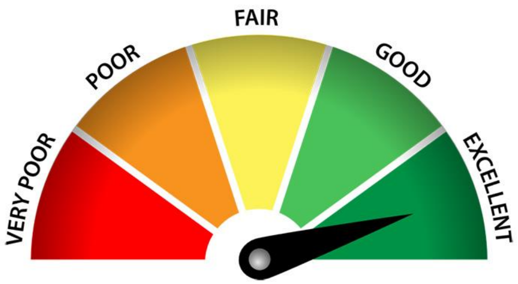
Informatica e Laboratorio di Programmazione

complessità degli algoritmi

Alberto Ferrari



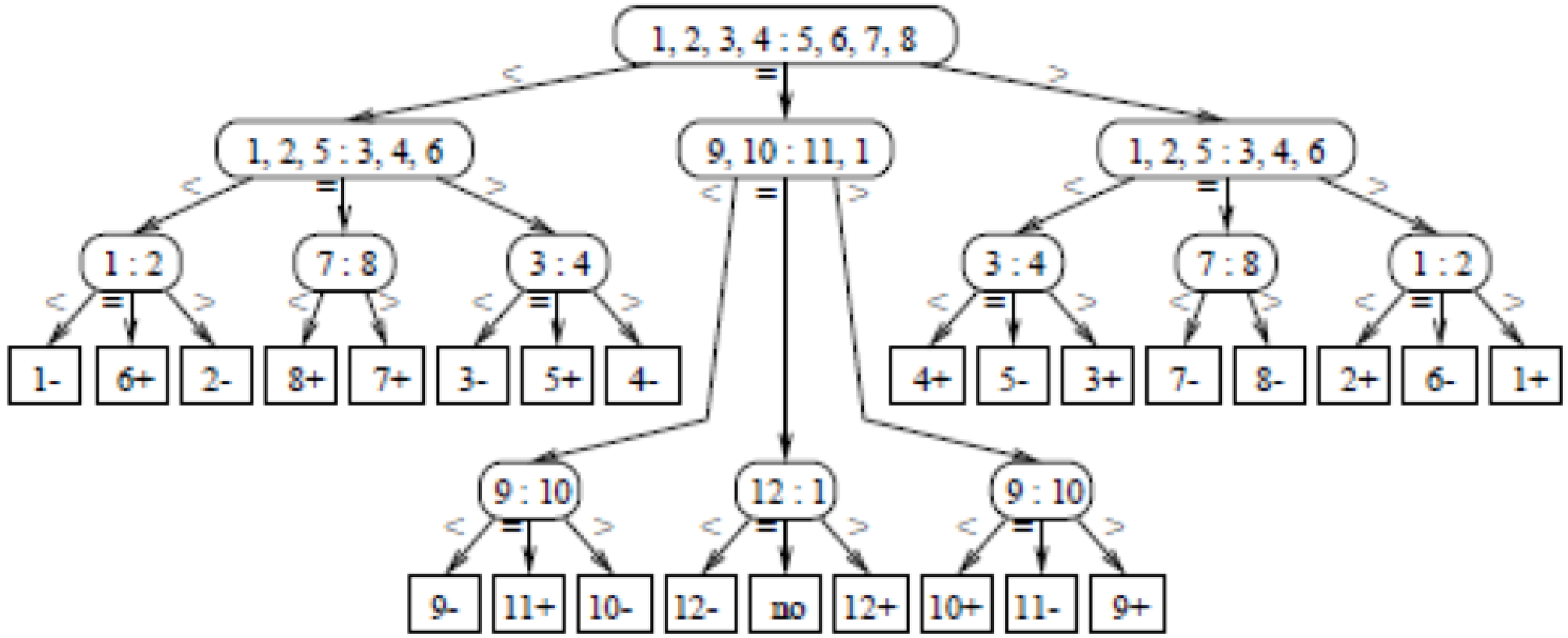
algoritmo

- 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

problema

- abbiamo ***12 monete*** che sembrano identiche ma non lo sono
- una di esse ha un peso diverso dalle altre ma non sappiamo qual è e neppure se è più ***pesante*** o più ***leggera*** delle altre
- dobbiamo scoprire qual è la moneta di peso diverso, con ***3 pesate*** comparative utilizzando una bilancia a due piatti

algoritmo


1, 2, 3, 4 : 5, 6, 7, 8
1, 2, 5 : 3, 4, 6
9, 10 : 11, 1
1, 2, 5 : 3, 4, 6
1 : 2
7 : 8
3 : 4
3 : 4
7 : 8
1 : 2
1-
6+
2-
8+
7+
3-
5+
4-
4+
5-
3+
7-
8-
2+
6-
1+
9 : 10
12 : 1
9 : 10
9-
11+
10-
12-
no
12+
10+
11-
9+

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***

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

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***

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**

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

esempio: complessità temporale fattoriale


```
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 **>=**

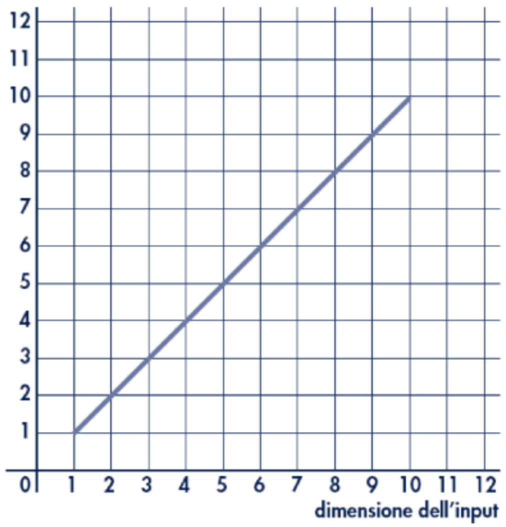
Ω(f(n)) equivale a “il tempo d’esecuzione dell’algoritmo è maggiore o uguale a f(n)”

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

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

complessità lineare

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



ricerca lineare

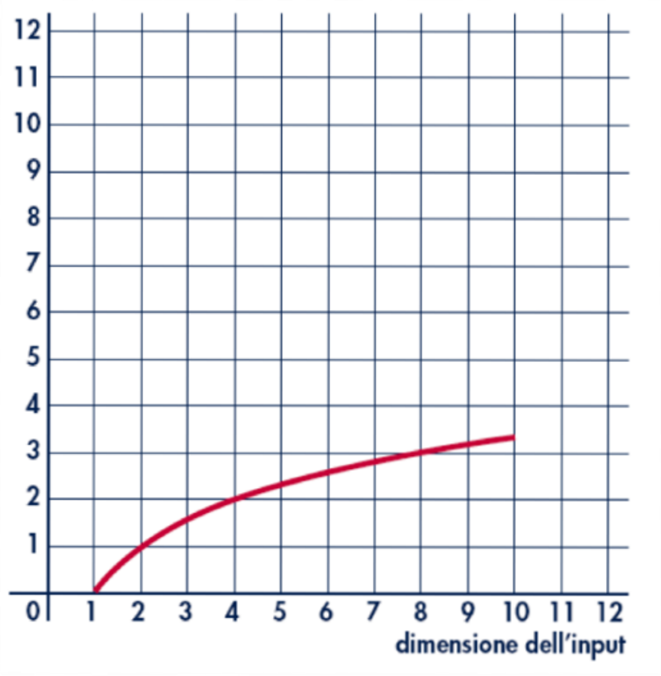

```
def linear_search(v: list, value) -> int:
    '''v: not necessarily sorted'''

    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 $O(\log_2(n))$

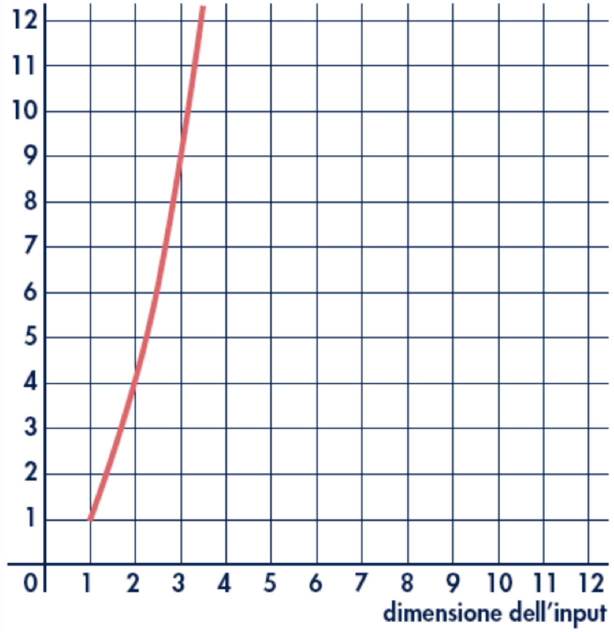


ricerca binaria (dicotomica)


```
def binary_search(v: list, value: int) -> int:
    ''' sorted list'''
    begin, end = 0, len(v) - 1
    while begin <= end:
        middle = (begin + end) // 2
        if v[middle] > value:
            end = middle - 1
        elif v[middle] < value:
            begin = middle + 1
        else:
            return middle
    return -1
```

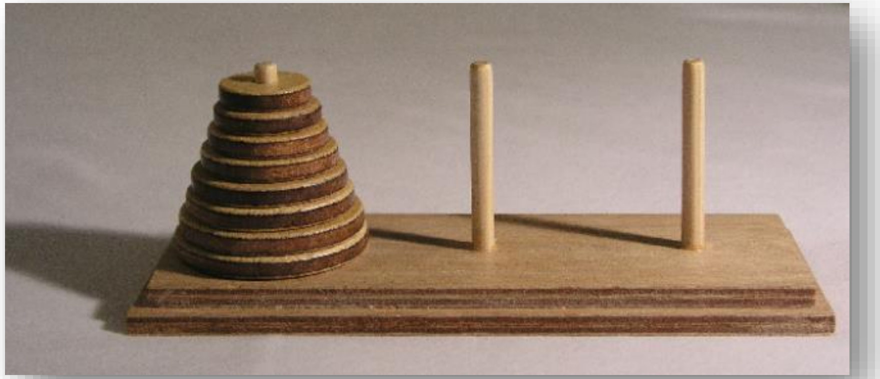
complessità quadratica

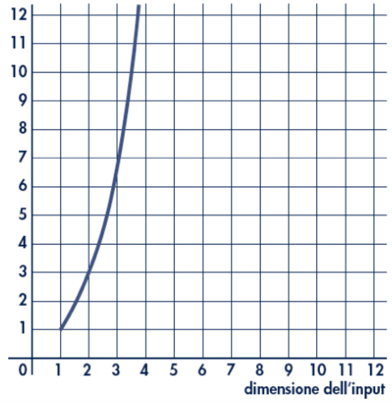
- un esempio è l'algoritmo di ordinamento ***bubblesort*** eseguito su un array di elementi
  - l'algoritmo ha complessità ***O(n²)***



complessità esponenziale

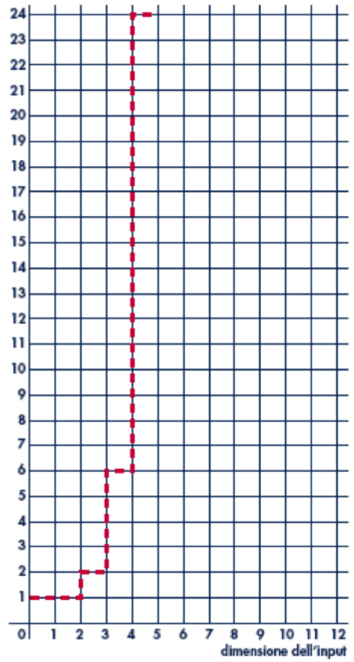
- l'algoritmo della ***Torre di Hanoi*** ha complessità $\boldsymbol{\Omega(2^n)}$
  - *la Torre di Hanoi è un rompicapo matematico composto da tre paletti e un certo numero di dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti.*
  - *il gioco inizia con tutti i 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*



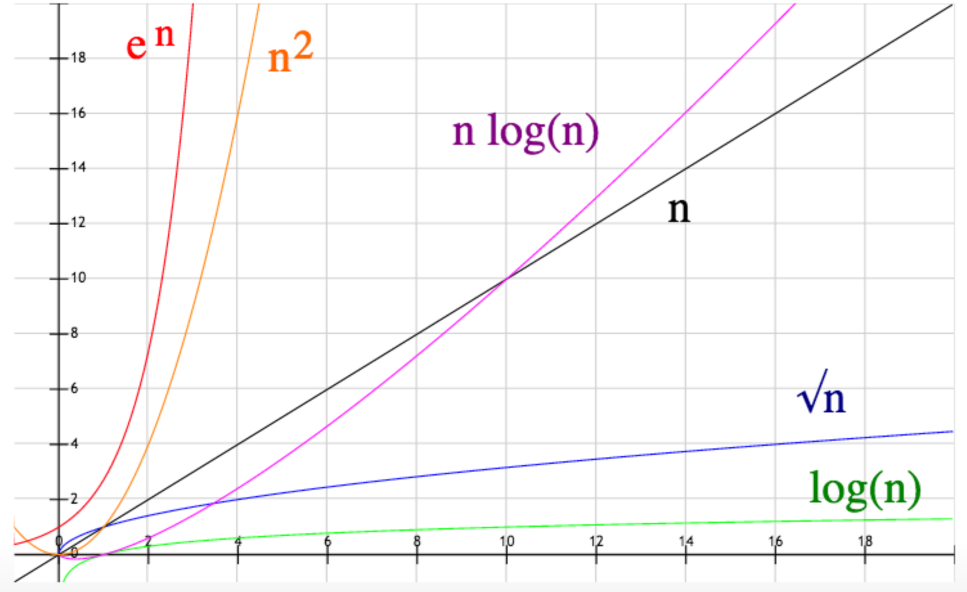


complessità fattoriale

- 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



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

complessità computazionale

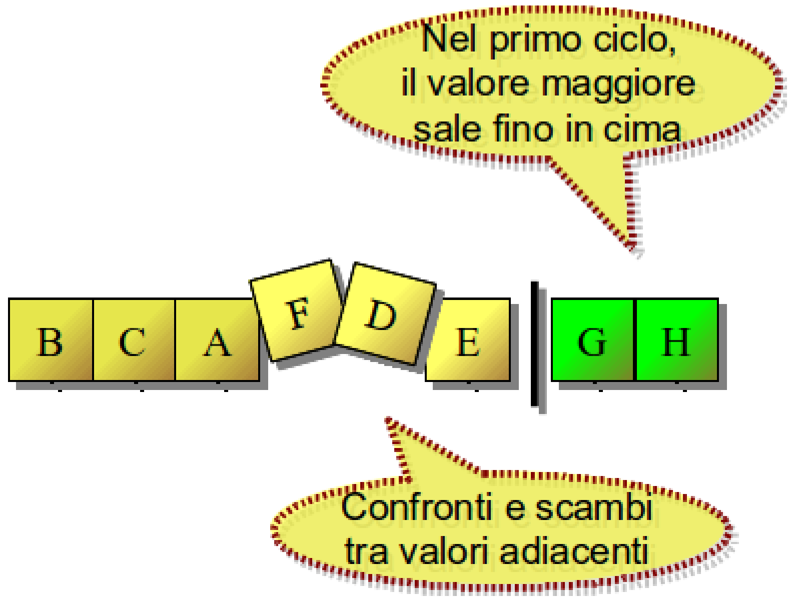
algoritmi di ordinamento

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

bubble sort


```
def swap(v: list, i: int, j: int):
    v[i], v[j] = v[j], v[i]

def bubble_sort(v: list):
    end = len(v) - 1
    while end > 0:
        for i in range(end):
            if v[i] > v[i + 1]:
                swap(v, i, i + 1)
        end -= 1
```



analisi bubble sort

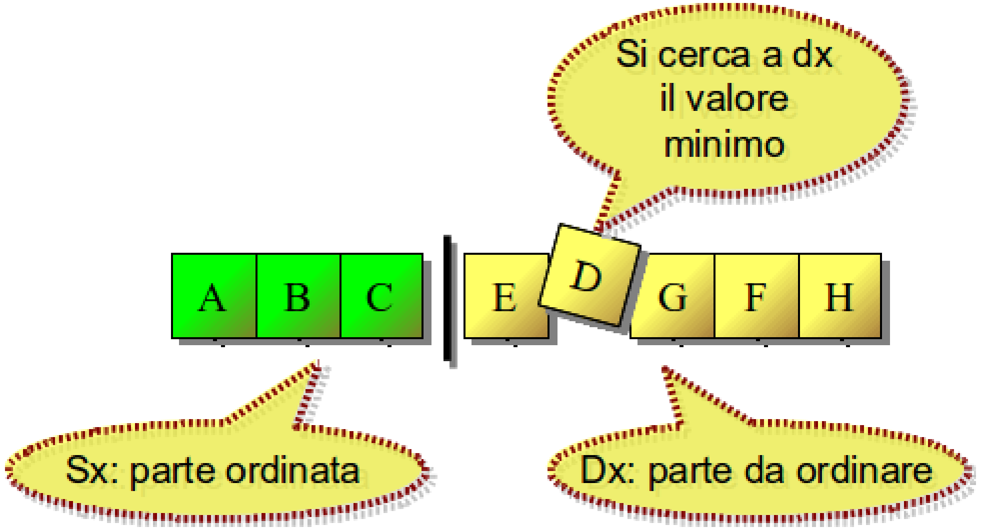
- gli elementi ***minori*** salgono rapidamente, "come ***bollicine***"
- caso peggiore: lista rovesciata
- numero di confronti e scambi: $\boldsymbol{n^2/2}$
- $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2 = n^2/2 - n/2 \approx n^2/2$
  - applicata la formula di Gauss per la somma dei primi numeri
- ***complessità***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

selection sort


```python
def selection_sort(v: list):
    for i in range(len(v) - 1):
        min_pos = i

        for j in range(i + 1, len(v)):
            if v[j] < v[min_pos]:
                min_pos = j

        swap(v, pos_min, i)
```



analisi selection sort

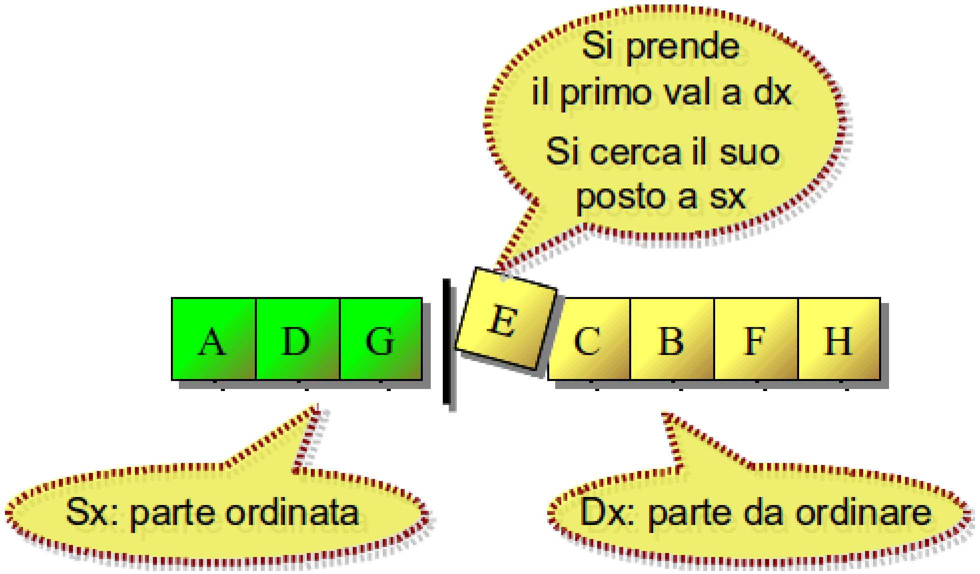
- ad ogni ***ciclo*** principale, si seleziona il valore ***minore***
- caso peggiore: lista rovesciata
- numero di confronti $n \cdot (n-1)/2$; ***complessità*** n^2
- numero di scambi: n-1 scambi
- Anche in media, circa stessi valori

insertion sort


```
def insertion_sort(v: list):
    for i in range(1, n):
        value = v[i]

        for j in range(i - 1, -1, -1):
            if v[j] <= value: break
            v[j + 1] = v[j]

        v[j + 1] = value
```

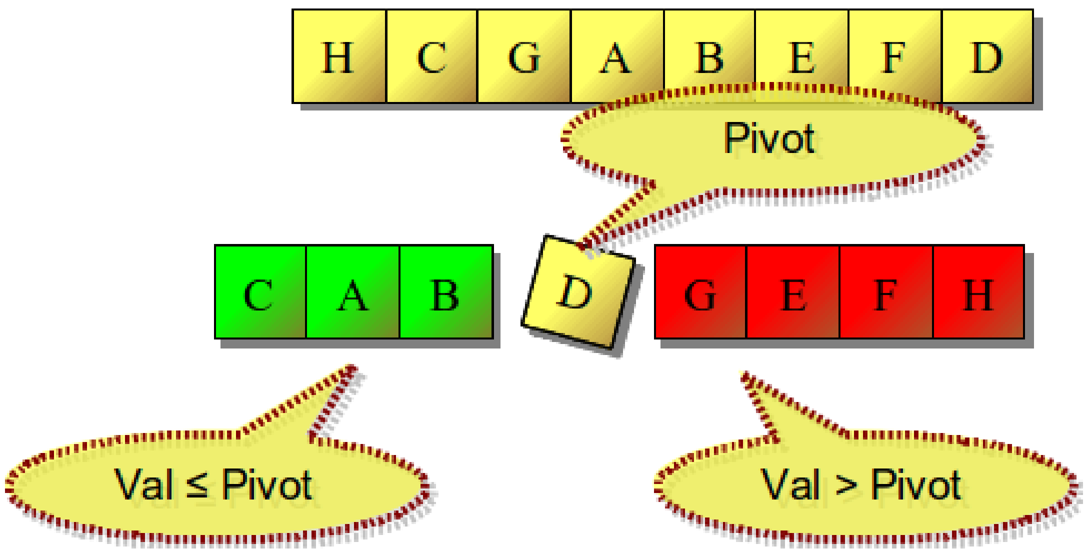


analisi insertion sort

- 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(v: list, begin=0, end=len(v)):
    if end - begin > 1:
        pivot = v[end - 1]
        j = begin
        for i in range(begin, end - 1):
            if v[i] < pivot:
                swap(v, i, j)
                j += 1
    swap(v, end - 1, j)
    quick_sort(v, begin, j)
    quick_sort(v, j + 1, end)
```

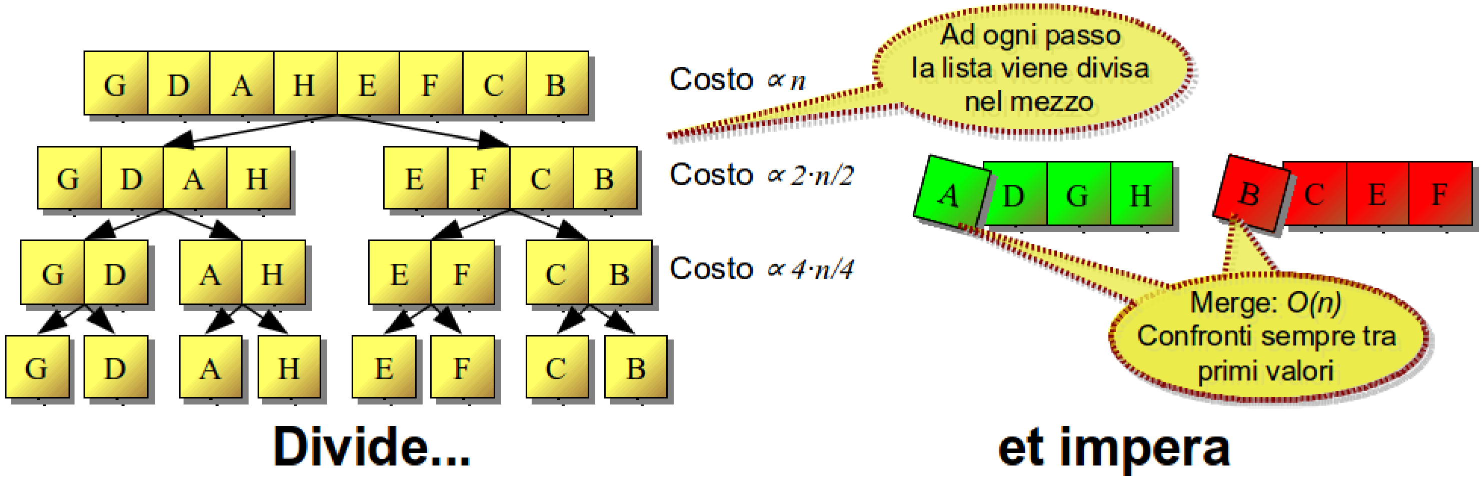


analisi quick sort

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

merge sort

```python
def merge_sort(v, begin=0, end=len(v)):
    '''
    In v, sort elements in range(begin, end)
    '''
    if end - begin > 1:
        middle = (begin + end) / 2
        merge_sort(v, begin, middle)
        merge_sort(v, middle, end)
        merge(v, begin, middle, end)
```



merge (fusione)


```
def merge(v, begin, middle, end):
    '''Merge two sorted portions of a single list'''
    i1, i2, n = begin, middle, end - begin
    result = []

    for k in range(n):
        if i1 < middle and (i2 >= end or v[i1] <= v[i2]):
            result.append(v[i1])
            i1 += 1
        else:
            result.append(v[i2])
            i2 += 1

    for k in range(n):
        cards[begin + k] = result[k]
```

analisi merge sort

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

confronto

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