3. L'ORDINAMENTO

- **PROBLEMA DELL'ORDINAMENTO:**
  - **INPUT**: una sequenza $A = a_1, a_2, \dots, a_m$ di m valori
  - **OUTPUT**: una sequenza $B = b_1, b_2, \dots, b_m$ che sia una permutazione di A tale che:

    $b_1 \leq b_2 \leq \dots \leq b_m$.

Esistono tantissimi algoritmi in grado di risolvere questo problema, ciascuno con diverse complessità computazionale.
Vediamo adesso alcuni algoritmi di ORDINAMENTO.

- **STUPID SORT:**

Prende la sequenza A; verifica se è ordinata, altrimenti genera una permutazione casuale di A e ricontrolla:

```
STUPIDSORT(A):
  while not sorted(A):
    A ← random_permutation(A)
```

Ancora più "stupido" è il Bozo Sort:

```
BOZOSORT(A):
  while not sorted(A):
    A ← invert_two_elements(A)
```

ANALISI:

- CASO PESSIMO: $T(m) = \infty$
- CASO MEDIO: $T(m) = O(m \cdot m!)$, le possibili permutazioni sono $m!$
- CASO OTTIMO: $T(m) = O(m)$.

- **SELECTION SORT** (approccio naïf):

Ispirato a come lo ordinerebbe un umano.
Cerco il minimo e lo metto in 1ª posizione ed ordino così i restanti $n-1$ elementi:

```
SELECTIONSORT(A, m):
  for i ← 0 to m-1:
    min ← i
    for j ← (i+1) to m:
      if A[j] < A[min] then:
        min ← j
    if min != i then:
      tmp ← A[i]
      A[i] ← A[min]
      A[min] ← tmp
```

ANALISI

Per calcolare la complessità nel caso medio, pessimo ed ottimo, ragiono sull'operazione dominante:

$$T(m) = \sum_{i=0}^{n-1} (n-i) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2} =$$

$$= O(n^2)$$

• BUBBLE SORT:

Ordina gli elementi facendo "salire" come BOLLE quelli più piccoli, mentre quelli più "pesanti" scendono verso il basso.

Compara ogni elemento adiacente e li inverte di posizione se sono nell'ordine sbagliato; alcuni elementi raggiungono la posizione corretta più velocemente di altri.

```
BUBBLESORT(A, n):
    scambio ← true
    while scambio do:
        scambio ← false
        for i ← 0 to n-1 do:
            if A[i] > A[i+1] then:
                swap(A[i], A[i+1])
                scambio ← true
```

L'operazione dominante è il confronto nel ciclo più interno.

Vengono effettuati $\frac{n^2}{2}$ scambi sia nel caso medio che nel caso pessimo

$\Rightarrow T(n) = \Theta(n^2)$

• INSERTION SORT:

Ordina in modo non decrescente; inserisce l'elemento A[i] al posto giusto nel vettore ordinato $A[0, \dots, i-1]$:

```
INSERTIONSORT(A, n):
    for i ← 1 to n:
        key ← A[i]
        j ← i-1
        while j >= 1 and A[j] > key:
            A[j+1] ← A[j]
            j ← j-1
        A[j+1] ← key
```

COMPLESSITÀ

- CASO OTTIMO: $\Theta(n)$
- CASO PESSIMO: $O(n^2)$
- CASO MEDIO: $\Theta(n^2)$

• MERGE SORT:

Utilizza la tecnica del DIVIDE ET IMPERA ed opera in maniera ricorsiva;
Proposto da John Von Neumann nel 1945.

IDEA DI BASE:

Se la sottosequenza ha lunghezza 1 è già ordinata, altrimenti:

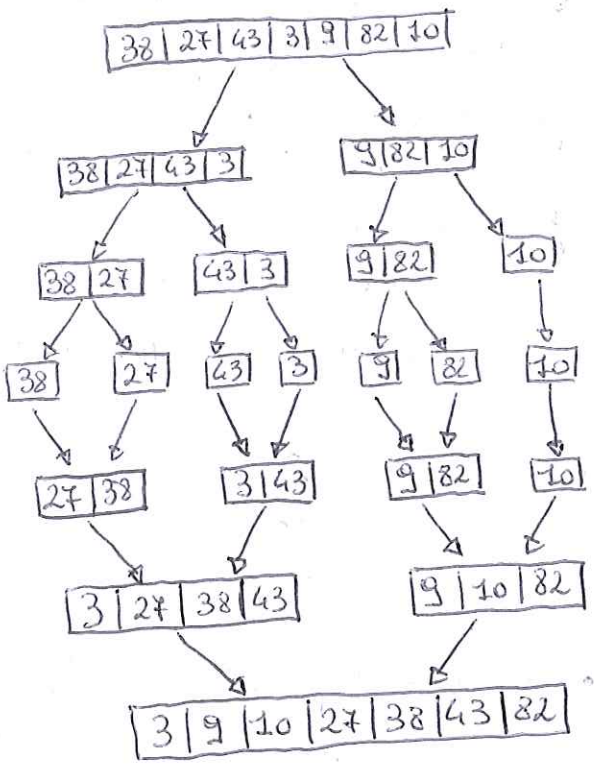
1) Si divide la sequenza in 2 metà;
2) Ciascuna delle 2 sottosequenze viene ordinata ricorsivamente
3) Le 2 sottosequenze vengono "fuse", estraendo ripetutamente il minimo delle 2 sottosequenze.

```
MERGESORT (A, left, right):
    if left < right then:
        center ← (left + right)/2
        MERGESORT (A, left, center)
        MERGESORT (A, center+1, right)
        MERGE (A, left, center, right)
```

Dove MERGE è la funzione seguente, per "fondere" le 2 sottosequenze ordinate:

```
MERGE (A, left, center, right):
    i ← left
    J ← center + 1
    k ← left
    while i ≤ center and J ≤ right:
        if A[i] ≤ A[J] then:
            B[k] ← A[i]
            i ← i+1
        else
            B[k] ← A[J]
            J ← J+1
        k ← k+1
    J ← right
    for h ← center downto i:
        A[J] ← A[h]
        J ← J-1
    for J ← left to k-1:
        A[J] ← B[J]
```

Esempio: $A = [38, 27, 43, 3, 9, 82, 10]$



ANALISI:

SEMPLIFICAZIONI:
- $n = 2^k \Rightarrow k = \log n$
- Tutte le sottosequenze hanno un numero di elementi che sono potenze esatte di 2.

La relazione di ricorrenza è:

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + dn & \text{se } n > 1 \end{cases}$$

Si fanno k divisioni:

$$O\left(\sum_{i=1}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=1}^{k} n\right) = O(nk) = \underline{O(n \log n)}$$

Altrimenti, dal MASTER THEOREM: $a = 2$, $b = 2$, $\beta = 1$, $\alpha = \log_b a = 1$

$\Rightarrow \alpha = \beta = 1 \Rightarrow$ $\boxed{T(n) = \Theta(n \log n)}$ ok!!

• BUCKET SORT

Assume che gli elementi siano uniformemente distribuiti in un certo intervallo. È suddiviso in fasi:

(1) Genera un certo numero di buckets;

(2) SCATTER: scandisce il vettore ed inserisce gli elementi nei vari buckets;

(3) ORDINA (con "Insertion Sort") ciascun bucket;

(4) GATHER: estrae da ciascun bucket gli elementi e li inserisce nel vettore originale.

```
BUCKETSORT (A, k):
    buckets ← vettore di k buckets
    M ← elemento più grande nel vettore A
    for i ← 1 to len(A):
        inserisci A[i] in buckets [⌊k*A[i]/M⌋]
    for i ← 1 to k:
        SORT (buckets[i])          # tipicamente con Insertion Sort
    return concatenazione di buckets[1], ..., buckets[k]
```

• ANALISI:

- **CASO PESSIMO:** Se tutti gli elementi vanno in 1 bucket?
  Il costo è dominato dal costo di ordinamento del bucket e, per Insertion Sort, il costo è: $O(n^2)$

- **CASO MEDIO:** per determinare M si può usare ARRAYMAX(): $O(n)$
  lo scatter costa: $O(n)$
  L'ordinamento di ciascun bucket dipende dal numero di elementi presenti nel bucket i-esimo: $O\left(\sum_{i=1}^{k} n_i^2\right)$

Nel CASO MEDIO $\longrightarrow E(n_i^2)$

Sia una v. aleatoria $X_{ij} = \begin{cases} 1 & \text{se } A[j] \xrightarrow{\in} \text{bucket}[i] \\ 0 & \text{altrimenti} \end{cases}$ $\Rightarrow n_i = \sum_{j=1}^{n} X_{ij}$

$$\Rightarrow E(n_i^2) = E\left(\sum_{j=1}^{n} X_{ij} \cdot \sum_{k=1}^{n} X_{ik}\right) = E\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij} \cdot X_{ik}\right) = E\left(\sum_{j=1}^{n} X_{ij}^2\right) + E\left(\sum_{\substack{i \le j \\ k \le n}}\sum_{j \ne k} X_{ij}X_{ik}\right)$$

Per l'assunzione di uniformità dell'input, $X_{ij} = 1$ con probabilità $1/k$:

$$E(X_{ij}^2) = 1^2 \cdot \left(\frac{1}{k}\right) + 0^2 \cdot \left(1 - \frac{1}{k}\right) = \frac{1}{k} \quad ; \quad E(X_{ij}X_{ik}) = \left(\frac{1}{k}\right)^2$$

$$\Rightarrow E\left(\sum_{j=1}^{n} X_{ij}^2\right) + E\left(\sum_{\substack{i \le j \\ k \le n}}\sum_{j \ne k} X_{ij}X_{ik}\right) = n \cdot \frac{1}{k} + n(n-1) \cdot \frac{1}{k^2} = \frac{n^2 + nk - n}{k^2}$$

$$\Rightarrow \text{la complessità è: } O\left(\sum_{i=1}^{k} E(n_i^2)\right) = O\left(\sum_{i=1}^{k} \frac{n^2}{k^2} + \frac{n}{k} - \frac{n}{k^2}\right) = O\left(\frac{n^2}{k} + n\right)$$

Il costo della fase di gather e della concatenazione dei bucket è: $O(k)$

$\Rightarrow$ la COMPLESSITÀ TOTALE è: $O\left(\frac{m^2}{k} + m + k\right)$

* Se $k = \Theta(m)$ $\Rightarrow$ la COMPLESSITÀ è $O(m)$ → l'efficienza dipende dal numero di buckets!

• QUICK SORT:

È basato sul DIVIDE ET IMPERA; organizzato in 3 fasi:

(1) Seleziona un elemento del vettore, detto PIVOT;

(2) Partiziona il vettore: gli elementi più grandi del pivot vanno alla sua destra, i minori a sinistra;

(3) Applica l'algoritmo RICORSIVAMENTE alle parte destra e sinistra.

* La scelta del pivot influenza enormemente le performance dell'algoritmo.

```
PARTITION (A, low, high):
    pivot ← A[(low + (high - low)) / 2]   # per evitare overflow
    i ← low - 1
    j ← high + 1
    loop forever:
        i ← i + 1
        while A[i] < pivot:
            j ← j - 1
        while A[j] > pivot:
            if i ≥ j then:
                return j
        swap A[i] with A[j]
```

```
QUICKSORT (A, low, high):
    if low < high then:
        p ← PARTITION (A, low, high)
        QUICKSORT (A, low, p)
        QUICKSORT (A, p+1, high)


QUICKSORT (A, 0, len(A)-1)
```

ANALISI:

- CASO PESSIMO: quando una delle 2 partizioni ha dimensione $n-1$ (pivot = max o min); le chiamate ricorsive si riducono ad n chiamate su un vettore ciascuna volta di dimensione $n-1$.

L'i-esima chiamata ha costo $O(n-i)$ $\Rightarrow$ $T(n) = \sum_{i=1}^{n} (n-i) = \sum_{i=1}^{n} i = \frac{n^2+n}{2} = O(n^2)$

- CASO MIGLIORE: Vettore partizionato esattamente a metà; Ci saranno $\log n$ chiamate, ciascuna di costo $O(n)$

$\Rightarrow$ $T(n) = O(n \log n)$; $T(n) = O(n) + 2T\left(\frac{n}{2}\right) \Rightarrow O(n \log n)$ (Per Master Theorem)

Se si riesce ad evitare il caso pessimo, è un buon algoritmo;

SI SCEGLIE IL PIVOT RANDOMICAMENTE A CASO!

• Radix Sort:

Utilizza un approccio CONTROINTUITIVO per l'uomo, ma molto efficiente per il calcolatore: compie un ordinamento (di tipo "bucket sort") per ciascuna cifra degli elementi;
ha complessità $T(m) = O(mk)$

1° Iterazione	2° Iterazione	3° Iterazione	4° Iterazione	Risultato
253	10	5	5	5
346	253	10	10	10
1034	1034	127	1034	127
10	5	1034	127	253
5	346	346	253	346
127	127	253	346	1034

Altre Proprietà degli Algoritmi di Ordinamento

(1) **Stabilità**: è stabile se preserva l'ordine iniziale tra due elementi con la stesse chiave; per esempio, l'ordinamento per nome e per cognome.

(2) **Ordinamento in Place**: un algoritmo che non crea copie dell'input per generare la sequenza ordinata

(3) **Adattatività**: se trae vantaggio degli elementi già ordinati, non andando a distruggere il loro ordine.

Algoritmo	T(m) – Caso Ottimo	T(m) – Caso Medio	T(m) – Caso Pessimo	S(m)	Stabile	In Place	Adattativo
Stupid Sort	$O(m)$	$O(m \cdot m!)$	∞	$O(m)$	NO	NO	NO
Selection Sort	$\Omega(m^2)$	$O(m^2)$	$\Theta(m^2)$	$\Theta(1)$	SÌ	SÌ	NO
Bubble Sort	$\Omega(m)$	$O(m^2)$	$O(m^2)$	$\Theta(1)$	SÌ	SÌ	SÌ
Insertion Sort	$\Theta(m)$	$O(m^2)$	$O(m^2)$	$\Theta(1)$	SÌ	SÌ	SÌ
Merge Sort	$\Theta(m \log m)$	$\Theta(m \log m)$	$\Theta(m \log m)$	$\Theta(m)$	SÌ	NO	NO
Bucket Sort	$O(m)$	$O(m)$	$O(m)$	$\Theta(m)$	SÌ	NO	NO
Quick Sort	$O(m \log m)$	$O(m \log m)$	$O(m^2)$	$\Theta(1)$	NO	SÌ	NO
Radix Sort	$O(mk)$	$O(mk)$	$O(mk)$	$\Theta(m)$	NO	NO	NO

Algoritmi di ordinamento: un riassunto

Algoritmo	T(n) - Caso ottimo	T(n) - Caso medio	T(n) - caso pessimo	S(n) (escluso input)	Stabile	In-Place	Adattativo
Stupid sort	$O(n)$	$O(n \cdot n!)$	∞	$O(n)$	No	No	No
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	No
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	Sì
Insertion sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì	Sì
Merge sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n)$	Sì	No	No
Bucket sort	$O(n)$	$O(n)$	$O(n^2)$	$\Theta(n)$	Sì	No	No
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$\Theta(1)$	No	Sì	No
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$\Theta(n)$	No	No	No

$O\left(\frac{n^2}{k} + n + k\right)$

Se pivot == max o min

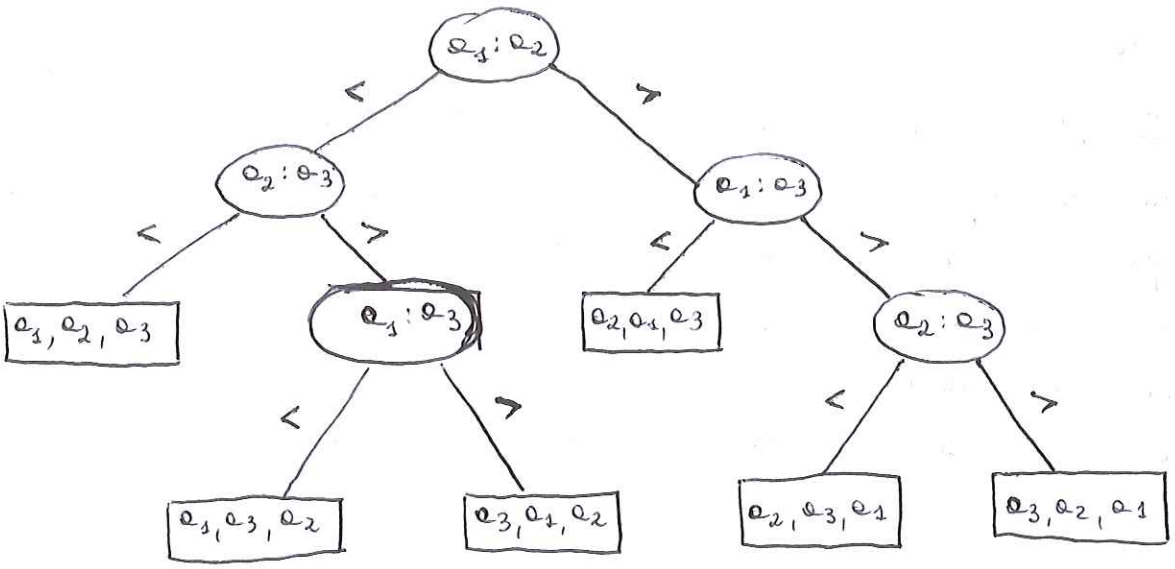
Tutti gli elementi in 1 bucket → Insertion Sort = $O(n^2)$

• Complessità Computazionale del Problema:

Teorema: La complessità temporale di un qualsiesi algoritmo di ordinamento per confronto è $T(n) = \Omega(N \log N)$, dove N è il numero di elementi da ordinare.

Infatti, l'algoritmo Bucket Sort ha complessità $O(n)$ poiché non è per confronto.

Dim Dimostriamolo tramite un albero di decisione:



Data una sequenza di input $a_1, \dots, a_n$, per ogni sequenza ci sarà un cammino all'interno dell'albero. Il numero possibile di permutazioni è $n!$, quindi $n!$ cammini.

$\Rightarrow$ L'albero avrà $n!$ foglie; l'altezza dell'albero sarà $h(T) \geq \lceil \log n! \rceil$, che corrisponde al numero di confronti che vengono eseguiti.

Approssimiamo $n!$ con la Formula di Stirling: $\lim_{n \to +\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{n!} = 1 \Rightarrow n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Quindi, $n! > \left(\frac{n}{e}\right)^n$; da cui $h(T) \geq \log\left(\frac{n}{e}\right)^n = n \log \frac{n}{e} = n \log n - n \log e = \underline{\Omega(n \log n)}$

ok!!!

4. Strutture Dati di Base

- **Definizioni:**
  - **Struttura Dati:** organizzazione sistematica dei dati e del loro accesso, che ne facilita la manipolazione;
  - **Algoritmo:** procedure suddivise in passi elementari che, eseguiti in sequenza, consentono di svolgere un compito in tempo finito.

- **Abstract Data Type:**

Il tipo di dato astratto (Abstract Data Type - ADT) è un insieme di oggetti ed un insieme di operazioni definite su di esso.

Specifica COSA fa ogni operazione, NON necessariamente COME.

Tipicamente un ADT definisce delle operazioni che possono organizzare tipi di dato differenti.

* In Python, possiamo usare il concetto di **Abstract Base Class (ABC)**:
- definiamo delle CLASSI che sono ASTRATTE per natura
- definiamo dei METODI all'interno di queste classi
- se una nuova classe estende la classe ABC, diventa OBBLIGATORIO implementare questi metodi.

- **Recap sulla Tipizzazione in Python:**

Python si basa sul concetto di "**Duck Typing**": Se parla e si comporta come una papera, allora è una papera.

Nei linguaggi interpretati, l'interprete tenta di invocare un metodo su un oggetto appartenente ad una classe: se il metodo esiste, tutto ok! Altrimenti vengono generate delle ECCEZIONI, che possono essere gestite con i costrutti `try ... except ... finally`.
Si può anche utilizzare `hasattr` per verificare se un oggetto dispone dell'implementazione di un determinato metodo.

- Il DUCK TYPING ha effetti interessanti:
  - possiamo avere oggetti che si comportano come file semplicemente implementando il METODO `read` all'interno della classe;
  - possiamo avere oggetti ITERABILE implementando il metodo `__iter__`.
- Un oggetto, indipendentemente dalla sua classe o tipo, può essere conforme ad una certa interfaccia in funzione del PROTOCOLLO che implementa;

Esempio:

```
class Team:
    def __init__(self, members):
        self.__members = members

    def __len__(self):
        return len(self.__members)

    def __contains__(self, member)
        return member in self.__members

# members è una lista
# la classe include i metodi di len e contains
```