complessità computazionale

classificazione degli algoritmi

- algoritmi ***sequenziali***
  - eseguono un solo passo alla volta
- algoritmi ***paralleli***
  - possono eseguire più passi per volta
- algoritmi ***deterministici***
  - ad ogni punto di scelta, intraprendono una sola via determinata dalla valutazione di un'espressione
- algoritmi ***probabilistici***
  - ad ogni punto di scelta, intraprendono una sola via determinata a caso
- algoritmi ***non deterministici***
  - ad ogni punto di scelta, esplorano tutte le vie contemporaneamente

- dato un problema, possono esistere ***più algoritmi*** che sono ***corretti*** rispetto ad esso
- *… e un numero illimitato di algoritmi errati :(*
- gli algoritmi corretti possono essere ***confrontati*** rispetto alla loro complessità o ***efficienza computazionale***

complessità di un algoritmo in base all'uso delle risorse

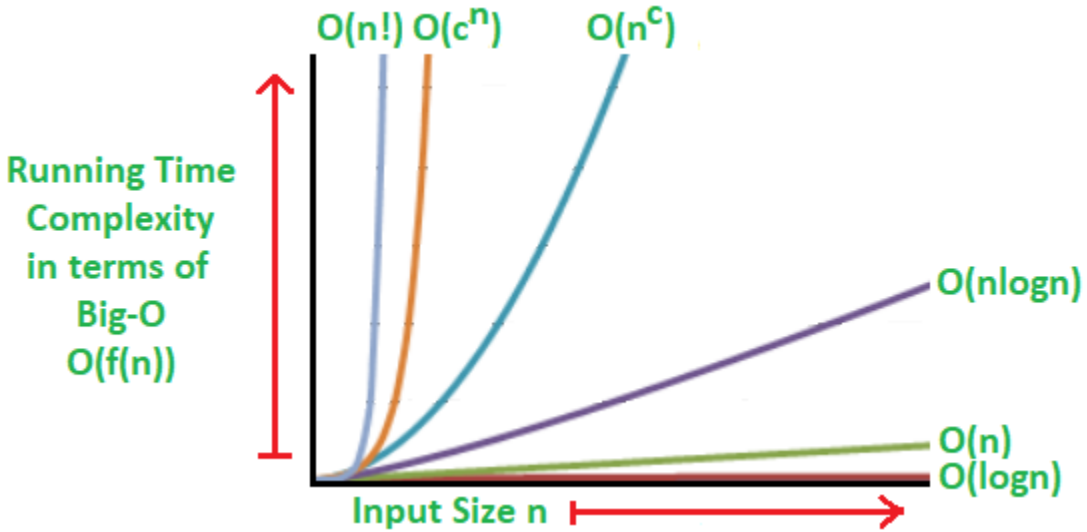
- l'algoritmo viene valutato in base alle ***risorse*** utilizzate durante la sua esecuzione:
  - ***tempo*** di calcolo
  - spazio di ***memoria*** (risorsa riusabile)
  - ***banda*** trasmissiva (risorsa riusabile)

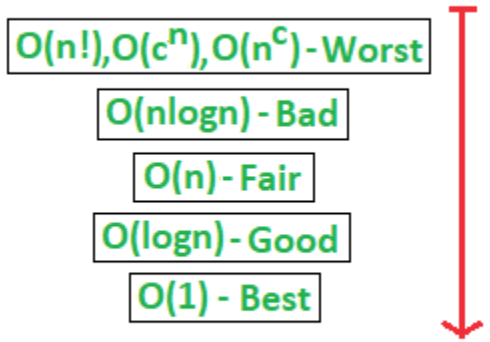
domanda

- ***esiste*** sempre un algoritmo risolutivo per un problema?

problemi decidibili e indecidibili

- problema ***decidibile***
  - se esiste un algoritmo che produce la ***soluzione in tempo finito*** per ogni istanza dei dati di ingresso del problema
- problema ***indecidibile***
  - se ***non*** esiste nessun algoritmo che produce la ***soluzione in tempo finito*** per ogni istanza dei dati di ingresso del problema





complessità temporale

- confronto fra algoritmi che risolvono lo stesso problema
  - si valuta il ***tempo di esecuzione*** (in numero di passi) in modo indipendente dalla tecnologia dell'esecutore
- in molti casi la ***complessità*** è legata al tipo o al numero dei dati di input
  - ad esempio la ricerca di un valore in una struttura ordinata dipende dalla dimensione della struttura
- il tempo è espresso in funzione della ***dimensione dei dati in ingresso T(n)***
  - per confrontare le funzioni tempo ottenute per i vari algoritmi si considerano le funzioni asintotiche

funzione asintotica

- data la funzione polinomiale ***f(n)*** che rappresenta il tempo di esecuzione dell'algoritmo al variare della dimensione n dei dati di input
- la funzione asintotica ***ignora le costanti moltiplicative*** e i ***termini non dominanti*** al crescere di n
  - **esempio: $f(x) = 3x^4 + 6x^2 + 10$**
  - **funzione asintotica = x^4**
- l'***approssimazione*** di una funzione con una funzione asintotica è molto utile per semplificare i calcoli
- la notazione asintotica di una funzione descrive il comportamento in modo semplificato, ignorando dettagli della formula
  - nell'esempio: per valori sufficientemente alti di x il comportamento di **$f(x) = 3x^4 + 6x^2 + 10$** è ***approssimabile*** con la funzione **$f(x) = x^4$**

- il tempo di esecuzione può essere calcolato in caso
  - ***pessimo*** – dati d'ingresso che massimizzano il tempo di esecuzione
  - ***ottimo*** – dati d'ingresso che minimizzano il tempo di esecuzione
  - ***medio*** – somma dei tempi pesata in base alla loro probabilità

complessità temporale

- ***O(1)*** Complessità ***costante***
- ***O(log n)*** Complessità ***logaritmica***
- ***O(n)*** Complessità ***lineare***
- ***O(n*log n)*** Complessità ***pseudolineare***
- $O(n^2)$ Complessità ***quadratica***
- $O(n^k)$ Complessità ***polinomiale***
- $O(a^n)$ Complessità ***esponenziale***

algoritmi non ricorsivi

- calcolo della complessità
  - vengono in pratica "***contate***" le operazioni eseguite
- calcolo della complessità di algoritmi non ricorsivi
  - il tempo di esecuzione di un'istruzione di ***assegnamento*** che non contenga chiamate a funzioni è ***1***
  - il tempo di esecuzione di una ***chiamata*** ad una ***funzione*** è ***1*** **+** il ***tempo*** di esecuzione della ***funzione***
  - il tempo di esecuzione di un'istruzione di selezione è il tempo di valutazione dell'***espressione*** **+** il tempo ***massimo*** fra il tempo di esecuzione del ramo ***True*** e del ramo ***False***
  - il tempo di esecuzione di un'istruzione di ***ciclo*** è dato dal tempo di valutazione della ***condizione*** **+** il tempo di esecuzione del ***corpo*** del ciclo moltiplicato per il numero di ***volte*** in cui questo viene eseguito

es. calcolo del fattoriale complessità lineare

A. Ferrari

```
int fattoriale(int n) {
  int fatt;
  fatt = 1;
  for (int i = 2; i <= n; i++)
    fatt  = fatt * i;
  return fatt;
}
```

$$T(n) = 1 + (n-1)(1+1+1)+1 = 3n - 1 = O(n)$$

- ***confrontare algoritmi corretti*** che risolvono lo stesso problema, allo scopo di scegliere quello ***migliore*** in relazione a uno o più parametri di valutazione

valutazione con un parametro

- se si ha a disposizione ***un solo parametro*** per valutare un algoritmo, per esempio il tempo d'esecuzione, è semplice la scelta: il più veloce
- ogni ***altra caratteristica non*** viene ***considerata***

valutazione con più parametri

- nel caso di ***due parametri*** normalmente si considera
- il ***tempo***
  - numero di passi (istruzioni) che occorrono per produrre il risultato finale
    - passi e non secondi o millisecondi perché il tempo varia al variare delle potenzialità del calcolatore
- lo ***spazio***
  - occupazione di memoria

terminologia

O (O grande) equivale al simbolo **<=** corrisponde a “al più come”

O(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è minore o uguale a f(n)”

o (o piccolo) equivale al simbolo **<**

o(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è strettamente minore a f(n)”

Θ (teta) corrispondente al simbolo **=**

Θ(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è uguale a f(n)”

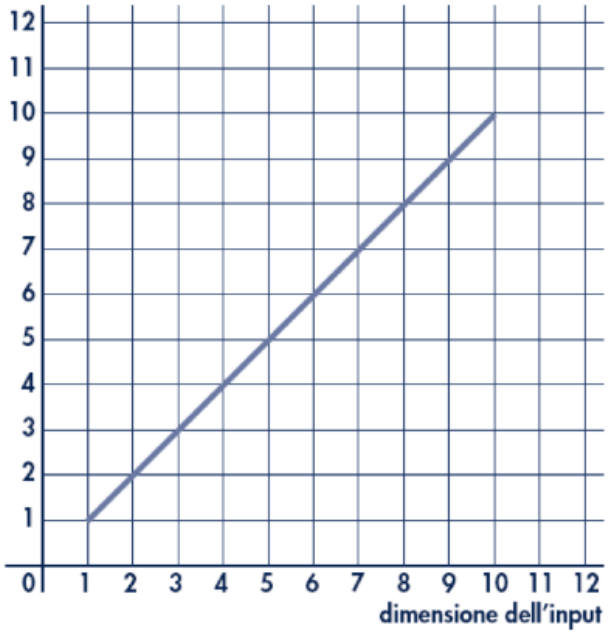
Ω (omega grande) equivale al simbolo **>=**

Ω(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è maggiore o uguale a f(n)”

ω (omega piccolo) equivale al simbolo **>**

ω(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è strettamente maggiore di f(n)”

- l'algoritmo ha complessità ***O(n)***
- esempio:
  - algoritmo di ricerca lineare (sequenziale) di un elemento in una lista

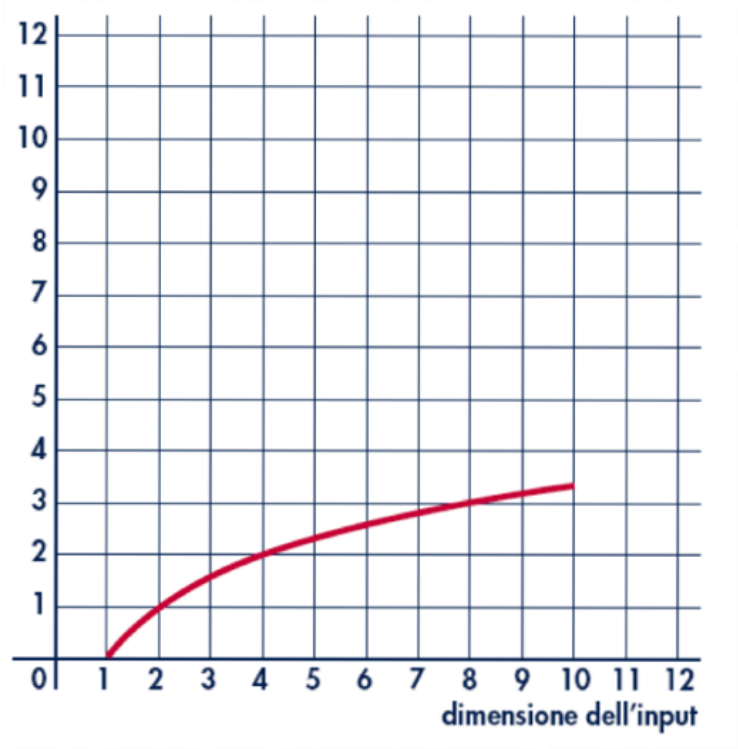


ricerca lineare


```
def linear_search(v: list, value) -> int:
    '''v: not necessarily sorted'''

    for i in range(len(v)):
        if v[i] == value:
            return i

    return -1
```

complessità logaritmica

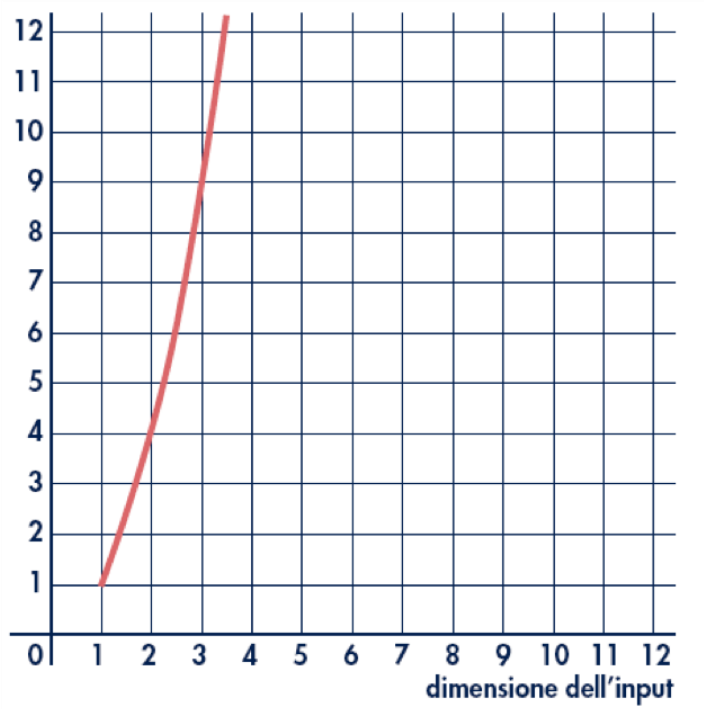
- esempio ricerca ***dicotomica*** in una lista ordinata
  - la ricerca dicotomica ha complessità $O(log_2(n))$



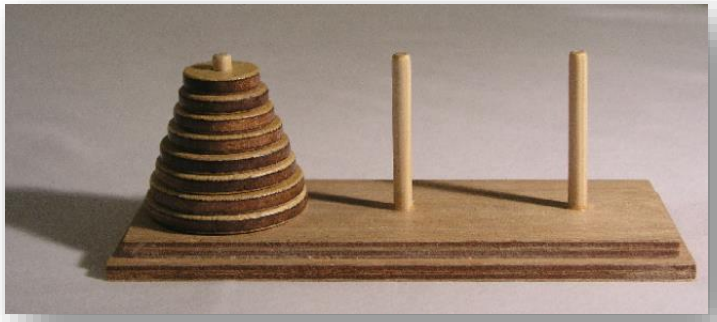
```
def binary_search(v: list, value) -> int:
    '''v: sorted list'''

    begin, end = 0, len(v)
    while begin < end:
        middle = (begin + end) // 2
        if v[middle] > value:
            end = middle
        elif v[middle] < value:
            begin = middle
        else:
            return middle

    return -1
```

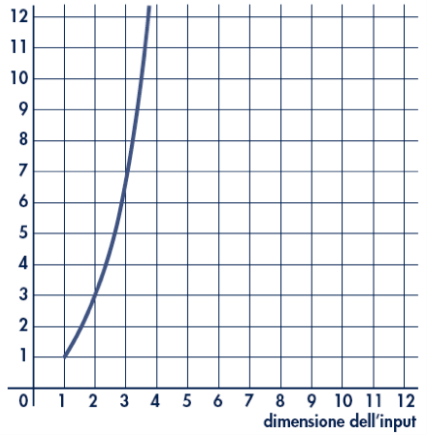
complessità quadratica

- un esempio è l'algoritmo di ordinamento ***bubblesort*** eseguito su un array di elementi
  - l'algoritmo ha complessità $O(n^2)$



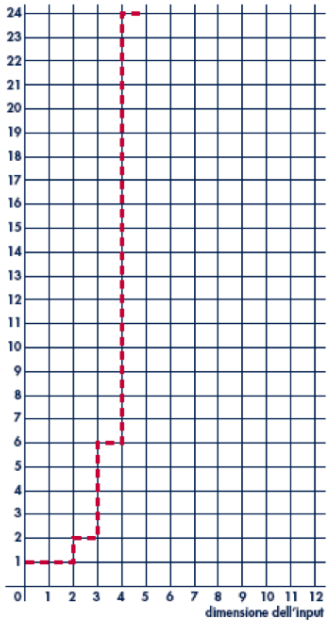
- l'algoritmo della ***Torre di Hanoi*** ha complessità $\Omega(2^n)$
  - *la Torre di Hanoi è un rompicapo matematico composto da tre paletti e un certo numero di dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti.*
  - *il gioco inizia con tutti i dischi incolonnati su un paletto in ordine decrescente, in modo da formare un cono.*
  - *lo scopo è portare tutti dischi sull'ultimo paletto, potendo spostare solo un disco alla volta e potendo mettere un disco solo su uno più grande, mai su uno più piccolo*





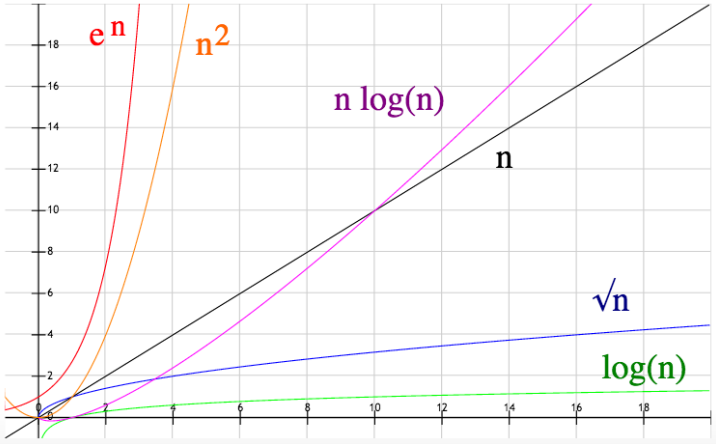
complessità fattoriale

- è quella che cresce ***più velocemente*** rispetto a tutte le precedenti
- esempio: algoritmo che calcola tutti gli ***anagrammi*** di una parola di n lettere distinte ha complessità ***Θ(n!)***



confronto

A. Ferrari



n	n/2	log(n)
10	5	3,321928
20	10	4,321928
30	15	4,906891
40	20	5,321928
50	25	5,643856
60	30	5,906891
70	35	6,129283
80	40	6,321928
90	45	6,491853
100	50	6,643856
300	150	8,228819
1000	500	9,965784
10000	5000	13,28771
100000	50000	16,60964

un esempio

SUDOKU

sudoku

A. Ferrari

- il ***sudoku*** (数独) è un gioco di logica nel quale al giocatore viene proposta una ***griglia*** di 9×9 celle, ciascuna delle quali può contenere un numero da 1 a 9, oppure essere vuota
- la griglia è suddivisa in 9 ***righe*** orizzontali, nove ***colonne*** verticali e, da bordi in neretto, in 9 "sottogriglie", chiamate ***regioni***, di 3×3 celle contigue
- le griglie proposte al giocatore hanno da 20 a 35 celle contenenti un numero
- scopo del gioco è quello di ***riempire le caselle bianche*** con numeri da 1 a 9, in modo tale che in ogni riga, colonna e regione siano presenti tutte le cifre da 1 a 9 e, pertanto, ***senza ripetizioni***

inizio del gioco

A. Ferrari

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

gioco risolto

A. Ferrari

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

soluzione ingenua

A. Ferrari

- si inserisce un numero in ogni cella vuota (inizialmente 1 in tutte le celle)
- si verifica che le regole del gioco siano rispettate
- se qualche regola non è rispettata si prova con altri numeri (si passa nell'ultima cella al numero successivo)
- complessità
  - il numero totale di celle è 81
  - supponendo riempite 31 celle ne rimangono 50
  - il numero di combinazioni possibili da provare è 9^{50} (circa $5*10^{47}$)
  - supponendo 1 nanosecondo per formulare una combinazione il tempo previsto è circa . . .

… calcoliamo

- 1 anno = 365 giorni
- 1 anno = 8760 ore
- 1 anno = 525600 minuti
- 1 anno = 3.154×10^7 secondi
- 1 anno = 3.154×10^{16} nanosecondi
- $5 \times 10^{47} / 3.154 \times 10^{16} =$ ***10^{31} anni***

- *comparsa dell'uomo sulla terra 2×10^6 anni fa*