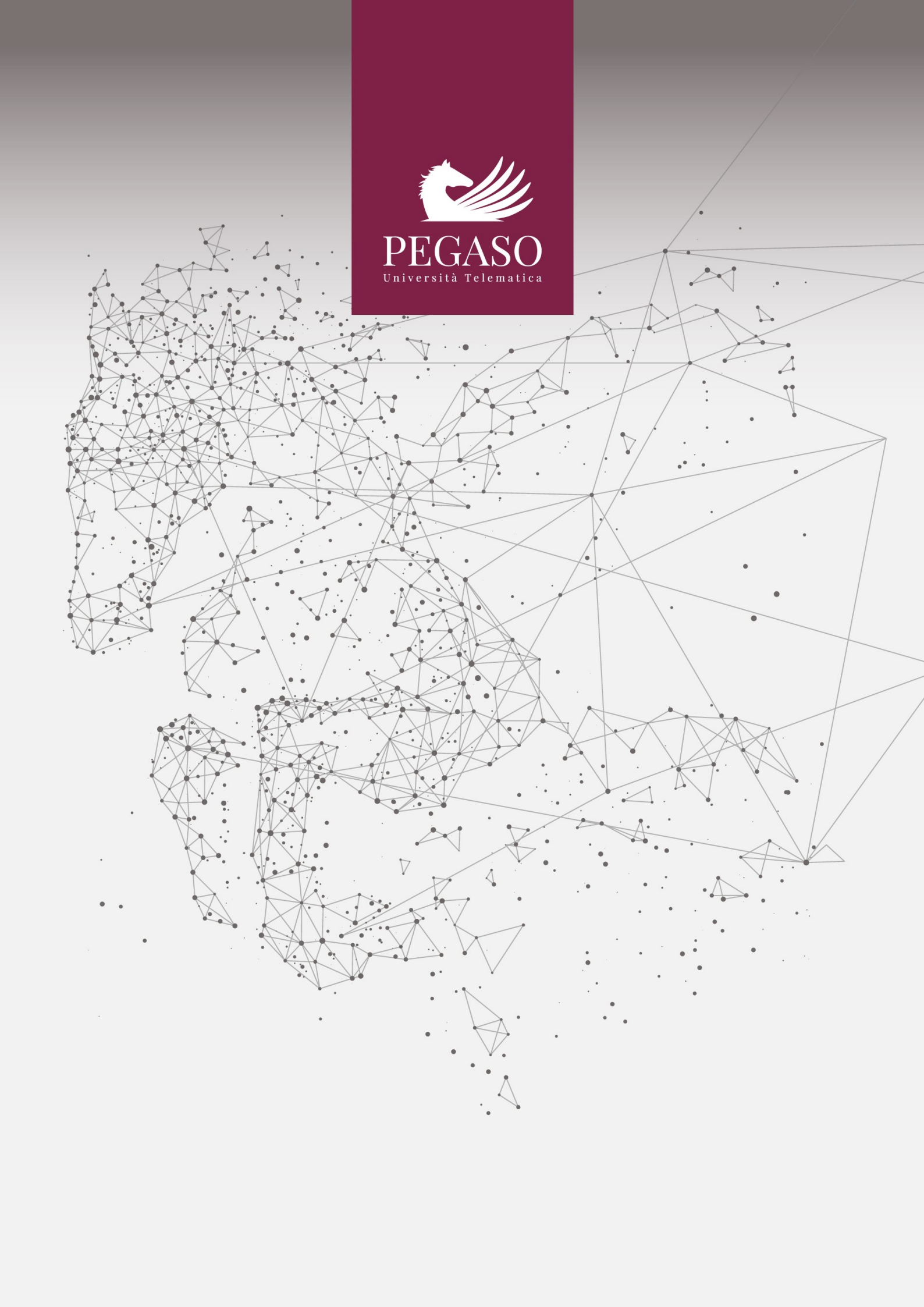
PEGASO
Università Telematica

Indice

1. IL PROBLEMA DELL'ORDINAMENTO 3

2. QUICK SORT 5

3. IMPLEMENTAZIONE 11

BIBLIOGRAFIA 15

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. Quick Sort

Il Quick sort è un algoritmo di ordinamento ricorsivo proposto da Hoare nel 1962.

L'idea che sta alla base del Quick sort è la seguente:

1. gli elementi del vettore vengono spostati in modo che la prima metà (quella a sinistra) contiene elementi tutti più piccoli della seconda metà (quella a destra)
2. il vettore viene decomposto nelle due metà
3. le due metà vengono ordinate separatamente
4. il vettore originario è ordinato mettendo insieme le due metà ordinate separatamente

Il Quick è l'algoritmo di ordinamento che ha, nel caso medio, prestazioni migliori tra quelli basati su confronto.

Di seguito il dettaglio della strategia adottata:

1. il vettore da ordinare viene delimitato dall'indice del primo elemento $\boldsymbol{inf}$ e dall'indice dell'ultimo elemento ***sup***
2. viene scelto un elemento del vettore di indice compreso tra inf e *sup*, chiamato **pivot** (una scelta potrebbe essere ad es. l'elemento che ha come indice $(inf + sup)/2$)
3. vengono utilizzati due indici i, j, che vengono inizializzati a $i = inf$ e $j = sup$
4. l'indice i viene incrementato di 1 fino a quando l'elemento di indice i non è maggiore o uguale al $pivot$
5. l'indice j viene decrementato di 1 fino a quando l'elemento di indice j non è minore o uguale al $pivot$
6. nel caso in cui $i < j$, gli elementi di indici i e j vengono scambiati, e poi i viene incrementato e j decrementato
7. nel caso in cui $i = j$, non si effettua lo scambio, ma i viene incrementato di 1 e j viene decrementato di 1.
8. se $i > j$ allora l'algoritmo termina. In questo modo risulta che:
    a. tutti gli elementi di indice appartenente a $[inf, \dots, j]$ sono minori o uguali del $pivot$
    b. tutti gli elementi di indice appartenente a $[i, \dots, sup]$ sono maggiori o uguali del $pivot$

c. tutti gli elementi (se esistono) di indice appartenente a $[j+1, \dots, i-1]$ sono uguali del $pivot$

9. L'algoritmo viene applicato ricorsivamente al sotto-vettore individuato dagli indici $[inf, j]$, se tale sotto-vettore contiene almeno un elemento, ossia se $inf < j$
10. L'algoritmo viene applicato ricorsivamente al sotto-vettore individuato dagli indici $[i, sup]$, se tale sotto-vettore contiene almeno un elemento, ossia se $i < sup$

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

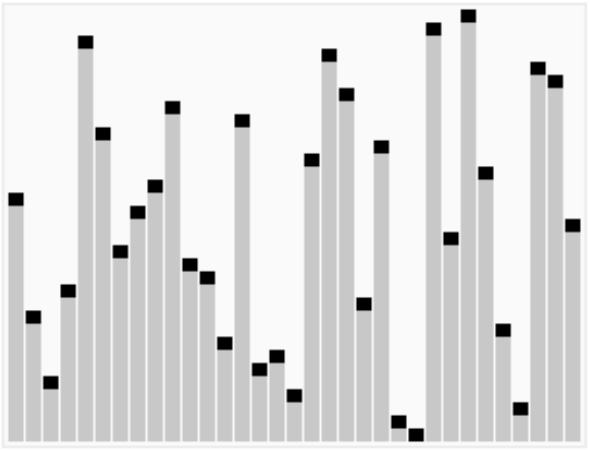

Figura 1 *- Quick Sort Animation*

Analizziamo il suo funzionamento; supponiamo di avere il seguente array di 10 elementi (indice da 0 a 9):

13, 10, 1, 45, 15, 12, 21, 15, 29, 34

Essendo $inf = 0$ e $sup = 9$, il $pivot = \frac{sup - inf}{2} = 4$, corrispondente all'elemento 15 dell'array.

Poniamo $i = inf = 0$ e $j = sup = 9$

L'indice i viene incrementato finchè non si trova un elemento $\geq pivot$; il primo elemento che soddisfa questa condizione è il 45, pertanto $i = 3$

L'indice j viene decrementato finchè non si trova un elemento $\leq pivot$; il primo elemento che soddisfa questa condizione è il 15, pertanto $j = 7$

[1] Sorting quicksort anim - Quicksort - Wikipedia

13, 10, 1, 45, **15**, 12, 21, 15, 29, 34

Gli elementi che si trovano nelle posizioni i e j vengono scambiati:

13, 10, 1, 15, **15**, 12, 21, 45, 29, 34

L'indice i viene incrementato e l'indice j viene decrementato: $i = 4$ e $j = 6$

13, 10, 1, 15, **15**, 12, 21, 45, 29, 34

L'indice i viene ora arrestato perché corrisponde al $pivot$, mentre j può essere decrementato e ci si ferma quando si arriva a 12 (primo elemento $\leq pivot$); pertanto $j = 5$.

13, 10, 1, 15, **15**, 12, 21, 45, 29, 34

Gli elementi che si trovano nelle posizioni i e j vengono scambiati:

13, 10, 1, 15, 12, **15**, 21, 45, 29, 34

L'indice i viene incrementato e l'indice j viene decrementato: $i = 5$ e $j = 4$

13, 10, 1, 15, 12, **15**, 21, 45, 29, 34

A questo punto ci si deve fermare perché $i > j$

Si evince che alla fine della prima iterazione:

- tutti gli elementi di indice appartenente a $[inf, j]$ sono $\leq pivot$
- tutti gli elementi di indice appartenente a $[i, sup]$ sono $\geq pivot$
- non ci sono elementi di indice appartenente a $[j + 1, \dots, i - 1]$ (se ci fossero stati, sarebbero stati uguali al $pivot$).

Come si vede l'esempio considerato rappresenta il caso migliore perché il vettore originario è stato decomposto in due vettori che hanno entrambi dimensione uguale e pari a metà della dimensione iniziale.

L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici $[inf, j]$ e $[i, sup]$.

Vi sono altri casi "interessanti" da analizzare, ad es. quello in cui il pivot coincide con l'elemento più grande del vettore; supponiamo ad es. di avere il seguente vettore:

13, 20, 1, 15, 34, 28, 21, 14, 29, 3

Poniamo $i = inf = 0$ e $j = sup = 9$ → $pivot = \frac{sup - inf}{2} = 4$ corrispondente all'elemento 34

Procedendo con la prima iterazione si ottiene:

13, 20, 1, 15, **34**, 28, 21, 14, 29, 3	- incremento i (è minore del $pivot$) - j è fermo (è minore del $pivot$)
13, 20, 1, 15, **34**, 28, 21, 14, 29, 3	- incremento i (è minore del $pivot$)
13, 20, 1, 15, **34**, 28, 21, 14, 29, 3	- incremento i (è minore del $pivot$)
13, 20, 1, 15, **34**, 28, 21, 14, 29, 3	- incremento i (è minore del $pivot$)
13, 20, 1, 15, **34**, 28, 21, 14, 29, 3	- incremento i (è minore del $pivot$): mi devo fermare
13, 20, 1, 15, 3, 28, 21, 14, 29, **34**	- scambio i valori corrispondenti ad i e j
13, 20, 1, 15, 3, 28, 21, 14, 29, **34**	Step obbligatorio dopo lo scambio - incremento i - decremento j
13, 20, 1, 15, 3, 28, 21, 14, 29, **34**	- incremento i (è minore del $pivot$) - j è fermo (è minore del $pivot$)
13, 20, 1, 15, 3, 28, 21, 14, 29, **34**	- incremento i (è minore del $pivot$) - j è fermo (è minore del $pivot$)
13, 20, 1, 15, 3, 28, 21, 14, 29, **34**	- incremento i (è minore del $pivot$) - j è fermo (è minore del $pivot$) - attenzione: i e j coincidono
13, 20, 1, 15, 3, 28, 21, 14, 29, **34**	- incremento i (è minore del $pivot$): mi devo fermare perché ho raggiunto il $pivot$

Alla fine della prima iterazione si ha dunque che:

- tutti gli elementi di indice appartenente a $[inf, j]$ sono $\leq pivot$
- tutti gli elementi di indice appartenente a $[i, sup]$ sono $\geq pivot$
- non ci sono elementi di indice appartenente a $[j+1, \dots, i-1]$.

L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici $[inf, j]$ e $[i, sup]$. Questo esempio è un caso peggiore, perché il vettore originario è stato decomposto in due vettori di cui il primo ha dimensione quasi uguale a quella originaria.

Consideriamo infine il seguente esempio:

13, 10, 1, 45, 15, 28, 21, 15, 29, 34

Poniamo $i = inf = 0$ e $j = sup = 9$ → $pivot = \frac{sup - inf}{2} = 4$ corrispondente all'elemento 15

Procedendo con la prima iterazione si ottiene:

13, 10, 1, 45, **15**, 28, 21, 15, 29, 34	- incremento i (è minore del $pivot$) - decremento j (è maggiore del $pivot$)
13, 10, 1, 45, **15**, 28, 21, 15, 29, 34	- incremento i (è minore del $pivot$) - decremento j (è maggiore del $pivot$): mi devo fermare
13, 10, 1, 45, **15**, 28, 21, 15, 29, 34	- incremento i (è minore del $pivot$): mi devo fermare - j è fermo (è uguale al $pivot$)
13, 10, 1, 45, **15**, 28, 21, 15, 29, 34	- scambio i valori corrispondenti ad i e j
13, 10, 1, 15, **15**, 28, 21, 45, 29, 34	Step obbligatorio dopo lo scambio - incremento i - decremento j
13, 10, 1, 15, **15**, 28, 21, 45, 29, 34	- i è fermo (è uguale al $pivot$) - decremento j (è maggiore del $pivot$)
13, 10, 1, 15, **15**, 28, 21, 45, 29, 34	- i è fermo (è uguale al $pivot$) - decremento j (è maggiore del $pivot$): mi devo fermare - attenzione: i e j coincidono
13, 10, 1, 15, **15**, 28, 21, 45, 29, 34	- dovrei scambiare gli indici ma è inutile visto che coincidono; procedo con lo step obbligatorio

	- incremento i - decremento j
13, 10, 1, 15, **15**, 28, 21, 45, 29, 34	- mi devo fermare perché $i > j$

Alla fine della prima iterazione si ha dunque che:

- tutti gli elementi di indice appartenente a $[inf, j]$ sono $\leq pivot$
- tutti gli elementi di indice appartenente a $[i, sup]$ sono $\geq pivot$
- vi è un solo elemento di indice appartenente a $[j+1, \dots, i-1]$, uguale al $pivot$

L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici $[inf, j]$ e $[i, sup]$

3. Implementazione

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

```
QUICKSORT(A, p, r)
IF p < r THEN
      q <- PARTITION(A, p, r)
      QUICKSORT(A, p, q – 1)
            QUICKSORT(A, q + 1, r)


PARTITION(A, p, r)
x <- A[r]
i <- p – 1
FOR j <- p TO r – 1
            IF A[j] <= x THEN
            i <- i + 1
            SCAMBIA(A[i],A[j])
NEXT j
SCAMBIA(A[i + 1],A[r])
RETURN i + 1
```

ATTENZIONE: questo pseudocode è puramente a carattere dimostrativo; all'interno di Replit CIE Pseudocode non è funzionante.

Di seguito l'implementazione in C++:

```
#include <iostream>

using namespace std;

void quicksort(int a[],int sx,int dx) {
  int i=sx;
  int j=dx;
  int pivot=pivot = a[(sx + dx) / 2];

  while (i<=j) {
    while (a[i] < pivot)
      i++;
    while (a[j] > pivot)
      j--;
    if (i <= j) {
      if (i < j) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
      i++;
      j--;
    }
  }

  if (sx < j)
    quicksort(a,sx,j);
  if (i < dx)
    quicksort(a,i,dx);
}

int main() {
  int a[8]={41, 37, 10, 74, 98, 22, 83, 66};
  int n=8;
  quicksort(a,0,n-1);
  for (int i=0;i<n;i++)
    cout<<a[i]<<" ";
}
```

L'implementazione in Python è invece la seguente:

```
def quicksort(a,sx,dx):
  i=sx
  j=dx
  pivot = a[(sx + dx) // 2]

  while i<=j:
    while a[i] < pivot:
      i+=1
    while a[j] > pivot:
      j-=1
    if i <= j:
      if i < j:
        tmp = a[i]
```

```
            a[i] = a[j]
            a[j] = tmp
          i+=1;
          j-=1;

      if sx < j:
        quicksort(a,sx,j)
      if i < dx:
        quicksort(a,i,dx)


a=[41, 37, 10, 74, 98, 22, 83, 66]
n=8
quicksort(a,0,n-1)
print(a)
```

Analizzando l'implementazione ragioniamo in termini di:

- **Stabilità**: l'algoritmo non è stabile; l'implementazione mostra che non è detto che venga preservato l'ordine iniziale (ovvero nella sequenza di input) tra elementi aventi la stessa chiave
- **In place**: l'algoritmo opera sul posto, infatti non sono usate strutture dati ausiliare di dimensione proporzionali alla grandezza dell'input

Relativamente alla complessità, ragioniamo nella seguente maniera partendo dal caso medio:

- La partizione di una sequenza in due parti può essere effettuata in tempo lineare (rispetto alla lunghezza n della sequenza).
- Per dividere in due parti l'intera sequenza occorrono quindi n passi; mediamente, la sequenza risulterà divisa in due parti di dimensioni circa n/2 ciascuna.
- Per dividere in due ciascuna di esse occorreranno quindi circa n/2 passi; quindi, per dividere entrambe le parti occorreranno 2(n/2) = n passi.
- Per dividere in due ognuna delle quattro porzioni così ottenute occorreranno mediamente 4(n/4) = n passi.
- [...]
- Ad ogni "livello di suddivisione" si fanno in media n passi. Quanti sono i "livelli di suddivisione"?
- Assumiamo per semplicità che la lunghezza n della sequenza sia una potenza di 2: $n = 2^k$. Poiché ad ogni livello si divide all'incirca per 2, al k-esimo livello si ottengono porzioni di lunghezza 1, su cui non vi è da fare nulla. I livelli sono quindi k ed il numero totale di passi è quindi nk. Ma per definizione è $k = \log_2 n$.
- Quindi il numero totale di passi, in funzione di n, è $\mathbf{n \cdot \log_2 n}$.

Se ci riferiamo invece al caso peggiore:

- La partizione di una sequenza in due parti può essere effettuata in tempo lineare (rispetto alla lunghezza n della sequenza). Per dividere in due parti l'intera sequenza occorrono quindi n passi.
- Nel caso peggiore, la sequenza risulterà divisa in due parti di cui una di dimensione 0 e l'altra di dimensione n-1. Sulla parte di dimensione 0 non si fa nulla, ma per dividere in due l'altra parte occorrono evidentemente n-1 passi.
- Nel caso peggiore tale parte risulterà a sua volta divisa in due parti di cui una di dimensione 0 e l'altra di dimensione n-2. Per dividere in due parti quest'ultima, occorreranno n-2 passi, e così via.
- Nel caso peggiore si ha quindi un numero di passi dato da:

  $$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 = n(n+1)/2 = \Theta(n2)$$

  Nel caso peggiore il Quick sort è quindi quadratico come l'Insertion sort e il Selection sort.

Nel caso migliore, ad ogni partizione la porzione di array risulta divisa in due parti di lunghezze esattamente uguali, cioè il pivot risulta essere sempre l'elemento mediano della porzione. Il ragionamento fatto "approssimativamente" nel caso medio diventa allora rigoroso, e quindi la complessità è $\mathrm{n} \cdot \log_2 n$

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