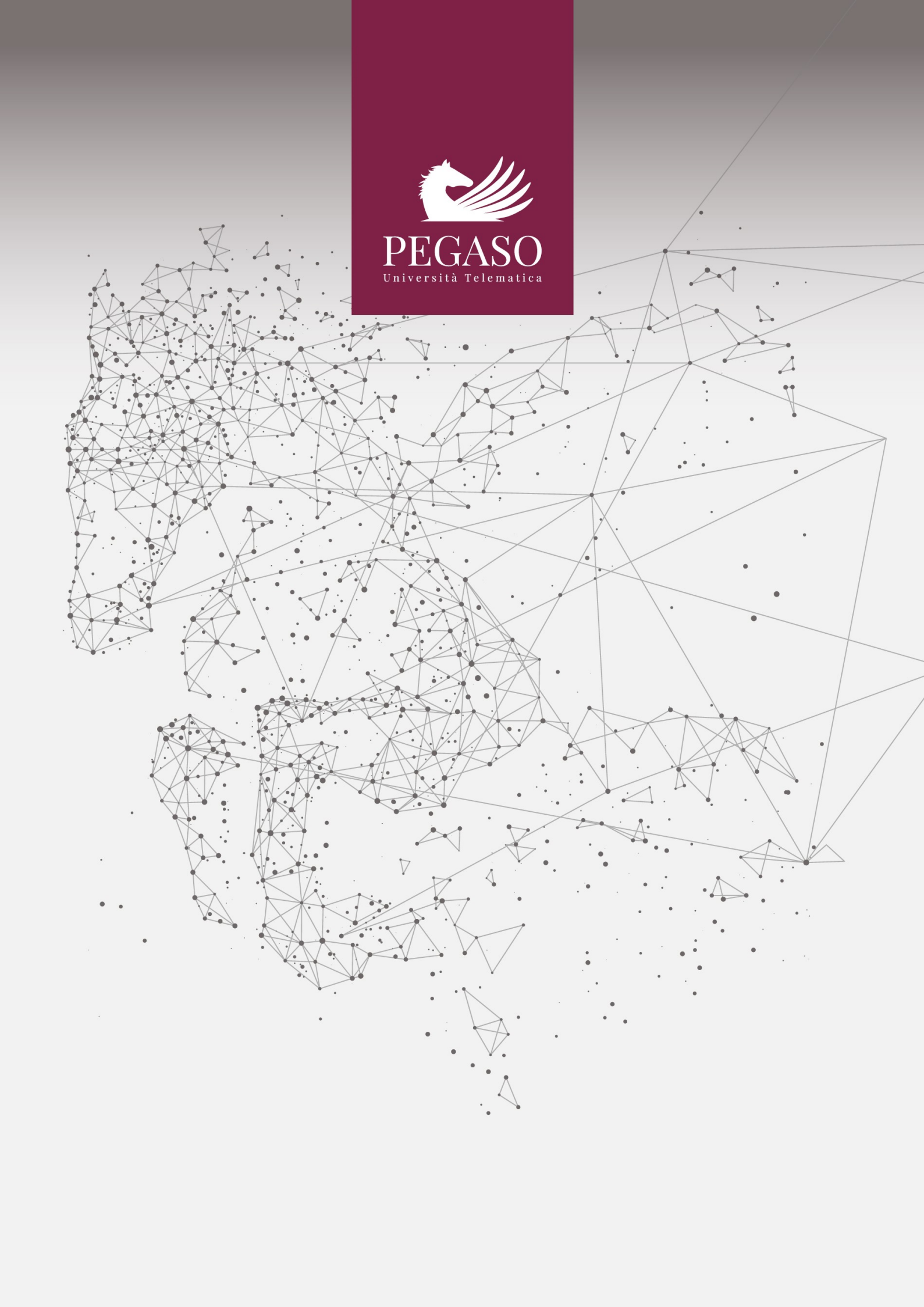
PEGASO
Università Telematica

Indice

1. Il problema dell'ordinamento

Il problema dell'ordinamento può essere definito "formalmente" nella seguente maniera: data una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$ un ordinamento è una permutazione (un ri-arrangiamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ 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 $\Omega(g(n))$)

2. Selection sort

L'algoritmo Selection Sort è anche chiamato Naive Sort dal momento che è un algoritmo "ingenuo", molto semplice ed intuitivo.

Tale algoritmo è basato su un processo iterativo che ha come obiettivo la selezione dell'elemento della sequenza di origine che contiene il valore maggiore (nel caso di ordinamento crescente) e di scambiare tale valore con il valore contenuto nell'ultima posizione, in modo da ridurre la sequenza di origine di un elemento.

Proviamo ad immaginare di essere di fronte a un tavolo da biliardo con tutte le palle sul tavolo, disposte a caso. Se volessimo ordinarle, una tecnica potrebbe essere quella di individuare quella con il numero più alto (selection) e metterla "in fondo" a una ipotetica fila, proseguendo nella stessa modalità con le palle rimanenti. Questo è come opera l'algoritmo Selection Sort.

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

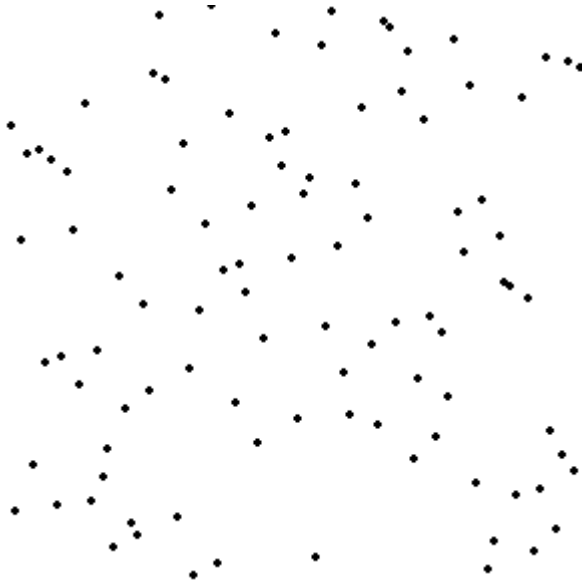

Figura 1 *- Selection Sort Animation*

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

[1]Selection sort animation - Selection sort - Wikipedia

Si individua il massimo della sequenza (in questo caso il 98, evidenziato in rosso):

41 37 10 74 98 22 83 66

Lo si porta in ultima posizione (quella di indice 7) scambiandolo con l'elemento ivi presente:

41 37 10 74 66 22 83 98

Si cerca a questo punto l'elemento più grande tra quelli nelle prime 7 posizioni (da indice 0 a 6):

41 37 10 74 66 22 83 98

Si individua il numero 83 che si trova già in ultima posizione:

41 37 10 74 66 22 83 98

Si cerca a questo punto l'elemento più grande tra quelli nelle prime 6 posizioni (da indice 0 a 5):

41 37 10 74 66 22 83 98

Lo si porta in ultima posizione scambiandolo con l'elemento ivi presente:

41 37 10 22 66 74 83 98

Si prosegue in questa modalità fino ad arrivare all'ordinamento completo:

10 22 37 41 66 74 83 98

Si è arrivati a questa conclusione alla settima iterazione.

Per valutare la complessità analizziamo il numero di confronti.

$$(N-1)+(N-2)+(N-3)+\cdots+2+1 = N*(N-1)/2 \;=\; O(N^2/2)$$

3. Implementazione

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

```
// FIX -2

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 max: INTEGER
DECLARE temp: INTEGER

FOR i <- n-1 TO -2 STEP -1
 max <- i
 FOR j <- i-1 TO -2 STEP -1
  IF a[j] > a[max] THEN
   max <- j
  ENDIF
 NEXT j
 IF max <> i THEN
```

```
    temp <- a[i]
    a[i] <- a[max]
    a[max] <- temp
  ENDIF
NEXT i

FOR i<-0 TO n-1
  OUTPUT a[i]
NEXT i
```

Attenzione: a causa di un BUG dell'ambiente di sviluppo è necessario inserire nel ciclo for il valore sul TO a -2 anziché a 0.

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=n-1;i>=0;i--) {
    int max=i;
    for (int j=i-1;j>=0;j--) {
      if (a[j]>a[max])
        max=j;
    }
    if (max!=i) {
      int temp=a[i];
      a[i]=a[max];
      a[max]=temp;
    }
  }
```

```
        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(n-1,-1,-1):
  max=i
  for j in range(i-1,-1,-1):
    if a[j]>a[max]:
      max=j
  if max!=i:
    temp=a[i]
    a[i]=a[max]
    a[max]=temp

print(a)
```

Analizzando l'implementazione ragioniamo in termini di:

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

- 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