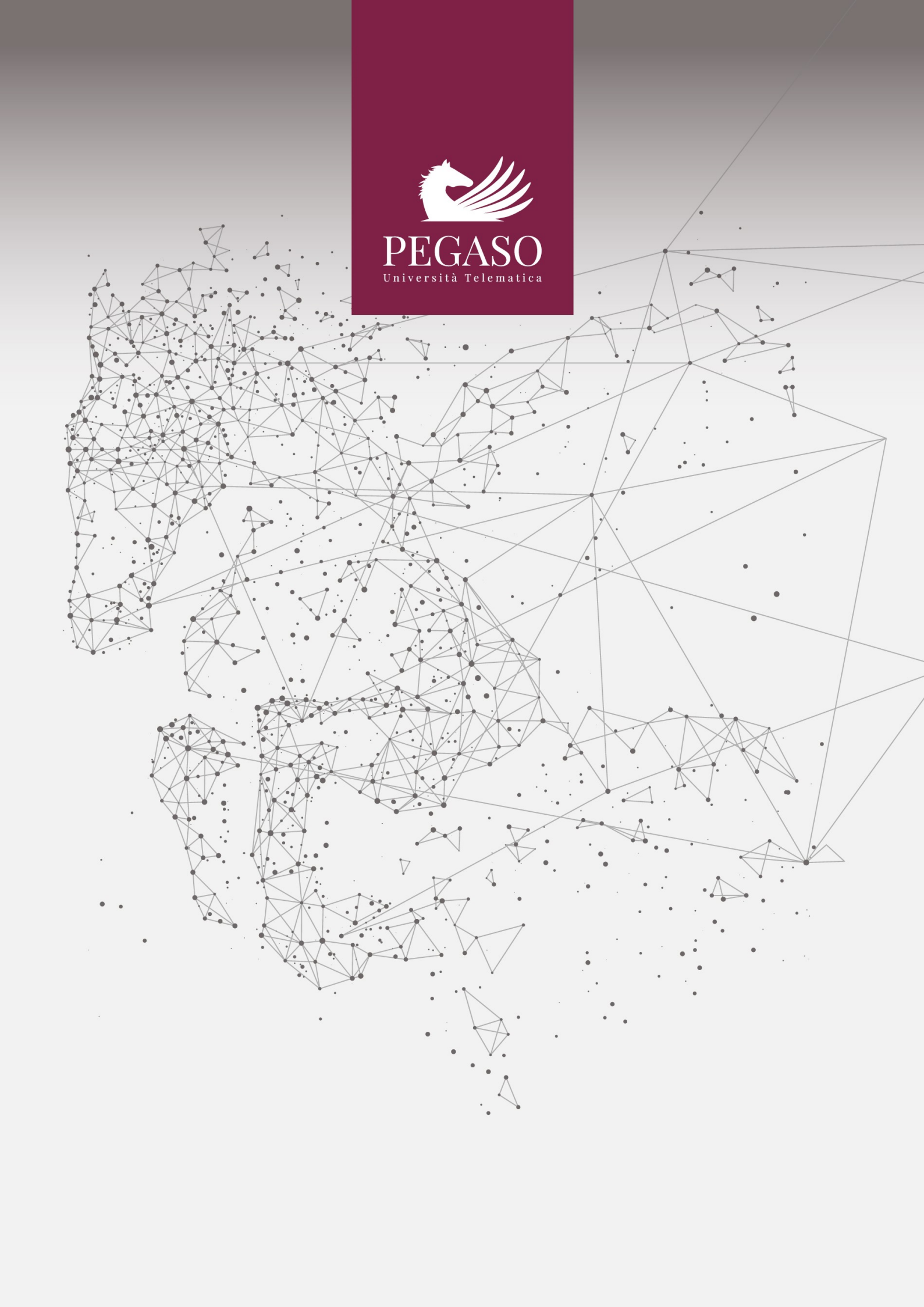
PEGASO
Università Telematica

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. Bubble sort

L'algoritmo Bubble sort tenta di corregge il difetto principale del Selection sort che quello di non accorgersi se l'array, a un certo punto, è già ordinato.

Bubble Sort (ordinamento a bolla) è un semplice algoritmo di ordinamento basato sullo scambio di elementi adiacenti: ogni coppia di elementi adiacenti viene comparata e invertita di posizione se sono nell'ordine sbagliato. L'algoritmo continua nuovamente a rieseguire questi passaggi per tutta la sequenza di elementi finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata.

L'algoritmo deve il suo nome al modo in cui gli elementi vengono ordinati cioè quelli più piccoli "risalgono" verso un'estremità della lista, così come fanno le bollicine in un bicchiere di spumante; al contrario quelli più grandi "affondano" verso l'estremità opposta della sequenza.

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

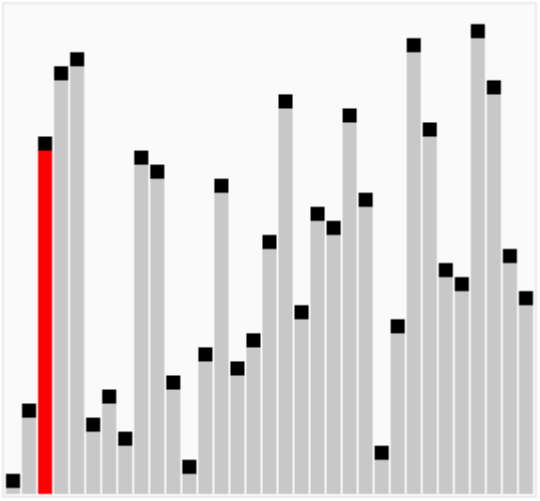

Figura 1 *- Bubble Sort Animation*

Il Bubble sort è un algoritmo iterativo, ossia basato sulla ripetizione di un procedimento fondamentale: la singola iterazione dell'algoritmo prevede che gli elementi dell'array siano confrontati a due a due procedendo in un verso stabilito (la scelta del verso non è significativa): per esempio, saranno confrontati il primo e il secondo elemento, poi il secondo e il terzo, poi il terzo e il quarto, e così via fino al confronto fra il penultimo e l'ultimo elemento.

[1] Sorting bubblesort anim - Bubble sort - Wikipedia

Ad ogni confronto, se i due elementi confrontati non sono ordinati secondo il criterio prescelto, vengono scambiati di posizione: durante ogni iterazione almeno un valore viene spostato rapidamente fino a raggiungere la sua collocazione definitiva, in particolare, alla prima iterazione il numero più grande raggiunge l'ultima posizione dell'array, alla seconda il secondo numero più grande raggiunge la penultima posizione, e così via.

Al generico passo i si ha pertanto l'array diviso in una sequenza di destinazione $a[0], a[1], \dots a[i-1]$ già ordinata, e una sequenza di origine $a[i], a[i+1], \dots a[n-1]$ ancora da ordinare. Come già detto, l'obiettivo è di far emergere (come se fosse una bollicina) il valore minimo della sequenza di origine confrontando e scambiando sistematicamente i valori di elementi adiacenti a partire dalla fine dell'array, in modo da ridurre la sequenza di origine di un elemento.

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 primo elemento della sequenza (in questo caso il 41 evidenziato in rosso) e lo si confronta con il secondo elemento della sequenza (in questo caso il 37 evidenziato in verde):

41 37 10 74 98 22 83 66

Essendo 41>37 si deve eseguire lo scambio:

37 41 10 74 98 22 83 66

Si prosegue confrontando il secondo elemento (il 41 in rosso) con il terzo (il 10 in verde):

37 41 10 74 98 22 83 66

Essendo 41>10 si esegue lo scambio:

37 10 41 74 98 22 83 66

Si prosegue confrontando il terzo elemento con il quarto, ma in questo caso 41<74 e quindi non si esegue alcuno scambio:

37 10 41 74 98 22 83 66

Si prosegue confrontando il quarto elemento con il quinto ed essendo 74<98, anche in questo caso non si esegue alcuno scambio:

37 10 41 74 98 22 83 66

Si prosegue confrontando il quinto elemento con il sesto:

37 10 41 74 98 22 83 66

Essendo 98<22 si esegue lo scambio:

37 10 41 74 22 98 83 66

Si prosegue confrontando il sesto elemento con il settimo:

37 10 41 74 22 98 83 66

Essendo 98>83 si esegue lo scambio:

37 10 41 74 22 83 98 66

Si prosegue confrontando il settimo elemento con l'ottavo:

37 10 41 74 22 83 98 66

Essendo 98>66 si esegue lo scambio:

37 10 41 74 22 83 66 98

Quello che è accaduto al termine della prima iterazione è che l'elemento più grande della sequenza (il 98) è andato nel fondo della sequenza (come la bollicina raggiunge la sommità). Se facciamo ora una seconda iterazione, accadrà che il secondo elemento più grande finirà in penultima posizione; illustriamo rapidamente:

37 10 41 74 22 83 66 98 → 10 37 41 74 22 83 66 98

10 37 41 74 22 83 66 98 → nessun cambio

10 37 41 74 22 83 66 98 → nessun cambio

10 37 41 74 22 83 66 98 → 10 37 41 22 74 83 66 98

10 37 41 74 22 83 66 98 → nessun cambio

10 37 41 74 22 83 66 98 → 10 37 41 74 22 66 83 98

Proseguendo in questa modalità si otterrà l'array ordinato dopo 7 iterazioni.

Generalizzando:

- se i numeri sono in tutto n, dopo $n-1$ iterazioni si avrà la garanzia che l'array sia ordinato
- alla iterazione $i-esima$, le ultime $i-1$ celle dell'array ospitano i loro valori definitivi, per cui la sequenza di confronti può essere terminata col confronto dei valori alle posizioni $n-1-i$ e $n-i$.

Come si evince dall'esempio precedente, può accadere che più numeri vengano spostati nel corso di una singola iterazione, per cui, oltre a portare il numero più grande in fondo, ogni singola iterazione può contribuire anche a un riordinamento parziale degli altri valori. Può dunque accadere (e normalmente accade) che l'array risulti effettivamente ordinato prima che si sia raggiunta la $n-1esima$ iterazione.

Il Bubble sort non è comunque un algoritmo efficiente, effettuando all'incirca $\frac{n^2}{2}$ confronti ed $\frac{n^2}{2}$ scambi sia in media che nel caso peggiore; il tempo di esecuzione dell'algoritmo è pertanto $O(n^2)$.

3. 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 scambio: BOOLEAN
DECLARE i: INTEGER
DECLARE temp: INTEGER

scambio <- TRUE
WHILE scambio DO
  scambio <- FALSE
  FOR i <- 0 TO n-2
    IF a[i]>a[i+1] THEN
      temp <- a[i]
      a[i] <- a[i+1]
      a[i+1] <- temp
      scambio <- TRUE
    ENDIF
  NEXT i
ENDWHILE

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

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;

  bool scambio=true;

  while (scambio) {
    scambio=false;
    for (int i=0;i<n-1;i++) {
```

```
            if (a[i]>a[i+1]) {
              int temp=a[i];
              a[i]=a[i+1];
              a[i+1]=temp;
              scambio=true;
            }
          }
        }

        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

scambio=True

while (scambio):
  scambio=False
  for i in range(n-1):
    if a[i]>a[i+1]:
      temp=a[i]
      a[i]=a[i+1]
      a[i+1]=temp
      scambio=True

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:

  $$T(n) = O(n^2)$$

  Dovendo iterare su tutti gli elementi e per ogni iterazione deve iterare sul sottoinsieme relativo.

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