

# Appunti di Algoritmi e Strutture dati

A.A. 2022/2023

## 1 Fondamenti

### 1.1 Algoritmi e loro rappresentazione

Un **problema** é un quesito che richiede la determinazione o la costruzione di uno o piú enti matematici che soddisfino le condizioni specificate nell'enunciato. Con *problema* si denota l'enunciato generale, con *istanza* si denota un caso particolare del problema, ovvero un insieme specifico di dati per il quale si vuole ottenere una soluzione.

Un **algoritmo** é una sequenza di azioni non ambigue che risolve un problema utilizzando un insieme di azioni elementari, eseguibili da un opportuno esecutore. Con **programma** si denota la rappresentazione di un algoritmo utilizzando un linguaggio (con opportune "traduzioni") direttamente comprensibile da un elaboratore.

L'algoritmo é specificato da un insieme ben definito di dati in input e in output, deve essere eseguibile in un numero finito di passi, fornire il risultato corretto per ogni possibile input ed essere abbastanza generale da essere applicabile a un'intera classe di problemi.

Lo **pseudocodice** é un linguaggio astratto ed informale, inteso per uso umano, utilizzato per descrivere un algoritmo. Utilizza la struttura di un linguaggio di programmazione normale ma non é vincolato nella sintassi ed é integrabile con linguaggio naturale o notazioni matematiche compatte.

### 1.2 Confronto di algoritmi

Nel confrontare gli algoritmi, ci si astrae da tutti gli aspetti dipendenti dall'implementazione.

La risorsa principale su cui ci si basa é il **tempo di esecuzione**, quantificato non in secondo ma in accessi alla RAM.

L'algoritmo viene visto come una funzione  $f(n)$  (dove  $n$  é l'input fornito) di cui si considera solamente il termine dominante (solitamente si trascura anche il coefficiente). Per confrontare due algoritmi, si confrontano le due rispettive funzioni per  $n \rightarrow \infty$ .

### 1.2.1 Notazione O-grande

La crescita asintotica delle funzioni viene descritta con diverse notazioni, la cui maggiormente usata è la notazione  $O$  grande (limite asintotico **superiore**), dove  $O$  sta per ordine di grandezza. L'**ordine di grandezza** è la più piccola funzione maggiorante (siccome ne esistono infinite).

Definizione: siano  $f : \mathbb{R} \rightarrow \mathbb{R}, g : \mathbb{R} \rightarrow \mathbb{R}, f(x) = O(g(x))$  se  $\exists c, x_0 : \forall x > x_0$  si ha  $f(x) \leq c \cdot g(x)$ .

Esempi

- Se si vuole dimostrare  $f(x) = x^2 + 2x + 1 \in O(x^2) \rightarrow x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2 \iff 3x^2 - 2x - 1 \geq 0 \iff x \geq 1 \left( \frac{2 \pm \sqrt{16}}{6} \rightarrow \frac{2+4}{6} = 1 \right)$ . Ho dimostrato che la definizione vale per  $n_0 = 1, c = 4$ .
- Per dimostrare invece che  $f(n) = \frac{n(n+1)}{2}$  non è  $O(n) \rightarrow \frac{1}{2}(n^2 + n) \leq Cn \iff n^2 + n \leq 2Cn \iff n^2 \leq n(2C - 1) \iff n \leq 2C - 1$ . Essendo  $c$  costante,  $\forall c \in \mathbb{R}, \exists n \in \mathbb{R} : n > 2c - 1$ , quindi  $f(n)$  non è  $O(n)$ .

Alcuni ordini di grandezza ben noti in ordine crescente: **costante** ( $O(1)$ ), **logaritmico** ( $O(\log n)$ ), **lineare** ( $O(n)$ ), **log lineare** ( $O(n \log n)$ ), **polinomiale** ( $O(n^k)$  con  $k$  costante), **esponenziale** ( $O(c^n)$  con  $c$  costante).

### 1.2.2 Altre notazioni

Altre notazioni utilizzate nell'analisi asintotica sono  $\Omega$  (Omega grande, limite asintotico **inferiore**) e  $\Theta$  (Theta grande).

Definizione di  $\Omega$ : siano  $f : \mathbb{R} \rightarrow \mathbb{R}, g : \mathbb{R} \rightarrow \mathbb{R}, f(n) = \Omega(g(n))$  se  $\exists c, n_0 : f(n) \geq c \cdot g(n) \forall n \geq n_0$ .

È utilizzata nei limiti inferiori di complessità e per l'analisi del tempo di esecuzione nel caso ottimo.

Definizione di  $\Theta$ : siano  $f : \mathbb{R} \rightarrow \mathbb{R}, g : \mathbb{R} \rightarrow \mathbb{R}, f(n) = \Theta(g(n))$  se  $\exists c_1, c_2, n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0$ . Da notare che  $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge \Omega(g(n))$ .

$O(f(n))$  è spesso usato erroneamente al posto di  $\Theta(f(n))$ .

Esistono degli analoghi di  $O$  e  $\Omega$  che sono  $o$  ( $o$  piccolo) e  $\omega$  (omega piccolo), dove al posto della disuguaglianza ( $\leq$ ) si ha la disuguaglianza stretta ( $<$ ).

Equivalentemente:

$$\text{se } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow c, \text{ allora } f(n) = \Theta(g(n))$$

$$\text{se } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow 0, \text{ allora } f(n) = o(g(n))$$

se  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow \infty$ , allora  $f(n) = \omega(g(n))$

### 1.2.3 Caso ottimo, medio, pessimo

Nell'analisi degli algoritmi si possono analizzare 3 casi: caso ottimo (migliore possibile), medio e pessimo (peggiore possibile). Principalmente si analizza il caso pessimo e il caso medio, anche se l'analisi di quest'ultimo é spesso molto piú complicata delle altre due perché richiede un'analisi statistica.

Esempio: Ricerca sequenziale.

Problema: dato un array  $v$  e un valore  $x$ , restituire l'indice della prima occorrenza di  $x$  in  $v$ , o -1 se  $x$  non é presente.

1. Caso ottimo: l'elemento é all'inizio della lista ( $O(1)$ )
2. Caso pessimo: l'elemento é in fondo o non é presente, quindi si itera su tutti gli elementi ( $O(n)$ )
3. Caso medio: per ipotesi, l'elemento é sempre presente e la probabilità  $p_i$  che l'elemento si trovi alla posizione  $i$  sia la stessa per ogni  $i$ , quindi  $p_i = \frac{1}{n}$ . Se l'elemento é nella prima posizione, devo controllare una sola volta, nella seconda due volte, nella terza tre volte e cosí via fino a  $n$ : questo lo posso esprimere con la somma dei primi  $n$  numeri, ovvero con la serie geometrica  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ . Moltiplico il tutto per la probabilità  $p_i$ , che é la stessa per ogni elemento e ottengo  $\frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{1}{2}(n+1)$ , ovvero  $O(n)$ .

## 2 Analisi asintotica

Un algoritmo  $A$  ha **costo di esecuzione**  $O(f(n))$  rispetto ad una risorsa di calcolo, su istanze di ingresso di dimensione  $n$  se la quantità  $r(n)$  di risorsa sufficiente per eseguire  $A$  su una **qualunque istanza di dimensione  $n$**  verifica la relazione  $r(n) = O(f(n))$ .

Dato lo pseudocodice, é possibile ottenere il costo di esecuzione analizzando la sua struttura. Ad esempio, data una serie di istruzioni,  $t(\text{istruzione } 1) + \dots + t(\text{istruzione } n)$ , negli if-else  $\max(t(\text{sequenza } 1), t(\text{sequenza } 2))$ , nei for e nei while bisogna vedere se il ciclo viene eseguito un numero di volte funzione di  $n$  o meno.

## 2.1 Algoritmi ricorsivi

Negli algoritmi ricorsivi, il tempo di esecuzione dell'algoritmo può essere descritto come la somma dei tempi di esecuzione di  $f(n_1), \dots, f(n_k)$  con  $n_i < n$ , ovvero richiede di calcolare tutti i termini precedenti, fino al caso base.

### 2.1.1 Ricerca binaria

La ricerca binaria ha classe di complessità  $O(\log n)$ .

Il tempo di esecuzione può essere espresso come:

$$\begin{cases} c_1 & \text{se } n = 1 \text{ (caso base)} \\ T(\lfloor \frac{n}{2} \rfloor) + c_2 & \text{se } n > 1 \end{cases} \quad (1)$$

È dimostrabile con diversi metodi, tra cui:

1. Metodo iterativo. Partendo da  $T(n)$ , devo arrivare al caso base  $T(1) = T(\frac{n}{n})$  dimezzando  $n$  ad ogni passo.  $T(n) = T(\frac{n}{2}) + c_2 = T(\frac{n}{4}) + 2c_2 = T(\frac{n}{8}) + 3c_2 = \dots$   
Mi fermo quando  $n = 2^k$ , quindi dopo  $k = \log_2 n$  chiamate ricorsive, per un totale di  $ck$  chiamate ( $c = c_1 + kc_2$ ). Quindi,

$$T(n) = c \cdot \log_2 n = O(\log n)$$

2. Dimostrazione per induzione. Prima di dimostrare, si "indovina" (grazie anche all'esperienza) la soluzione:  $T(n) \leq c \cdot \log_2 n$  ( $T(n) = O(\log n)$ ).  
Assumendo  $T(n') \leq c \cdot \log_2 n' \forall n > n'$  (**ipotesi induttiva**), voglio dimostrare che  $T(n) \leq c \cdot \log_2 n$ .

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \text{ (costante qualsiasi)} \leq c \cdot \log_2(\frac{n}{2}) + 1 \text{ (ipotesi induttiva } (\frac{n}{2} \text{ é } n')) \\ &= c \cdot \log_2 n - c \cdot \log_2 2 \text{ (proprietá del logaritmo)} + 1 \\ &= c \cdot \log_2 n - c \cdot 1 + 1. \end{aligned}$$

Se  $c \geq 1$ , allora

$$c \cdot \log_2 n + c - 1 \leq c \cdot \log_2 n \rightarrow T(n) \leq c \cdot \log_2 n$$

## 3 Grafi

Un grafo  $G = (V, E)$  é composto da un insieme di **vertici**  $V$  e un insieme di **archi**  $E \subset V \times V$  (non  $\subseteq$  perché non si ha un arco tra un vertice e se stesso) che connettono i vertici.

### 3.1 Terminologia

#### 3.1.1 Grafo orientato/non orientato

In un grafo **non orientato**, due vertici  $u, v$  sono **adiacenti** se  $\{u, v\} \in E$ .

In un grafo **orientato**, se  $(u, v) \in E \rightarrow v$  adiacente a  $u$  e  $(u, v)$  é **incidente** in  $v$ .

### 3.1.2 Grado, cammino e ciclo

Il **grado** di un vertice é il numero di vertici adiacenti ad esso.

Il **cammino** é una sequenza di vertici  $v_1, \dots, v_n$  tale che per ogni coppia di vertici consecutivi  $v_i, v_{i+1}$ ,  $v_{i+1}$  é adiacente a  $v_i$ . Un cammino é detto **elementare** se non ci sono vertici ripetuti. Un **ciclo** é un cammino elementare in cui il primo vertice coincide con l'ultimo (torna all'inizio).

### 3.1.3 Grafo connesso, sottografo

É detto **grafo connesso** qualsiasi coppia di vertici unita da almeno un cammino. Un **sottografo** é un sottoinsieme di vertici e archi di un grafo dato. Una **componente connessa** é un sottografo connesso massimale.

### 3.1.4 Grafo completo

Un grafo é detto **completo** se ogni coppia di vertici é connessa da un arco. In questo caso, il numero di archi  $m = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$  (ogni nodo é connesso a  $n - 1$  nodi ( $n(n - 1)$ ); infine divido per 2 siccome altrimenti sto considerando gli stessi archi 2 volte).

### 3.1.5 Grafo trasposto

Un grafo é  $G^T = (V, E^T)$  é il **trasposto** di  $G = (V, E)$  se  $E^T = \{(u, v) : (v, u) \in E\}$  (inverte il verso di percorrenza degli archi di  $G$ ).

### 3.1.6 Grafo pesato

In un grafo pesato, agli archi viene associato un valore, detto **peso** (es. costo necessario per percorrerlo). Il peso é determinato da  $p : V \times V \rightarrow \mathbb{R}$ . Se non specificato, si assume il costo infinito.

## 3.2 Alberi

Un albero é un **grafo aciclico** (non é possibile compiere un ciclo al suo interno) con un numero di nodi uguale al numero di archi + 1. Può esistere un nodo particolare chiamato **radice**, a partire dal quale é possibile raggiungere qualsiasi altro nodo.

A partire dalla radice, viene definita una *relazione di ordinamento* fra i nodi, basata sulla distanza dalla radice, in cui ogni nodo può avere un padre e dei figli. Inoltre, é detto **foglia** un nodo senza successori, mentre se presenta successori é detto **nodo interno**.

Si definisce **profondità** di un nodo la lunghezza del cammino dalla radice al nodo, **altezza** di un nodo la massima lunghezza di un cammino dal nodo a una foglia discendente dal nodo stesso, altezza dell'albero l'altezza della radice (massima lunghezza del cammino che collega la radice a qualsiasi nodo).

Un **livello**/strato consiste di nodi con la stessa profondità. Il numero di livelli é dato dall'altezza + 1 (radice).

La rappresentazione grafica di un albero é detta **arborescenza**.

## 4 Ordinamento

Input: sequenza di  $n$  numeri  $a_1, a_2, \dots, a_n$

Output: i numeri ricevuti in input in ordine crescente:  $a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}$ ,  
dove  $\pi$  é una permutazione degli indici.

Esempio:

$a = [7, 32, 88, 21, 92, -4]$

$\pi = [6, 1, 4, 2, 3, 5]$  (1-based)

$a[\pi[]] = [-4, 7, 21, 32, 88, 92]$

Piú in generale, dato un array di  $n$  elementi, ogni elemento é composto da una **chiave** (confrontabili tra loro) e un **valore**. Si vuole permutare l'array delle chiavi in modo che appaiano nell'ordine desiderato.

### 4.1 Definizioni

- ordinamento **in place**: l'algoritmo permuta gli elementi dell'array senza utilizzare un array di appoggio
- ordinamento **stabile**: l'algoritmo preserva l'ordine in cui gli elementi con la stessa chiave appaiono nell'array originale

### 4.2 Teorema del lower-bound per algoritmi comparison-sort

Qualsiasi algoritmo comparison-sort (basato su confronti di coppie di elementi) effettua, nel caso pessimo,  $\Omega(n \cdot \log n)$  (non può fare meglio di così) confronti per ordinare una sequenza di  $n$  numeri.

#### 4.2.1 Dimostrazione

Alla base c'è l'idea che, dato un array di  $n$  numeri, esistono  $n!$  permutazioni, quindi  $n!$  possibili ordinamenti, e ad ogni confronto si dimezzano le possibilità rimaste.

Il caso pessimo é dato dall'altezza dell'albero decisionale associato a questo algoritmo. Siccome l'albero decisionale in questo caso é un **albero binario** (albero in cui ogni nodo ha al massimo 2 figli), date  $n!$  foglie, ha un'altezza  $\Omega(n \log n)$ , quindi nel caso pessimo vengono eseguiti  $\Omega(n \log n)$  confronti.

## Dimostrazioni

1. Siccome un albero binario alto  $h$  ha massimo  $2^h$  foglie, bisogna ottenere  $2^h \geq n!$ . Usando la formula di Stirling:  $n! > (\frac{n}{e})^n \implies$

$$h \geq \log\left(\frac{n}{e}\right)^n = n \log\left(\frac{n}{e}\right) = n \log n - n \log e = \Omega(n \log n)$$

Si noti che il passaggio da  $\log_2$  a  $\log_e$  é semplicemente dato dalla moltiplicazione per una costante

2. Bisogna ottenere  $2^h \geq n! \implies h \geq \log(n!)$ . Siccome ho  $\geq$ , se sostituisco a  $\log(n!)$  qualcosa di piú piccolo, la disuguaglianza resta sempre vera, quindi:

$$\log(n!) = \log(1 \cdot 2 \cdots \frac{n}{2} \cdot (\frac{n}{2} + 1) \cdots n) \geq \log(1 \cdot 1 \cdots \frac{n}{2} \cdots \frac{n}{2}) = \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log\left(\frac{n}{2}\right) = \Omega(n \log n)$$

(sostituisco tutti i termini da 1 a  $\frac{n}{2}$  con 1 e da  $\frac{n}{2}$  a  $n$  con  $\frac{n}{2}$ ).

## 4.3 Insertion sort

In questo algoritmo, si parte disponendo gli elementi ordinati nella parte sinistra dell'array, si prende un elemento dalla parte non ordinata e lo si inserisce in maniera ordinata.

### 4.3.1 Pseudocodice

```
for  $j = 2$  to  $\text{size}(A)$  do
   $key \leftarrow A[j]$ 
   $i \leftarrow j - 1$ 
  while  $i > 0$  and  $A[i] > key$  do    ▷ fino a quando il numero é piú grande
     $A[i + 1] \leftarrow A[i]$           ▷ shift a destra di 1
     $i \leftarrow i - 1$ 
  end while
   $A[i + 1] \leftarrow key$ 
end for
```

## 4.4 Merge sort

Il merge sort é un algoritmo di ordinamento che si basa sul metodo **divide et impera**, che consiste nel dividere un problema in sottoproblemi, risolverli e combinare le soluzioni dei sottoproblemi per ottenere la soluzione del problema complesso.

Dato un vettore  $S$ , l'algoritmo consiste nel:

1. dividere  $S$  nei vettori  $S_1$  e  $S_2$ , ognuno con la metà degli elementi di  $S$  fino a quando  $S$  ha almeno 2 elementi
2. ordinare gli elementi in  $S_1$  e  $S_2$  (merge)
3. mettere insieme gli elementi di  $S_1$  e  $S_2$

Si parte ordinando 2 vettori da 1 elemento, poi 2 vettori da 2 elementi e così via fino ad ordinare i due sottovettori iniziali  $S_1$  e  $S_2$ .

Il merge é la parte che si occupa di ordinare i sottovettori. Ad ogni ordinamento di 2 vettori di  $n$  e  $m$  elementi, alloca un vettore di  $n + m$  elementi (non in-place). L'ordinamento viene fatto inserendo ogni volta nel vettore allocato il minore degli **affioranti** (elementi piú a sinistra di ogni vettore, ovvero i piú piccoli).

#### 4.4.1 Pseudocodice

Versione ricorsiva non in-place:

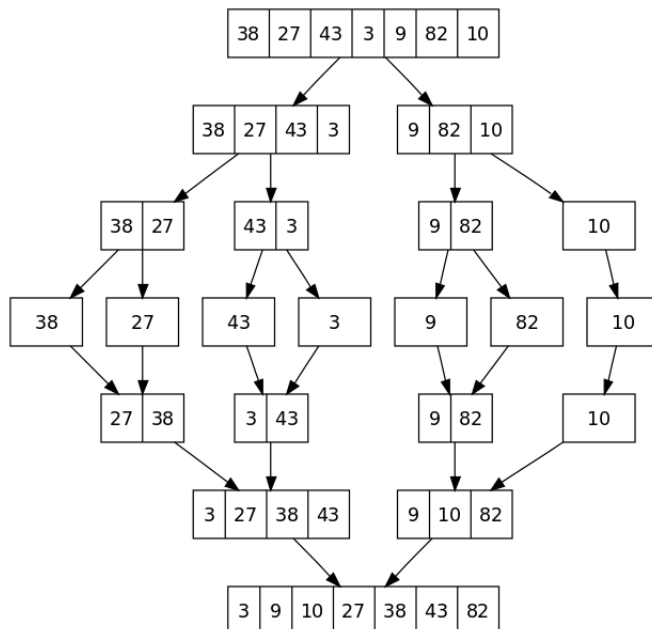
```

function MERGESORT(A,p,r)
  if  $p < r$  then
     $q \leftarrow (p + r)/2$ 
    MERGESORT(A,p,q)           ▷ ordino la metà di sinistra
    MERGESORT(A,q+1,r)         ▷ ordino la metà di destra
    MERGE(A,p,q,r)
  end if
end function

```



#### 4.4.2 Esempio



#### 4.4.3 Complessità

La complessità del merge sort non dipende dalla configurazione iniziale. L'algoritmo fa lo stesso procedimento ogni volta, quindi non esiste un caso pessimo, medio o ottimo:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

#### 4.5 Quick sort

Il quick sort é un altro algoritmo di ordinamento sempre basato sulla tecnica divide et impera.

Consiste nel dividere l'array in 2 parti non vuote basandosi su un elemento detto **pivot** (selezionato arbitrariamente (es. il primo) o casualmente (minimizza le possibilità di avere il caso pessimo)), spostando tutti gli elementi più piccoli del pivot a sinistra e quelli più grandi a destra. Il procedimento viene iterato fino a quando tutti i gruppi non sono ordinati.

Il lavoro principale é svolto dalla funzione che si occupa di partizionare:

- partendo dall'inizio del vettore, ci si sposta verso il fondo (destra) fino a quando non si trova un elemento più grande ( $\geq o >$  arbitrariamente, a patto che dopo si usi la sua negazione (es.  $\geq$  e  $<$ )) del pivot, utilizzando un indice  $i$

- partendo dal fondo del vettore, ci si sposta verso l'inizio (sinistra) fino a quando non si trova un elemento piú piccolo del pivot, utilizzando un indice  $j$
- si scambiano gli elementi e si continua il procedimento dalla stessa posizione fino a quando non si ottiene  $i > j$  (gli elementi sono posizionati correttamente)

Una volta ordinati gli elementi di un sottogruppo, la funzione che partiziona ritorna un indice  $q$  tale che gli elementi fino a posizione  $q$  sono piú piccoli del pivot, mentre quelli da  $q + 1$  in poi sono piú grandi.

#### 4.5.1 Pseudocodice

Versione ricorsiva:

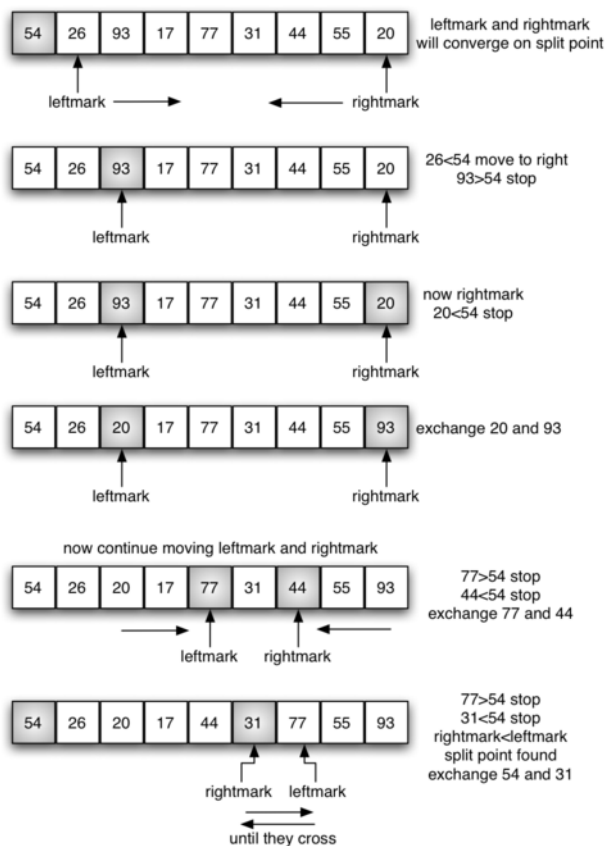
```

function QUICKSORT( $A, p, r$ )
  if  $p < r$  then
     $q \leftarrow partition(A, p, q)$            ▷ sceglie il pivot e sposta gli elementi
    QUICKSORT( $A, p, q$ )                   ▷ ordino la metà di sinistra
    QUICKSORT( $A, q+1, r$ )                 ▷ ordino la metà di destra
  end if
end function

```

Dallo pseudocodice si può notare una differenza rispetto al merge sort: il merge sort prima divide i vettori in sottogruppi, poi li riordina, mentre il quick sort prima sposta gli elementi (a destra o sinistra del pivot), poi itera ricorsivamente per ordinarli.

### 4.5.2 Esempio



### 4.5.3 Complessità

Nel caso ottimo, il pivot ha esattamente metà degli elementi maggiori e metà minori. L'algoritmo é analogo al merge sort, quindi

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

Nel caso pessimo, l'array é (paradossalmente) già ordinato. Ogni elemento ha tutti gli elementi a destra maggiori: ad ogni ciclo si decrementa  $j$  (ci si sposta dal fondo verso l'inizio) fino a raggiungere  $i$ , siccome il primo elemento di ogni ciclo é già nella posizione corretta.

$$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$$

Nel caso medio, considerando il caso in cui il pivot abbia  $\frac{1}{10}$  degli elementi più piccoli e  $\frac{9}{10}$  più grandi, si può costruire un albero di decisione (qui considera il caso di  $\frac{1}{4}$  e  $\frac{3}{4}$  ma sono analoghi):

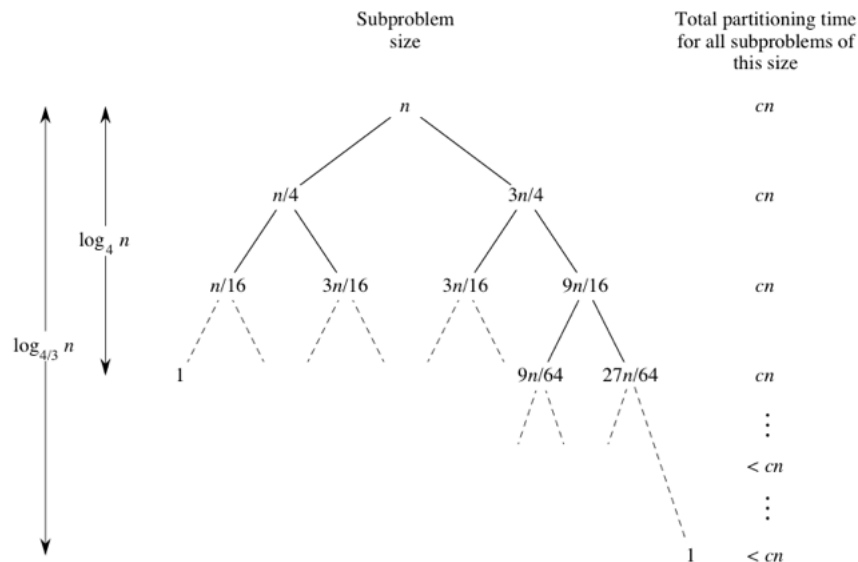


Figure 1: Fonte: Khan Academy

Fino a quando ci sono tutti i rami, si paga  $\log_{10} n$  volte  $n$  ( $\frac{1}{10}n + \frac{9}{10}n, \frac{1}{100}n + \frac{9}{100}n + \frac{81}{100}n, \dots$ ); una volta che una parte dei rami termina, si paga  $\log_{10} n$  volte una quantità  $< n$ , quindi la complessità totale é  $\Theta(n \log n)$ .