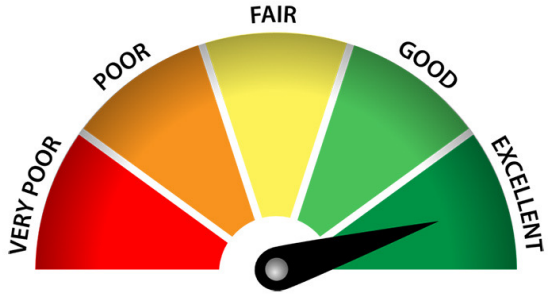
UNIVERSITÀ
DI PARMA

complessità degli algoritmi



algoritmo

- matematico persiano Muhammad ***al-Khwarizmi*** (IX secolo)
- un ***algoritmo*** è una *sequenza finita di passi interpretabili da un esecutore*
- l'esecuzione di un algoritmo potrebbe richiedere un tempo non necessariamente finito
- un algoritmo ***non*** deve necessariamente essere espresso in un ***linguaggio di programmazione***
- l'algoritmo si trova ad un livello di ***astrazione*** più alto rispetto ad ogni programma che lo implementa

UNIVERSITÀ DI PARMA

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

problemi e algoritmi

- 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***

UNIVERSITÀ DI PARMA

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

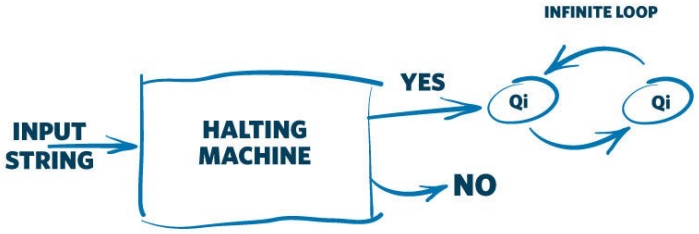
- ***esiste*** sempre un algoritmo risolutivo per un problema?

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

esempio di problema indecidibile

- il problema dell'arresto (Halting Problem):
  - dato un programma e un input, esiste un algoritmo che decide sempre se il programma terminerà o andrà in loop infinito?
  - Alan Turing ha dimostrato che non esiste un algoritmo generale che possa stabilirlo per ogni possibile programma e input



problemi trattabili e intrattabili

- problemi ***intrattabili***
  - ***non*** sono ***risolvibili*** in tempo polinomiale nemmeno da un ***algoritmo non deterministico***
- problemi ***trattabili***
  - si dividono in due categorie
    - ***P*** - insieme dei problemi risolvibili in tempo polinomiale da un algoritmo ***deterministico***
    - ***NP*** - insieme dei problemi risolvibili in tempo polinomiale da un algoritmo ***non deterministico***

esempi problemi trattabili P e NP

- ***ordinamento di una lista*** (deterministico polinomiale ***P***)
  - data una lista disporre gli elementi in crescente (o decrescente)
  - esistono algoritmi (ad esempio, mergesort o quicksort) che risolvono l'ordinamento in tempo polinomiale rispetto alla dimensione della lista (ad esempio, O(n log n), dove n è il numero di elementi)
- ***problema del commesso viaggiatore*** (non deterministico polinomiale ***NP***)
  - dato un insieme di città e le distanze tra di esse trovare qual è il percorso più breve che visita tutte le città e torna alla partenza
  - verifica: se è fornita una soluzione, è possibile verificarne la correttezza (e calcolare la lunghezza totale) in tempo polinomiale
  - ma non si conosce un algoritmo che trovi sempre la soluzione ottima in tempo polinomiale

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

UNIVERSITÀ DI PARMA

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 **$f(x) = x^4$**

casi

- 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

UNIVERSITÀ DI PARMA

esempio: complessità temporale fattoriale

```
def factorial(n: int) ->int:
    ''' n! n factorial '''
    fact = 1
    for i in range(2,n+1):
        fact = fact * i
    return fact
```

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

complessità computazionale

- ***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

UNIVERSITÀ DI PARMA

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 **>=**

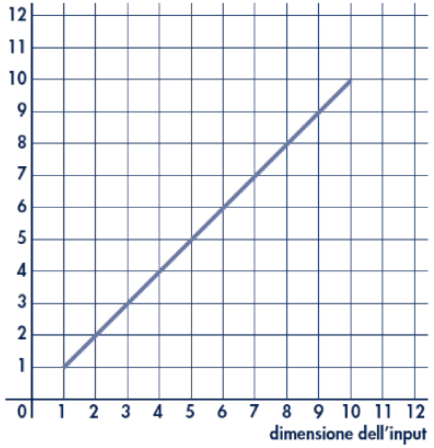
Ω(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è maggiore o uguale a f(n)”

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

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

complessità lineare

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



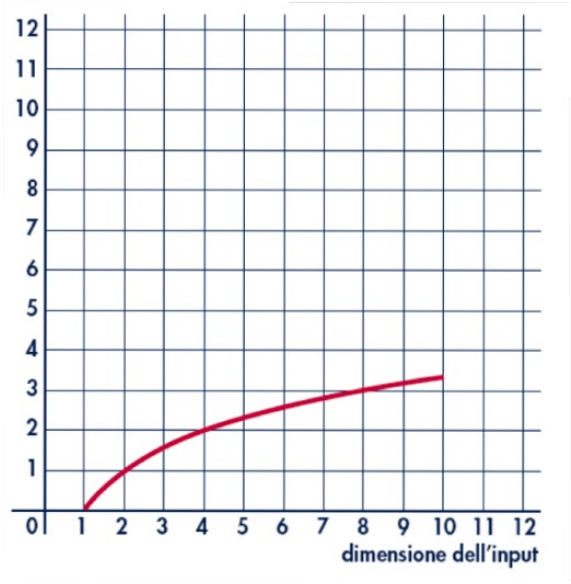
UNIVERSITÀ DI PARMA

ricerca lineare

```
def ricerca_lineare(l: list, x) -> int:
    ''' restituisce indice della posizione
        dell'elemento di l con valore x '''
    for i in range(len(l)):
        if l[i] == x:
            return i
    return -1
```

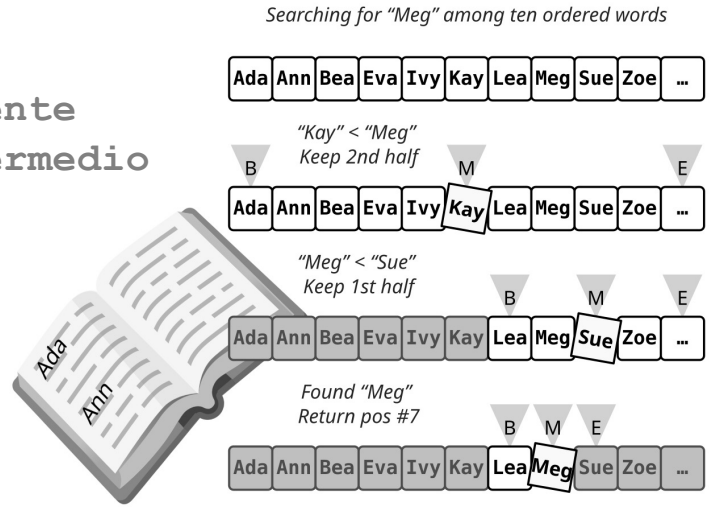
complessità logaritmica

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



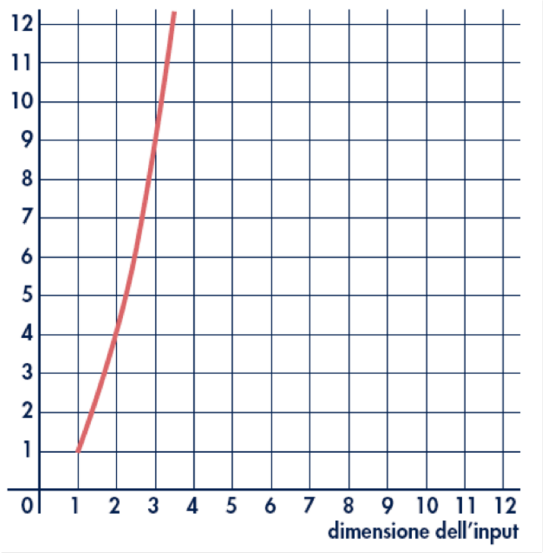
ricerca binaria (dicotomica)

```
def ricerca_binaria(l: list, v, inizio, fine) -> int:
    '''
        ricerca v nella lista ordinata l
        nell'intervallo fra gli indici [inizio,fine]
    '''
    if inizio >= fine:
        return -1                          # v non presente
    medio = (inizio + fine) // 2           # indice intermedio
    if l[medio] > v:
        return ricerca_binaria(l, v, inizio, medio)
    elif l[medio] < v:
        return ricerca_binaria(l, v, medio+1, fine)
    return medio
```



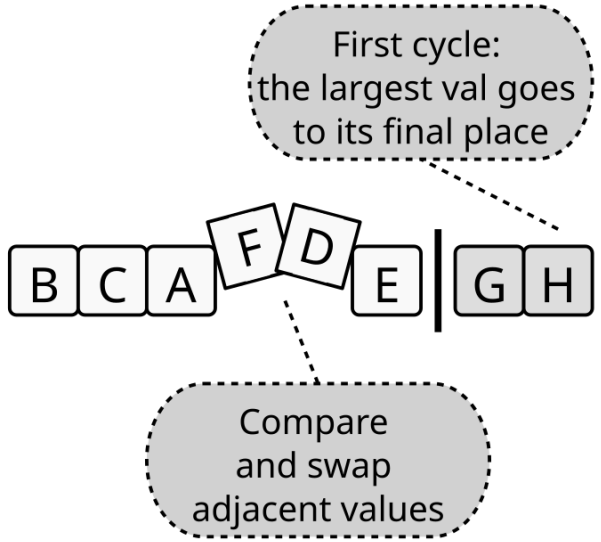
complessità quadratica

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



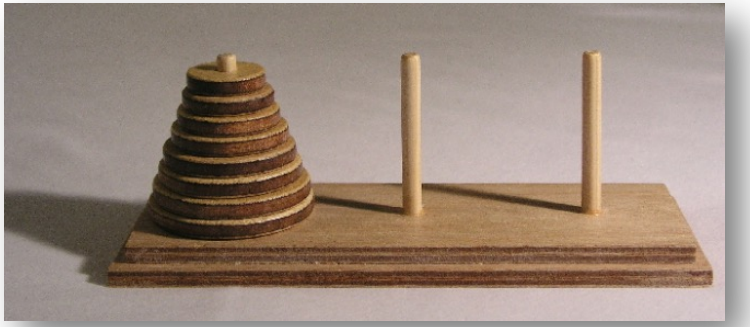
bubble sort

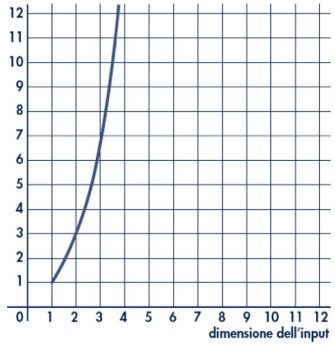
```
def bubble_sort(l: list):
    n = len(l)
    for i in range(n-1):
      for j in range(n-i-1):
        if l[j] > l[j+1]:
          l[j], l[j+1] = l[j+1], l[j]
      # print('passo',i,'lista',l)
```



complessità esponenziale

- 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*





hanoi

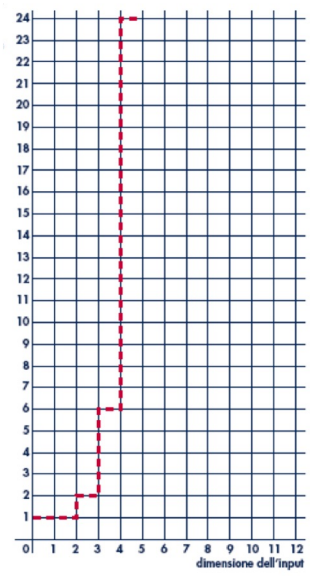
```
def move_towers(towers: list, n: int, src: int, tmp:
int, dst: int):
    # if there are discs above, move n-1 away
    if n > 1:
        move_towers(towers, n - 1, src, dst, tmp);

    # now move the largest disc (of n) to its dest
    top_disc = towers[src].pop()
    towers[dst].append(top_disc)
    print_towers(towers)

    # if there were discs above, move those on top
    if n > 1:
        move_towers(towers, n - 1, tmp, src, dst)
```

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!)***

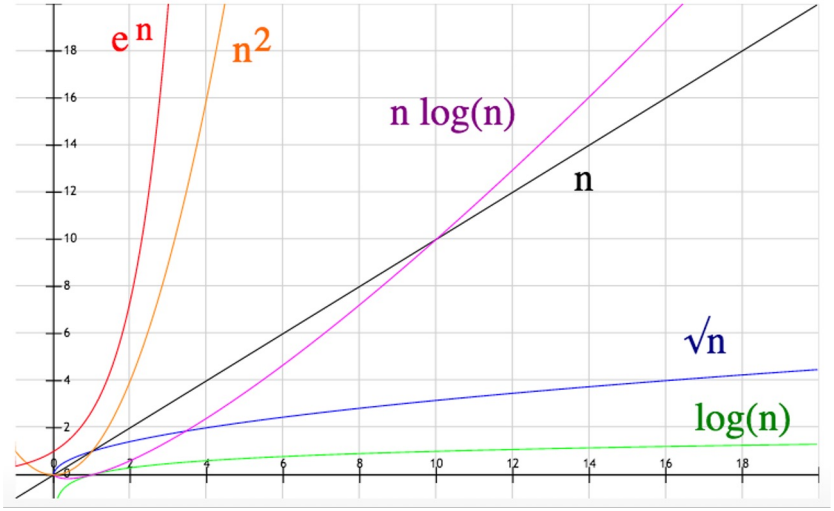


UNIVERSITÀ DI PARMA

anagrammi

```
def anagram_string(s: str) -> list:
    ''' return a list of all possible anagrams of s '''
    if len(s) <= 1:
        return [s]
    anagrams = []
    for i, c in enumerate(s):
        for sub_anagram in anagram_string(s[:i] + s[i+1:]):
            anagrams.append(c + sub_anagram)
    return list(set(anagrams))  # Removes duplicates if any
```

confronto



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

UNIVERSITÀ
DI PARMA

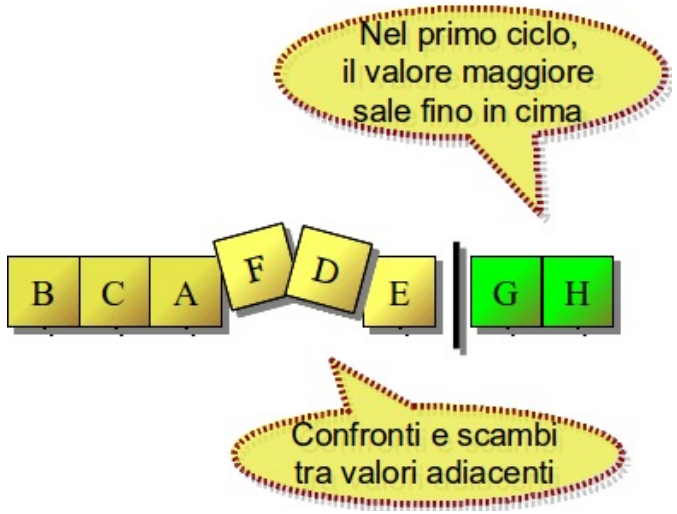
complessità computazionale

algoritmi di ordinamento

https://www.toptal.com/developers/sorting-algorithms

bubble sort

```
def bubble_sort(l: list):
    n = len(l)
    for i in range(n-1):
      for j in range(n-i-1):
        if l[j] > l[j+1]:
          l[j], l[j+1] = l[j+1], l[j]
        # print('passo',i,'lista',l)
```



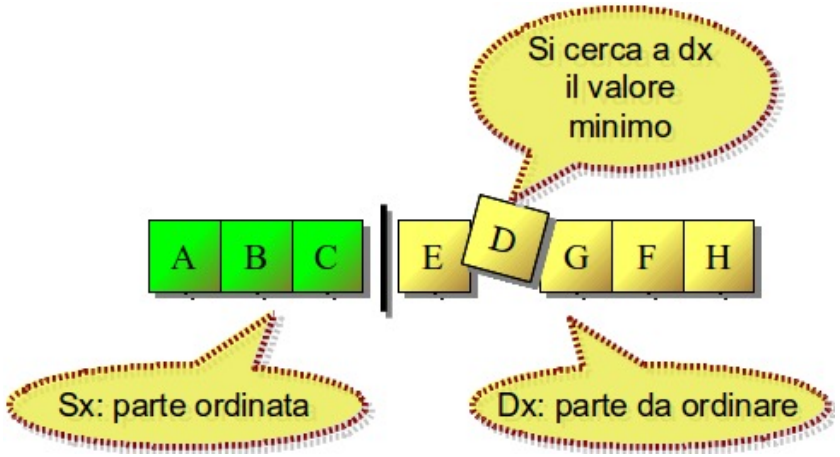
https://www.w3schools.com/python/python_dsa_bubblesort.asp

analisi bubble sort

- gli elementi ***minori*** salgono rapidamente, "come ***bollicine***"
- caso peggiore: lista rovesciata
- numero di confronti e scambi: $\boldsymbol{n^2/2}$
- $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2 = n^2/2 - n/2 \approx n^2/2$
  - applicata la formula di Gauss per la somma dei primi numeri
- ***complessità*** $\boldsymbol{n^2}$
  - anche in media, circa stessi valori

selection sort

```
def selection_sort(l: list):
    n = len(l)
    for i in range(n-1):
      min_index = i
      for j in range(i+1, n):
         if l[j] < l[min_index]:
           min_index = j
      min_value = l.pop(min_index)
      l.insert(i, min_value)
      # print('passo',i,'lista',l)
```



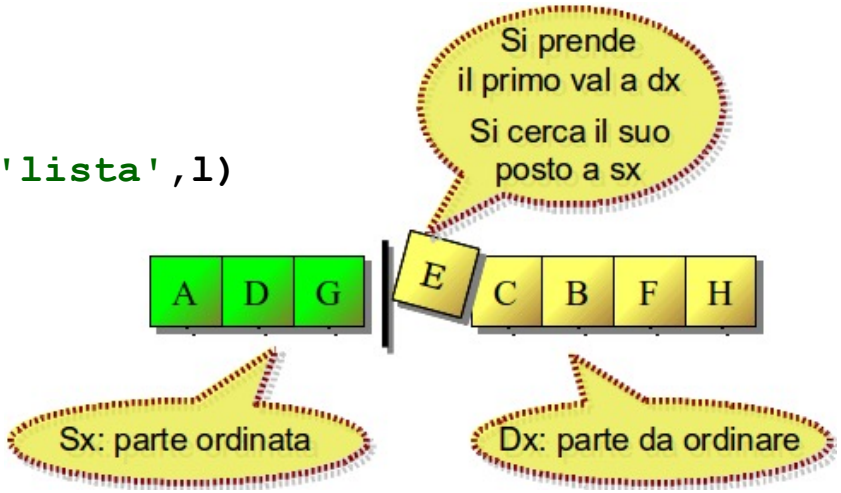
https://www.w3schools.com/python/python_dsa_selectionsort.asp

analisi selection sort

- ad ogni ***ciclo*** principale, si seleziona il valore ***minore***
- caso peggiore: lista rovesciata
- numero di confronti $n \cdot (n-1)/2$; ***complessità*** n^2
- numero di scambi: n-1 scambi
- Anche in media, circa stessi valori

insertion sort

```
def insertion_sort(l: list):
    for i in range(1, len(l)):
        val = l[i]                  # valore da inserire in posizione corretta
        j = i - 1
        # sposta gli elementi della lista da 0 a i-1] che sono più grandi di val
        while j >= 0 and l[j] > val:
            l[j + 1] = l[j]
            j -= 1
        l[j + 1] = val
        # print('inserimento elemento di indice',i,'lista',l)
```



https://www.w3schools.com/python/python_dsa_insertionsort.asp

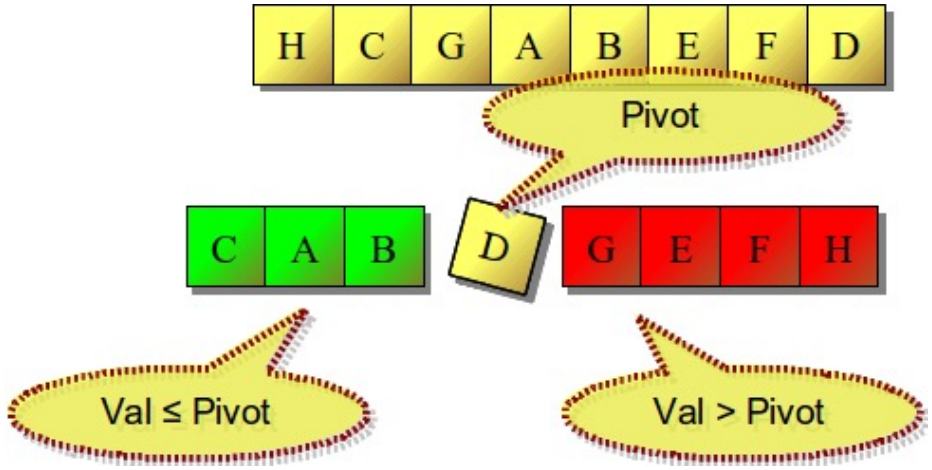
analisi insertion sort

- la ***prima parte*** è ***ordinata***, vi si inserisce un elemento alla volta, più facile trovare il posto
- caso peggiore: lista rovesciata
- cicli: $1+2+\ldots+(n-1) = n \cdot (n-1)/2$; ***complessità $O(n^2)$***
  - in media si scorre solo 1/2 della prima parte
- in media $n^2/4$ confronti e $n^2/4$ scambi
- ottimizzazioni
  - ricerca binaria in parte ordinata
  - inserimento a coppie, o gruppi

quick sort

```
def quick_sort(l: list, inizio, fine):
    ''' ordina la lista l dall'indice inizio all'indice fine '''
    print('ordino',l[inizio:fine])
    if fine - inizio <= 1: # lista vuota o 1 carattere -> ordinata
        return
    med, pivot = inizio, l[fine - 1]
    for i in range(inizio, fine):
        if l[i] <= pivot:
            l[i], l[med] = l[med], l[i]
            med += 1
    quick_sort(l, inizio, med - 1)
    quick_sort(l, med, fine)

lista = [38, 27, 43, 3, 9, 82, 10]
quick_sort(lista,0,len(lista))
```



https://www.w3schools.com/python/python_dsa_quicksort.asp

analisi quick sort

- dato un insieme, sceglie un valore ***pivot***
- crea due sottoinsiemi: $x \leq pivot$, $x > pivot$
- stesso algoritmo sui 2 insiemi (***ricorsione***)
- caso peggiore: lista rovesciata, n^2
  - dipende da scelta pivot, ma esiste sempre
- caso medio: $n \cdot log_2(n)$

UNIVERSITÀ DI PARMA

merge sort

```
def merge_sort(l: list) ->list:
  ''' restituisce una lista ordinata '''
  if len(l) <= 1:      # lista di 0 o 1 elementi -> ordinata
    return l
  med = len(l) // 2    # indice elemento centrale
  parte_sinistra = l[:med]
  parte_destra = l[med:]

  sinistra_ordinata = merge_sort(parte_sinistra)
  destra_ordinata = merge_sort(parte_destra)

  return merge(sinistra_ordinata, destra_ordinata)  # merge delle due liste
```

https://www.w3schools.com/python/python_dsa_mergesort.asp

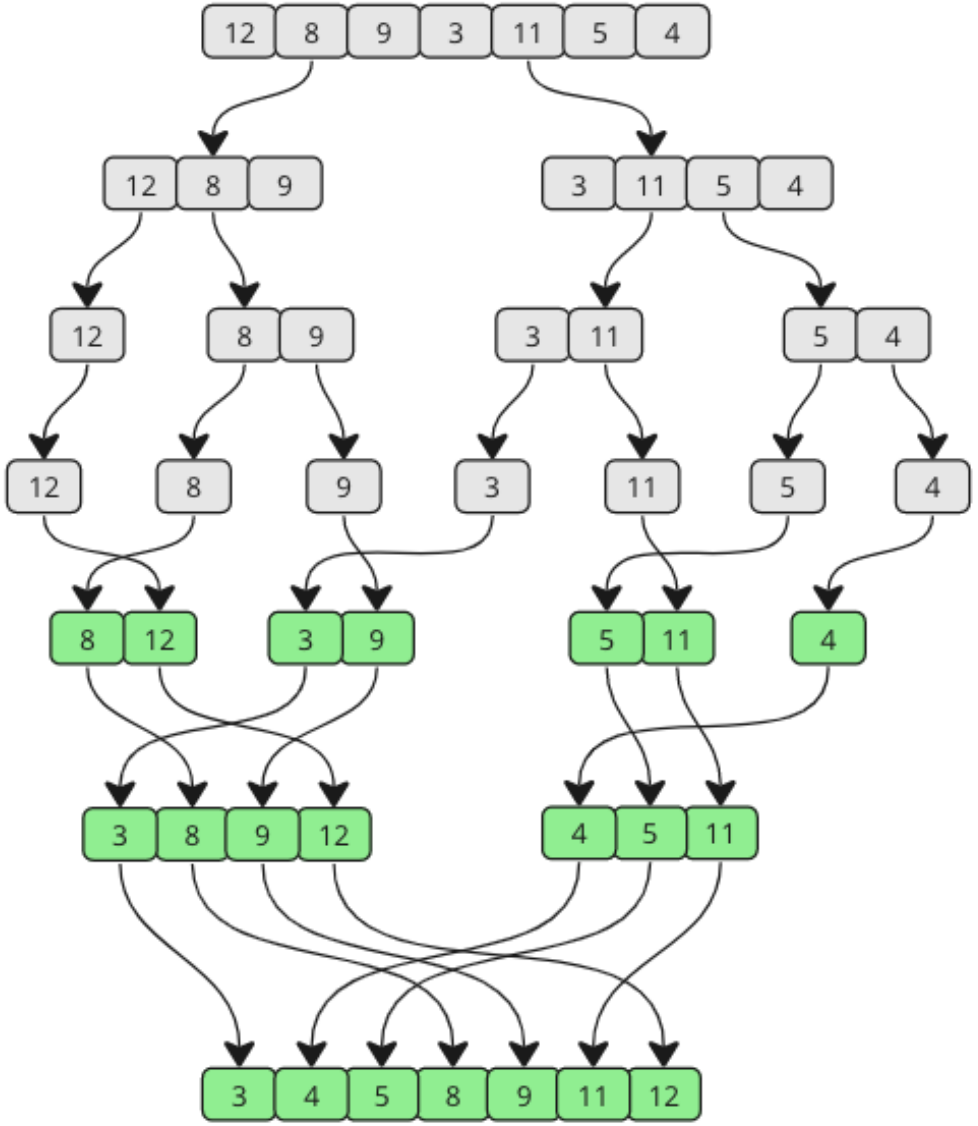
UNIVERSITÀ DI PARMA

merge_sort - fusione

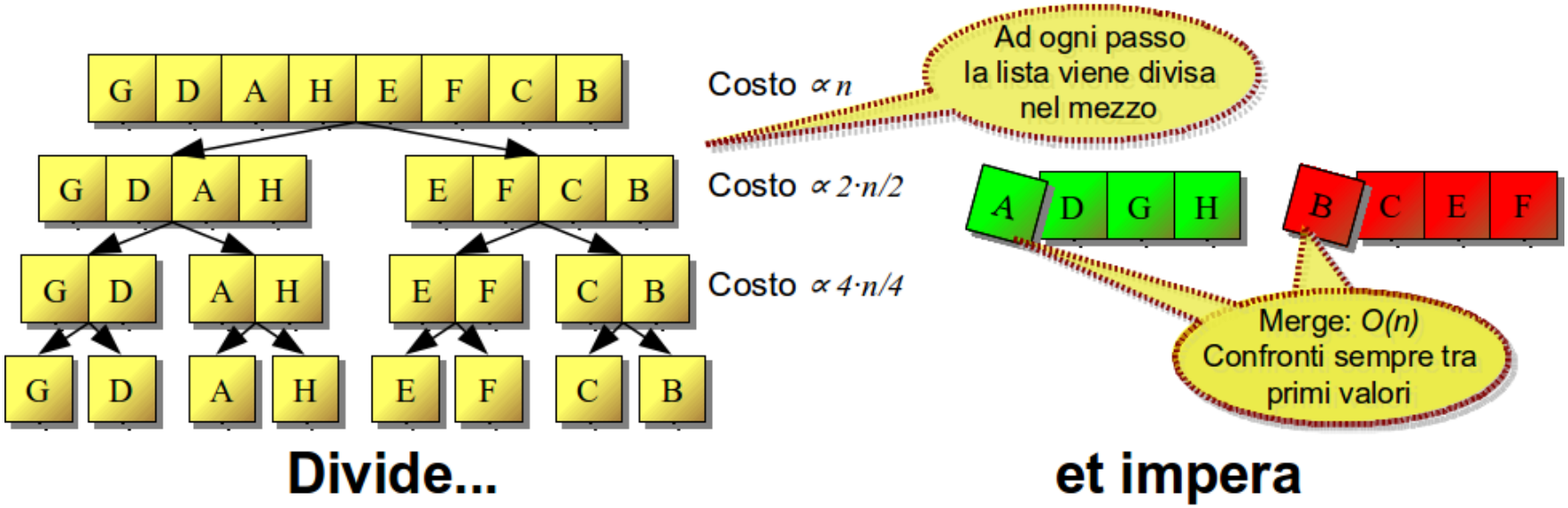
```
def merge(sinistra: list, destra: list) -> list:
  ''' restituisce una lista di fusione delle due liste ordinate '''
  risultato = []                # lista merge di sinistra e destra
  s = d = 0                     # indici iniziali delle due liste
  while s < len(sinistra) and d < len(destra):  # fino all'esaurimento delle due liste
    if sinistra[s] < destra[d]:
      risultato.append(sinistra[s]) # inserimento elemento da lista sinistra
      s += 1                        # elemento successivo della lista sinistra
    else:
      risultato.append(destra[d])   # idem destra
      d += 1

  risultato.extend(sinistra[s:])    # aggiunta eventuali elementi rimasti
  risultato.extend(destra[d:])
```

UNIVERSITÀ
DI PARMA



merge sort


G D A H E F C B
G D A H
E F C B
G D
A H
E F
C B
G
D
A
H
E
F
C
B
Costo ∝ n
Costo ∝ 2·n/2
Costo ∝ 4·n/4
Ad ogni passo
la lista viene divisa
nel mezzo
A D G H
B C E F
Merge: O(n)
Confronti sempre tra
primi valori
Divide...
et impera

analisi merge sort

- simile a Quick Sort, ma non si sceglie pivot
  - la fusione ha complessità lineare
- caso peggiore, caso medio: $n \cdot log_2(n)$
- ***spazio***
  - la fusione richiede ***altra memoria***: n
  - si può evitare il costo con spostamenti in place...
  - aumenta però la complessità (necessari più scambi)

Algorithm	Time complexity:Best	Time complexity:Average	Time complexity:Worst
Quick sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$
Merge sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

UNIVERSITÀ DI PARMA

confronto

https://www.toptal.com/developers/sorting-algorithms