



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



Complessità computazionale e analisi di complessità

Didattica Integrativa - Fondamenti di Informatica
AA 2023/2024

Giulio Martini

`giulio.martini@igi.cnr.it`

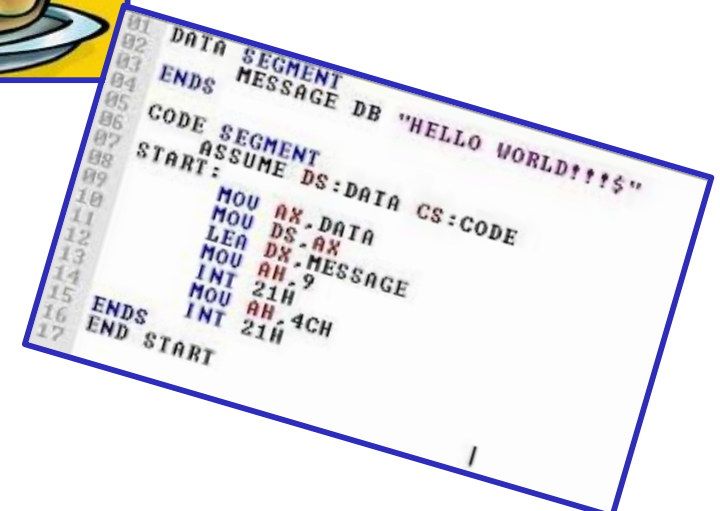
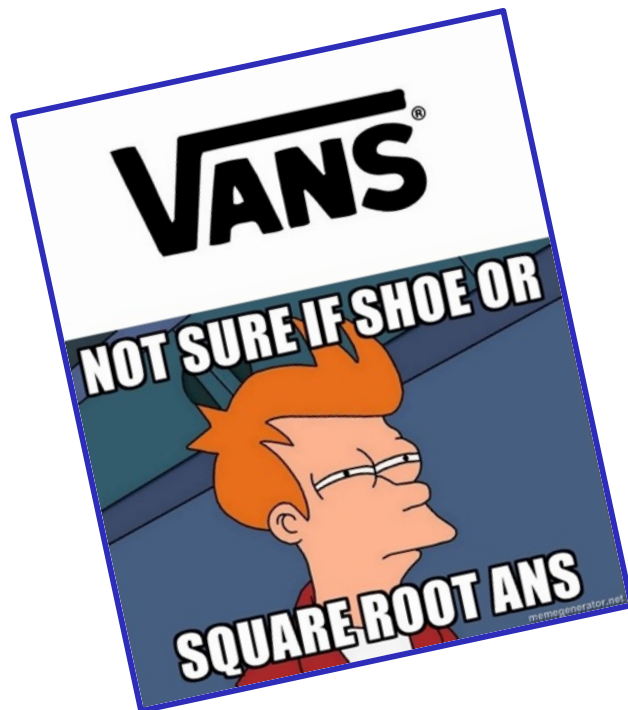
Cosa vedremo?

- **Complessità computazionale di un algoritmo**
 1. Cos'è?
 2. A cosa serve?
 3. Come si calcola?
- Analisi della **complessità di un problema**:
 - Come si relaziona alla complessità computazionale?

Definizione

La **complessità computazionale** di un **algoritmo** è una misura assoluta delle risorse necessarie alla sua esecuzione

Qualsiasi compito che può essere scomposto in una lista non infinita di operazioni.



La **complessità computazionale** di un algoritmo è una misura assoluta delle **risorse** necessarie alla sua esecuzione

- **Risorse**: il **tempo** impiegato a eseguire l'algoritmo e lo **spazio** occupato nel farlo.
- L'analisi della complessità computazionale si può quindi suddividere in:
 1. Analisi della **complessità temporale** (tempo impiegato).
 2. Analisi della **complessità spaziale** (spazio occupato in memoria).

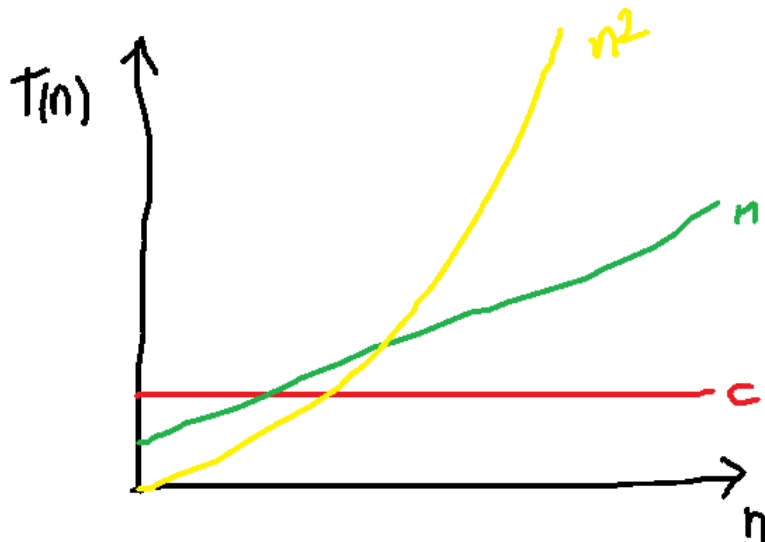
Da cosa dipende?

La **complessità computazionale** di un algoritmo è una **misura assoluta** delle risorse necessarie alla sua esecuzione.

- **Misura assoluta**: non dipende da chi o che cosa sta eseguendo l'algoritmo.
- Esiste un parametro di cui tenere conto quando si vuole calcolare la complessità computazionale di un algoritmo?

Un parametro di cui tenere conto

Più precisamente, la **complessità computazionale** di un algoritmo è la **funzione** di crescita del **tempo** impiegato e dello **spazio** occupato al crescere delle dimensioni dell'**input n** (o degli input).



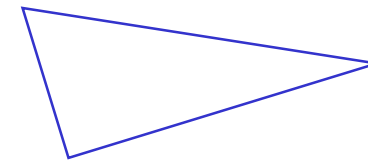
- La funzione di **complessità temporale** si indica con **$T(n)$** .
- La funzione di **complessità spaziale** si indica con **$S(n)$** .

A cosa serve?

L'analisi della **complessità computazionale** serve a farsi un'idea dell'efficienza di un algoritmo a prescindere da chi lo esegue.

Vantaggi:

- Analisi delle prestazioni svincolata dall'architettura
- Possibilità di confrontare algoritmi diversi



$$T_{x,y,z} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{pmatrix}$$



$$\begin{pmatrix} xaxis.x & yaxis.x & zaxis.x & 0 \\ xaxis.y & yaxis.y & zaxis.y & 0 \\ xaxis.z & yaxis.z & zaxis.z & 0 \\ -dot(xaxis, cameraPosition) & -dot(yaxis, cameraPosition) & -dot(zaxis, cameraPosition) & 1 \end{pmatrix}$$



$$\begin{pmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & far/(near - far) & -1 \\ 0 & 0 & (near * far)/(near - far) & 0 \end{pmatrix}$$

Complessità temporale



Partiamo dalla **complessità temporale**.

Se provassimo a misurare il **tempo di esecuzione**?

Dipende da:

1. Modalità di implementazione
2. Specifico input
3. Hardware utilizzato



```
tic  
  
funzioneMoltoPesante();  
  
toc
```

10 ms

```
long inizio = System.currentTimeMillis();  
  
funzioneMoltoPesante();  
  
long fine = System.currentTimeMillis();  
long tempoTrascorso = fine - inizio;  
  
System.out.println(tempoTrascorso);
```

3 ms

Misurare il **tempo di esecuzione** **non** è un metodo affidabile per la valutazione delle prestazioni!

Complessità temporale

Come individuare un metodo che permetta di ottenere una misura assoluta?

PRIMA IDEA!

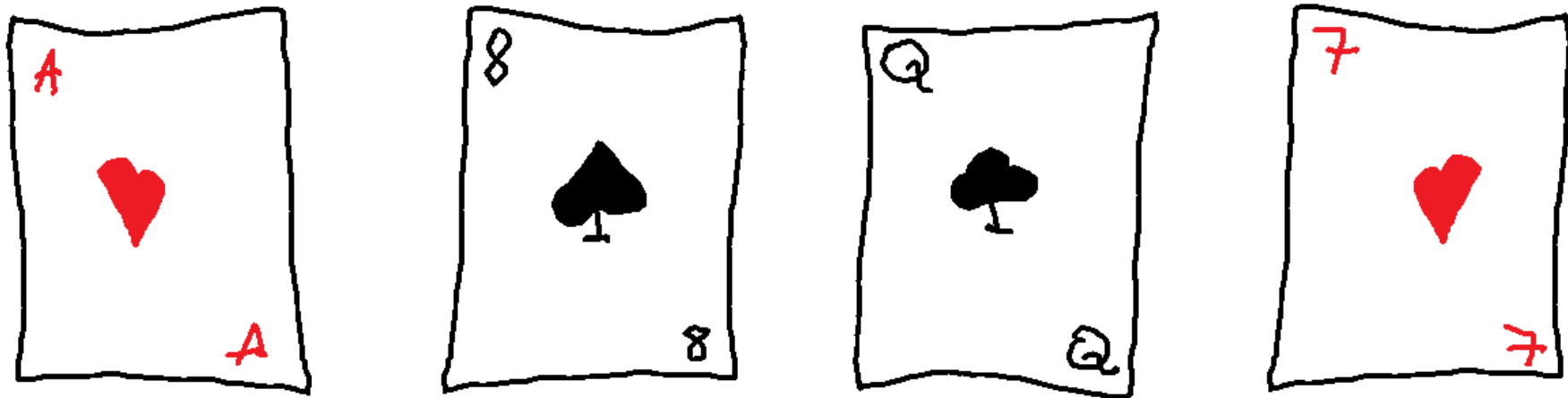
Misurare come **varia** il tempo di esecuzione di un algoritmo...

al variare delle dimensioni dell'input n !

Vediamo qualche esempio...

Esempio: bubble sort

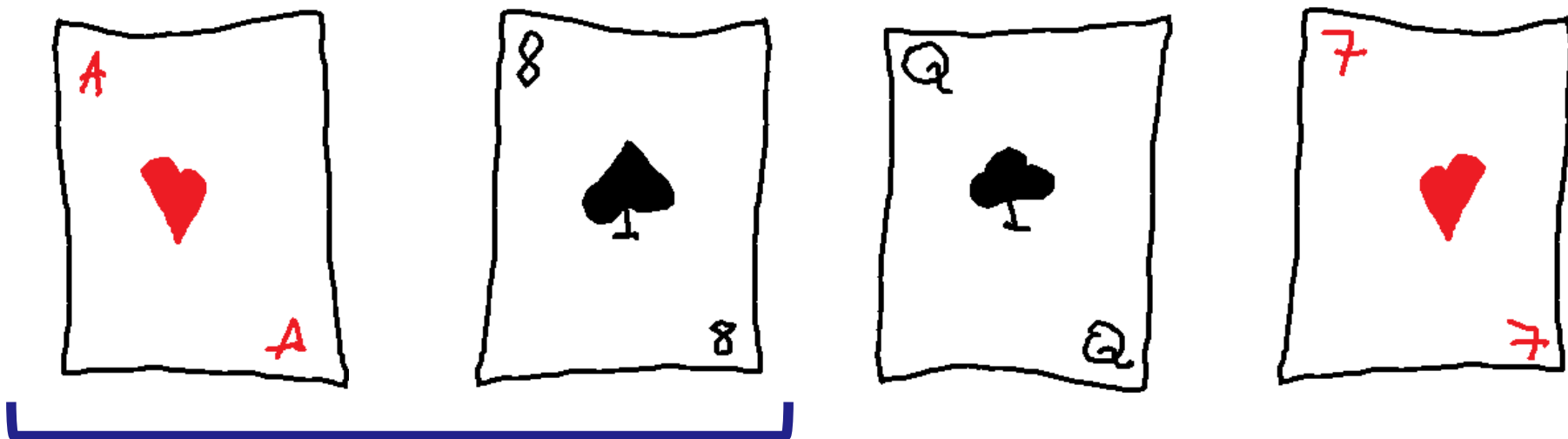
- Bubble sort (ordinamento a bolla)
 - Guardo le carte a coppie partendo da sinistra
 - Se la carta di sinistra è la più alta scambio le carte
 - Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

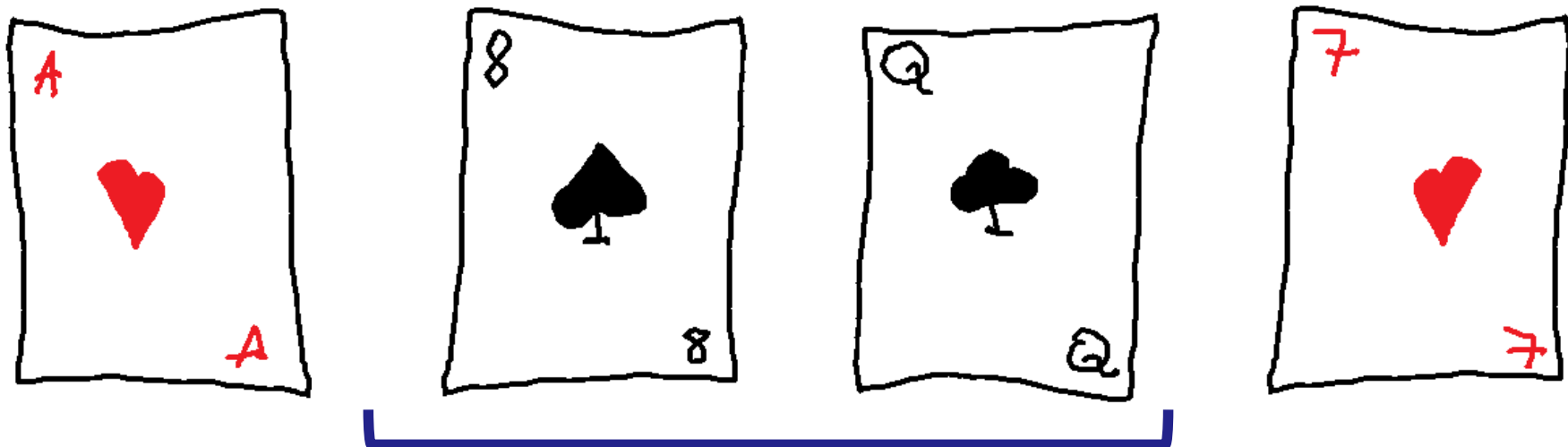
- Bubble sort

- Guardo le carte a coppie partendo da sinistra
- Se la carta di sinistra è la più alta scambio le carte
- Continuo finché non riesco a fare un intero giro senza scambiare carte



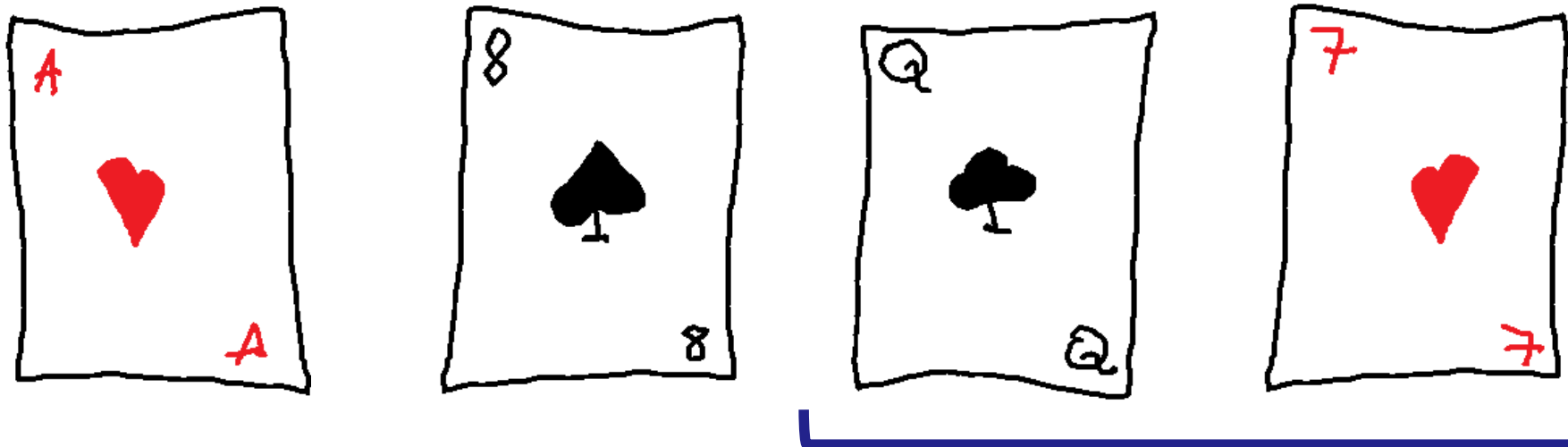
- Bubble sort

- Guardo le carte a coppie partendo da sinistra
- Se la carta di sinistra è la più alta scambio le carte
- Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

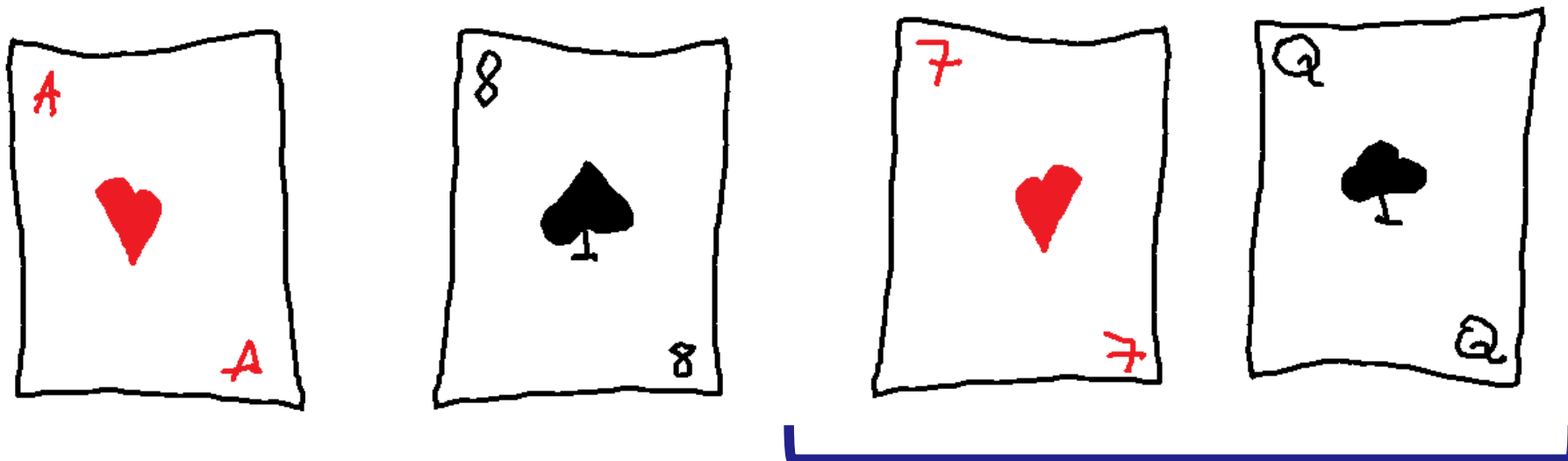
- Bubble sort
 - Guardo le carte a coppie partendo da sinistra
 - Se la carta di sinistra è la più alta scambio le carte
 - Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

- Bubble sort

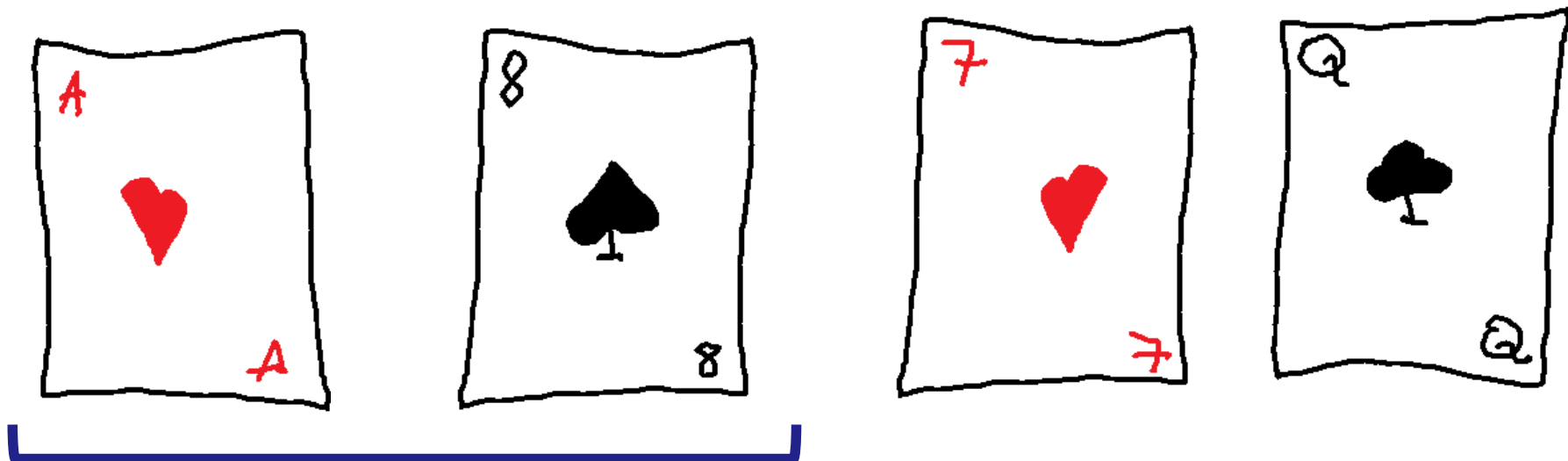
- Guardo le carte a coppie partendo da sinistra
- Se la carta di sinistra è la più alta scambio le carte
- Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

- Bubble sort

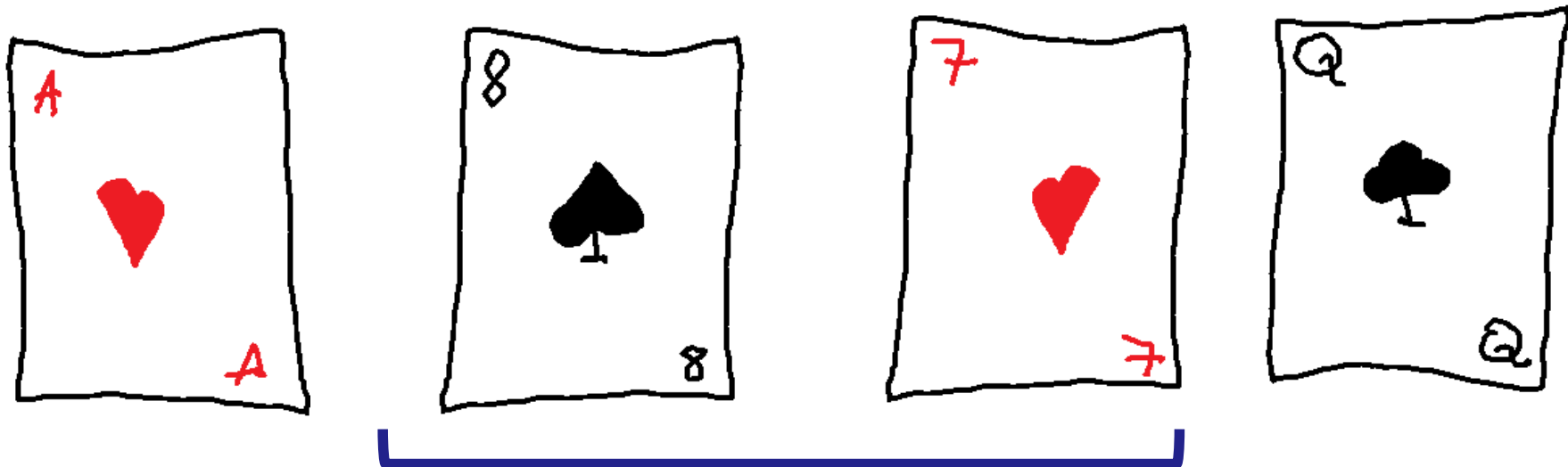
- Guardo le carte a coppie partendo da sinistra
- Se la carta di sinistra è la più alta scambio le carte
- Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

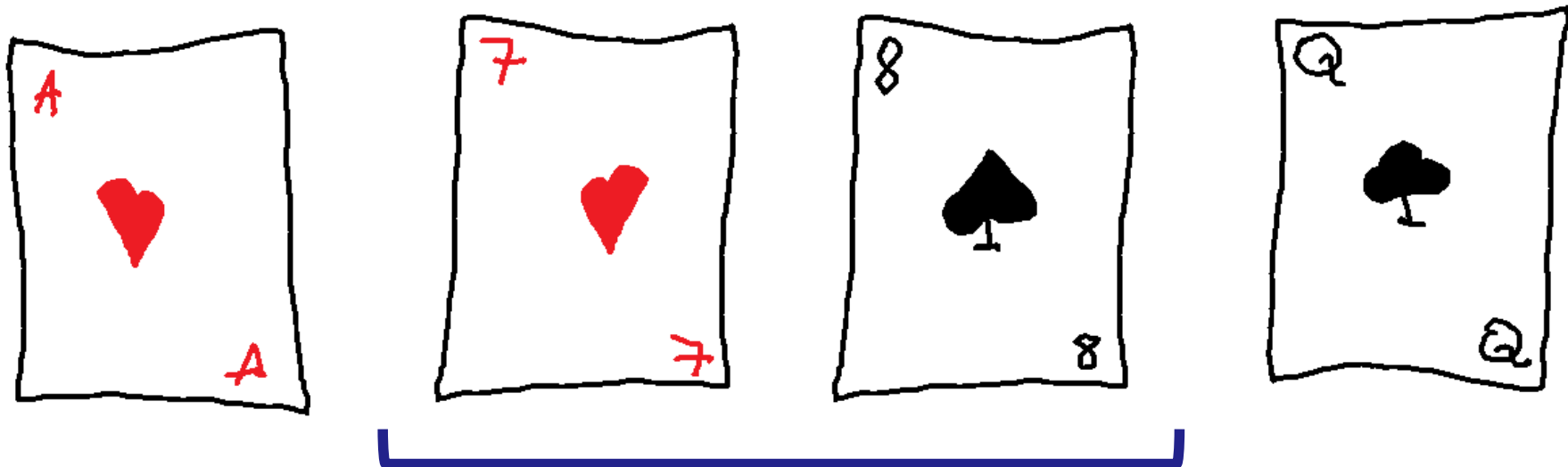
- Bubble sort

- Guardo le carte a coppie partendo da sinistra
- Se la carta di sinistra è la più alta scambio le carte
- Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

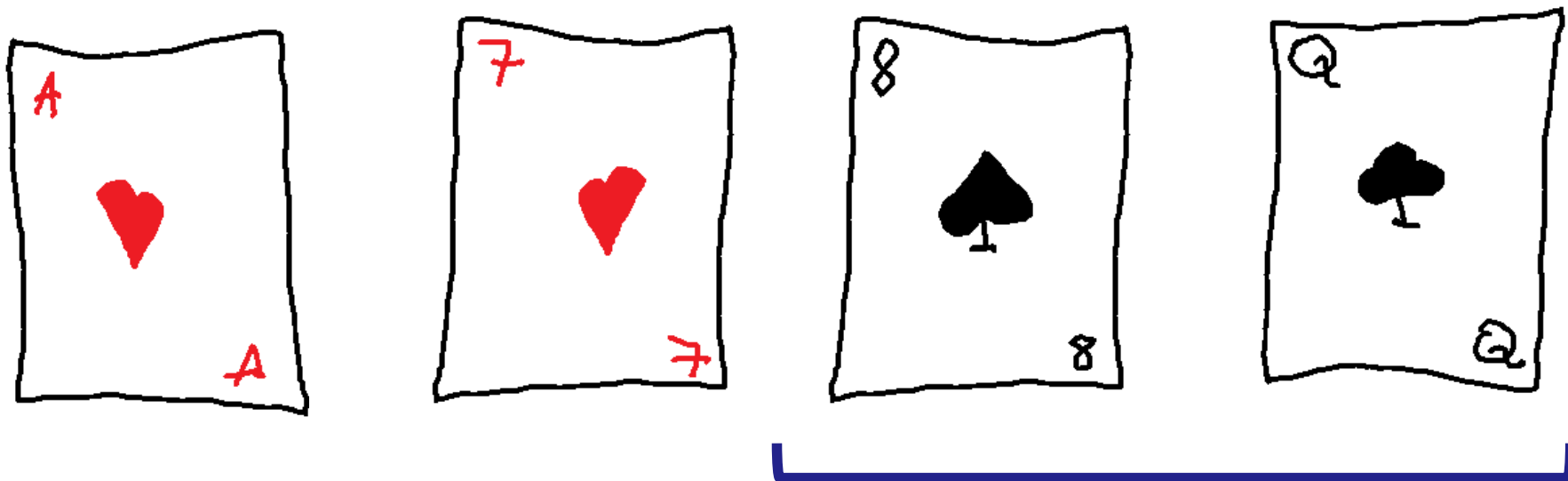
- Bubble sort
 - Guardo le carte a coppie partendo da sinistra
 - Se la carta di sinistra è la più alta scambio le carte
 - Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

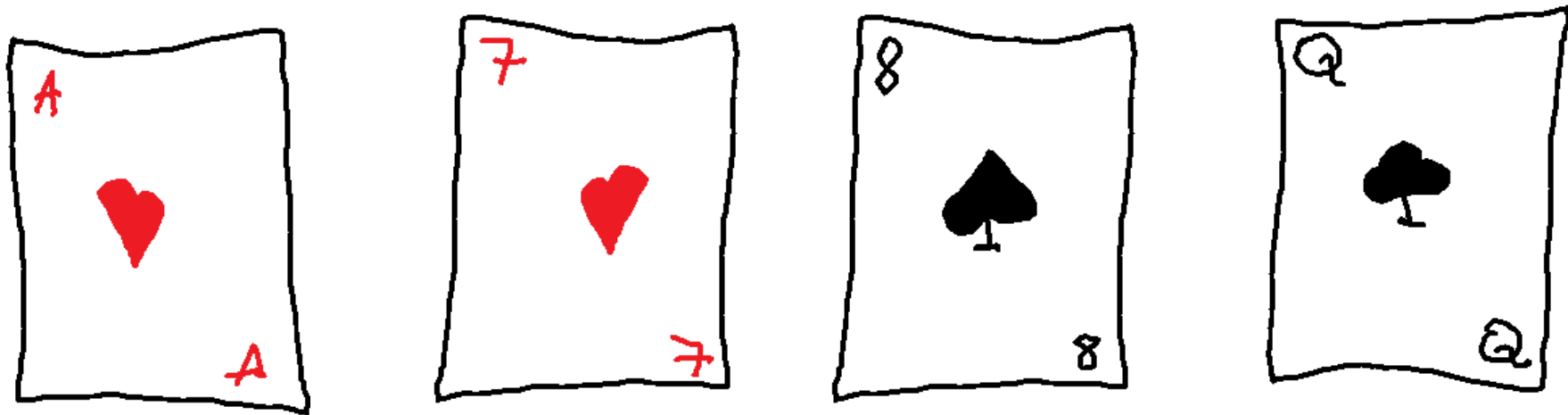
- Bubble sort

- Guardo le carte a coppie partendo da sinistra
- Se la carta di sinistra è la più alta scambio le carte
- Continuo finché non riesco a fare un intero giro senza scambiare carte



Esempio: bubble sort

- Bubble sort
 - Guardo le carte a coppie partendo da sinistra
 - Se la carta di sinistra è la più alta scambio le carte
 - Continuo finché non riesco a fare un intero giro senza scambiare carte



- Merge sort

6 5 3 1 8 7 2 4

- Se la **sequenza** di carte da ordinare ha lunghezza **0** oppure **1**, è **già ordinata**.
Altrimenti:
 - **Divido** la sequenza di carte a **metà** in due sequenze, e via via ciascuna di esse ancora a metà, fino ad ottenere sequenze di una sola carta.
 - **Fondo ordinatamente** a due a due queste sottosequenze, ottenendo sequenze ordinate di due carte (l'ultima può essere di una carta se sono dispari).
 - **Fondo** ordinatamente le sottosequenze ordinate, a due a due, ottenendo **sequenze ordinate** di quattro (o tre) carte. Per fare questo, si **estrae** ripetutamente **il minimo delle due sottosequenze** e lo si pone nella sequenza in uscita, che risulterà ordinata.
 - Continuo a fondere le sottosequenze in maniera ordinata fino a **riottenere una sequenza** di dimensione pari a quella **iniziale**, ma **ordinata**.

Implementazione ed analisi

- Vediamo un'implementazione del **bubble sort** e del **merge sort**.

https://www.youtube.com/watch?v=kgBjXUE_Nwc&t=31s

https://www.youtube.com/watch?v=kgBjXUE_Nwc&t=182s

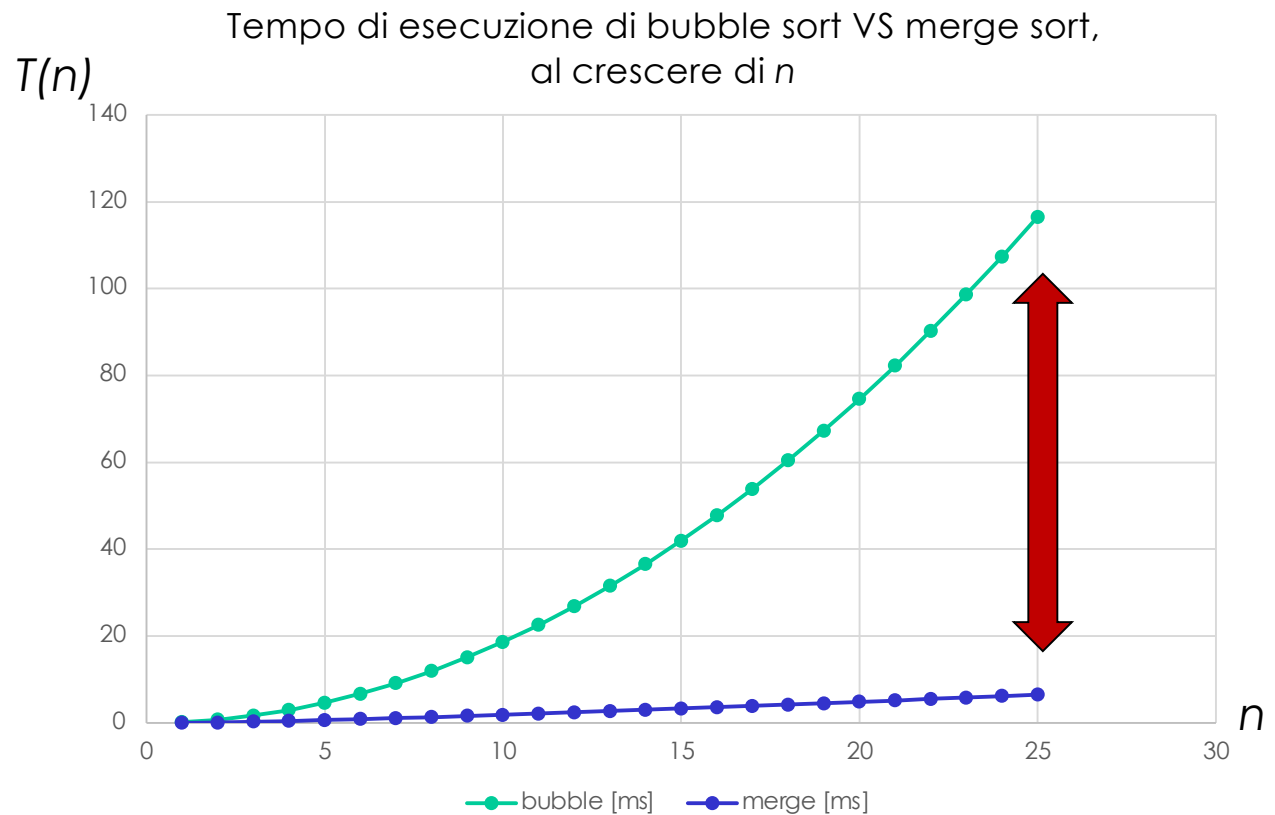
youtube.com/Computerphile

- Ora che abbiamo visto come funzionano questi algoritmi, è possibile:
 1. Misurare per entrambi come **varia** il **tempo** di esecuzione al **variare** della dimensione di **n** .
 2. Confrontare i risultati ottenuti.

Un rapido confronto

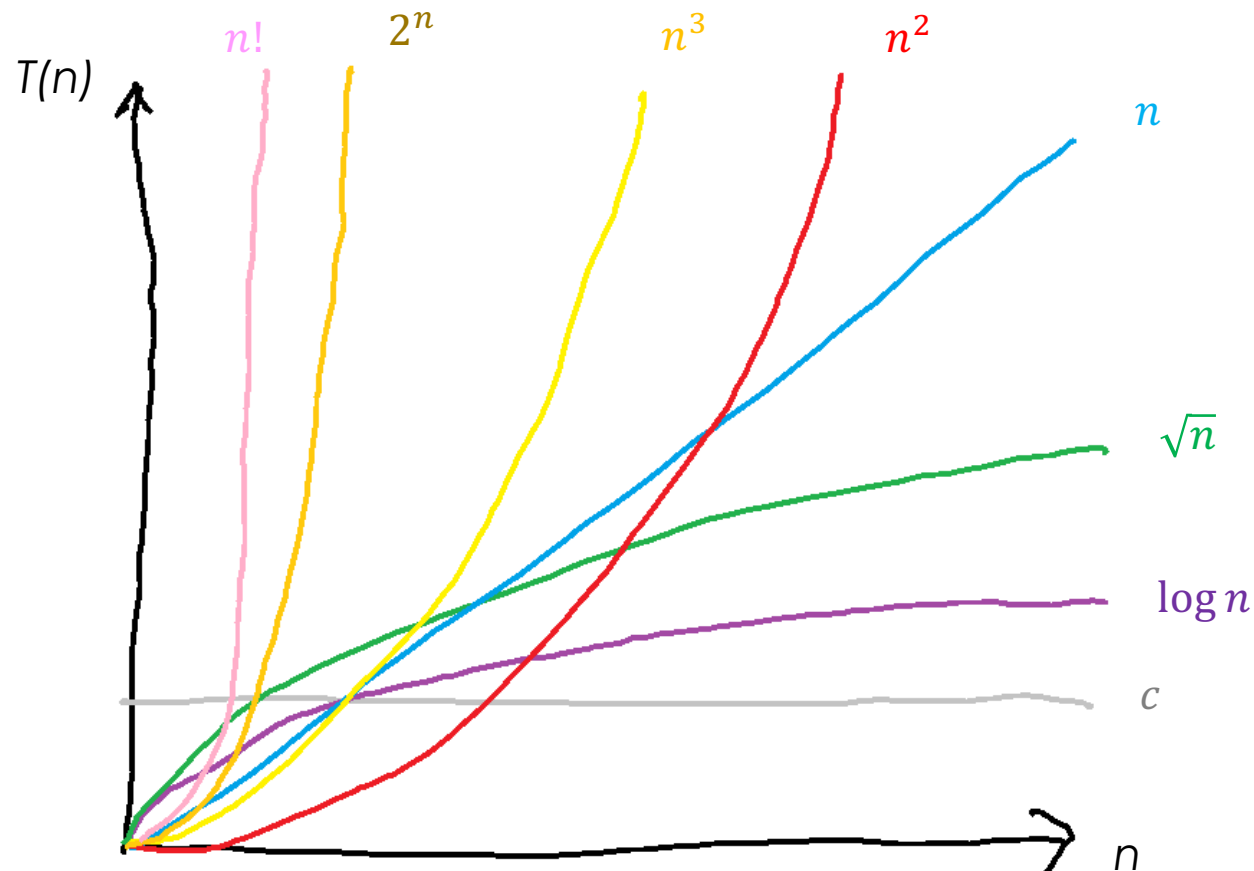
N	bubble [ms]	merge [ms]
1	0,186461562	0,052467259
2	0,745846247	0,112261046
3	1,678154055	0,266894323
4	2,983384987	0,449044185
5	4,661539042	0,651655193
6	6,712616221	0,870571784
7	9,136616523	1,103048102
8	11,93353995	1,347132555
9	15,1033865	1,601365937
10	18,64615617	1,864615617
11	22,56184896	2,13597677
12	26,85046488	2,414709846
13	31,51200392	2,700199022
14	36,54646609	2,991923528
15	41,95385138	3,289437193
16	47,73415979	3,592353479
17	53,88739133	3,900334267
18	60,41354599	4,21308129
19	67,31262377	4,530329475
20	74,58462467	4,851841696
21	82,2295487	5,177404566
22	90,24739586	5,506825049
23	98,63816613	5,839927675
24	107,4018595	6,176552246
25	116,5384761	6,516551929

È quindi possibile confrontare i tempi di esecuzione degli algoritmi al crescere di n :




La forma della curva

- Algoritmi diversi possono presentare complessità diverse anche risolvendo uno stesso problema!



Cosa è meglio?

- Come confrontare le curve tra loro?
- Per n piccolo non è chiaro... al crescere di n però divergono in maniera evidente!
- Per $n \rightarrow \infty$  la **curva migliore** è quella che cresce meno!
- **Problema:** siamo ancora vincolati dal dover effettuare una misura temporale...
Non sempre è fattibile misurare il tempo di esecuzione di un algoritmo!

SECONDA IDEA!

Se individuassimo un insieme di **operazioni fondamentali**,
potremmo utilizzarne il tempo di esecuzione come “unità di misura”!

Il passo base

- Un **passo base** è qualsiasi operazione che richiede un **tempo costante** per essere eseguita, cioè che non dipende dal valore e dal tipo dell'oggetto dell'operazione.
- Molto spesso è possibile scomporre (almeno parzialmente) un algoritmo in una **sequenza di passi base**, ognuno con il suo tempo costante di esecuzione.
- Con una piccola approssimazione, è possibile considerare i tempi di esecuzione di diversi passi base come uguali tra loro.
- È quindi possibile utilizzare il tempo di esecuzione di un passo base come unità di misura per il tempo di esecuzione di un algoritmo.
- Il tempo di esecuzione complessivo sarà quindi pari alla **somma** dei tempi di esecuzione dei singoli passi base.



VANTAGGIO: ci si è svincolati dalla misurazione del tempo!

Calcolo dei passi base

- Alcune operazioni che sono **passo base**:

- Assegnazione di un valore ad una variabile

```
i = 10u;
```

- Operazioni aritmetiche di base (addizione, sottrazione...)

```
i++;
```

```
i = 6 + var;
```

- Accesso ad un elemento di un vettore

```
i = array[10];
```

- Valutazione di un'espressione booleana (un confronto fra due numeri)

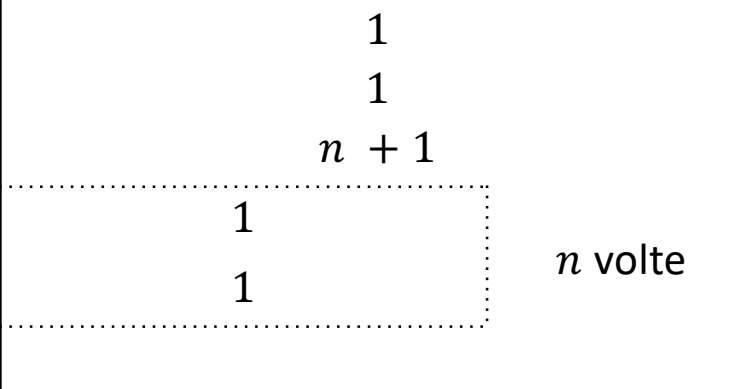
```
if (test)
```

```
while (test)
```

```
for (i = 0; i < 10; i++)
```

- Altre operazioni **non** sono passo base (esempio: **ricorsione**)

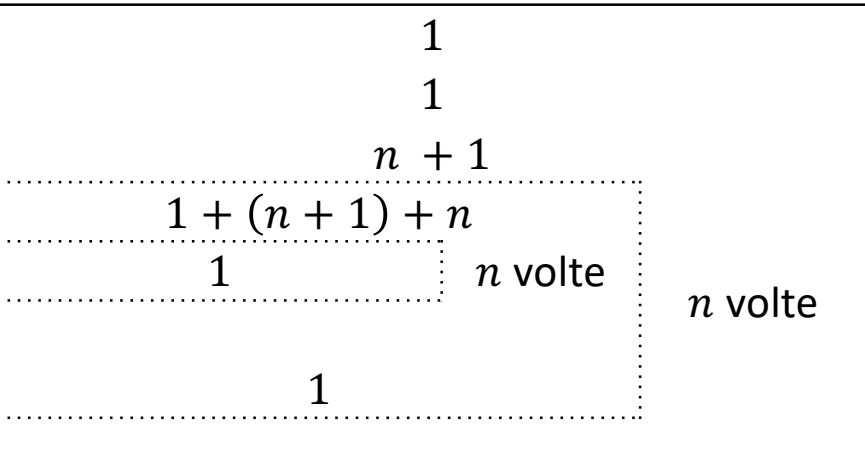
Esempio 1

<pre> int i = 0; int j = 0; while (i < n) { j = 3*j + 42; i = i + 1; } </pre>	
Somma :	$2 + (n + 1) + (1 + 1) \cdot n = 3n + 3$

$$T(n) = 3n + 3$$

Osservazione: l'introduzione di un ciclo iterativo basato su **n** ha portato alla presenza di un termine di complessità pari a **n**.

Esempio 2

<pre> int i = 0; int j = 0; while (i < n) { for (j = 0; j < n; j++) { System.out.println("Ciao"); } i = i + 1; } </pre>	
Somma:	$ \begin{aligned} &1 + 1 + (n + 1) + \\ &n \cdot [1 + (n + 1) + n + 1 \cdot n + 1] \\ &= 3n^2 + 4n + 2 \end{aligned} $

$$T(n) = 3n^2 + 4n + 2$$

Osservazione: annidando un secondo ciclo iterativo dentro al primo, il grado del termine di ordine più elevato ora è **n^2** ... è aumentato di uno!

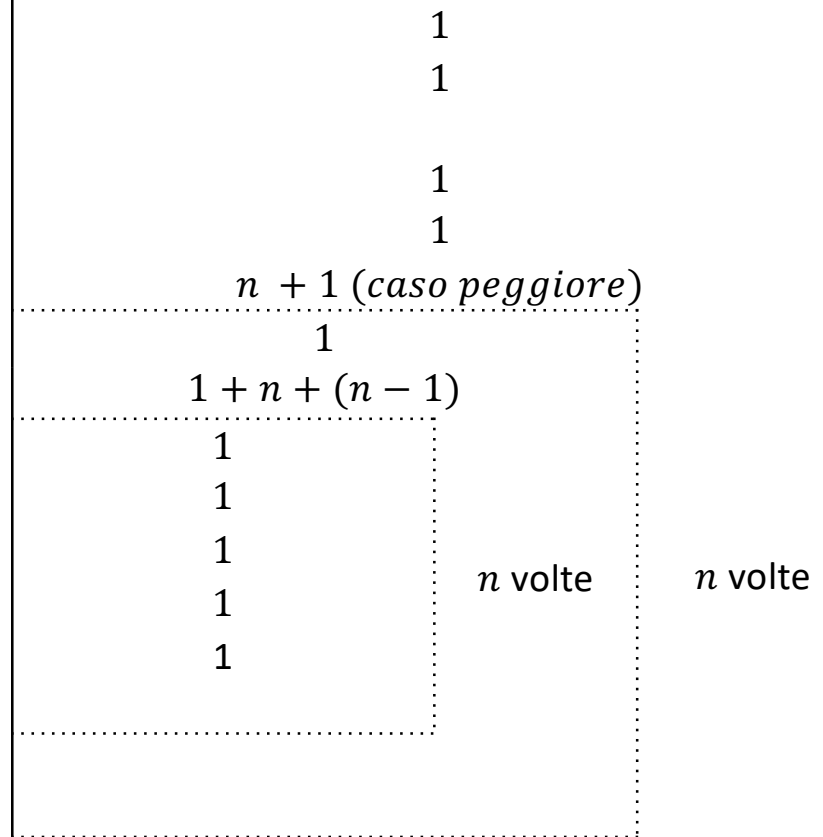
Esempio: bubble sort

Studiamo ora il **bubble sort**, nel caso peggiore (elementi in ordine inverso).

```

public static void bubbleSort(int[] cards) {
    int n = cards.length;
    int tempCard = 0;

    int i = 0;
    boolean swap = true;
    while (swap) {
        swap = false;
        for (int j = 0; j < n - 1; j++) {
            if (cards[j] > cards[j+1]) {
                tempCard = cards[j];
                cards[j] = cards[j + 1];
                cards[j + 1] = tempCard;
                swap = true;
            }
        }
    }
}
  
```



$$T(n) = 7n^2 + 2n + 5$$

Somma :

$$1 + 1 + 1 + 1 + (n + 1) + n \cdot [1 + 1 + n + (n - 1) + n \cdot (1 + 1 + 1 + 1 + 1)]$$

- Analizziamo ora la complessità temporale nel caso in cui l'algoritmo da analizzare sia **ricorsivo**.
- La **successione di Fibonacci** è una lista di numeri in cui ciascun numero è la somma dei due che lo precedono nella lista.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Per implementarla, è possibile scrivere un algoritmo ricorsivo:

```
public static long Fibonacci(long n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

1 + 1

$T(n - 1) + T(n - 2)$

Complessità temporale della ricorsione 2/2

<pre> public static long Fibonacci(long n) { if (n == 0 n == 1) return n; else return Fibonacci(n - 1) + Fibonacci(n - 2); } </pre>	$1 + 1$ $T(n - 1) + T(n - 2)$
Somma:	$T(n) = 2 + T(n - 1) + T(n - 2)$

Metodo della **sostituzione**:

$$T(6) = 2 + T(5) + T(4) = 2 + 27 + 16 = 45$$

$$T(5) = 2 + T(4) + T(3) = 2 + 16 + 9 = 27$$

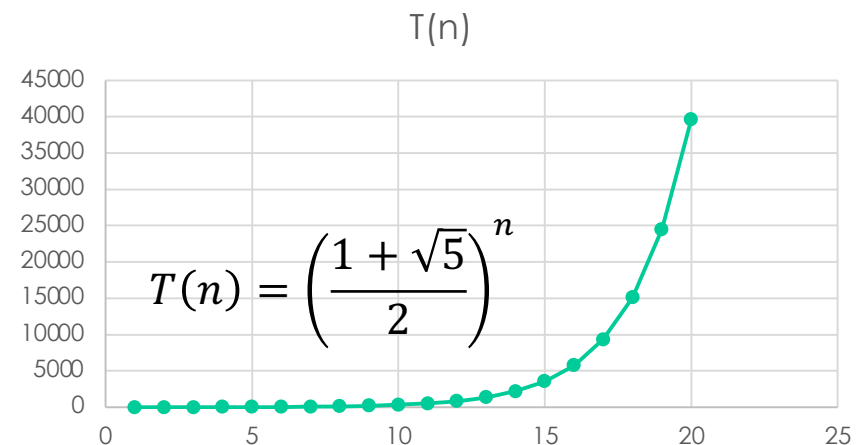
$$T(4) = 2 + T(3) + T(2) = 2 + 9 + 5 = 16$$

$$T(3) = 2 + T(2) + T(1) = 2 + 5 + 2 = 9$$

$$T(2) = 2 + T(1) + T(0) = 2 + 2 + 1 = 5$$

$$T(1) = 2$$

$$T(0) = 1$$



Fibonacci iterativo

```
public static long Fibonacci(long n)
{
    if (n < 2) return n;
    long fib0 = 0;
    long fib1 = 1;

    for (int i = 2; i <= n; i++) {

        long newFib = fib0 + fib1;
        fib0 = fib1;
        fib1 = newFib;
    }
    return fib1;
}
```

1
1
1

$1 + n + (n - 1)$

1
1
1

Somma:

$$T(n) = 3 + 1 + n + (n - 1) + 3 \cdot (n - 1) = 4 + n + n - 1 + 3n - 3 = 5n$$

Osservazione: andamento lineare!

L'utilizzo di espressioni come tempo di esecuzione costante, lineare, quadratico o esponenziale non è molto preciso...

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro!

Supponiamo di avere, per uno stesso problema:

- sette algoritmi diversi con diversa complessità.
- tempo di esecuzione di un passo base pari a 1 microsecondo (10^{-6} sec).

Classi di complessità

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro.

n	10	100	1000	1000000	
\sqrt{n}	3 μ s	10 μ s	30 μ s	1 ms	
$n + 5$	15 μ s	105 μ s	1 ms	1 s	
$2n$	20 μ s	200 μ s	2 ms	2 s	
n^2	100 μ s	10 ms	1 s	1000000 s	~ 12 gg
$n^2 + 2$	102 μ s	10 ms	1 s	1000000 s	~ 12 gg
n^3	1 ms	1 s	1000 s	1×10^{12} s	~ 300 secoli
2^n	1 ms	1×10^{24} s	1×10^{295} s	#NUM!	~ 3×10^{301016} secoli

Classi di complessità

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro.

n	10	100	1000	1000000	
\sqrt{n}	3 μ s	10 μ s	30 μ s	1 ms	
$n + 5$	15 μ s	105 μ s	1 ms	1 s	
$2n$	20 μ s	200 μ s	2 ms	2 s	
n^2	100 μ s	10 ms	1 s	1000000 s	~ 12 gg
$n^2 + 2$	102 μ s	10 ms	1 s	1000000 s	~ 12 gg
n^3	1 ms	1 s	1000 s	1×10^{12} s	~ 300 secoli
2^n	1 ms	1×10^{24} s	1×10^{295} s	#NUM!	~ 3×10^{301016} secoli

Classi di complessità

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro.

n	10	100	1000	1000000	
\sqrt{n}	3 μ s	10 μ s	30 μ s	1 ms	
$n + 5$	15 μ s	105 μ s	1 ms	1 s	
$2n$	20 μ s	200 μ s	2 ms	2 s	
n^2	100 μ s	10 ms	1 s	1000000 s	~ 12 gg
$n^2 + 2$	102 μ s	10 ms	1 s	1000000 s	~ 12 gg
n^3	1 ms	1 s	1000 s	1×10^{12} s	~ 300 secoli
2^n	1 ms	1×10^{24} s	1×10^{295} s	#NUM!	~ 3×10^{301016} secoli

Classi di complessità

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro.

n	10	100	1000	1000000	
\sqrt{n}	3 μ s	10 μ s	30 μ s	1 ms	
$n + 5$	15 μ s	105 μ s	1 ms	1 s	
$2n$	20 μ s	200 μ s	2 ms	2 s	
n^2	100 μ s	10 ms	1 s	1000000 s	~ 12 gg
$n^2 + 2$	102 μ s	10 ms	1 s	1000000 s	~ 12 gg
n^3	1 ms	1 s	1000 s	1×10^{12} s	~ 300 secoli
2^n	1 ms	1×10^{24} s	1×10^{295} s	#NUM!	~ 3×10^{301016} secoli

Classi di complessità

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro.

n	10	100	1000	1000000	
\sqrt{n}	3 μ s	10 μ s	30 μ s	1 ms	
$n + 5$	15 μ s	105 μ s	1 ms	1 s	
$2n$	20 μ s	200 μ s	2 ms	2 s	
n^2	100 μ s	10 ms	1 s	1000000 s	~ 12 gg
$n^2 + 2$	102 μ s	10 ms	1 s	1000000 s	~ 12 gg
n^3	1 ms	1 s	1000 s	1×10^{12} s	~ 300 secoli
2^n	1 ms	1×10^{24} s	1×10^{295} s	#NUM!	~ 3×10^{301016} secoli

Classi di complessità

Ci serve **un modo matematicamente rigoroso**
per confrontare queste curve tra loro.

n	10	100	1000	1000000	
\sqrt{n}	3 μ s	10 μ s	30 μ s	1 ms	
$n + 5$	15 μ s	105 μ s	1 ms	1 s	
$2n$	20 μ s	200 μ s	2 ms	2 s	
n^2	100 μ s	10 ms	1 s	1000000 s	~ 12 gg
$n^2 + 2$	102 μ s	10 ms	1 s	1000000 s	~ 12 gg
n^3	1 ms	1 s	1000 s	1×10^{12} s	~ 300 secoli
2^n	1 ms	1×10^{24} s	1×10^{295} s	#NUM!	~ 3×10^{301016} secoli

Conta l'**operazione dominante!!!**

Introduciamo il concetto di **O-grande**.

È un **criterio matematico** per partizionare gli algoritmi
in **classi di complessità**.

$$\begin{array}{ccc} T_A(n) = n^2 + 3n + 6 & \longrightarrow & O(n^2) \\ T_B(n) = 2n^2 & \nearrow & \end{array}$$

$n^2 + \cancel{3n} + \cancel{6}$ Togliamo tutto ciò che non c'entra con n

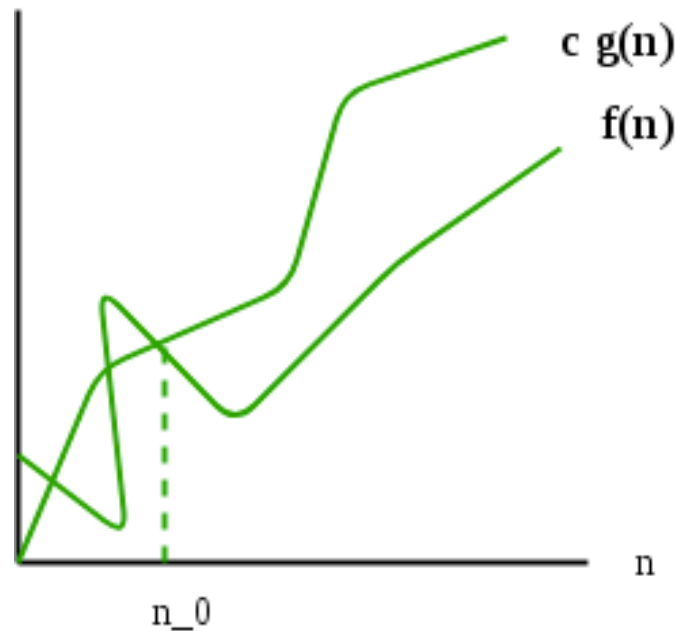
$n^2 + \cancel{n}$ Lasciamo solo l'operazione dominante (grado maggiore)

$$n^2 \rightarrow O(n^2)$$

È possibile trovare una **formulazione rigorosa** per questo criterio?

O-grande: definizione formale

Si dice che $f(n)$ (che per noi è $T(n)$) è di **ordine** $g(n)$, e si scrive $T(n) \in O(g(n))$,
se esistono delle costanti $c > 0$ e $n_0 > 0$ tali che
 $T(n) \leq c \cdot g(n)$ per qualsiasi $n > n_0$ (cioè trascurando la prima parte).



$$f(n) = O(g(n))$$

Si può anche dire che $T(n) \in O(g(n))$ se $\frac{T(n)}{g(n)} \leq c$ per $n > n_0$

Esempio di calcolo O-grande

$$\text{Bubble Sort: } T(n) = 7n^2 + 3n + 5$$

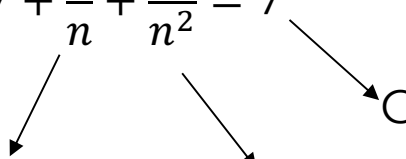
Ingegneristicamente:

$$\cancel{7}n^2 + \cancel{3}n + \cancel{5}$$

$$O(n^2)$$

Matematicamente:

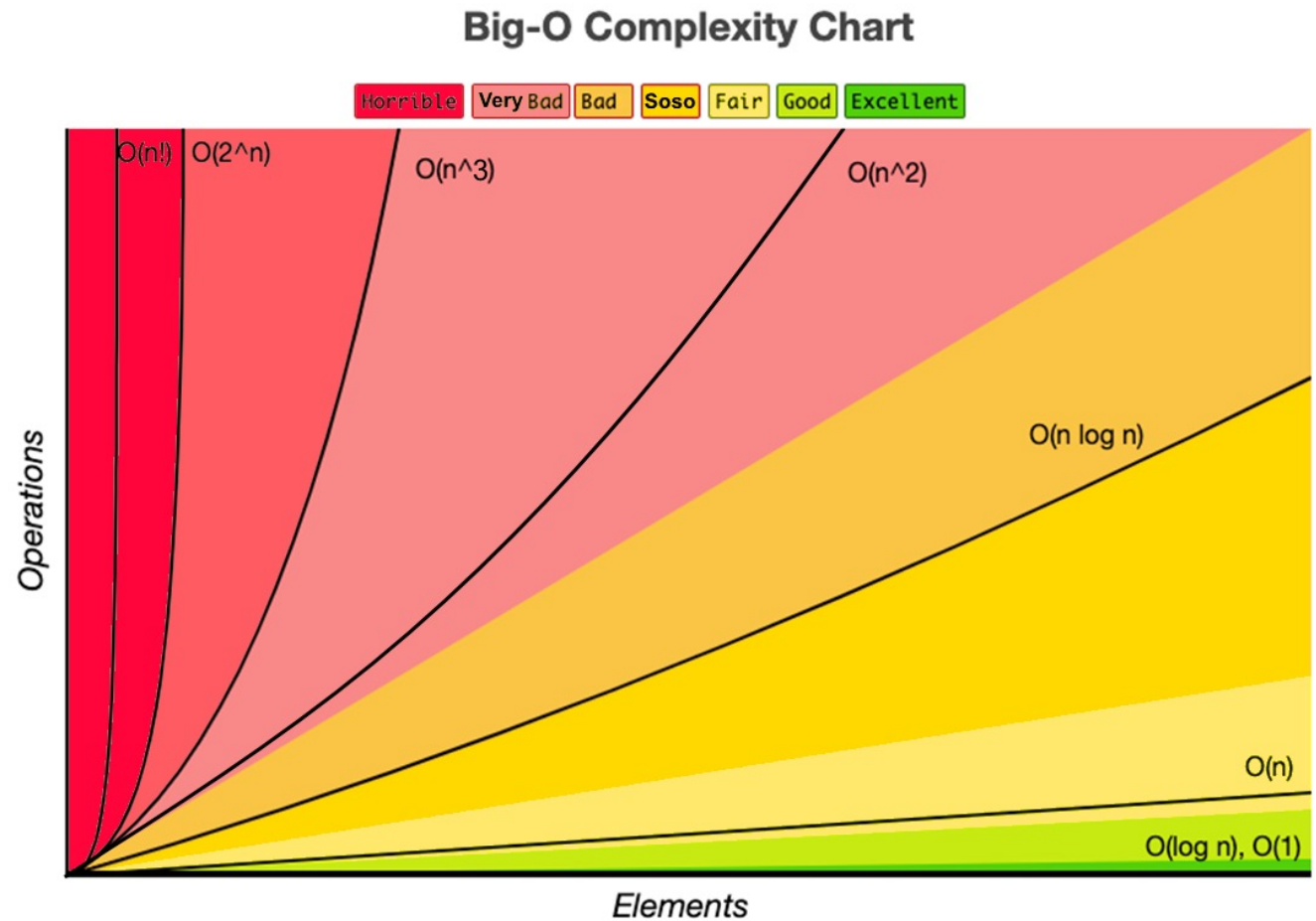
$$\lim_{n \rightarrow \infty} \frac{7n^2 + 3n + 5}{n^2} = \lim_{n \rightarrow \infty} 7 + \frac{3}{n} + \frac{5}{n^2} = 7$$



Tende a 0 Tende a 0 Costante

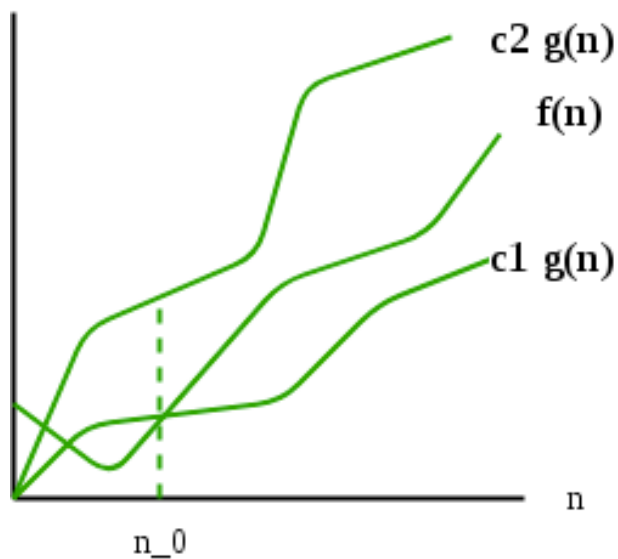
Classi di complessità 2

n	Classe
$\log(n)$	$O(\log(n))$
\sqrt{n}	$O(\sqrt{n})$
$n + 5$	$O(n)$
$2n$	$O(n)$
$n \log(n)$	$O(n \log(n))$
n^2	$O(n^2)$
$n^2 + 2$	$O(n^2)$
n^3	$O(n^3)$
2^n	$O(2^n)$
$n!$	$O(n!)$

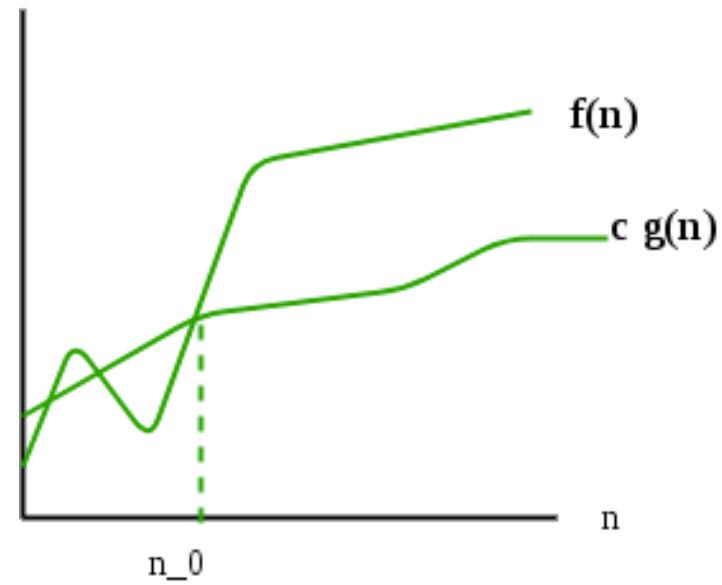


Definizione di Θ e Ω

- **Θ – Theta** (caso medio)
 - $f(n) \in \Theta(g(n))$: esistono delle costanti $c_1 > 0$, $c_2 > 0$ e $n_0 > 0$ tali che $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n > n_0$
- **Ω – Omega** (caso migliore)
 - $f(n) \in \Omega(g(n))$: se esistono delle costanti $c > 0$ e $n_0 > 0$ tali che $0 \leq c \cdot g(n) \leq f(n)$ per ogni $n > n_0$



$f(n) = \text{theta}(g(n))$



$f(n) = \text{Omega}(g(n))$

O-Grande, Theta o Omega?

- **ES. 1**

- $f(n) = n(\log n)^2$
- $g(n) = n^2/\log n$
- $f(n) \in \Theta(g(n))?$
- $f(n) \in O(g(n))?$
- $f(n) \in \Omega(g(n))?$

O-Grande, Theta o Omega?

- **ES. 1**

- $f(n) = n(\log n)^2$
- $g(n) = n^2/\log n$

$$\frac{f(n)}{g(n)} = \frac{n(\log n)^2}{n^2/\log n} = \frac{(\log n)^3}{n} \rightarrow 0 \text{ for } n \rightarrow \infty$$

- $f(n) \in O(g(n))$
- $g(n) \in \Omega(f(n))$

O-Grande, Theta o Omega?

- **ES. 2**

- $f(n) = 100n + \log n$
- $g(n) = n + (\log n)^2$
- $f(n) \in \Theta(g(n))?$
- $f(n) \in O(g(n))?$
- $f(n) \in \Omega(g(n))?$

O-Grande, Theta o Omega?

- **ES. 2**
- $f(n) = 100n + \log n$
- $g(n) = n + (\log n)^2$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{100n + \log n}{n + (\log n)^2} = \lim_{n \rightarrow \infty} \frac{100 + (\log n)/n}{1 + (\log n)^2/n} = 100.$$

- $f(n) \in \Theta(g(n))$

- **ES. 3**

- Vogliamo dimostrare che

$$n^2 - 3n \in \Theta(n^2)$$

- Quindi che

$$\exists n_0, c_1, c_2 > 0 \mid c_1 n^2 \leq n^2 - 3n \leq c_2 n^2, \forall n > n_0$$

Calcolo costanti caso Theta

- **ES. 3**

- Dividendo per n^2 si ottiene: $c_1 \leq 1 - \frac{2}{n} \leq c_2$

- Risolvendo le due disequazioni, si ha che:

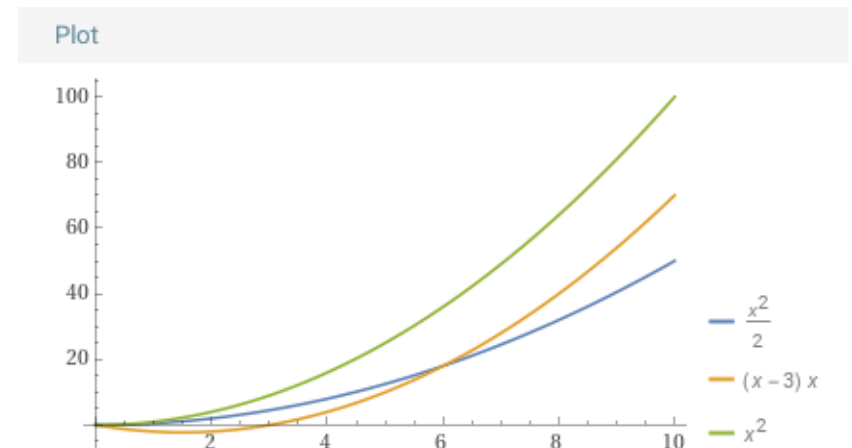
- 1) Quella di destra è verificata $\forall n \geq 1$, posto $c_2 \geq 1$

- 2) Quella di sinistra invece è verificata per alcuni valori di c_1 ,
posto $n > 3$

Per semplicità, basta quindi scegliere $n_0 = 6$ per ottenere $c_1 = \frac{1}{2}$

- Quindi, per $\forall n > n_0 = 6$, si ha

$$\frac{1}{2}n^2 \leq n^2 - 3n \leq n^2$$



O-Grande, Theta o Omega?

- **ES. 4**

- $f(n) = \log n$
- $g(n) = \log n^2$

- $f(n) \in \Theta(g(n))?$
- $f(n) \in O(g(n))?$
- $f(n) \in \Omega(g(n))?$

O-Grande, Theta o Omega?

- **ES. 4**

- $f(n) = \log n$
- $g(n) = \log n^2$
- $\log n^2 = 2\log n$
- $f(n) \in \Theta(g(n))$

O-Grande, Theta o Omega?

- **ES. 5**

- $f(n) = \sqrt{n}$
- $g(n) = (\log n)^5$

- $f(n) \in \Theta(g(n))?$
- $f(n) \in O(g(n))?$
- $f(n) \in \Omega(g(n))?$

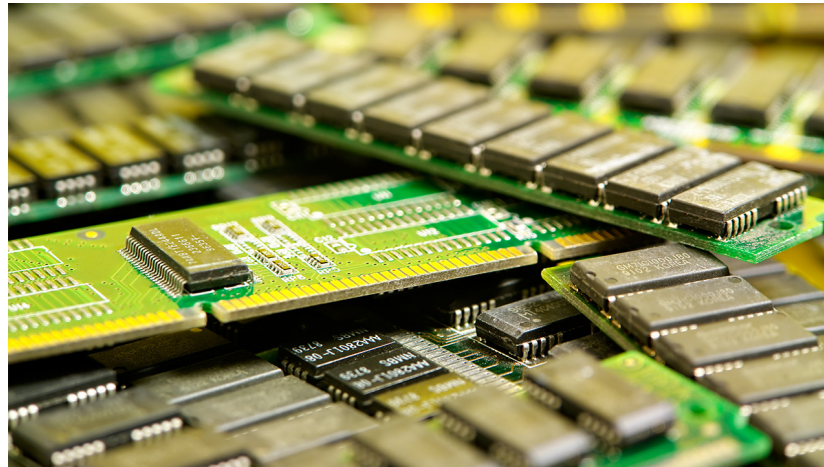
O-Grande, Theta o Omega?

- **ES. 5**

- $f(n) = \sqrt{n}$
- $g(n) = (\log n)^5$
- Polinomiale cresce sempre più del logaritmico
- $f(n) \in \Omega(g(n))$
- $g(n) \in O(f(n))$

Complessità spaziale

La **complessità spaziale** di un algoritmo è una misura assoluta della quantità di memoria necessaria alla sua esecuzione.



Calcolo della complessità spaziale

La **complessità spaziale** di un algoritmo è una misura assoluta della quantità di memoria necessaria alla sua esecuzione.

- Cambiano un po' le regole di calcolo, si distinguono:
 - azioni che occupano una **quantità costante** di memoria:
inizializzazione di variabili (booleane, intere, a virgola mobile)

```
int i;
```

```
bool ok;
```

- azioni che occupano una **quantità variabile** di memoria:
allocazione di vettori e stringhe

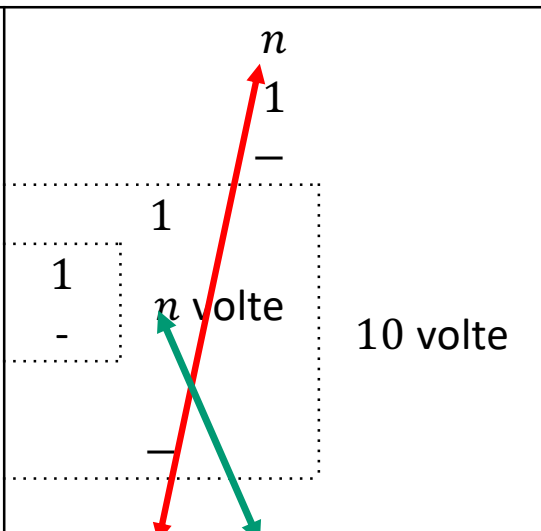
```
int[] array = {1, 2, 3};
```

```
int[][] array = { {1, 1, 1}, {2, 2, 2} };
```

- **memoria ausiliaria**: la memoria occupata dal codice stesso e dalle sue funzioni. Può essere una **costante** o assumere un ruolo **rilevante (ricorsione!)**.

Esempio 1

- Supponiamo che l'array **j** costituisca il nostro **input** di dimensione **n**.

<pre> int j[] = {0, 0, 0}; int i = 0; while (i < 10) { for (int idx = 0; idx < j.length; idx++) { int temp = i + j[idx]; j[idx] = temp; } i = i + 1; } </pre>	
Somma:	$1 + n + O(1) \rightarrow O(n)$

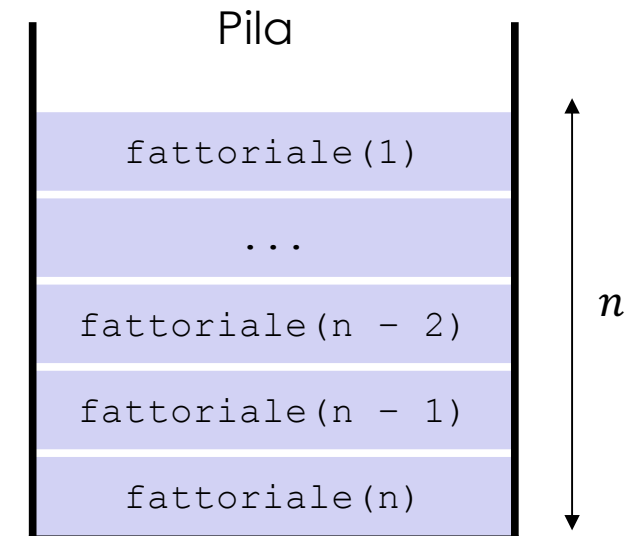
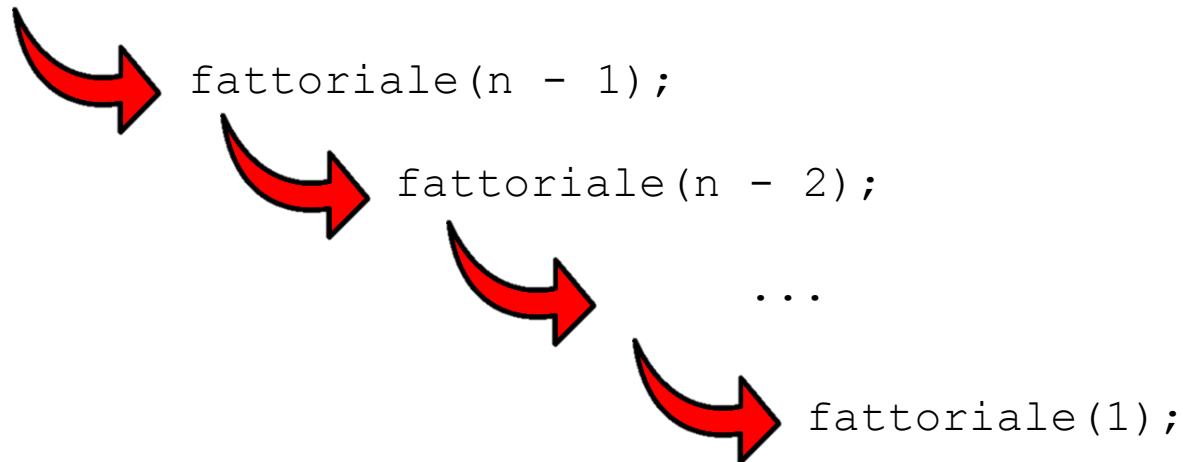
Osservazione: in questo esempio i cicli iterativi non accumulano spazio occupato in memoria ad ogni iterazione!

Esempio 2: fattoriale ricorsivo

```
fattoriale(n) {  
    if (n <= 1) {  
        return 1;  
    }  
    else {  
        return n*fattoriale(n - 1);  
    }  
}
```

1

fattoriale(n);



Complessità spaziale: $O(n)$

Esempio 3: fattoriale iterativo

```
long factorial(int n)
{
    long fact = 1;

    for (int i = 1; i <= n; i++)
    {
        fact = fact * i;
    }

    return fact;
}
```

1

1

1

Complessità spaziale: $O(1)$

$$S(n) = 2S\left(\frac{n}{2}\right) + n + O(1)$$

60

Esempio 3: merge sort ricorsivo

```

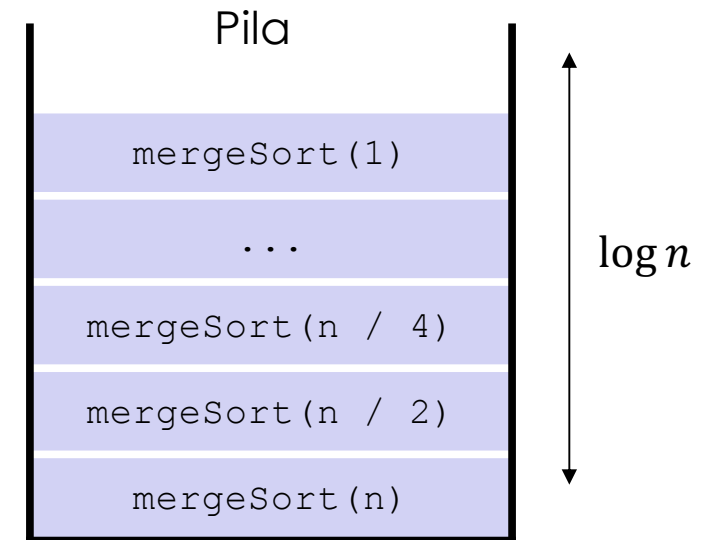
public static void mergeSort(int[] v)
{
    // caso base
    if (v.length < 2) return;

    // dividiamo (circa) a meta'
    int mid = v.length / 2;

    int[] left = new int[mid];
    int[] right = new int[v.length - mid];
    System.arraycopy(v, 0, left, 0, mid);
    System.arraycopy(v, mid, right, 0, v.length - mid);

    // passi ricorsivi: problemi piu' semplici
    // si tratta di doppia ricorsione
    mergeSort(left);
    mergeSort(right);

    // fusione
    merge(v, left, right);
}
  
```



Quante chiamate ricorsive ci sono?

$$S(n) \cong n + \frac{n}{2} + \frac{n}{4} + \dots + 1 \rightarrow \log n \rightarrow O(\log n)$$

Qual è il termine dominante per ogni chiamata ricorsiva?

$$\rightarrow O(n)$$

Per il Merge Sort Ricorsivo: $S(n) = O(n)O(\log n) = O(n \log n)$

Esempio 3: merge sort ricorsivo

Vediamo ora una **diversa** implementazione del **Merge Sort ricorsivo**, partendo dalla funzione di merge (N.B. è diversa da quella dell'implementazione precedente)...

<pre>//function to merge the two halves arr[l..m] and arr[m+1..r] of array arr[] static void merge(int arr[], int l, int m, int r) { //find sizes of two subarrays to be merged int i, j, k; int n1 = m - l + 1; int n2 = r - m; //create temp arrays int L[] = new int[n1]; int R[] = new int[n2]; //copy data to temp arrays L[] and R[] for (i = 0; i < n1; i++) L[i] = arr[l + i]; for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];</pre>	<p>$O(1)$</p> <p>$O(1)$</p> <p>$O(1)$</p> <p>$n/2$</p> <p>$n/2$</p>
--	---

Esempio 3: merge sort ricorsivo

```
//merge the temp arrays back into arr[l..r]

//initial indexes of first and second subarrays
i = 0;
j = 0;
//initial index of merged subarray array
k = l;

//take i-th element of n1 if it is smaller than the j-th one of n2,
//otherwise take j-th element of n2
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    }
    else {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

Esempio 3: merge sort ricorsivo

```
//copy the remaining elements of L[], if there are any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}
//copy the remaining elements of R[], if there are any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}
```

Per la funzione merge: $S(n) = \frac{n}{2} + \frac{n}{2} + O(1) \Rightarrow S(n) = O(n)$

Esempio 3: merge sort ricorsivo

Analizziamo il metodo mergeSort ricorsivo vero e proprio

<pre>//recursive implementation of mergeSort static void mergeSortRecursive(int arr[], int l, int r) { if (l < r) { //find the middle point int m = (l + r) / 2; //sort first and second halves mergeSortRecursive(arr, l, m); mergeSortRecursive(arr, m + 1, r); //merge the sorted halves merge(arr, l, m, r); } }</pre>	$O(1)$ $S(n/2)$ $S(n/2)$ $O(n)$
---	--

Per il Merge Sort Ricorsivo:
$$S(n) = 2S\left(\frac{n}{2}\right) + O(n) + O(1)$$

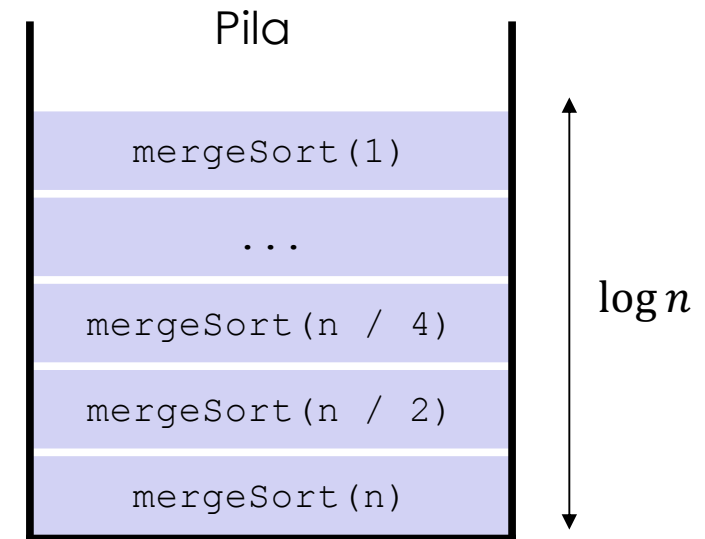
Esempio 3: merge sort ricorsivo

```

//recursive implementation of mergeSort
static void mergeSortRecursive(int arr[], int l, int r)
{
    if (l < r)
    {
        //find the middle point
        int m = (l + r) / 2;

        //sort first and second halves
        mergeSortRecursive(arr, l, m);
        mergeSortRecursive(arr, m + 1, r);

        //merge the sorted halves
        merge(arr, l, m, r);
    }
}
  
```



Quante chiamate ricorsive
ci sono?

$\xleftarrow{\log n \text{ volte}}$

$$S(n) \cong n + \frac{n}{2} + \frac{n}{4} + \dots + 1 \rightarrow \log n \rightarrow O(\log n)$$

⇒ Per il Merge Sort Ricorsivo:

$$S(n) = O(\log n) + O(n) = O(n)$$

Esempio 4: merge sort iterativo

//iterative implementation of mergeSort	
static void mergeSortIterative(int arr[], int n)	$O(1)$
{	
//for curr size of subarrays to be merged curr_size varies from 1 to n/2	$O(1)$
int curr_size;	
//for picking starting index of left subarray to be merged	$O(1)$
int left_start;	
//merge subarrays: from subs of size 1 to sorted subs of size 2	
//from subs of size 2 to sorted subs of size 4, and so on	
for (curr_size = 1; curr_size <= n-1; curr_size = 2*curr_size)	
{	
//pick starting point of different subarrays of current size	
for (left_start = 0; left_start < n-1; left_start += 2*curr_size)	
{	
//find ending point of left subarray	
//mid+1 is starting point of right	
int mid = Math.min(left_start + curr_size - 1, n-1);	$O(1)$
int right_end = Math.min(left_start + 2*curr_size - 1, n-1);	$O(1)$
//merge subarrays arr[left_start...mid] and arr[mid+1...right_end]	
merge(arr, left_start, mid, right_end);	$O(n)$
}	
}	
}	
⇒ Per il Merge Sort Iterativo: $S(n) = O(n)$	

Analisi di un problema

- Come ultimo spunto, è opportuno però domandarsi: cosa succede in una situazione pratica?
- Molto spesso non si presenta il caso in cui si parte dall'analisi della complessità di un codice...

si verifica la situazione in cui bisogna sviluppare un algoritmo per risolvere un dato problema!



Il punto di partenza è quindi **l'analisi del problema!**

- Come può essere utile quanto visto in merito alla **complessità computazionale** per l'analisi di un problema?

Complessità di un problema

Un problema ha **complessità** $O(n)$ se si può dimostrare che gli algoritmi per risolverlo devono avere **almeno** complessità $O(n)$.

Notazione O grande	Nome	Esempio
$O(1)$	Costante	Determinare se numero è pari o dispari
$O(\log n)$	Logaritmico	Cercare un elemento in un array ordinato tramite ricerca binaria
$O(\sqrt{n})$	Radice quadrata	Trovare il numero di divisori di un numero
$O(n)$	Lineare	Cercare il massimo in un array non ordinato
$O(n \log n)$	Linearitmico (Linearithmic)	Ordinare gli elementi di un array tramite merge sort
$O(n^2)$	Quadratico	Ordinare gli elementi di un array tramite bubble sort
$O(n^3)$	Cubico	Risolvere un'equazione in tre variabili
$O(2^n)$	Esponenziale	Determinare tutti i sottoinsiemi di un insieme
$O(n!)$	Fattoriale	Trovare tutte le possibili permutazioni di una stringa

Complessità di un problema

Un problema ha **complessità** $O(n)$ se si può dimostrare che gli algoritmi per risolverlo devono avere **almeno** complessità $O(n)$.

- La complessità è una caratteristica intrinseca!
- È fondamentale cercare di studiare la complessità del problema **prima** di iniziare a sviluppare un algoritmo per risolverlo.
- Fatto questo, si può **valutare** la dimensione dei dati su cui si basa il problema, a cui corrisponde la **dimensione dell'input n** dell'algoritmo:
 - Quanto è grande l'input per il nostro problema?
 - Quanto potrebbe diventare grande un domani?
 - Potrebbero verificarsi problemi prestazionali **rilevanti**?

Complessità di un problema

- In questo modo è possibile:
 - 1) individuare qual è la **classe di algoritmi** adatta alla soluzione.
 - 2) avendo considerato la **dimensione dell' input**, capire se la soluzione rispetta eventuali **vincoli temporali o spaziali**, o se invece il problema deve essere riformulato.
 - 3) capire se l'**algoritmo** successivamente implementato per risolvere il problema ha una **complessità adeguata**.
- Spesso, specie se il problema è complesso, non esiste una soluzione immediata o ideale. Bisogna:
 1. Cercare di scomporlo in sottoproblemi più semplici.
 2. Trovare un compromesso tra i requisiti di partenza e l'ottimizzazione dell'algoritmo.

*“Premature optimization
is the root of all evil” -
Donald Knuth*

V.S.

```
void RandomSort(int[] i) {  
  
    while(!isSorted(i)) {  
        shuffle(i);  
    }  
}
```

- ✓ La **complessità computazionale** di un algoritmo è una misura assoluta delle risorse necessarie alla sua esecuzione.
- ✓ La complessità computazionale è espressa in funzione della **dimensione dell'input** e serve ad analizzare l'**efficienza** di un algoritmo a prescindere da chi lo esegue.
- ✓ La complessità computazionale di un algoritmo può essere studiata tramite l'analisi della sua **complessità temporale** e della sua **complessità spaziale**.
- ✓ Lo studio della **complessità di un problema** è spesso il punto di partenza per progettare un algoritmo che ne costituisca una soluzione efficiente.

Grazie a tutti per l'attenzione!

e-mail: giulio.martini@igi.cnr.it