# ALGORITMI E STRUTTURE DATI

Laurea triennale in scienze informatiche

Magliani Andrea
Perego Luca

Università degli studi di Milano-Bicocca

A.A. 2022/2023

# INDICE

# Problema Computazionale e Algoritmi

## 1.1 Problema Computazionale

Relazione matematica tra input e output.
Un problema è definito come: $\pi \subseteq input \times output$

## 1.2 Istanza

Set di input specifici legati ad un determinato problema.

## 1.3 Algoritmo

Descrizione finita, composta da una sequenza di istruzioni elementari e non ambigue che, se eseguita, trasforma gli input in output.

## 1.4 Analisi degli Algoritmi

Gli algoritmi vengono **analizzati** per valutarne diversi aspetti:

- <u>Correttezza</u>: verificata con test e dimostrazioni;
- <u>Efficienza</u>: verificata misurando i tempi e lo spazio occupato;

Un algoritmo che risolve un problema per ogni sua istanza in un tempo finito è detto **corretto**.

Un algoritmo che, per almeno una delle istanze, non risolve correttamente il problema è detto **non corretto**.

## 1.5 Struttura dati

Un modo per memorizzare e manipolare dati.

# Correttezza & Efficienza

## 2.1 Dimostrazione di Correttezza

Invariante di ciclo: metodo per dimostrare la correttezza di un algoritmo contenente un loop. L'invariante di ciclo si divide in 3 fasi:

- **Inizializzazione**: dimostra la correttezza per la prima iterazione;
- **Conservazione**: l'algoritmo è corretto per ogni valore di $i$ e questa incrementa correttamente ad ogni iterazione;
- **Conclusione**: assumendo la condizione del ciclo False, l'algoritmo termina restituendo il risultato corretto;

## 2.2 Calcolo dell'Efficienza

Un algoritmo efficiente utilizza il minor **tempo** e **risorse** possibili.
È necessario definire una funzione $T(n)$, ovvero il tempo di calcolo impiegato per gestire un input di lunghezza $n$.

Ad ogni tipo di istruzione viene assegnato un **valore temporale** di esecuzione, per poi contarne le occorrenze nel codice.

$T(n)$ equivale alla somma delle occorrenze di tutti i valori temporali.

In un algoritmo è presente:

Caso Migliore: $T_{migl}(n)$ →
Sottoinsieme delle istanze in cui l'algoritmo impiega meno.

Caso Migliore: $T_{pegg}(n)$ →
Sottoinsieme delle istanze in cui l'algoritmo impiega di più.

# Notazioni Asintotiche

## 3.1 O-Grande

O$(n)$ rappresenta il **limite superiore** [asintoticamente] della funzione $T(n)$ di un algoritmo.

$O(g(n)) = \{f(n) \mid \exists\, c, n_0 > 0,\ f(n) \le c \cdot g(n)\ \ \forall n > n_0\}$ $\qquad f(n) \in N \ \wedge\ f(n) > 0\ def.$

## 3.2 Ω-Grande

Ω$(n)$ rappresenta il **limite inferiore** [asintoticamente] della funzione $T(n)$ di un algoritmo.

$\Omega(g(n)) = \{f(n) \mid \exists\, c, n_0 > 0,\ 0 \le c \cdot g(n) < f(n)\ \ \forall n > n_0\}$

## 3.3 θ-Grande

θ$(n)$ rappresenta la funzione che **delimita superiormente** ed **inferiormente** la funzione $T(n)$ di un algoritmo.

$\theta(g(n)) = \{f(n) \mid \exists\, c_1, c_2, n_0 > 0,\ \ 0 \le c_2 \cdot g(n) \le f(n)\ \ \le f(n) \le c_1 \cdot g(n)\ \forall n > n_0\}$

## 3.4 Gerarchie di crescita Asintotica

La crescita di $T(n)$ varia in base alla funzione a cui è associata.
La **scala di crescita** è:

$$c \to \log n \to \sqrt{n} \to n \to n \log n \to n^a [a > 1] \to a^n \to n! \to n^n$$

# Caratteristiche degli Algoritmi

## 4.1 Stabile

Algoritmo che, se incontrati **2 valori uguali** e adiacenti, ne mantiene l'ordine.

## 4.2 In-Place

Algoritmo che non utilizza una **struttura dati ausiliaria**, ma lavora direttamente sull'input.

# Algoritmi di Ordinamento

## 5.1 Definizione

Gli algoritmi di ordinamento sono utilizzati per posizionare gli elementi di un insieme secondo una **relazione d'ordine**.

## 5.2 Struttura del problema

Ogni algoritmo di ordinamento condivide problema e risultato.

<u>Problema</u>: Ordinamento di un vettore $V$ di $n$ elementi.
<u>Input</u>: Un vettore $V$ di $n$ elementi.
<u>Output</u>: Un vettore $V$ t.c.:

- L'output è una permutazione di $V$;
- $\forall\, i \in [1, n-1] \quad V'[i] \le V[i+1]$;

# Teorema dell’esperto

## 6.1 Enunciato

Sia $T(n) = a \cdot T\left(\frac{n}{b}\right) \cdot f(n)$ $\qquad a \geq 1,\ b > 1,\ f(n)\ asin.\ pos.$

1. se $\exists\, \varepsilon > 0\ t.c.\ f(n) = 0\left(n^{\log_b a - \varepsilon}\right)$ allora $T(n) = \theta\left(n^{\log_b a}\right)$
2. se $f(n) = \theta\left(n^{\log_b a}\right)$ allora $T(n) = \theta\left(n^{\log_b a} \cdot \log n\right)$
3. se $\exists \varepsilon > 0\ t.c.\ f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ e se $\exists\, c < 1\ t.c.\ a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ $\forall n > n_0$ allora $t(n) = \theta(f(n))$

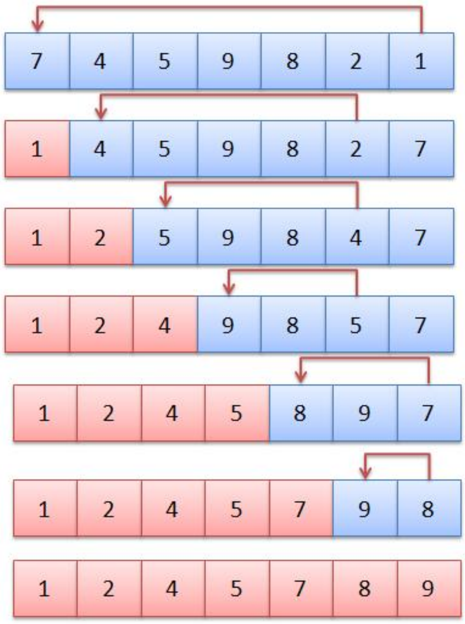
# Selection sort

## 7.1 Pseudocodice

```
SELECTION_SORT (V)

    for i := 1 to V.length - 1

        posmin := i

        for j := i + 1 to V.length - 1

            if V[posmin] > V[j] then

                posmin := j

        scambia V[i] con V[posmin]
```

## 7.2 Funzionamento

Si cerca il numero minore navigando tutto il vettore e si mette nella posizione i, con i che varia dalla prima posizione del vettore fino all'ultima.




## 7.3 Correttezza

I primi i - 1 elementi di V sono i più piccoli i - 1 elementi di V ordinati in ordine crescente.

INIZIALIZZAZIONE

I primi 0 elementi di $V^{I}$ sono i più piccoli 0 elementi di V ordinati in ordine crescente

CONSERVAZIONE

Corretto ad inizio e fine ciclo.

TERMINAZIONE

Corretto al termine dell'algoritmo.

## 7.4 Tempi di calcolo

CASO MIGLIORE e CASO PEGGIORE sono asintoticamente uguali.
$T(n) = \theta(n^2)$

## 7.5 Caratteristiche

STABILE

No, il selection sort non è un algoritmo di ordinamento stabile in quanto scambiando l'elemento di posizione i con l'elemento più piccolo dell'array, non sempre mantiene l'ordine originale degli elementi uguali nel vettore.

IN PLACE

Si, il selection sort è un algoritmo di ordinamento in place in quanto non utilizza altre strutture dati per ordinare il vettore in input.

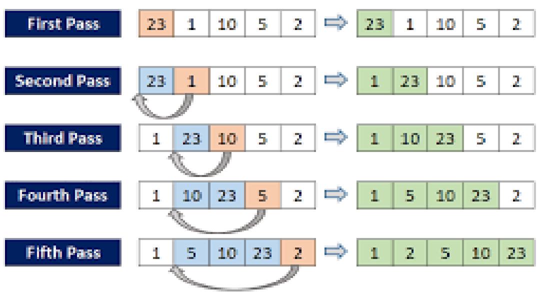
# Insertion sort

## 8.1 Pseudocodice

```
INSERTION_SORT (V)

     for i := 2 to V.length

          j := i - 1
          key := V[i]

          while j >= 1 AND V[j] > Key

               V[j + 1] := V[j]
               V[j] := Key
               j := j - 1
```

## 8.2 Funzionamento

Si parte dal secondo elemento del vettore in input e si controlla se l’elemento precedente è minore, nel caso si scambiano i due valori, si continua successivamente con l’elemento i + 1 fino alla fine dell’array.



## 8.3 Correttezza

All'inizio di ogni iterazione i primi i-1 elementi di $V^I$ sono i primi i-1 elementi di V in ordine crescente.

INIZIALIZZAZIONE

Il primo elemento di $V^I$ è il primo elemento di V in ordine crescente

CONSERVAZIONE

Vero ad inizio e fine ciclo

TERMINAZIONE

Vero a fine algoritmo

## 8.4 Tempi di calcolo

CASO MIGLIORE

Il vettore V è già ordinato. Tmigl(n) = $\theta(n)$ -> T(n) = $\Omega(n)$

CASO PEGGIORE

V è ordinato in senso decrescente. Tpegg(n) = $\theta(n^2)$ -> T(n) = $O(n^2)$

## 8.5 Caratteristiche

STABILE

Si l'insertion sort è un algoritmo di ordinamento stabile in quanto mantiene l'ordine degli elementi uguali tra di loro.

IN PLACE

Si l'insertion sort è un algoritmo di ordinamento in place in quanto non utilizza strutture d'appoggio per eseguire le sue operazioni.

# Mergesort

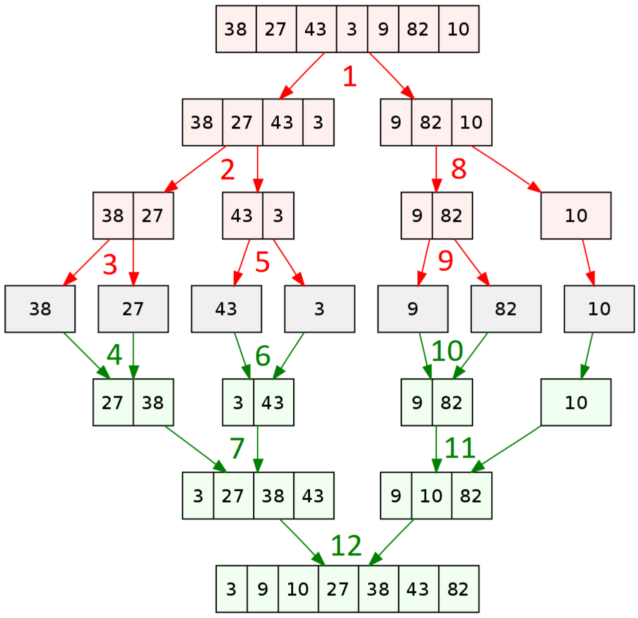
## 9.1 Pseudocodice

```
MERGESORT (V,l, r)

     if l < r then

          mid := floor[(l+r)/2]
          MERGESORT (V, l, mid)
          MERGESORT (V, mid+1, r)
          MERGE (V,l, mid, r)

MERGE (V, l, mid, r)

     T := Vettore di lunghezza r-l + 1
     i := l
     j := mid + 1
     k := 1

     while i <= mid AND j <= r do

          if V[i] <= V[j] then
               T[k] := V[i]
               i := i + 1

          else

               T[k] := V[j]
               j := j + 1

          k := k + 1

     while i <= min do

          T[k] := V[i]
          i := i + 1
          k := k + 1

     for k := 1 to T.length do

          V[l + k - 1] := T[k]
```

## 9.2 Funzionamento

Il mergesort utilizza la strategia di programmazione **divide et impera** per ridurre il problema in più sottoproblemi.

Una volta ottenuti i singoletti e arrivati nel caso base si esegue la procedura di merge, andando a unire i singoletti riordinandoli durante il processo.




## 9.3 Tempi di calcolo

a = 2 (volte che viene chiamato il metodo ricorsivo)
b = 2 (in quante porzioni divido l'array [mid = l+r / **2**]
f(n) = $\theta(n)$ (righe di codice che non c'entrano con la ricorsione)

quindi applico il secondo caso del teorema del maestro:

$$\theta(n) = \theta(n^{\log_2 2}) \rightarrow \theta(n) = \theta(n) \rightarrow T(n) = \theta(n^{\log_2 2} \cdot \log n) \rightarrow T(n) = \theta(n \cdot \log n)$$

# Ricerca Dicotomica

**Problema:** Ricerca di un elemento in un vettore $V$ ordinato.
**Input:** un vettore $V$ di $n$ elementi e un intero x.
**Output:** un intero $n$ t.c. $n \in V \wedge n = x$

## 10.1 Pseudocodice

```
Ricerca_Dicotomica(V, x, l, r)

    if r < l then
        return false

    if r = l then
        return x == V[l]

    mid := floor((l+r)/2)

    if x > V[mid] then
        return Ricerca_Dicotomica(V, x, mid + 1, r)
    else
        return Ricerca_Dicotomica(V, x, l, mid)
```

## 10.2 Tempi di calcolo

Caso migliore e caso peggiore sono **asintoticamente uguali**: $T(n) = \Theta(log_2 n)$.