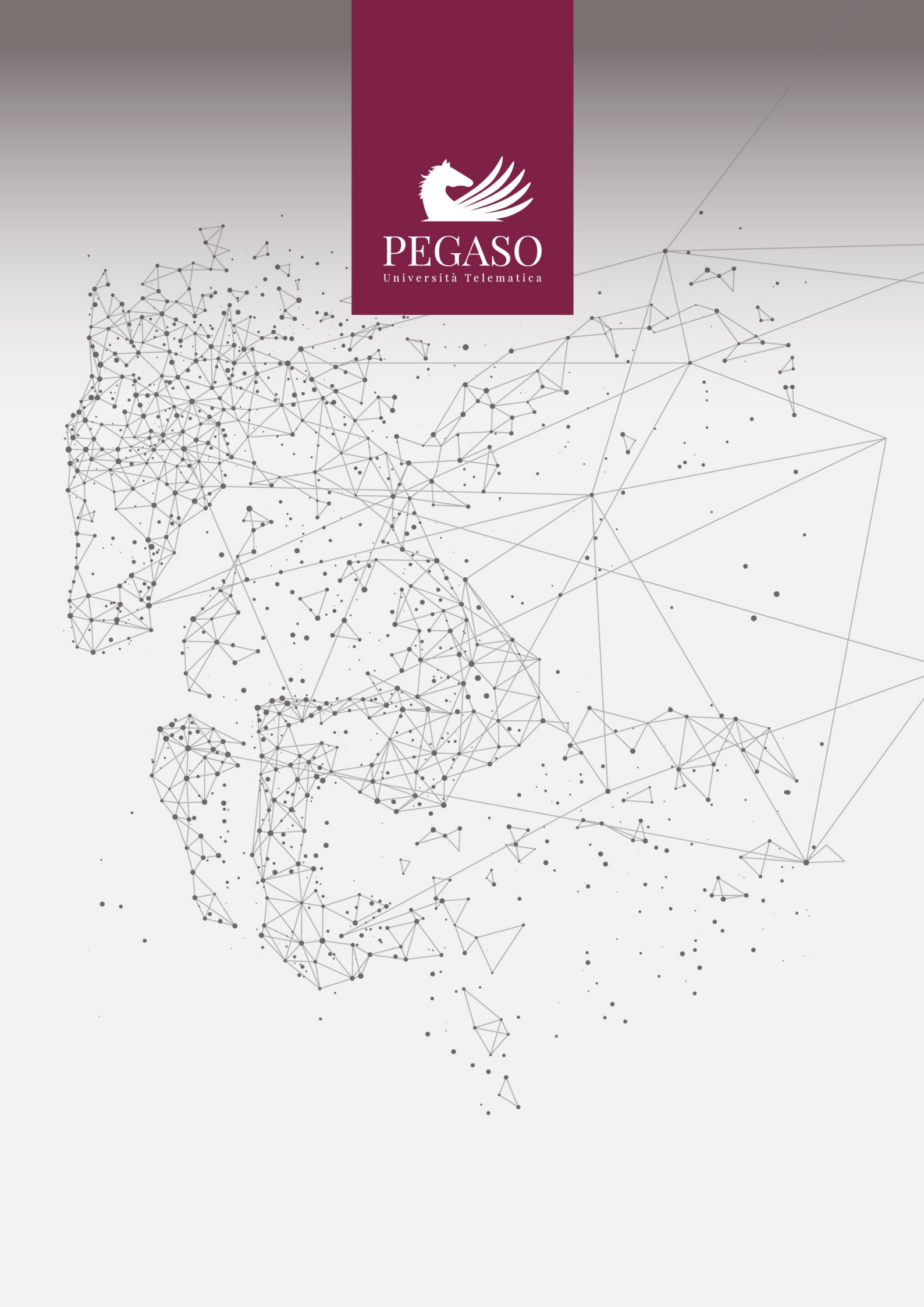
PEGASO
Università Telematica

IL PROBLEMA DELL'ORDINAMENTO

Il problema dell'ordinamento può essere definito "formalmente" nella seguente maniera: data una sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$ un ordinamento è una permutazione (un ri-arrangiamento) $\langle a'_1, a'_2, \dots, a'_n \rangle$ della sequenza di input tale che $1 \leq a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Più semplicemente possiamo dire che l'ordinamento di una sequenza di informazioni consiste nel disporre le stesse informazioni in modo da rispettare una qualche relazione d'ordine di tipo lineare (ad esempio una relazione d'ordine "minore o uguale" dispone le informazioni in modo "non decrescente").

Oltre che per il loro principio di funzionamento e per la loro efficienza, gli algoritmi di ordinamento possono essere confrontati in base ai seguenti criteri:

- **Stabilità**: un algoritmo di ordinamento è stabile se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave. Algoritmi di questo tipo evitano interferenze con ordinamenti pregressi dello stesso array basati su chiavi secondarie. Se ad esempio si ordina per anno di corso una lista di studenti già ordinata alfabeticamente, un metodo stabile produce una lista in cui gli alunni dello stesso anno sono ancora in ordine alfabetico mentre un ordinamento instabile probabilmente produrrà una lista senza più alcuna traccia del precedente ordinamento.
- **Sul posto (in place)**: un algoritmo di ordinamento opera in place se la dimensione delle strutture ausiliarie di cui necessita è indipendente dal numero di elementi dell'array da ordinare. In altre parole: un algoritmo in place non crea una copia dell'input per raggiungere l'obiettivo (l'ordinamento), pertanto un algoritmo in place risparmia memoria rispetto ad un algoritmo non in place. Si intuisce quanto sui grandi numeri la proprietà "in place" sia rilevante.

È inoltre possibile classificare in base alla complessità del tempo di calcolo. La complessità di calcolo si riferisce soprattutto al numero di operazioni necessarie all'ordinamento (principalmente operazioni di confronto e scambio), in funzione del numero di elementi da ordinare:

- **Algoritmi Semplici di Ordinamento**: algoritmi che presentano una complessità proporzionale a n^2, essendo n è il numero di informazioni da ordinare; essi sono generalmente caratterizzati da poche e semplici istruzioni.
- **Algoritmi Evoluti di Ordinamento**: algoritmi che offrono una complessità computazionale proporzionale ad $n \log_2 n$ (che è sempre inferiore a n^2). Tali algoritmi sono molto più complessi e fanno molto spesso uso di ricorsione. La convenienza del loro utilizzo si ha unicamente quando il numero n di informazioni da ordinare è molto elevato.

Da precisare che è possibile dimostrare che il problema dell'ordinamento non può essere risolto con un algoritmo di complessità asintotica inferiore a quella pseudo-lineare: per ogni algoritmo che ordina un array di n elementi, il tempo d'esecuzione soddisfa $T(n) = \Omega(n \cdot \log_2 n)$.

Riportiamo schematicamente le caratteristiche dei principali algoritmi in termini di complessità e criteri di confronto:

Nome	*Migliore*	*Medio*	*Peggiore*	*Memoria*	*Stabile*	*In place*
Bubble sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Sì	Sì
Heap sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$\Theta(1)$	No	Sì
Insertion sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Sì	Sì
Merge sort	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$O(n)$	Sì	No
Quick sort	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$O(n^2)$	$O(n)$	No	Sì
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	No	Sì

Ricordiamo brevemente il significato delle notazioni asintotiche:

- **Notazione asintotica O** (notazione O grande): limite superiore asintotico
- **Notazione asintotica Ω** (notazione Omega): limite inferiore asintotico
- **Notazione asintotica θ** (notazione Theta): limite asintotico stretto (se una funzione è $\theta(g(n))$ allora è anche $O(g(n))$ e $\Omega(g(n))$)

SELECTION SORT

L'algoritmo di ordinamento Insert Sort (ordinamento diretto) nasce dall'idea che per ottenere un array ordinato basta costruirlo ordinato, inserendo gli elementi al posto giusto fin dall'inizio.

Idealmente, il metodo costruisce un nuovo array, contenente gli stessi elementi del primo, ma ordinato. In pratica, non è necessario costruire un secondo array, in quanto le stesse operazioni possono essere svolte direttamente sull'array originale: così, alla fine esso risulterà ordinato.

Insert sort, è dunque un algortmo di ordinamento iterativo che al generico passo i vede l'array diviso in una sequenza di destinazione $v[0], \ldots, v[k-1]$ già ordinata e una sequenza di origine $v[k], \ldots, v[n-1]$ ancora da ordinare: l'obiettivo è di inserire il valore contenuto in $a[k]$ al posto giusto nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ridurre la sequenza di origine di un elemento.

Di seguito riportiamo una animazione[1] che ci dà un'idea di come opera Insertion Sort:

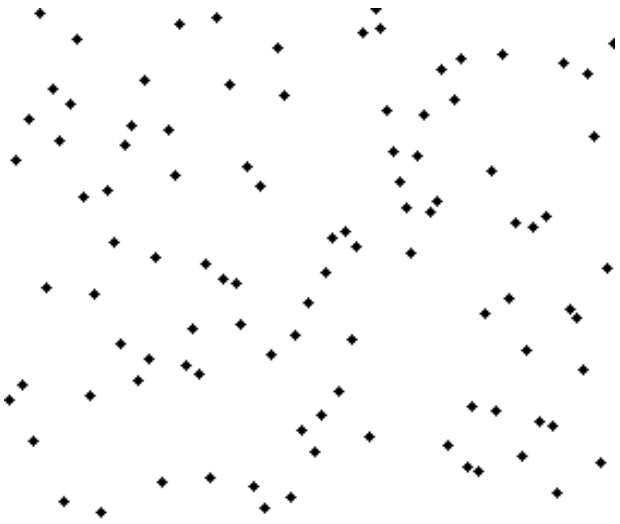

Figura 1 - Insertion Sort Animation

Pertanto "vecchio" e "nuovo" array condividono lo stesso array fisico di N celle (da 0 a $N-1$) ed in ogni istante, le prime K celle (numerate da 0 a *K-1*) costituiscono il nuovo array mentre le successive *N-K* celle costituiscono la parte residua dell'array originale.

[1] Insertion sort animation - Insertion sort - Wikipedia

Come conseguenza della scelta di progetto fatta, in ogni istante il nuovo elemento da inserire si trova nella cella successiva alla fine del nuovo array, cioè la $(K+1)-esima$ (il cui indice è K)

Analizziamo con un esempio il suo funzionamento; supponiamo di avere il seguente array di 8 elementi (indice da 0 a 7):

41 37 10 74 98 22 83 66

Si considera il secondo elemento della sequenza (in questo caso il 37, evidenziato in rosso):

41 37 10 74 98 22 83 66

Tale elemento lo si deve portare nella posizione "giusta" (a sinistra) e quindi va portato in prima posizione e scambiato con il 41

37 41 10 74 98 22 83 66

Si considera il terzo elemento della sequenza (in questo caso il 10, evidenziato in rosso): 0 a 6):

37 41 10 74 98 22 83 66

Tale elemento lo si deve portare nella posizione "giusta" (a sinistra) e quindi va portato in prima posizione eseguendo così 2 scambi:

10 37 41 74 98 22 83 66

Si considera il quarto elemento della sequenza (in questo caso il 74, evidenziato in rosso):

10 37 41 74 98 22 83 66

Tale elemento lo si deve portare nella posizione "giusta" che è esattamente quella in cui si trova; si considera dunque il quinto elemento della sequenza (in questo caso il 98, evidenziato in rosso):

10 37 41 74 98 22 83 66

Tale elemento lo si deve portare nella posizione “giusta” che è esattamente quella in cui si trova; si considera dunque il sesto elemento della sequenza (in questo caso il 22, evidenziato in rosso):

10 37 41 74 98 22 83 66

Tale elemento lo si deve portare nella posizione “giusta” che è la seconda e pertanto dovranno essere eseguiti 4 scambi:

10 22 37 41 74 98 83 66

Si considera il settimo elemento della sequenza (in questo caso l’83, evidenziato in rosso):

10 22 37 41 74 98 83 66

Tale elemento lo si deve portare nella posizione “giusta” che è la sesta e pertanto dovrà essere eseguito 1 scambio:

10 22 37 41 74 83 98 66

Si considera l’ultimo elemento della sequenza (in questo caso il 66, evidenziato in rosso):

10 22 37 41 74 83 98 66

Tale elemento lo si deve portare nella posizione “giusta” che è la quinta e pertanto dovranno essere eseguiti 3 scambi:

10 22 37 41 66 74 83 98

L’array è dunque ordinato.

Il caso ottimo si verifica quando l'array è già ordinato e la complessità è lineare $O(n)$; il caso pessimo si verifica quando l'array è inversamente ordinato ed in questo caso la complessità è quadratica: $O(n^2)$

IMPLEMENTAZIONE

Passiamo all'implementazione dell'algoritmo usando prima lo pseudocode:

```
DECLARE a: ARRAY[0:10] OF INTEGER
a[0]<-41
a[1]<-37
a[2]<-10
a[3]<-74
a[4]<-98
a[5]<-22
a[6]<-83
a[7]<-66

DECLARE n: INTEGER
n<-8

DECLARE i: INTEGER
DECLARE j: INTEGER
DECLARE value: INTEGER
```

Figura 2 - Pseudocode - dichiarazione strutture dati

```
FOR i <- 1 TO n-1
  value <- a[i]
  j <- i-1

  DECLARE continua: BOOLEAN
  continua <- TRUE

  WHILE continua DO
    IF a[j]>value THEN
      a[j+1] <- a[j]
      j <- j-1
    ELSE
      continua <- FALSE
    ENDIF
    IF j<0 THEN
      continua <- FALSE
    ENDIF
  ENDWHILE

  a[j+1] <- value
NEXT i
```

Figura 3 – Algoritmo in Pseudocode

Attenzione: a causa di un BUG dell'ambiente di sviluppo se nella condizione while inserisco (cond1 AND cond2), il fatto che cond1 non è verificata non fa automaticamente

uscire dal ciclo ma esegue comunque la cond2 (causando un errore); per questo si è dovuto aggirare il problema con l'implementazione di cui sopra.

Di seguito l'implementazione in C++:

```
#include <iostream>

using namespace std;

int main() {

  int a[8]={41, 37, 10, 74, 98, 22, 83, 66};
  int n=8;

  for (int i=1;i<n;i++) {
    int value=a[i];
    int j=i-1;
    while (j>=0 && a[j]>value) {
      a[j + 1]=a[j];
      j=j-1;
    }
    a[j+1]=value;
  }

  for (int i=0;i<n;i++)
    cout<<a[i]<<" ";
}
```

L'implementazione in Python è invece la seguente:

```
a=[41, 37, 10, 74, 98, 22, 83, 66]
n=8

for i in range(1,n):
  value=a[i]
  j=i-1;
  while j>=0 and a[j]>value:
    a[j + 1]=a[j]
    j=j-1

  a[j+1]=value

print(a)
```

Analizzando l'implementazione ragioniamo in termini di:

- **Stabilità**: usando la condizione di $<$ e non $\leq$ non altera gli elementi chiave ed in generale l'algoritmo è stabile

- **In place**: non introduce strutture dati ausiliarie che dipendono dalla grandezza dell'array da ordinare pertanto è "in place"

Chiaramente la complessità sarà pari a:

$$T(N) = O(N^2/2)$$

Avendo a che fare con 2 cicli for uno annidato all'altro.

Bibliografia e sitografia

Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione

C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione

Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley

Sedgewick: Algoritmi in C, Pearson, 2015

Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione