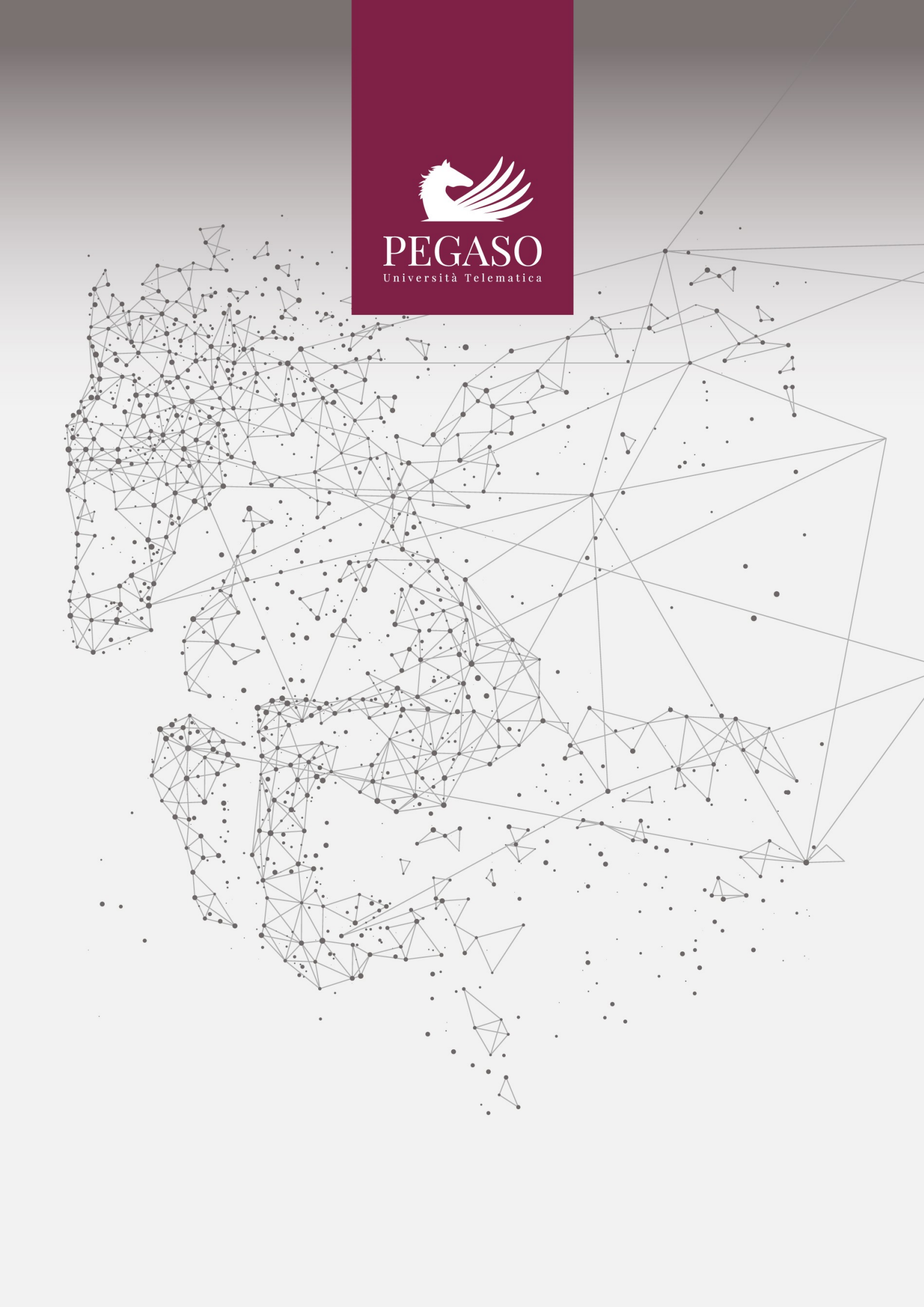
PEGASO
Università Telematica

Indice

1. Il problema dell'ordinamento

Il problema dell'ordinamento può essere definito "formalmente" nella seguente maniera: data una sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$ 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 $\Omega(g(n))$)

2. Merge Sort

Il merge sort è un algoritmo di ordinamento ricorsivo proposto da Von Neumann nel 1945; è basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sotto-problemi della stessa natura di dimensione via via più piccola.

Concettualmente, l'algoritmo funziona nel seguente modo:

- Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata
- Altrimenti:
    - la sequenza viene divisa (**divide**) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sotto-sequenze di cui la prima ha un elemento in più della seconda)
    - ognuna di queste sotto-sequenze viene ordinata, applicando ricorsivamente l'algoritmo (**impera**)
    - le due sotto-sequenze ordinate vengono fuse (**combina**). Per fare questo, si estrae ripetutamente il minimo delle due sotto-sequenze e lo si pone nella sequenza in uscita, che risulterà ordinata

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

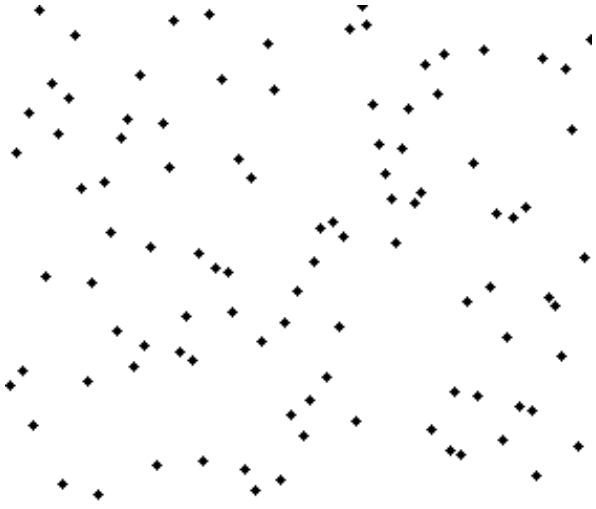

Figura 1 *- Merge Sort Animation*

[1] Merge sort animation2 - Merge sort - Wikipedia

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 esegue una suddivisione in 2 sotto-sequenze:

[41 37 10 74] [98 22 83 66]

Per ognuna delle sotto-sequenze si esegue una nuova suddivisione:

[41 37] [10 74] [98 22] [83 66]

Per ognuna delle sotto-sequenze si esegue una nuova suddivisione:

[41] [37] [10] [74] [98] [22] [83] [66]

Si procede in sostanza fino ad ottenere elementi singoli. A questo punto si procede alla combinazione dei singoli elementi, 2 a 2 (ordinandoli):

[41] + [37] → [37] [41]

[10] + [74] → [10] [74]

[98] + [22] → [22] [98]

[83] + [66] → [66] [83]

Si procede poi con la combinazione delle coppie:

[37] [41] + [10] [74] → [10] [37] [41] [74]

[22] [98] + [66] [83] → [22] [66] [83] [98]

Si finisce con la combinazione degli ultimi 2 set:

[10] [37] [41][10] [74] + [22] [66] [83] [98] → [10] [22] [37] [41] [66] [74] [83] [98]

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 b: ARRAY[0:10] OF INTEGER

DECLARE n: INTEGER
n<-8

PROCEDURE merge(left: INTEGER, center: INTEGER, right: INTEGER)

  DECLARE i: INTEGER
  DECLARE j: INTEGER
  DECLARE k: INTEGER
  DECLARE size: INTEGER

  i <- left
  j <- center + 1
  k <- 0
  size <- right-left+1

  DECLARE b: ARRAY[0:right-left+1] OF INTEGER

  WHILE i <= center AND j <= right DO
    IF a[i] <= a[j] THEN
      b[k] <- a[i]
      i <- i + 1
    ELSE
      b[k] <- a[j]
      j <- j + 1
    ENDIF
    k <- k + 1
  ENDWHILE

  WHILE i <= center DO
     b[k] <- a[i]
     i <- i + 1
     k <- k + 1
  ENDWHILE

  WHILE j <= right DO
     b[k] <- a[j]
     j <- j + 1
     k <- k + 1
  ENDWHILE
```

```
  FOR k <- left TO right
    a[k] <- b[k-left]
  NEXT k

ENDPROCEDURE

PROCEDURE mergesort(left: INTEGER, right: INTEGER)
  IF left < right THEN
    DECLARE center: INTEGER
      center <- (left + right) DIV 2
      CALL mergesort(left, center)
      CALL mergesort(center+1, right)
      CALL merge(left, center, right)
  ENDIF
ENDPROCEDURE

CALL mergesort(0,7)

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

ATTENZIONE: lo pseudocode di cui sopra è puramente a carattere dimostrativo; all'interno di Replit CIE Pseudocode non è funzionante dal momento che vi sono delle restrizioni:

- all'interno di Replit CIE Pseudocode non è funzionante dal momento che vi sono delle restrizioni nell'esecuzione di codici ricorsivi;
- vi è inoltre una problematica legata al passaggio di array come parametri di funzione; pertanto, è necessario lavorare con array globali;
- vi sono problemi nella definizione di array all'interno di funzioni.

Di seguito l'implementazione in C++:

```
#include <iostream>

using namespace std;

void merge (int a[], int left, int center, int right) {
  int i=left;
  int j=center + 1;
  int k=0;
  int b[right-left+1];
  for (int i=0;i<right-left+1;i++)
    b[i]=0;

  while ((i <= center) && (j <= right)) {
    if (a[i] <= a[j]) {
      b[k] = a[i];
      i++;
    } else {
          b[k] = a[j];
          j++;
```

```
      }
      k++;
    }

    while (i <= center) {
       b[k] = a[i];
       i++;
       k++;
    }

    while (j <= right) {
      b[k] = a[j];
      j++;
      k++;
    }

    for (k=left;k<=right;k++)
      a[k] = b[k-left];
}

void mergesort(int a[], int left, int right) {
  if (left < right) {
       int center = (left + right) / 2;
       mergesort(a, left, center);
       mergesort(a, center+1, right);
       merge(a, left, center, right);
  }
}

int main() {
  int a[8]={41, 37, 10, 74, 98, 22, 83, 66};
  int n=8;
  mergesort(a,0,n-1);

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

L'implementazione in Python è invece la seguente:

```
def merge(a,left,center,right):
  i=left
  j=center + 1
  k=0
  b=[0]*(right-left+1)

  while i <= center and j <= right:
    if a[i] <= a[j]:
      b[k] = a[i]
      i+=1
    else:
          b[k] = a[j]
          j+=1
    k+=1

  while i <= center:
```

```
            b[k] = a[i]
            i+=1
            k+=1

        while j <= right:
          b[k] = a[j]
          j+=1
          k+=1

        for k in range(left,right+1):
          a[k] = b[k-left]


      def mergesort(a, left,right):
        if left < right:
            center = (left + right) // 2
            mergesort(a, left, center)
            mergesort(a, center+1, right)
            merge(a, left, center, right)



      a=[41, 37, 10, 74, 98, 22, 83, 66]
      n=8
      mergesort(a,0,n-1)
      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**: l'algoritmo non opera sul posto in quanto nella fusione ordinata viene usato un array di appoggio il cui numero di elementi µe proporzionale al numero di elementi dell'array da ordinare.

La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell'array. Se $n = right - left + 1$ è la dimensione della porzione di array in esame, poiché il tempo d'esecuzione dell'operazione di fusione ordinata è proporzionale ad n, il tempo d'esecuzione dell'algoritmo è dato dalla relazione di ricorrenza lineare:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + (c_1 n + c_2)$$

Infatti, la funzione merge qui presentata ha complessità temporale $O(n)$ e mergesort richiama sé stessa due volte, e ogni volta su (circa) metà della sequenza in input; da questo segue che il tempo di esecuzione dell'algoritmo è dato dalla ricorrenza di cui sopra, la cui soluzione è:

$$T(n) = O(n \log_2 n)$$

Precisiamo che la tecnica di implementazione illustrata sopra, è di tipo: Top-Down; si opera cioè su un insieme e lo si divide in sottoinsiemi fino ad arrivare all'insieme contenente un solo elemento, per poi riunire le parti scomposte.

Vi è anche un'altra tecnica: Bottom-Up; si considera l'insieme come composto da un vettore di n sequenze e ad ogni passo vengono fuse due sequenze.

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