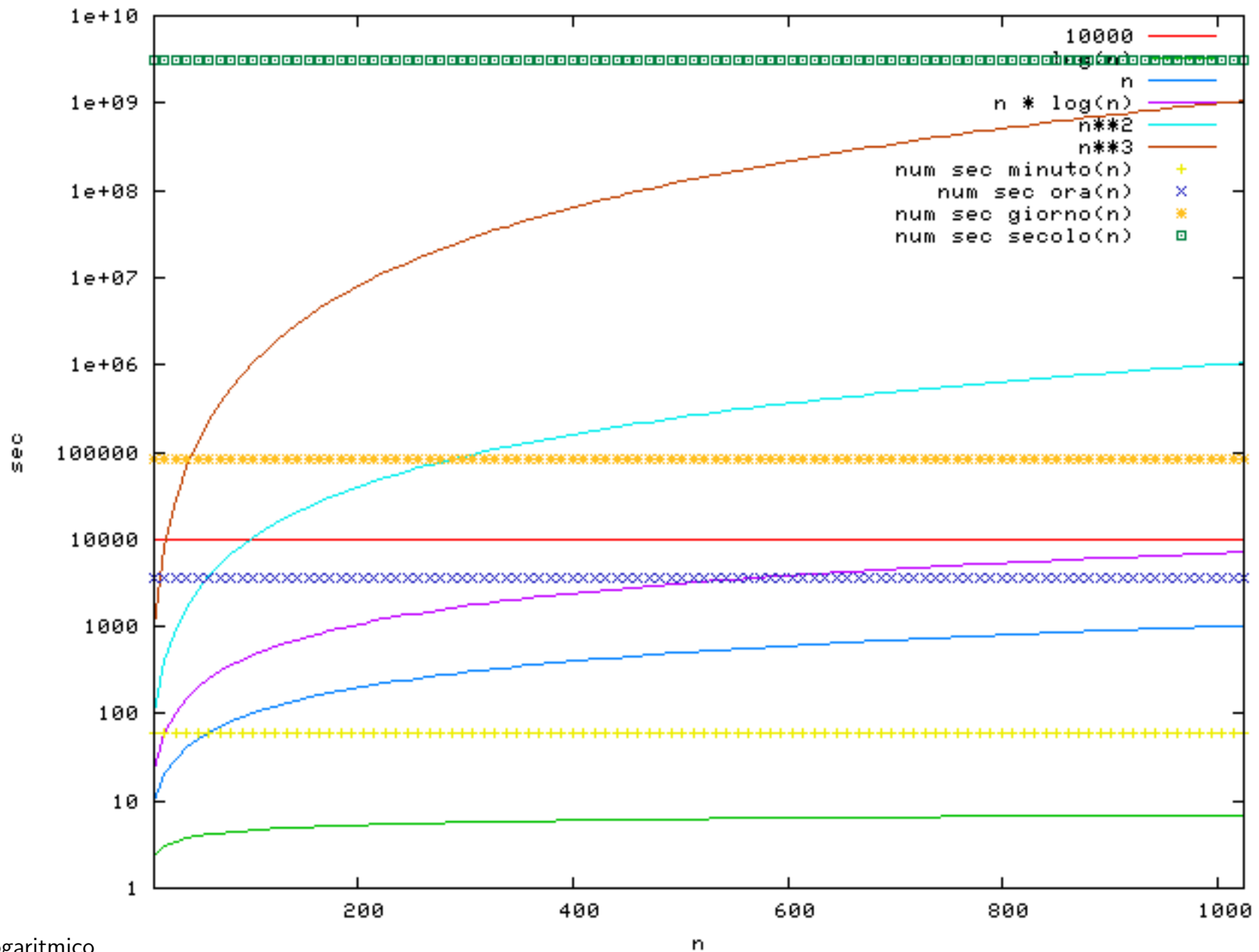
## Classificazione degli Ordini di Grandezza

I seguenti ordini di grandezza sono via via crescenti:

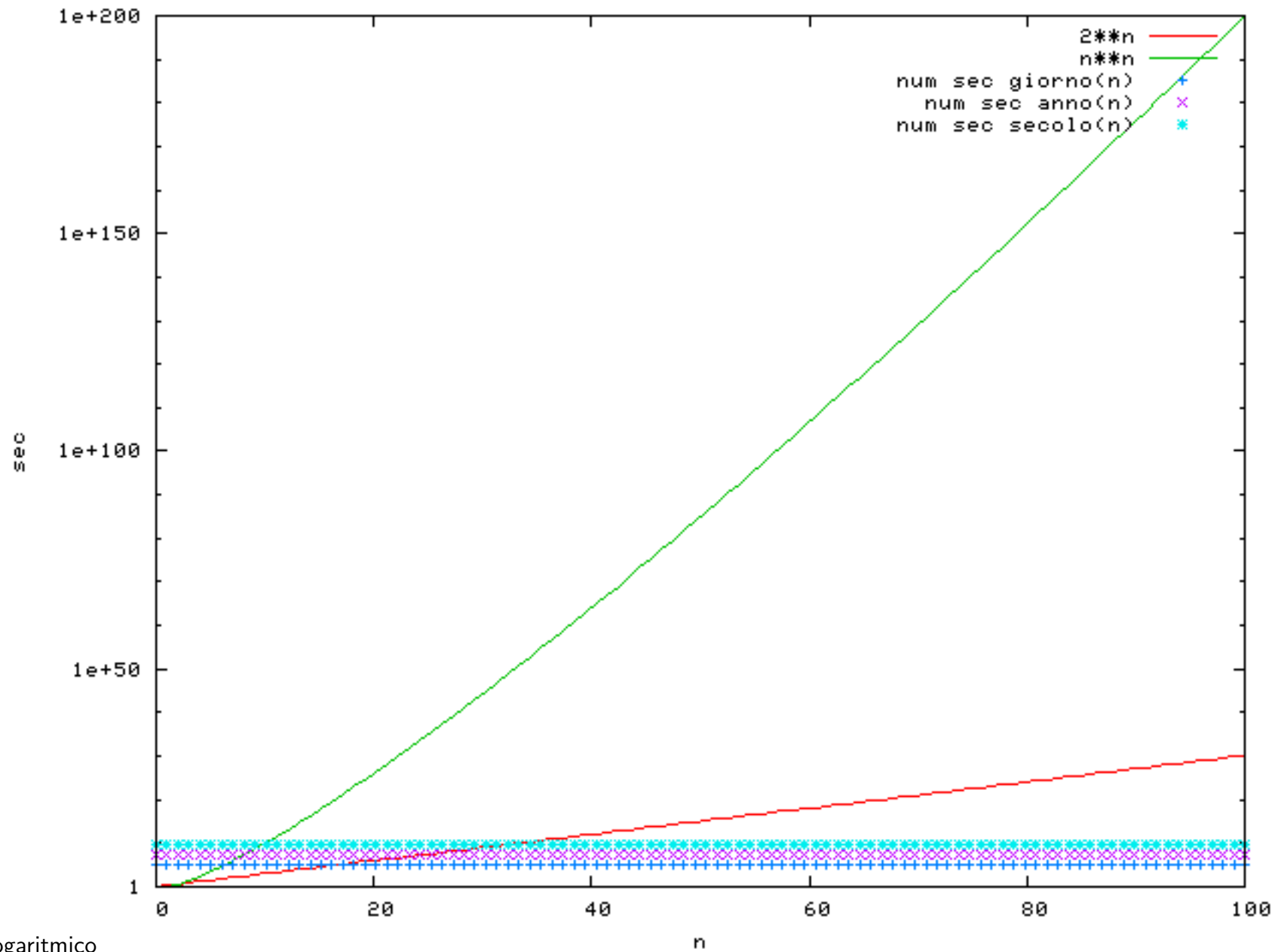
- ▷  $\mathcal{O}(1)$ , ordine costante
- ▷  $\mathcal{O}(\log n)$ , ordine logaritmo
- ▷  $\mathcal{O}(n)$ , ordine lineare
- ▷  $\mathcal{O}(n \log n)$ , ordine pseudolineare
- ▷  $\mathcal{O}(n^2)$ , ordine quadratico
- ▷  $\mathcal{O}(n^3)$ , ordine cubico
- ▷  $\mathcal{O}(2^n)$ , ordine esponenziale base 2
- ▷  $\mathcal{O}(n!)$ , ordine fattoriale
- ▷  $\mathcal{O}(n^n)$ , ordine esponenziale base  $n$

in generale

- ▷ un ordine  $\mathcal{O}(n^k)$ , con  $k$  costante positiva é detto **polinomiale**
- ▷ un ordine  $\mathcal{O}(a^n)$ , con  $a$  costante maggiore di uno é detto **esponenziale**
- ▷ ordini esponenziali o maggiori sono anche detti **superpolinomiali**



N.B.: asse  $y$  logaritmico



N.B.: asse  $y$  logaritmico



## Valutazione di Complessità

Utilizzando gli ordini di grandezza, ogni operazione elementare ha complessità  $\mathcal{O}(1)$ , e le uniche istruzioni che danno un contributo diverso sono: le istruzioni **condizionali** e le istruzioni **iterative** (**for** e **while**).

Esempio:

```
if ( < F_condizione > ) {  
    < F_then >  
} else {  
    < F_else >  
}
```

supponiamo che:

- ▷  $F\_condizione \in \mathcal{O}(f(n))$
- ▷  $F\_then \in \mathcal{O}(g(n))$
- ▷  $F\_else \in \mathcal{O}(h(n))$

Allora la complessità di esecuzione dell'intero blocco di istruzioni è

$$\mathcal{O}(\max(f(n), g(n), h(n)))$$

## Valutazione di Complessità

Esempio:

```
for (i=0; i < n; i++) {  
    < F_for >  
}  
  
for (j=0; j < n; j++) {  
    for (k=0; k < m; k++ {  
        < F_forfor >  
    }  
}
```

supponiamo che:

- ▶  $F\_for \in \mathcal{O}(f(n))$
- ▶  $F\_forfor \in \mathcal{O}(g(m))$

Allora la complessità dell'intero programma é

$$\mathcal{O}(\max(n \cdot f(n), \quad nm \cdot g(m)))$$

### Attenzione:

La definizione di complessità è data in termini asintotici ed in alcuni casi può essere fuorviante nel senso che le costanti moltiplicative ed additive possono giocare un ruolo fondamentale.

Ad esempio, si supponga di avere due algoritmo  $A$ ,  $B$ , tali che

$$\triangleright t_A(n) = f(n) \in \mathcal{O}(n^2)$$

$$\triangleright t_B(n) = g(n) \in \mathcal{O}(2^n)$$

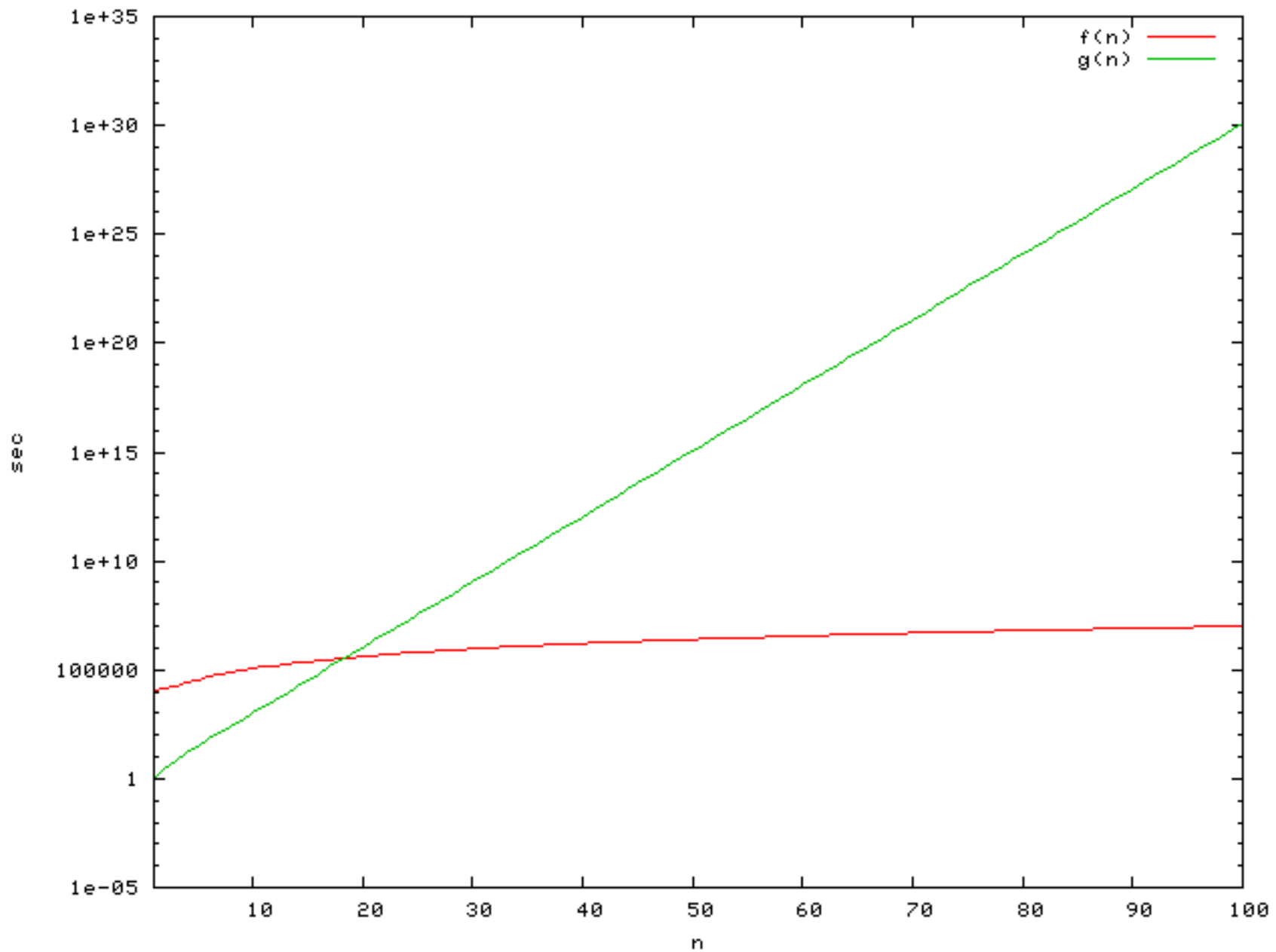
Allora, secondo le definizioni date, diremo che  $A$  è **migliore** di  $B$ .

Ma se

$$\begin{aligned} f(n) &= 10^{1000} \cdot n^2 + 10^4 \\ g(n) &= 10^{-19} \cdot 2^n - 1 \end{aligned}$$

$g(n)$  per molti valori di  $n$  assume valori decisamente inferiori a  $f(n)$ .

Ovvero se il problema richiede di essere risolto solo per alcune particolari istanze,  $n$  **piccolo**, allora é conveniente usare l'algoritmo  $B$  anziché l'algoritmo  $A$ .



## Valutazione di Complessità

Ricapitolando:

- ▷ si stima il tempo di esecuzione (lo spazio occupato) dell'algoritmo in funzione della dimensione dell'input:

$$T(n) \quad S(n)$$

- ▷ si valuta il tempo di esecuzione (o lo spazio occupato) dall'algoritmo nel caso peggiore.  
Cioé si considera l'istanza di input che fa lavora di più l'agoritmo: ad esempio, per un algoritmo di ordinamento  $\leq$  il caso peggiore é dato da un istanza di input in cui i dati siano ordinati in senso inverso

$$a_1 \geq a_2 \geq \dots \geq a_n$$

Quindi siamo interessati a valutare  $\mathcal{O}(\dots)$ .

- ▷ si stima il tempo di esecuzione in modo asintotico, cioè al crescere della dimensione dell'input

Stimare lo spazio occupato da un algoritmo é importante, ad esempio lo spazio occupato dall'algoritmo per il calcolo del minimo di un vettore può essere stimato  $\mathcal{O}(n)$ , ovvero lo spazio necessario a memorizzare l'intero vettore.

## Algoritmi efficienti ed inefficienti

In pratica, un algoritmo è **efficiente** se la sua complessità è **polinomiale**, mentre è **inefficiente** se la sua complessità è **superpolinomiale**.

Scrivere algoritmi **inefficienti** è molto facile poichè si basano sul principio di **provare tutte le possibili combinazioni e poi selezionare quella che soddisfa un certo criterio**.

Anche se usati in alcuni casi rari, hanno una scarsa utilità pratica poichè in generale richiedono un tempo computazionale **irragionevole**.

Esempio:

**Problema:** ordinare un vettore di  $n$  numeri interi in modo non decrescente

**Algoritmo:** genero tutte le possibili permutazioni di  $n$  numeri e poi verifico quale di queste è ordinata.

- ▷  $v = [9, 2, 7, 4]$
- ▷  $p[1] = [9, 2, 7, 4], p[2] = [9, 7, 2, 4], p[3] = [9, 4, 2, 7] \dots$
- ▷  $\dots p[k] = [2, 4, 7, 9] \dots$

Poiché le permutazioni da generare sono  $n!$ , e verificare se una permutazione è ordinata o meno richiede un tempo **lineare**, l'algoritmo proposto richiede  $\mathcal{O}(n \cdot n!)$  ovvero un tempo **superpolinomiale**.

Consideriamo adesso il seguente algoritmo:

```
void ordina ( int v[] , int l ) {  
    int i, p, t;  
    for ( i=0; i < l; i++ ) {  
        p = Min ( v, i, l );  
        t = v[i];  
        v[i] = v[p];  
        v[p] = t;  
    }  
}
```

L'algoritmo calcola il minimo tra gli  $n$  elementi iniziali del vettore e quindi scambia l'elemento minimo con quello in posizione iniziale; ripete lo stesso procedimento per i rimanenti  $n - 1, n - 2, \dots$  elementi.

Poichè il tempo di calcolo richiesto dall'algoritmo Min è  $\mathcal{O}(k)$ , ove  $k$  è il numero degli elementi del vettore, a meno di costanti, il tempo di calcolo  $T(n)$  dell'algoritmo ordina è la sommatoria del tempo di calcolo dell'algoritmo Min per dimensioni del vettore variabili tra  $n$  ed 1, ovvero:

$$\sum_{k=n}^1 k = \sum_{i=0}^{n-1} (n - i) = n + \sum_{i=1}^n (n - i) = n + \frac{n(n - 1)}{2} = n^2/2 + n/2$$

Quindi, il tempo computazionale  $T(n)$  è  $\mathcal{O}(n^2)$ , ovvero l'algoritmo ha complessità temporale **polinomiale** e quindi **efficiente**. **Esiste un algoritmo più efficiente per il problema dell'ordinamento ?**

## Strutture Dati: Tipo di Dato

In un linguaggio di programmazione:

- ▷ un **dato** é un particolare valore che una variabile può assumere
- ▷ un **tipo di dato** é un modello matematico che caratterizza l'insieme di valori che una variabile può assumere, e le operazioni che su di essa possono essere effettuate.

Esempio: **integer**, **boolean**, **float**, **char** sono tipi di dato primitivi di molti linguaggi di programmazione.

Una variabile  $v$  dichiarata di tipo intero **integer** assume valori numerici interi e su di essa possono essere effettuate tutte le usuali operazioni matematiche definite sui numeri interi:  $*$ ,  $/$ ,  $+$ ,  $-$ ,  $\%$ ,  $\dots$

In generale le proprietà di un **tipo di dato** dipendono esclusivamente dal tipo di dato e sono **indipendenti** dall'implementazione: in questo senso si parla di **tipo di dato astratto**.

Esempio: le proprietà dei numeri interi non devono dipendere dal modo in cui sono rappresentati (little vs big endian) in un particolare linguaggio di programmazione.



## Strutture Dati: Specifica ed Implementazione

Data Structure:

Any method of organising a collection of data to allow it to be manipulated effectively.

Examples data structures are: array, dictionary, graph, hash, heap, linked list, matrix, object, queue, ring, stack, tree, vector.

Una **struttura dati** é un particolare tipo di dato astratto caratterizzato da una organizzazione imposta agli elementi che la compongono.

Piú precisamente una struttura dati consiste di:

1. un modo sistematico di organizzare i dati
2. un insieme di operatori che permettono di manipolare gli elementi della struttura

## Classificazione delle Strutture Dati

Le strutture dati possono essere classificate in base alla disposizione dei dati, al loro numero ed al loro tipo:

- ▷ **lineari**: i dati sono disposti in sequenza e possono essere nominati come primo, secondo, terzo, . . .
- ▷ **non lineari**: i dati non sono disposti in sequenza
- ▷ **a dimensione fissa**: il numero degli elementi rimane sempre costante
- ▷ **a dimensione variabile**: il numero degli elementi può variare nel tempo
- ▷ **omogenee**: i dati sono tutti dello stesso tipo
- ▷ **non omogenee**: i dati sono di tipi diverso

Esempio: il tipo di dato **vettore** rappresenta una struttura dati **lineare, omogenea, a dimensione fissa**.

## Strutture Dati: Specifica ed Implementazione

Nel descrivere le strutture dati é importante distinguere tra la **specifica astratta** della struttura e l'**implementazione** stessa della struttura dati.

La specifica di una struttura dati si divide in due parti fondamentali:

▷ **specifica sintattica** definisce

1. i nomi dei tipi di dato utilizzati per definire la struttura
2. i nomi, i domini e codomini delle operazioni che possono essere applicate alla struttura
3. i nomi delle costanti

▷ **specifica semantica**

1. associa un insieme matematico ad ogni nome di tipo
2. associa un valore ad ogni costante
3. associa una funzione ad ogni operazione. La funzione é definita matematicamente esplicitando una coppia di condizioni definite sui domini degli operandi e dei risultati:

- ▷ **precondizione** stabilisce quando un operatore é applicabile. Se la precondizione manca l'operatore é sempre applicabile.
- ▷ **postcondizione** indica come il risultato sia vincolato agli argomenti dell'operatore.

## Strutture Dati: Specifica ed Implementazione

L'implementazione di una struttura dati riconduce la specifica ai tipi di dato e agli operatori disponibili (**primitivi**) in un linguaggio di programmazione.

Gli operatori che modificano una struttura dati sono implementati tramite **procedure** (return value **void** in C).

Gli operatori che calcolano un valore sono implementati tramite **funzioni** (return value  $\neq$  **void** in C).

Ad una data specifica di una struttura dati possono corrispondere diverse implementazioni più o meno efficienti, a seconda dei dati e delle procedure utilizzate.

L'implementazione degli operatori deve essere **trasparente all'utente**, in principio tale realizzazione può essere cambiata (ad esempio resa più efficiente) senza che l'utente abbia necessità di modificare il proprio programma.

## Struttura Dati Vettore

### Specifica Sintattica:

- ▷ **nomi dei tipi:** vettore, intero, tipoelem
- ▷ **nomi degli operatori e dei relativi domini :**
  - ▷ **CREAVETTORE:**  $() \longrightarrow \text{vettore}$
  - ▷ **LEGGIVETTORE:**  $(\text{vettore}, \text{intero}) \longrightarrow \text{tipoelem}$
  - ▷ **SCRIVIVETTORE:**  $(\text{vettore}, \text{intero}, \text{tipoelem}) \longrightarrow \text{vettore}$

### Specifica Semantica:

- ▷ **vettore:** insieme di sequenze i cui elementi sono di tipo tipoelem
- ▷ siano  $v, i, e$  tre generici valori dei tipi vettore, intero e tipoelem:
  - ▷ **CREAVETTORE** $=v$   
Post:  $\forall i, 1 \leq i \leq n, v(i)$  é di tipo tipoelem
  - ▷ **LEGGIVETTORE** $(v, i)=e$   
Pre:  $1 \leq i \leq n$   
Post:  $e = v(i)$
  - ▷ **SCRIVIVETTORE** $(v, i, e)=v'$   
Pre:  $1 \leq i \leq n$   
Post:  $v'(i) = e, v'(j) = v(j), \forall j \mid 1 \leq j \leq n, j \neq i$

## Struttura Dati Vettore: Realizzazione in C

La specifica del tipo di dato vettore in C é immediata in quanto é una struttura di dati primitiva del linguaggio.

- ▷ CREAETTORE  $\iff$  `tipoelem v[n]`
- ▷ LEGGIVETTORE  $\iff$  `v[i]`
- ▷ SCRIVIVETTORE  $\iff$  `v[i] = e`

Esempio: realizzazione di un vettore di interi

- ▷ CREAETTORE  $\iff$  `int v[N];`
- ▷ LEGGIVETTORE  $\iff$  `v[i];`
- ▷ SCRIVIVETTORE  $\iff$  `v[i] = e;`

## Struttura Dati Matrice

Una matrice é una estensione del tipo di dato vettore in cui ci possono essere due o piú indici.

Specifica Sintattica: matrici  $n_1 \times n_2$

- ▷ **nomi dei tipi:** matrice, intero, tipoelem
- ▷ **nomi degli operatori e dei relativi domini :**
  - **CREAMATRICE:**  $() \longrightarrow \text{matrice}$
  - **LEGGIMATRICE:**  $(\text{matrice}, \text{intero}, \text{intero}) \longrightarrow \text{tipoelem}$
  - **SCRIVIMATRICE:**  $(\text{vettore}, \text{intero}, \text{intero}, \text{tipoelem}) \longrightarrow \text{matrice}$

Specifica Semantica: matrici  $n_1 \times n_2$

- ▷ **matrice:** insieme i cui elementi sono di tipo tipoelem
- ▷ siano  $M, i, j, e$  generici valori dei tipo matrice, intero e tipoelem:
  - **CREAMATRICE**= $m$   
Post:  $\forall i, j \ 1 \leq i \leq n_1, 1 \leq j \leq n_2, \ v(i, j) \text{ é di tipo tipoelem}$
  - **LEGGIMATRICE**( $v, i, j$ ) =  $e$   
Pre:  $1 \leq i \leq n_1, \ 1 \leq j \leq n_2$   
Post:  $e = v(i, j)$

○  $\text{SCRIVIMATRICE}(v, i, j, e) = v'$

Pre:  $1 \leq i \leq n_1, \quad 1 \leq j \leq n_2$

Post:

$v'(i, j) = e$

$v'(i', j') = v(i, j), \quad \forall (i', j') \mid 1 \leq i' \leq n_1, \quad 1 \leq j' \leq n_2, \quad (i', j') \neq (i, j)$

Esempio: Realizzazione in C di una matrice di interi

```
int M[10][20]; // CREAMATRICE
int a, b;

a = M[5][5];    // LEGGIMATRICE

M[6][6] = b;    // SCRIVIMATRICE
```



## Struttura Dati Record a $k$ campi

Data una costante  $k$ , la struttura dati record é una  $k$ -pla i cui dati appartengono al dominio definito dal prodotto cartesiano

$$t_1 \times t_2 \times \dots \times t_k$$

Specifica Sintattica:

- ▷ **CREARECORD**:  $() \longrightarrow \text{record}$
- ▷ **LEGGICAMPO**<sub>1</sub>:  $\text{record} \longrightarrow t_1$
- ▷  $\vdots$
- ▷ **LEGGICAMPO** <sub>$k$</sub> :  $\text{record} \longrightarrow t_k$
- ▷ **SCRIVICAMPO**<sub>1</sub>:  $(\text{record}, t_1) \longrightarrow \text{record}$
- ▷  $\vdots$
- ▷ **SCRIVICAMPO** <sub>$k$</sub> :  $(\text{record}, t_k) \longrightarrow \text{record}$

Specifica semantica: sia  $r = (e_1, \dots, e_k)$  un dato di tipo record:

▷  $\text{CREARECORD} = r$

Post:  $r = (e_1, \dots, e_k), \forall 1 \leq h \leq k, e_h \in t_h$

▷ una funzione  $\text{LEGGICAMPO}$  e  $\text{SCRIVICAMPO}$  per ogni campo del record:

○  $\text{LEGGICAMPO}_h(r) = e$

Post:  $e = e_h$

○  $\text{SCRIVICAMPO}_h(r, e) = r'$

Post:  $r' = (e_1, \dots, e_{h-1}, e, e_{h+1}, \dots, e_k)$

In C il tipo di dato record é un tipo di dato primitivo e viene realizzato tramite il costrutto **struct**:

```
struct data = {  
    int giorno;  
    int mese;  
    int anno;  
};  
  
data nineEleven;  
  
nineEleven.giorno = 9;  
nineEleven.mese   = 11;  
nineEleven.anno   = 2001;
```

- ▷ `CREARECORD`  $\iff$  data `r`;
- ▷ `LEGGICAMPO.giorno(r)`  $\iff$  `r.giorno`;
- ▷ `SCRIVICAMPO.giorno(r)`  $\iff$  `r.giorno` = integer;

Le strutture dati finora viste hanno le seguenti caratteristiche:

- ▷ sono primitive nella maggior parte dei linguaggi e quindi facile da usare
- ▷ sono strutture **statiche**, cioè il numero di elementi che ad esse appartengono deve essere fissato a priori.

Quindi, ad esempio, se vogliamo scrivere un programma che legge su STDIN un insieme di dati e calcola il minimo siamo costretti a dichiarare a priori una dimensione massima della struttura che conterrà i dati.

Questa caratteristica conduce, in generale, ad un uso inefficiente delle risorse hw.

## Rappresentazione in Memoria

Per valutare l'efficienza di procedure che usano tipi di dato primitivi si prescinde dalle caratteristiche specifiche di una macchina e si assume una organizzazione abbastanza generica:

- ▷ i dati sono contenuti in memoria
- ▷ la memoria é divisa in **celle**, tutte di uguali ampiezza, ognuna delle quali può contenere un dato elementare (eg. un intero)
- ▷ si accede ad una cella specificandone l'indirizzo
- ▷ il tempo di accesso ad una cella si assume costante

Memorizzazione di un vettore  $v$  di  $n$  elementi di tipo primitivo (int, char, ...):

- ▷ é memorizzato in  $n$  celle consecutive a partire da un indirizzo  $a_v$ ,
- ▷ il tempo di accesso ad un generico elemento  $i$  é uguale al tempo di accesso della cella di indirizzo  $a_v + i$ , quindi é **costante** o  $\mathcal{O}(1)$
- ▷ il passaggio per **variabile** di un vettore costa  $\mathcal{O}(1)$  poiché si passa l'indirizzo  $a_v$
- ▷ il passaggio del vettore per **valore** costa  $\mathcal{O}(n)$ , in quanto bisogna copiare su un nuovo vettore tutti gli elementi del vettore  $v$

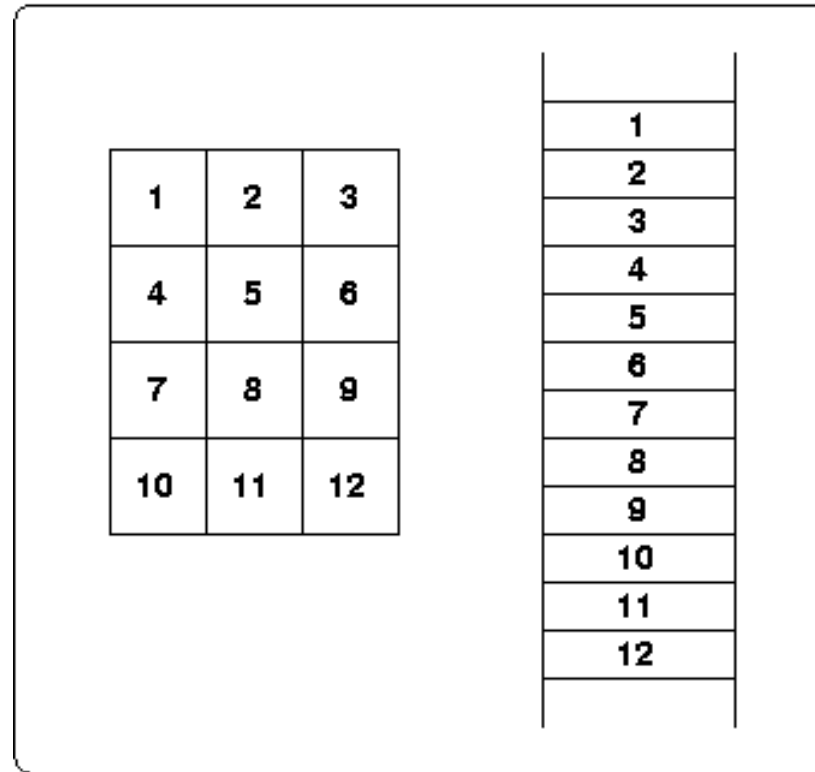
Memorizzazione di una matrice di  $n_1 \times n_2$  elementi di tipo primitivo:

- ▷ la matrice é memorizzata linearizzando la struttura per righe. Supposto che l'indirizzo di partenza sia  $a_m$ :
  - ▷ la riga di indice 0 é memorizzata nel range di indirizzi  
 $[a_m \dots a_m + n_2 - 1]$
  - ▷ la riga di indice 1 é memorizzata nel range di indirizzi  
 $[a_m + n_2 \dots a_m + n_2 + n_2 - 1] = [a_m + n_2 \dots a_m + 2 * n_2 - 1]$
  - ▷ la riga di indice  $i$  é memorizzata nel range di indirizzi  
 $a_m + i * n_2 \dots a_m + i * n_2 + n_2 - 1$
- ▷ il tempo di accesso all'elemento  $(r, c)$  é uguale al tempo di accesso alla cella di indirizzo  $a_m + c * n_2 + r$ , quindi é  $\mathcal{O}(1)$
- ▷ il costo del passaggio per **variabile** di una matrice é  $\mathcal{O}(1)$  poiché si passa il valore di  $a_m$
- ▷ il costo del passaggio per **valore** di una matrice é  $\mathcal{O}(n_1 \cdot n_2)$

```
int[10][20] M;  
  
void pippo (int myM[10][20] ) {  
    myM[0][0] = 0xfedececa; // modifica elemento (0,0) di M  
}  
  
pippo (M);
```

## Rappresentazione in Memoria di una Matrice

Rappresentazione di una matrice  $4 \times 3$  di numeri interi in memoria:



## Memorizzazione di un Record

Memorizzazione di un record a  $k$  campi

I campi del record sono memorizzati in celle di memoria consecutive:

- ▷ se i campi del record sono tutti dello stesso tipo allora la rappresentazione in memoria é analoga a quella di un vettore di  $k$  elementi
- ▷ se i campi del record sono di tipo diverso allora ogni campo occuperá un numero di celle diverso
- ▷ l'accesso al campo  $i$  del record ha un costo  $\mathcal{O}(1)$  e l'indirizzo é calcolato nel seguente modo:  
$$m_r + \sum_{k=0}^{i-1} \text{sizeof}(\text{campo}_k)$$
- ▷ il passaggio per variabile di un record ha costo costante
- ▷ mentre quello per valore ha un costo proporzionale alla dimensione di ciascun campo del record.

## Uso dei Puntatori

Per ottenere strutture dati a dimensione variabile abbiamo bisogno di utilizzare i **puntatori**

Il puntatore é un tipo di dato che rappresenta l'indirizzo di una cella di memoria. Una variabile di tipo puntatore ad un intero non per valore quello dell'intero, ma l'indirizzo di memoria ove l'intero é memorizzato.

Esempio

```
int * pi;  
int i;
```

La variabile *i* denota un valore intero, mentre la variabile *pi* denota un indirizzo di memoria destinato a contenere un valore intero.

Quando dichiariamo una variabile il compilatore alloca una area di memoria opportunamente grande (**sizeof**) a contenere un valore del tipo della variabile e la associa alla variabile in questione.

Quando dichiariamo un puntatore il compilatore alloca una area di memoria destinata a contenere un **indirizzo di memoria** e la associa alla variabile puntatore.

Successivamente é compito del programmatore **allocare** una area di memoria opportunamente grande a contenere un valore del tipo puntato dalla variabile puntatore, e **dealloca** quando tale area non serve piú.



Esempio: nel nostro caso possiamo effettuare le seguenti operazioni:

```
int * pi;  
int i;  
  
pi = malloc(sizeof(int)); /* alloca una area di memoria */  
*pi = 4;  
printf("%d\n", *pi);  
free(pi);                /* dealloca l'area di memoria puntata da pi */
```

**WARNING** si raccomanda di utilizzare i puntatori con molta cautela:

- ▷ un uso improprio conduce facilmente a **segmentation fault**, ovvero un indirizzamento erraneo dello spazio di memoria.
- ▷ l'uso dei puntatori conduce alla scrittura di codice a volte poco comprensibile.
- ▷ un uso sbagliato può condurre all'impossibilità di indirizzare, durante l'esecuzione del programma, zone di memoria precedentemente allocate.

## Esempio: Record concatenati

Tramite i puntatori possiamo costruire strutture dati dinamiche che non sono allocate in porzioni di memoria contigue.

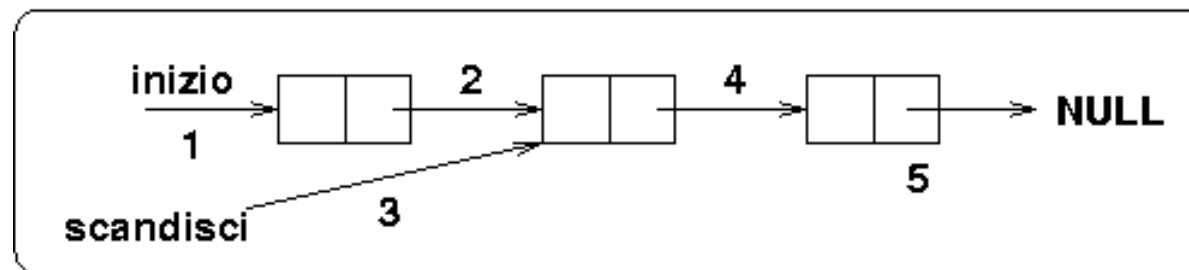
```
typedef struct studente * catena;  
  
struct studente {  
    int      matricola;  
    catena   successivo;  
};
```

- ▷ Il tipo **catena** denota un puntatore ad un record di tipo **studente**.
- ▷ Il tipo **studente** denota un record composto da un campo intero e da un campo puntatore ad un record di tipo **studente**.

Eseguendo adesso la seguente sequenza di operazioni:

```
1 inizio = malloc(sizeof(struct studente));  
2 inizio->successivo = malloc(sizeof(struct studente));  
3 scandisci = inizio->successivo;  
4 scandisci->successivo = malloc(sizeof(struct studente));  
5 (scandisci->successivo)->successivo = NULL;
```

1. il puntatore **inizio** viene inizializzato con un indirizzo di una area memoria grande quando il tipo di dato **studente**
2. il campo **successivo** del record puntato da **inizio** viene inizializzato con l'indirizzo di un'altro record di tipo **studente**
3. la variabile puntatore **scandisci** contiene l'indirizzo di memoria del record puntato da **inizio**→**successivo**
4. il campo **successivo** del record puntato da **scandisci** viene inizializzato con l'indirizzo di un'altro record di tipo **studente**
5. il campo **successivo** del record puntato da **scandisci**→**successivo** viene inizializzato ad un valore speciale detto **NULL**



## Esercizi

- ▷ scrivere un programma C che inizializza una matrice  $3 \times 3$  e stampa la linearizzazione
- ▷ scrivere un programma C che inizializza una catena e stampa gli indirizzi di ciascuna cella