Università di Trieste

Laurea in ingegneria elettronica e informatica

Enrico Piccin - Corso di Algoritmi e Strutture dati - Prof. Andrea Sgarro

Anno Accademico 2021/2022 - 2 Marzo 2022

Indice

2 Marzo 2022

1 Introduzione

Algoritmo è una parola molto antica, non connessa all'utilizzo e all'invenzione del calcolatore.
Alla base della teoria della computazione si pone il **Liber abaci** (tradotto "Libro della computazione"), scritto nel 1200 da Leonardo Bonacci, in contatto con la popolazione araba, in quanto mercante; egli è venuto a conoscenza della **numerazione araba**, introducendola in Occidente e spiegandola dettagliatamente all'interno del **Liber abaci**.
La numerazione araba è uno straordinario passo in avanti nella scienza, in quanto con essa viene introdotto il concetto di **notazione posizionale**, così come l'importanza del numero 0: i numeri non servono solamente per contare, come si pensava in precedenza, e per questo rinnegando il numero 0.
A Firenze, sempre negli stessi anni, ci fu una **protesta sindacale** contro l'innovazione tecnologica, contro questa nuova scoperta, facendo pressione affinché il governo abolisse il nuovo sistema di numerazione, in quanto avrebbe fatto perdere il posto di lavoro a tutti coloro che prima eseguivano difficili calcoli con la numerazione romana: tuttavia, tale proteste, com'é noto, possono rallentare il progresso, ma mai arrestarlo.
Leonardo Bonacci, nei suoi viaggi in Oriente, venne a conoscenza del **Liber abaci** di Al-Gorasmy, proveniente dalla Coresmia, ma che parlava persiano, da cui poi sarebbe stato tratto il nome **Algoritmo**, che letteralmente significa **procedimento di calcolo**.

ALGORITMO

Algoritmo significa letteralmente **procedimento di calcolo**. Tuttavia, bisogna chiarire che un algoritmo è un procedimento di calcolo non necessariamente numerico, ma molto più generale, che va ben al di là dei numeri.
Un altro importante elemento che contraddistingue l'algoritmo è la **meccanicità**, ovvero la sua esecuzione può essere affidata ad una macchina: ciò significa che un algoritmo non deve necessariamente essere meccanizzato, ma deve essere **meccanizzabile**; in altre parole, l'esecuzione (bada bene, l'esecuzione e non la sua ideazione) dell'algoritmo è completamente **stupida**.

Esempio 1: L'algoritmo della moltiplicazione è molto chiaro e semplice: basta solamente accedere ad una **base di dati** in cui sono memorizzati i prodotti elementari fra numeri molto piccoli, e quindi molto più semplici da trattare.
Una volta eseguite le operazioni di moltiplicazione tramite quanto esposto in precedenza, è necessario eseguire delle operazioni di addizione, che era necessario aver precedentemente memorizzato.
Ecco che quello che si è appena eseguito è un **procedimento di calcolo**.

Esempio 2: L'algoritmo della divisione fa sempre uso di una base di dati, nella quale devono essere memorizzati i risultati del prodotto del divisore con tutti i numeri decimali da 0 a 9 e confrontare ciascun prodotto con il termine da dividere per ottenere il quoziente.
Alternativamente, si sarebbe potuto creare un ciclo da 0 a 9 in cui per ogni indice si sarebbe dovuto verificare se questo fosse il fattore moltiplicativo corretto per ottenere la quantità giusta da sottrarre.

Osservazione: In ciascuno di tali esempi è essenziale la meccanicità del processo esecutivo, che appare evidente.

Quando si rappresentano delle quantità e, a maggior ragione, quando si effettuano dei calcoli, è fondamentale fissare una base di rappresentazione, da cui poi dipendono le cifre che si possono impiegare per la rappresentazione stessa.
La notazione posizionale permette anche di comprendere la rappresentazione di qualsiasi quantità con qualsiasi base, effettuando anche delle conversioni di base a seconda della maggiore o minore convenienza di rappresentazione.

Per esempio, volendo convertire una quantità rappresentata in base $\mathcal{B} = 7$ in una base $\mathcal{C} = 10$ si deve procedere come segue

$$(5203)_7 = 5 \cdot 7^3 + 2 \cdot 7^2 + 0 \cdot 7^1 + 3 \cdot 7^0 = 1715 + 98 + 0 + 3 = (1816)_{10}$$

Ovviamente le basi di rappresentazione sono almeno binarie, in quanto la **base unaria** non può, per ovvie ragioni, rappresentare alcuna quantità se non quella unica che viene permessa dalla base scelta, ossia lo 0.
Tuttavia, il processo inverso, atto a passare dalla rappresentazione di una quantità in base 10 ad una in base 3, non risulta essere così immediato.
Per cercare un algoritmmo che permette di effettuare tale conversione, si effettua un primo **passaggio controintuitivo** (che suggerisce, tuttavia, il corretto processo esecutivo), che prevede di rappresentare una quantità in base 10 in una quantità ancora in base 10, tramite un processo di divisioni successive. Si consideri, a tal proposito

$$(3412)_{10}$$

e si divida progessivamente tale numero per 10, come segue

3412	10
341	2
34	1
3	4
0	3

Leggendo, ora, i resti, al contrario si ottiene il numero cercato all'inizio. Se ora si prova a considerare un'altra base, come 3, l'operazione porta ad un risultato analogo

3412	3
1137	1
379	0
126	1
42	0
14	0
4	2
1	1
0	1

Per cui si è ottenuto

$$(3412)_{10} = (11200101)_3$$

Scegliendo la base 2 si ottiene, per esempio

241	2
120	1
60	0
30	0
15	0
7	1
3	1
1	1
0	1

Per cui si è ottenuto

$$(241)_{10} = (11110001)_2$$

Ovviamente la lunghezza di rappresentazione in base 2 prevede un numero di cifre pari a circa il triplo di quelle impiegate per rappresentare la medesima quantità in base 10, proprio perché

$$\log_2(10) \cong 3.3$$

Per passare da base 10 a base 100, le operazioni sono molto semplici

$$(375712)_{10} = [(37)\,(57)\,(12)]_{100}$$

usando come simboli

$$(00)\,,(01)\,,...,(75)\,,...,(99)$$

Si consideri, ora la base 8 e si scriva un numero binario in base ottale:

$$(010101010)_2 = [(010)\,(101)\,(010)]_8 = (252)_8$$

Ancora una volta, le cifre impiegate per la rappresentazione sono state ridotte ad un terzo, sempre perché

$$\log_2(8) = 3$$

E se ora si volesse impiegare la base 16 si otterrebbe:

$$(010101010)_2 = [(1010)\,(1010)]_{16} = (\mathrm{AA})_{16}$$

Osservazione: Si consideri una lunghezza $l = 5$. Allora usando 5 cifre, non tutte nulle, in base 10, i numeri n che si possono rappresentare sono

$$10000 \leq n \leq 99999 \quad \equiv \quad 10^4 \leq n < 10^5 \quad \equiv \quad 10^{l-1} \leq n < 10^l$$

Da ciò si può estrapolare un risultato importante

$$\log_{10}\left(10^{l-1}\right) \leq \log_{10}(n) < \log_{10}\left(10^l\right) \quad \equiv \quad l-1 \leq \log_{10}(n) < l$$

che è una relazione esatta. Tuttavia, approssimativamente, si può scrivere che

$$l_{10}(n) \cong \log_{10}(n)$$

3 Marzo 2022

Com'è noto, il matematico indiano **Ramanujan** ha affermato che la matematica esatta non rappresenta una base solida per la realtà, mentre la matematica vera è fatta di approssimazioni.
Hardy scoprì quanto fosse importante lo studio di **Ramanujan** e insieme a lui portò avanti la teoria dei numeri, una teoria **asintotica** che, come lui stesso affermava, non può essere esatta, ma fatta di approssimazioni.
Se, per esempio, si considera una quantità scritta in base 5, quale $n = (412)_5$

$$(412)_5 = 4 \cdot 5^2 + 1 \cdot 5^1 + 2 \cdot 5^0 = (107)_{10}$$

Se, ora, si fissa una lunghezza $l = 4$, una lunghezza rigida, senza considerare zeri in testa, si può capire che con 4 cifre si possono rappresentare, in base 5 numeri n nell'intervallo

$$1000 \leq n < 10000$$

ovvero tale per cui

$$5^{l-1} \leq n < 5^l$$

e ciò funziona con qualsiasi base, per cui, in generale, fissata una lunghezza l e una base $\mathcal{B}$ si ha che le quantità che possono essere rappresentate con l cifre, non tutte uguali a 0 è

$$\mathcal{B}^{l-1} \leq n < \mathcal{B}^l$$

Traducendo tale risultato tramite il logaritmo in base $\mathcal{B}$, sfruttando la crescenza in senso stretto della funzione logaritmica, si ottiene, equivalentemente

$$l - 1 \leq \log_{\mathcal{B}}(n) < l$$

in cui, ovviamente,

$$\log_{\mathcal{B}}(n) < l \leq \log_{\mathcal{B}}(n) + 1$$

che si può scrivere che

$$l_{\mathcal{B}}(n) \cong \log_{\mathcal{B}}(n)$$

Per cui l'**errore massimo** che si può commettere è di 1 cifra in base $\mathcal{B}$, nel caso peggiore, ma sarà sempre un po' maggiore del $\log_{\mathcal{B}}(n)$, per cui il logaritmo è una **sottostima della lunghezza**. Tuttavia, nello spirito di Hardy, sarà utile anche scrivere che la lunghezza binaria di n è circa uguale al logaritmo binario di n, ovvero

$$\log_{\mathcal{B}}(n) \cong \log_{\mathcal{B}}(n)$$

in cui si può interpetare il $\log_{\mathcal{B}}(n)$ come una **lunghezza analogica**, mentre $l_{\mathcal{B}}(n)$ è una **lunghezza digitale**, in quanto **intera**, con precisione alla cifra (senza nulla in mezzo): in molti casi sarà più utile la lunghezza analogica di quella digitale, in quanto molto più precisa.
Grazie a questa formula è possibile capire facilmente come si alterano le lunghezze quando si effettua un cambiamento di base. Per esempio, si può osservare che

$$\log_2(10) \cong 3.38$$

per cui la lunghezza in base 2 è circa tre volte la lunghezza in base 10.

Esempio: Per trasformare un numero da base 10 in base 5 si deve procedere per divisioni successive per 5, considerando i resti (per questo si parla di *divisione intera*). Per esempio si ha che

$$10 \div 3 = 3 \text{ con resto di } 1$$

in cui, ovviamente, il resto r può essere

$$0 \leq r < D$$

con D divisore. Convertendo 32 da base 10 a base 5 ci si aspetta di ottenere un resto $0 \leq r \leq 4$.

1.1 Architettura dei calcolatori

Il calcolatore, naturalmente, si basa sulla logica binaria, ovvero opera impiegando la rappresentazione in base 2.
Il metodo più utilizzato per rappresentare caratteri diversi da quelli binari, tramite una codifica binaria, è il metodo ASCII (dall'inglese, American Standard Code For Information Interchange). Naturalmente, siccome la codifica tramite ASCII fa uso di soli 7 bit (sarebbero 8, ma un bit è riservato alla parità, per la rilevazione degli errori), il numero di n-uple binarie che si possono ottenere è 2^7, un numero certamente irrisorio per la rappresentazione di tutti i caratteri alfanumerici necessari per la comunicazione multilinugua.
In generale, fissata una lunghezza n, il numero di n-uple binarie distinte è, ovviamente, 2^n, che rappresenta una crescita esponenziale, praticamente infinita, anche se, ovviamente, in teoria sono un numero ben limitato.

Esempio: Si consideri una macchina calcolatrice che considera delle istruzioni di 9 bit come di seguito esposto:

NOME ISTRUZIONE	ISTRUZIONE
ADD	$010 \times \times \times \times \times \times$
PUNCH	$100 \times \times \times \times \times \times$

Tabella 1: Tabella di istruzioni operative per un calcolatore

Naturalmente, tale linguaggio è **Assembly**, ovvero un linguaggio molto simile al linguaggio macchina, che risulta particolarmente complesso da impiegare per lo sviluppo di software.

1.2 Diagramma di flusso

Si consideri il seguente **flowchart**, o **diagramma di flusso**:

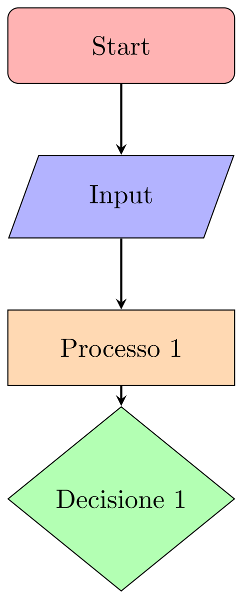

Figura 1: Diagramma di flusso

Tuttavia, tale tecnica di progettazione algoritmica è oramai superata, lasciando il posto allo **pseudocodice**, ossia un linguaggio di definizione delle istruzioni slegato da qualsiasi specifico linguaggio di programmazione di riferimento, che permette di esporre una serie di istruzioni esecutive molto simili a quelle di un programma vero e proprio.

2 Ordinamento (sorting)

Si espongono, di seguito, i principi di algoritmica dei più importanti algoritmi di ordinamento. Ciascuno di tali algoritmi prevede di effettuare l'ordinamento di n numeri forniti come input, in modo debolmente crescente, in caso di uguaglianza.

2.1 Bubble-Sort

Si espone di seguito l'algoritmo di ordinamento **bubble-sort** impiegando lo *pseudocodice*:

Algorithm 1 Bubble-sort

```
do the following n − 1 times
    point to the 1st element
    do the following n − 1 times
        compare with next
        if wrong order exchange
        point to the next
```

Naturalmente tale algoritmo è corretto e lo si può verificare immediatamente, considerando, per esempio, i seguenti 5 elementi, così ordinati:

$$\boxed{2}\ \boxed{3}\ \boxed{1}\ \boxed{4}\ \boxed{5}$$

Ovviamente il procedimento ci porta ad eseguire l'algoritmo 4 volte. Nella prima iterazione si ottiene

$$\boxed{2}\ \boxed{1}\ \boxed{3}\ \boxed{4}\ \boxed{5}$$

la seconda iterazione, invece, porta ad ottenere

$$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}$$

mentre le ultime due iterazioni sono superflue. Si capisce facilmente che tale algoritmo non risulta essere pienamente efficiente, in quante alcune iterazioni potrebbero essere evitate, tramite un **flag**, per esempio. Allo stato attuale, il numero delle iterazioni da eseguire è

$$\#\text{iterazioni} = (n-1)\cdot(n-1)$$

Se, invece, si facesse in modo di evitare alcune iterazioni si avrebbe un numero di iterazioni

$$(n-1) \leq \#\text{iterazioni} \leq (n-1)^2$$

considerando $n-1 \cong n$, e quindi $(n-1)^2 \cong n^2$ si può dire che la complessità del bubble-sort è **quadratica**, in quanto il numero delle iterazioni è n^2.

4 Marzo 2022

L'algoritmo bubble-sort non viene utilizzato, ad oggi, così come non si impiega lo pseudocodice in *pseudo-english* (da leggere psude-inglish). Esso è funzionale, ma non efficiente, in quanto la sua complessità è n^2.
Ecco che per definire un algoritmo di ordinamento non è necessaria solamente la sua funzionalità, ma anche l'efficienza.

2.2 Insertion-sort

L'insertion-sort è un algoritmo di ordinamento che prevede di considerare ciascuna quantità da ordinare ad una ad una e di effettuare un confronto solo quando ci sono dei cambiamenti.
La prima quantità è ovviamente già in ordine con se stessa. Se la seconda è più piccola della prima, si effettua uno scambio, per cui ora i primi due numeri sono ordinati. Si considera, ora, il terzo numero e se questo è più piccolo del secondo si effettua uno scambio e un nuovo confronto tra la seconda e la prima e così via.
Pertanto si effettuano tutti i confronti solamente quando si ha uno scambio delle quantità: questo comporta che il minimo numero di iterazioni è $n-1$, se n è il numero delle quantità da ordinare. Se, invece, tutte le quantità sono in disordine si effettua un numeri di iterazioni pari

$$1+2+...+(n-2)+(n-1)=\frac{n\cdot(n+1)}{2}$$

una formula molto semplice che Gauss determinò come segue, ovverosia scrivendo la somma dei numeri da 1 a k in ordine crescente e poi decrescente, come mostrato di seguito:

$$\begin{array}{ccccccccccc} 1 & + & 2 & + & 3 & + & ... & + & k-1 & + & k \\ \\ k & + & k-1 & + & k-2 & + & ... & + & 2 & + & 1 \end{array}$$

essendo k numeri in ambedue le righe, sommando i due termini corrispondenti, uno sotto l'altro, si ottiene sempre $k+1$ che, sommato per k volte produce $k\cdot(k+1)$. Tuttavia, dal momento che tale quantità è il doppio di quella richiesta si ottiene

$$\frac{k\cdot(k+1)}{2}$$

Pertanto si ha che il numero di iterazioni dell'algoritmo *insertion-sort* è

$$n\cong n-1\leq \#\text{iterazioni}\leq\frac{n\cdot(n-1)}{2}\cong n^2$$

che, in maniera approssimata è

$$n\leq \#\text{iterazioni}\leq n^2$$

pertanto, nel caso migliore, la **complessità è lineare**, mentre nel caso peggiore, la **complessità è quadratica**.

2.3 Pseudocodice

Per la scrittura dello pseudocodice si devono impiegare delle notazioni e dei simboli ben specifici, che di seguito vengono riportati.

2.3.1 Assegnazione

L'**assegnazione** viene indicata con il simbolo = (oppure := o ←), anche se l'assegnazione non è un'uguaglianza. Per esempio, la notazione

$$A=3$$

significa che nella cella di memoria A viene inserito il valore 3. Analogamente, se si scrive

$$A = A + 1$$

significa che il valore presente nella cella di memoria A viene incrementato di 1 unità.

2.3.2 Condizione

La specifica della **condizione** avviene tramite l'istruzione **if**, secondo la notazione seguente:

```
if C then
    istruzioni
else
    istruzioni
```

2.3.3 Ciclo for

Il **ciclo for** è un'istruzione di ciclo in cui vengono indicate specificatamente le iterazioni che devono essere eseguite, secondo la notazione seguente

for $i = 0$ **to** n **do**

2.3.4 Ciclo while

Il **ciclo while** è un'istruzione di ciclo in cui si effettuano le istruzioni fintantoché la condizione specifcata è vera

while C **do**

Si consideri lo pseudocodice dell'algoritmo **INSERTION-SORT(A)**, esposto di seguito, dove **A** sta ad indicare **array**, ovvero un record di $length[A] = n$ valori da ordinare, già forniti in input. Di seguito si espone lo pseudocodice, in cui si parte da 1 come posizione iniziale dell'array:

Algorithm 2 Insertion-sort

```
for j = 2 to length[A] = n
    do key = A[j]
        ...
        i = j − 1
    while i > 0 ∧ A[i] > key
        do A[i + 1] = A[i]
            i = i − 1
            A[i + 1] = key
```

Tale codice é concluso e il suo funzionamento può essere facilmente verificato come segue, considerando l'array A di lunghezza $length[A] = 6$:

$$\boxed{5}\ \boxed{2}\ \boxed{4}\ \boxed{6}\ \boxed{1}\ \boxed{3}$$

Partendo con $j = 2$ si fissa $key = A[j] = 2$ e imponendo $i = 1$, si entra all'interno del ciclo *while*, in quanto $i > 0$ e $A[i] > key$ e si effettua l'istruzione $A[i + 1] = A[i]$ e $A[i + 1] = key$, trovandosi nella configurazione seguente

$$\boxed{2}\ \boxed{5}\ \boxed{4}\ \boxed{6}\ \boxed{1}\ \boxed{3}$$

Terminato il ciclo while e il ciclo for si procede con $j = 3$ si fissa $key = A[j] = 4$ e imponendo $i = 2$, si entra all'interno del ciclo *while*, in quanto $i > 0$ e $A[i] > key$ e si effettua l'istruzione $A[i + 1] = A[i]$ e $A[i + 1] = key$, trovandosi nella configurazione seguente

$$\boxed{2}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{1}\ \boxed{3}$$

Adesso, terminato il ciclo while e for, si considera $j = 4$, specificando $key = A[j] = 6$ e $i = 3$. In questo caso, tuttavia, non si entra nel ciclo while, in quanto $i > 0$, ma $A[i] < key$. Si procede direttamene con $j = 5$, $key = A[j] = 1$ e $i = 4$ e si entra nel ciclo while, compiendo tutte le iterazioni

$$\boxed{2}\ \boxed{4}\ \boxed{5}\ \boxed{1}\ \boxed{6}\ \boxed{3}$$

$$\boxed{2}\ \boxed{4}\ \boxed{1}\ \boxed{5}\ \boxed{6}\ \boxed{3}$$

$$\boxed{2}\ \boxed{1}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{3}$$

$$\boxed{1}\ \boxed{2}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{3}$$

e così via fino ad arrivare all'ordinamento finale

$$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$$

Ecco che, come si può vedere, tale algoritmo ha una complessità che, nel caso migliore, è **lineare** (n) e nel caso peggiore è **quadratica** (n^2).
Mentre si chiama **complessità tipica** la **complessità media**, ovvero la complessità dell'algoritmo nel caso intermedio. In questo caso la complessità tipica è n^2, che può essere calcolata, in maniera non propriamente corretta, come segue

$$\frac{n + n^2}{2} = n^2$$

Esiste, infine, anche una **complessità empirica**, basata sull'uso pratico dell'algoritmo: in particolare, l'algoritmo insertion-sort funziona particolarmente bene quando il **numero degli elementi da ordinare è ridicolmente basso**, il che potrebbe essere un controsenso; tuttavia, potrebbe essere particolarmente utile ricorrere all'ordinamento di pochi numeri all'interno di una procedura particolarmente complessa: ecco, allora, che l'utilizzo di insertion-sort diviene conveniente (cosa che non accade per bubble-sort).

Osservazione: È importante osservare che l'algoritmo di insertion-sort è un **algoritmo di ordinamento in loco**, ovvero tale per cui non si impiega un altro array per l'ordinamento, ma tutte le operazioni si effettuano sullo stesso array di partenza.

Osservazione: Quando si parla di algoritmica, non è possibile parlare di **completezza** senza parlare di **complessità**.

3 Grafi

Il **grafo** è una **struttura finita**. Gli elementi costituitvi di un grafo sono i **vertici** (o **nodi**) e gli **archi** (o **lati**) (dall'inglese *arcs* o *edges*). Per indicare i vertici si impiega la lettera v, mentre per indicare gli archi si usa la lettera ξ.
Un arco collega due vertici distinti che, per il momento, non è orientato, non rappresenta una freccia, in quanto si parla di **grafi semplici**, come illustrato di seguito:

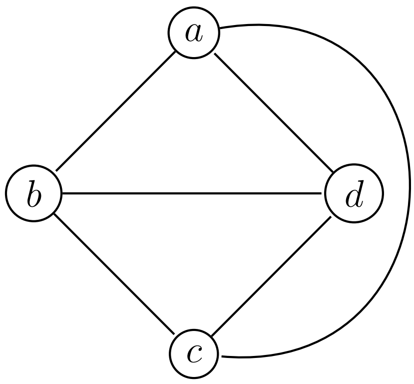

Figura 2: Esempio di grafo semplice

Il calcolo della copertura dei vertici prende il nome di **vertex cover** e prevede di definire il numero minimo di vertici essenziali per coprire tutti gli archi. L'ottimizzazione del grafo, in questo caso, prevede di determinare la copertura minima.
Si supponga di avere a disposizione k vertici (che si indica come $|v| = k$, in cui $|v|$ rappresenta la **cardinalità** dell'insieme dei vertici). Naturalmente, la copertura minima pari a 0 si ha quando i vertici del grafo sono tutti scollegati, ovvero non ci sono archi.
Il numero di archi in un grafo completo è pari a

$$\frac{|v| \cdot (|v| - 1)}{2}$$

per cui si potrebbe, in questo caso limite, affermare che la copertura minima sia k, ma se si elimina un nodo ancora la copertura sussiste, in quanto su ogni arco vi sarà sempre almeno un nodo coperto. Quindi si può affermare che la *vertex cover*, nel caso generale è compresa tra 0 e $k - 1$, con k numero dei vertici.

9 Marzo 2022

Il problema del **vertex cover**, ovvero di “ricoprimento dei vertici”, è un proplema che riguarda la teoria dei grafi.
Gli elementi costituitvi di un grafo sono i **vertici** (o **nodi**, molto più raramente chiamati *punti*) e gli **archi** (dall’inglese *edges*, traducibili in **spigoli** o, più impropriamente, in *lati*), per il momento non orientati, che collegano due vertici, per il momento necessariamente distinti.
La notazione per indicare vertici e archi è la seguente

- L’insieme dei vertici si denota con v
- L’insieme degli archi si denota con ξ

Naturalmente, sussiste la possibilità che in un grafo tutti i vertici siano sconnessi ed **isolati**, ovvero il numero degli archi sia nullo, per cui si ottiene che $|\xi| = 0$.
Analogamente, volendo collegare tutti i nodi con un arco (ottenendo un **grafo completo**), si procede come seugue: partendo da un primo vertice se ne collega un secondo, necessariamente distinto; ma non volendo considerare ogni arco due volte, si divide per due, ottenendo

$$\frac{|v| \cdot (|v| - 1)}{2} \cong |v|^2$$

da cui si evvince che il numero degli archi, in un grafo, è compreso tra

$$0 \leq |\xi| \leq \frac{|v| \cdot (|v| - 1)}{2}$$

considerando $|v|$ la cardinalità, ossia il numero dei vertici considerati che costituiscono il grafo.
Il problema di **vertex cover** è un problema di ottimizzazione: ridurre il numero minimo di vertici tale per cui nel grafo ad ogni arco deve essere collegato almeno un vertice coperto. Per esempio, in un grafo dove ogni nodo è isolato, il numero minimo dei vertici da coprire è 0, in quanto non ci sono archi. Nel caso di un **grafo completo**, come quello esposto di seguito

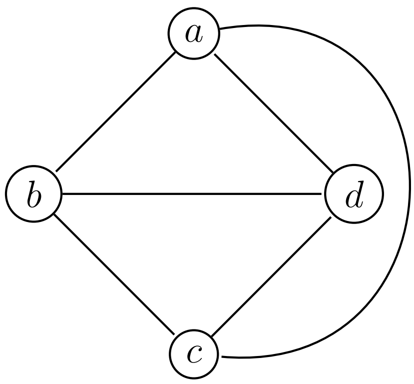

Figura 3: Esempio di grafo semplice completo

la copertura minima non è, come contrariamente si potrebbe pensare all’inizio, pari a $|v|$, in quanto eliminando uno qualsiasi dei nodi, ancora ad ogni arco sarà collegato almeno un nodo comperto. Pertanto si ha che il numero $\#nodi$ dei nodi che si dovranno coprire al fine di risolvere il problema del vertex cover in un grafo avente $|v|$ vertici sarà sempre compreso tra

$$0 \leq \#nodi \leq |v| - 1$$

Per la risoluzione meccanica del **vertex cover** vi sono due algoritmi, di cui solo il secondo realmente efficace. Si consideri, a titolo esemplificativo, il grafo seguete

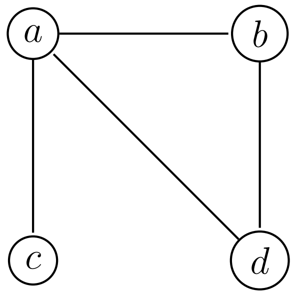

Figura 4: Esempio di grafo semplice

in cui, naturalmente, per coprire tutti gli archi sarà sufficiente considerare il vertice a e il vertice d (oppure il nodo b), ottenendo, come possibile copertura

$$\boxed{a}\ \boxed{b}\ \boxed{c}\ \boxed{d}$$

$$\boxed{1}\ \boxed{0}\ \boxed{0}\ \boxed{1}$$

in cui con $\boxed{1}$ si rappresenta la copertura del rispettivo vertice. Ecco che questa è una k-upla binaria di **peso** $w = 1 + 1 = 2$, in cui $k = 4$: usando tale notazione, quindi, un procedimento meccanico, atto a verificare la corretta copertura prevede di considerare tutti gli archi e verificare per ciascuno di essi che almeno ad un vertice collegato dall'arco in questione corrisponda un 1; se un arco è collegato a due vertici cui corrisponde 0, la copertura è scorretta, ovviamente.
Tuttavia, la k-upla 1001 è anche una codifica binaria su 4 bit del numero $(9)_{10}$, che suggerisce la procedura di controllo seguente, definita a partire da k numero di vertici

Algorithm 3 Vertex-cover

for $i = 0$ **to** 2^k
  $i \rightarrow$ binario
  $check(i)$

ove *check(i)* è una procedura che svolge il compito precedentemente esposto: presa in ingresso una k-upla binaria, cui si fa corrispondere la copertura di rispettivi vertici, verifica per ciascun arco che ad esso sia collegato un vertice effettivamente coperto, ovvero cui corrisponde un 1 nella k-upla binaria considerata; alla fine della procedura si ottiene la copertura di peso minore, ovvero quella con meno 1 e quindi che prevede meno vertici coperti.
Tale algoritmo non risulta propriamente efficiente; pertanto, al fine di eliminare alcune iterazioni, si potrebbe pensare di partire con il peso w ed effettuare il medesimo *check* per tutte n-uple di peso w, come mostrato di seguito:

Algorithm 4 Vertex-cover

for $w = 0$ **to** $k - 1$
  for all

che è un procedimento più razionale, in quanto si controlla progressivamente se sono sufficienti un numero sempre maggiore di vertici da coprire (basta 1 vertice? Bastano 2 e così via...) e si esce dal ciclo quando si trova il primo, in quanto quella copertura sarà certamente la minima.
Da notare che si cicla fino a $k - 1$ e non fino a k, in quanto è noto dalla teoria che il numero di vertici che si andranno a coprire per risolvere un qualsiasi problema di vertex-cover è sempre compreso tra 0 e $k - 1$, con k numero di vertici: ma l'unica n-upla con peso massimo è quella con tutti 1, la quale costituisce una sola configurazione tra le 2^k possibili. Pertanto, nel caso peggiore, si potrebbe procedere ad effettuare un numero di iterazioni pari a $2^k - 1$, che non è molto dissimile dal caso precedente, in cui si facevano inevitabilmente 2^k iterazioni.
Questo algoritmo, pertanto, pur essendo corretto, è ispirato al meccanismo dell'**exaustive search** (dall'inglese, ricerca esauriente), che prevede di controllare tutti gli archi al fine di verificarne la

corretta copertura.
Il motivo per cui tale algoritmo è inutilizzabile è che presenta un numero di **iterazioni esponenziale** e, conseguentemente, una **complessità esponenziale**, la quale è intollerabile, dal momento che il numero delle iterazioni cresce esponenzialmente al variare dell'imput.
Di seguito si espone, invece, un nuovo algoritmo che risolve il problema del **vertex cover**; alla base di tale algoritmo si pone la seguente idea: si considerino dapprima due nodi connessi da un arco e si eliminino tutti gli archi che incidono sul primo e sul secondo vertice e si proceda a considerare un nuovo arco che insiste su due nodi ancora non considerati e si eliminino tutti gli altri archi che incidono sui nodi stessi e così via, fino ad esaurire tutti gli archi a disposizione, tale che alla fine della procedura si ottiene un ricoprimento vero e proprio.
Come di consueto, nello pseudocodice esposto di seguito, l'input non viene specificato a priori, ma il numero delle iterazioni per specificare l'input è noto, ossia è pari a $\cong |v| + |\xi|$.
Lo pseudocodice è il seguente:

Algorithm 5 Vertex-cover

$C \leftarrow \varnothing$
$E' \leftarrow \xi(\mathcal{G})$
while $E' \neq \varnothing$
  do ... (u, v) in E
    $C \leftarrow C \cup \{u, b\}$
    delete incidienti
return C

In cui C è un contenitore, inizialmente vuoto, all'interno del quale successivamente andranno inseriti i vertici necessari per la vertex cover. Invece, E' è un contenitore, inizialmente pieno, in quanto contiene tutti gli archi che dovranno essere esaminati/scartati. Dopodiché, fintantoché non ci sono più archi da analizzare, si considera un primo arco, designato con la notazione (u, v), i cui due estremi, appunto u e v andranno ad essere inseriti all'interno di C, mentre verrano eliminati da E' tutti gli archi incidienti in u o in v, finché non si avranno più archi a disposizione.
Il numero di iterazioni in cui tale algoritmo si impegna è, approssimativamente, pari a $|v| + |\xi|$, ovvero si ha una **complessità lineare**: questa è un'ottima notizia, in quanto significa che tale algoritmo è velocissimo; l'unico problema è che esso è scorretto.
Si consideri, a titolo di esempio, il seguente grafo semplice:

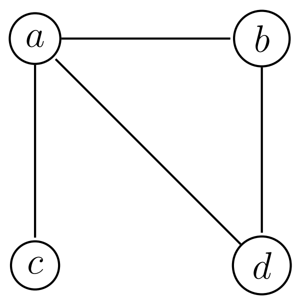

Figura 5: Esempio di scorrettezza dell'agoritmmo considerato

In questo caso, l'algoritmo, considerando per primo l'arco $a - b$, elimina tutti gli altri archi e produce un risultato corretto.
Tuttavia, se il primo arco considerato dall'algoritmo fosse $a - c$, l'algoritmo è costretto a considerare anche l'arco $b - d$, non producendo un risultato corretto.
Tale algoritmo prende il nome di **algoritmo approssimato**, in quanto non sempre produce un risultato corretto, ma mai disastroso.
Tuttavia, non esiste un algoritmo che sia corretto e anche accettabile dal punto di vista del numero delle iterazioni e quindi della complessità: pertanto ci si deve accontentare dell'algoritmo approssimato appena esposto.
Per capire il range di risposta di tal algoritmo, si assuma che il numero di nodi ottimali necessari alla copertura sia $|\mathcal{O}|$ e il numero dei nodi effettivamente ottenuti dalla procedura algoritmica sia

$|\mathcal{P}|$, legati dalla seguente relazione, dimostrata con l'esempio precedente:

$$|\mathcal{O}| \leq |\mathcal{P}|$$

Naturalmente, per quanto visto con l'esempio precedente, tale disuguaglianza può anche essere stretta. Ovviamente, se ora si considerano gli archi privilegiati dall'algoritmo esposto, denotati con $|\mathcal{A}|$, ovverosia l'insieme degli archi che l'algoritmo ha via via considerato, esso, naturalmente, presenta la seguente cardinalità

$$|\mathcal{A}| = \frac{|\mathcal{P}|}{2}$$

in quanto su ogni arco vi sono due nodi (u e v nello pseudocodice), per cui basta considerarne la metà. Ora, il numero di nodi ottimale $|\mathcal{O}|$ deve essere necessariamente almeno uguale al numero di archi privilegiati $|\mathcal{A}|$, dal momento che ciascuno di tali archi deve insistere almeno su un vertice coperto. Pertanto $|\wr|$ e $|\mathcal{A}|$ sono legati dalla seguente relazione

$$|\mathcal{O}| \geq |\mathcal{A}|$$

per cui si ottiene che

$$\frac{|\mathcal{P}|}{2} \leq |\mathcal{O}| \leq |\mathcal{P}|$$

4 Algoritmi aritmetici

L'**algoritmo di Euclide** permette di calcolare il **Massimo Comune Divisore** (**M.C.D.**) tra due numeri, ma non procedendo alla fattorizzazione in fattori primi tra le due quantità considerate. Infatti, normalmente, per determinare l'M.C.D. tra due quantità è necessario procedere alla scoposizione delle due in fattori primi, come mostrato di seguito per i numeri 12 e 9:

$$12 = 3 \cdot 2 \cdot 2$$
$$9 = 3 \cdot 3$$

da cui si evince che

$$\mathrm{MCD}(12, 9) = (12, 9) = 3$$

Tuttavia, non è possibile procedere attraverso la fattorizzazione per la risoluzione di tale problema, in quanto gli algoritmi per la fattorizzazione sono estremamente lenti. L'algoritmo di Euclide, invece, risolve tale problematica in maniera corretta, veloce ed efficiente, basandosi sul meccanismo della **ricorsività** e delle divisioni intere. Infatti, com'è noto, la divisione può essere di due tipologie

1. Divisione esatta: $10 \div 3 = 3,\overline{3}$
2. Divisione intera: $10 \div 3 = 3$ con resto 1

10 Marzo 2022

Naturalmente, un algortimo non può essere applicato concretamente se ha una crescita esponenziale: esso è inutilizzabile, in quanto il tempo di risposta è troppo elevato per avere un impiego pratico.
L'esposto seguente, tratto da un lavoro di Gary & Johnson, ne dà una fondamentale prova pratica, considerando un calcolatore le cui istruzioni durano $0,000001$ s per essere processate; naturalmente l'algoritmo considera input variabili, di lunghezza 10, 20, 30, 40, 50 e 60 e si suppone, per semplicità, che la lunghezza dell'input determini in modo quanto più preciso e linearmente dipendente il numero delle operazioni che devono essere eseguite: pertanto, se l'input ha lunghezza 60 significa che sono necessarie 60 operazione per terminare l'algoritmo e quindi $0,00006$ s sarà il tempo impiegato per l'esecuzione.
Se, invece, la complessità dell'algoritmo è quadratica, allora ciò significa che se l'input è di lunghezza 60, il numero di operazioni diviene $60 \cdot 60 = 3600$ e quindi il numero di secondi diviene $0,0036$ s.
Se la complessità è cubica, allora con lunghezza dell'input di 60 il numero di operazioni diviene $60 \cdot 60 \cdot 60$ e quindi il tempo impiegato è di $0,216$ s.
Con una complessità quintica, a 60 di lunghezza corrispondono 60^5 istruzioni e quindi 13 minuti di esecuzione.
Passando ad una complessità esponenziale, come 2^n, con lunghezza dell'imput pari a 60 si hanno 2^{60} istruzioni e quindi un tempo esecutivo di 360 secoli. Passando appena a 3^n, con lunghezza dell'imput pari a 60 si hanno 3^{60} istruzioni e quindi un tempo esecutivo di $1,3 \cdot 10^{13}$ secoli.

Osservazione: Si osservi che considerare un calcolatore 1000 volte più veloce può essere significativo se la complessità dell'algoritmo è polinomiale, ma è totalmente ininfluente se la complessità è esponenziale.

4.1 Algoritmo di Euclide

Si consideri il seguente algoritmo, noto come **algoritmo di euclide**, estremamente fulmineo:

Algorithm 6 Euclide

begin
$a, b = m, n$
while $b \neq 0$ **do** $a, b = b, a \bmod b$
mcd $= a$
end

In questo caso l'algoritmo considera come input **due numeri interi** m e n di cui ha senso determinare l'**m.c.d**: pertanto essi devono essere necessariamente interi per ipotesi, in quanto nell'algoritmo non è previsto un controllo sintattico a monte. Generalmente si considerano $m > n$, ma ciò è ininfluente, in quanto il programma provvede ad effettuare un cambiamento del loro ordine in automatico.
Nell'algoritmo vi sono delle assegnazioni composte, in cui

$$a, b = m, n$$

ovvero ad a si assegna il valore m, mentre a b si assegna il valore n. Così come in seguito si ha

$$a, b = b, a \bmod b$$

ovvero ad a si assegna il vecchio valore b, mentre a b si assegna il resto della divisione intera tra a e b. Alla fine del ciclo si ottiene che l'**m.c.d.** cercato è proprio a.
Si consideri, a tal proposito, il seguente esempio:

$$\underset{a}{\boxed{12}}\ \underset{b}{\boxed{9}}$$

Dopo il primo passo si ottiene

$$\underset{a}{\boxed{9}}\ \underset{b}{\boxed{3}}$$

ed infine

$$\underset{a}{\boxed{3}}\ \underset{b}{\boxed{0}}$$

ecco che nella prima cella si ha proprio l'm.c.d. cercato. Ora, tuttavia, bisogna verificare se tale algoritmo sia effettivamente corretto e che quella considerata non sia solo una combinazione; inoltre, per determinare la complessità dell'algoritmo è necessario considerare, essenzialmente, il numero di **iterazioni libere** del ciclo while, in quanto la complessitì dell'algoritmo dipende unicamente da tale fattore.

Osservazione: Si osservi, innanzitutto, che il ciclo while si chiude visto che in posizione b sarà presente un resto, ovvero una quantità intera che progressivamente diminuisce fino a diventare 0. L'ultimo passaggio, naturalmente, è il più semplice, in quanto l'algoritmo, prima dell'ultima iterazione, si ritroverà sempre nella situazione desiderata:

$$\boxed{\alpha \cdot \text{m.c.d.}}\ \boxed{\text{m.c.d.}}$$

per cui nell'ultima iterazione, effettuando la divisione intera tra una quantità e un suo multiplo non si può che ottenere resto 0 e, quindi, nella cella a si ha proprio l'm.c.d. che si sta cercando, come mostrato di seguito:

$$\boxed{\text{m.c.d.}}\ \boxed{0}$$

Per la dimostrazione della correttezza dell'algoritmo, anche nelle fasi intermedie, si deve dimostrare essenzialmente che, a qualunque fase del processo esecutivo

$$\text{M.C.D.}(a, b) = \text{M.C.D.}(b, a \bmod b)$$

ovvero che quello che si trova nelle celle $a - b$ nell'istante t e ciò che si trova in $a - b$ all'istante successivo hanno entrambi lo stesso massimo comune divisore: bisogna, di fatto, dimostrare quella che in matematica prende il nome di **invariante**; in questo caso l'invariante è proprio l'm.c.d. Infatti, le quantità che si trovano progressivamente nelle due celle hanno sempre lo stesso massimo comune divisore, a qualunque istante, fino ad arrivare alla configurazione finale in cui nella prima cella è presente un multiplo del valore della seconda cella.
Per dimostrare, quindi, l'invarianza seguente:

$$\text{M.C.D.}(a, b) = \text{M.C.D.}(b, a \bmod b)$$

si deve dimostrare che un diviore di (a, b) è anche divisore di $(b, a \bmod b)$ e viceversa, per cui hanno lo stesso massimo comune divisore. Si supponga, allora, che α divida sia a che b, per cui si ha che $a = \alpha a'$ e $b = \alpha b'$; inoltre, considerando $r = a \bmod b$ quale il resto della divisione tra a e b, si può scrivere, per il teorema del quoziente e resto

$$a = qb + r$$

Pertanto, sapendo che α divide sia a che b, bisogna verificare che α divida anche $r = a \bmod b$ per concludere la dimostrazione dell'invarianza. Tuttavia, il teorema del quoziente e resto permette di ricavare r come segue in cui, ovviamente, si ha che

$$r = a - qb = \alpha \cdot (a' - qb')$$

in cui è evidente come α sia, effettivamente, un divisore anche di $r = a \bmod b$. Procedendo, ora, al contrario, supponendo che $b = \alpha b'$ e $r = \alpha r'$, sempre per il teorema del quoziente e resto si può scrivere $a = qb + r = \alpha \cdot (qb' + r')$, per cui, ancora una volta, α divide anche a, come volevasi dimostrare.

Osservazione: Avendo dimostrato la correttezza dell'algoritmo di Euclide, bisogna ora procedere alla verifica della sua straordinara rapidità. Il problema della complessità dell'algoritmo di Euclide, il quale era un alessandrino, venne risolto brillantemente da un matematico francese di nome **Lamé**, nel 1844, il quale, a tutti gli effetti, è da reputarsi il padre della **teoria della complessità**, quando **Reynaud**, nel 1811 lo aveva già preceduto, ma si parla sempre della prima metà dell'800.
Per comprendere il meccanismo alla base dello studio della complessità, si consideri un numero rappresentato in base 10, quale il seguente $n = (37855)_{10}$, il quale viene progressivammente ridotto

di **almeno** 10 volte ad ogni iterazione, portandolo progressivamente a 3785, 378, 37, 3 ed infine 0. Pertanto, al più, il numero di iterazioni necessarie per annientare questo valore e portarlo a 0 è pari alla **lunghezza decimale** del numero n (in questo caso pari a 5) che, con una buona dose di approssimazione può essere considerata

$$l_{10}(n) \cong \log_{10}(n)$$

Se, ora, il numero n considerato viene scritto in binario e ad ogni iterazione viene annientato della metà, al più serviranno un numero di iterazioni pari alla lunghezza binaria del numero n considerato per annientarlo, ovvero

$$\#\text{iterazioni} \leq l_2(n) \cong \log_2(n)$$

Con questa premessa, considerando l'algoritmo di Euclide e procedendo con i calcoli di **Lamé**, si prendano ad esempio tre iterazioni successive del ciclo while, che producono i seguenti tre stati corrispondenti a tre istanti consecutivi dell'esecuzione:

$$\boxed{a}\ \boxed{b}$$

$$\boxed{b}\ \boxed{c}$$

$$\boxed{c}\ \boxed{d}$$

in cui, ovviamente, si ha che $a \geq b \geq c \geq d$; inoltre, per come sono stati ottenuti a, b, c e d appare evidente che

$$a = bq + c \ \text{ e } \ b = qc + d$$

A parte il caso iniziale in cui i due numeri $a \leq b$, cui l'algoritmo provvede cambiandoli d'ordine, in generale si ha sempre che $a > b$ e, quindi, il **quoziente** della divisione tra i due numeri è sempre **almeno 1**, quindi, siccome $q \geq 1$ deve essere che

$$b = qc + d \geq c + d$$

ma non solo: è anche noto che il valore della prima posizione è sempre maggiore o uguale del valore nella seconda posizione, ovvero $c \geq d$, per cui si ha che

$$b = qc + d \geq c + d \geq 2d$$

Questo significa che in ogni passo doppio, ovvero ogni due iterazioni, il contenuto della seconda cella di memoria è dimezzato, o peggio, in quanto si ha che $b \geq 2d$. Volendo far sì che l'elemento nella seconda cella divenga zero, poiché ad ogni doppia iterazione il suo valore viene almeno dimezzato, il numero di passi doppi necessari è al più uguale al logaritmo in base 2 di n, quindi

$$\#\text{passi doppi} \cong \log_2(n) \longrightarrow \#\text{passi singoli} < 2\log_2(n)$$

pertanto, il numero di passi di cui si necessità per terminare il ciclo while è estremamente piccolo, anche nel caso in cui il numero n considerato è enormemente grande.

Osservazione: Un altro modo per descrivere l'algoritmo di Euclide è quello che riguarda la **ricorsività**, come mostrato di seguito

Algorithm 7 Euclide

```
Procedura Euclid(a,b)
if b = 0 then
    return a
else
    return Euclid(b, a mod b)
```

Esempio: Per esempio, si considerino le iterazioni seguenti

$$\text{Euclid}(30, 21) \longrightarrow \text{Euclid}(21, 9) \longrightarrow \text{Euclid}(9, 3) \longrightarrow \text{Euclid}(3, 0) = 3$$

4.2 Notazione O grande

Si considerno due funzione $f(x)$ e $g(x)$ infinite a $+\infty$, ovvero

$$\lim_{x\to+\infty} f(x) = +\infty \quad \text{e} \quad \lim_{x\to+\infty} g(x) = +\infty$$

Allora le due funzioni si rassomiglieranno per $x \to +\infty$ quando hanno lo stesso ordine di infinito, ovvero

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = \alpha \in \mathbb{R}^+ - \{0\}$$

Tuttavia, taluna è una generalizzazione molto spartana, ma la notazione O grande lo è ancora di più. Si considerino, nuovamente, due funzioni $f(x)$ e $g(x)$, le quali non debbono più essere infinite a $+\infty$, ma è sufficiente che esse siano **definitivamente positive**, ovvero

$$\exists x_n : \forall x > x_n, f(x) > 0 \wedge g(x) > 0$$

In questo caso, pertanto, affermare che $f(x)$ "assomiglia" a $g(x)$ significa affermare che $f(x)$ appartiene alla stessa **classe di equivalenza** di $g(x)$ (ovvero la classe delle funzioni che rassomigliano a $g(x)$), che si indica come segue

$$f(x) \in \Theta\left(g(x)\right)$$

che, generalmente, verrà denotato con

$$f(x) = \Theta\left(g(x)\right)$$

nonostante sia più propriamente corretto impiegare un segno di appartenenza in luogo di uno di uguaglianza.

APPARTENENZA ALLA CLASSE DI EQUIVALENZA DI UNA FUNZIONE

Si considerino due funzioni $f(x)$ e $g(x)$ **definitivamente positive**; si dirà che $f(x)$ appartiene alla stessa classe di equivalenza di $g(x)$ e si scriverà

$$f(x) = \Theta\left(g(x)\right)$$

se

$$\exists c_1 > 0, c_2 > 0, \overline{x} : c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x), \quad \forall x \geq \overline{x}$$

ovvero si riesce a descrivere ipoteticamente, con la funzione $g(x)$, una guaina all'interno della quale racchiudere la funzione $f(x)$.

Osservazione: Dalla definizione appena fornita, appare evidente come questa sia a tutti gli effetti una **relazione di equivalenza**, in quanto è soddisfatta la proprietà **riflessiva**

$$g(x) = \Theta\left(g(x)\right)$$

in quanto basta considerare $c_1 = c_2 = 1$; inoltre è soddisfatta anche la proprietà **simmetrica**:

$$f(x) = \Theta\left(g(x)\right) \longrightarrow g(x) = \Theta\left(f(x)\right)$$

giacché se si ha che

$$c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x)$$

allora si può scrivere

$$\frac{1}{c_2} \cdot f(x) \leq g(x) \leq \frac{1}{c_1} \cdot f(x)$$

e infine quella **transitiva**

$$f(x) = \Theta\left(g(x)\right), g(x) = \Theta\left(h(x)\right) \longrightarrow f(x) = \Theta\left(h(x)\right)$$

in quanto se

$$c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x) \;\text{ e }\; c_3 \cdot h(x) \leq g(x) \leq c_4 \cdot h(x)$$

è facile concludere che

$$c_1 \cdot c_3 \cdot h(x) \leq c_1 \cdot g(x) \leq f(x) \text{ e } f(x) \leq c_2 \cdot g(x) \leq c_2 \cdot c_4 \cdot h(x)$$

per cui si ottiene che

$$c_1 \cdot c_3 \cdot h(x) \leq f(x) \leq c_2 \cdot c_4 \cdot h(x)$$

Esempio: Si consideri la classe di equivalenza

$$\Theta(1)$$

Allora ad essa vi apparterranno tutte le costanti positive, così come le funzioni oscillanti che assumono valori positivi, come $f(x) = 2 + \sin(x)$, in quanto

$$2 \cdot 1 \leq f(x) \leq 3 \cdot 1, \quad \text{con } g(x) = 1$$

Osservazione: Si osservi che se due funzioni hanno lo stesso ordine di infinito, allora appartengono alla stessa classe di equivalenza. Tuttavia non è vero il contrario.
Si considerino, infatti, due funzioni $f(x)$ e $g(x)$ tali che

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} = \alpha \in \mathbb{R}^+ - \{0\}$$

quindi definitivamente, per $x \geq \overline{x}$, si ha che

$$\alpha - \epsilon \leq \frac{f(x)}{g(x)} \leq \alpha + \epsilon$$

dal momento che dalla definizione di limite sia ha $\forall \epsilon > 0$, è possibile anche considerare $\epsilon = \frac{\alpha}{2}$, da cui

$$\frac{\alpha}{2} \leq \frac{f(x)}{g(x)} \leq \frac{3}{2}\alpha$$

ma allora, moltiplicando ambo i membri per $g(x)$, essendo per definizione necessariamente definitivamente positiva, la disuguaglianza si conserva e diviene

$$\frac{\alpha}{2} \cdot g(x) \leq f(x) \leq \frac{3}{2} \cdot \alpha \cdot g(x)$$

che corrisponde esattamente alla definizione di due funzioni che appartengono alla stessa classe di equivalenza, con

$$c_1 = \frac{\alpha}{2} \text{ e } c_2 = \frac{3}{2}\alpha$$

Per cui si è dimostrato che avere lo stesso ordine di infinito significa anche appartenere alla stessa classe di equivalenza, ma non è vero il contrario.

11 Marzo 2022

4.3 Funzioni di complessità

Si consideri una funzione di complessità temporale $f(n)$ definita in funzione n della lunghezza dell'input: è molto articolato, nonché scarsamente utile, provvedere al calcolo di valori precisi di una funzione di complessità temporale; presumibilmente, però, al crescere di n, ossia al crescere della lunghezza dell'input, aumenta anche il valore della funzione stessa.
Le funzioni $f(n)$ di complessità temporale oggetto di studio, comunque, sono molto generali e, comunemente, sono **definitivamente positive** e, generalmente, sono infinite per $x \to +\infty$ e vengono studiate per $x \geq 0$.
Com'é noto, è stata definita la classe di complessità $\Theta(f(n))$ come l'insieme di tutte le funzioni che rassomigliano alla funzione $f(n)$. In questo caso, in particolare, affermare che

$$g(n) = \Theta(f(n))$$

significa affermare che $\exists c_1 > 0, c_2 > 0, \overline{n}$ tali che

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n), \quad \forall n \geq \overline{n}$$

in cui $\Theta(f(n))$ è, a tutti gli effetti, una **classe di equivalenza**, in quanto gode delle proprietà di **riflessività**, **simmetria** e **transitività**.
Se si considera una lunghezza $l = 5$ in base 10, una lunghezza effettiva senza zeri in testa, allora il numero n che si può descrivere con lunghezza $l = 5$ è compreso tra

$$10000 \leq n \leq 100000 - 1 \longrightarrow 10^{l-1} \leq n < 10^l$$

e passando ai logaritmi in base 10, ciò si traduce in

$$l_{10}(n) - 1 \leq \log_{10}(n) < l_{10}(n)$$

in cui $\log_{10}(n)$ viene considerata una **lunghezza continua**, decisamente più pratica nel suo utilizzo rispetto alla **lunghezza intera** $l_{10}(n)$, la quale è la lunghezza effettiva, ma inutilmente precisa. Naturalmente si può scrivere che

$$\frac{1}{2} \cdot l_{10}(n) \leq l_{10}(n) - 1 \leq \log_{10}(n) \leq l_{10}(n)$$

per cui si può osservare come $l_{10}(n)$ e $\log_{10}(n)$ appartengono alla stessa classe di equivalenza $\Theta(l_{10}(n))$ in quanto sono state usate le costanti $c_1 = \frac{1}{2}$ e $c_2 = 1$. Dal punto di vista della notazione Θ, quindi, tali lunghezze sono perfettamente equivalenti.
Non solo, ma se si considerano le quantità $\log_{10}(n)$ e $\log_2(n)$, è noto che

$$\log_2(n) = \frac{\log_{10}(n)}{\log_{10}(2)}$$

in cui

$$\frac{1}{\log_{10}(n)}$$

è una costante certamente positiva, allora si evince come tali quantità siano perfettamente equivalenti secondo la notazione Θ.
Quando si impiega la notazione Θ, infatti, una funzione $f(n)$ e $\alpha \cdot f(n), \alpha \geq 0$ appartengono alla stessa classe Θ ed è per questo che normalmente viene omessa la base dei logaritmi, che comunque deve essere maggiore di 1, la quale, di fatto, è ininfluente.
Inoltre si è osservato come due funzioni che hanno lo stesso ordine di infinito appartegono anche alla stessa classe Θ, semplicemente applicando la definizioen di limite. Pertanto, le 3 funzioni

$$3n^2 + 1 \qquad \frac{n^2}{1000} \qquad 8945n^2 - n$$

appartengono alla stessa classe $\Theta(n^2)$, in quanto presentano lo stesso ordine di infinito, ovvero il limite del loro rapporto è una costante positiva.

Si consideri una funzione $f(n)$ infinita per $x \to +\infty$ e la funzione $g(n) = f(n) \cdot [\sin(n) + 2]$ anch'essa infinita per $x \to +\infty$ che appartengono alla stessa classe Θ; tuttavia, si ha che

$$\lim_{n \to +\infty} \frac{g(n)}{f(n)} = \lim_{n \to +\infty} \sin(n) + 2 = \nexists$$

per cui due funzioni che appartengono alla stessa classe di equivalenza Θ non è detto che abbiano lo stesso ordine di infinito.

FUNZIONE O-GRANDE DI UN'ALTRA

Si considerino due funzioni $f(x)$ e $g(x)$ **definitivamente positive**; allora indicare

$$f(n) = O(g(n))$$

significa affermare che definitivamente si ha che

$$f(n) \leq g(n)$$

ovvero che

$$\exists \overline{n}, c \geq 0 : f(n) \leq c \cdot g(n), \quad \forall n \geq \overline{n}$$

per cui se $f(n)$ e $g(n)$ appartengono alla stessa classe di equivalenza Θ, allora $f(n)$ è O-grande di $g(n)$ e $g(n)$ è O-grande $f(n)$ e viceversa: se $f(n)$ è O-grande di $g(n)$ e $g(n)$ è O-grande $f(n)$ allora $f(n)$ e $g(n)$ appartengono alla stessa classe di equivalenza Θ.

Esempio: Si osservi che, ovviamente

$$2n^2 = O(n^2) \quad \text{e} \quad \frac{n}{17} = O(n^2)$$

o ancora, che quando

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$$

significa affermare che $g(n) = O(f(n))$.

Ossevazione: Si osservi, per quanto si è detto, che

$$\begin{cases} f(n) = O(g(n)) \\ g(n) = O(f(n)) \end{cases}$$

se e solo se appartengono alla stessa classe Θ. Analogamente si ha che

$$\begin{cases} f(n) = O(g(n)) \\ g(n) = O(h(n)) \end{cases}$$

allora si ha che $f(n) = O(h(n))$.

FUNZIONE Ω-GRANDE DI UN'ALTRA

Si considerino due funzioni $f(x)$ e $g(x)$ **definitivamente positive**; allora indicare

$$f(n) = \Omega(g(n))$$

significa affermare che definitivamente si ha che

$$f(n) \geq g(n)$$

ovvero che

$$\exists \overline{n}, c \geq 0 : f(n) \geq c \cdot g(n), \quad \forall n \geq \overline{n}$$

per cui se $f(n)$ e $g(n)$ appartengono alla stessa classe di equivalenza Θ, allora $f(n)$ è Ω-grande di $g(n)$ e $g(n)$ è Ω-grande $f(n)$ e viceversa: se $f(n)$ è Ω-grande di $g(n)$ e $g(n)$ è Ω-grande $f(n)$ allora $f(n)$ e $g(n)$ appartengono alla stessa classe di equivalenza Θ.

Esempio: Si osservi che indicare

$$f(n) = \Omega(g(n))$$

se e solo se

$$g(n) = O(f(n))$$

Analogamente si ha che

$$\begin{cases} f(n) = O(g(n)) \\ f(n) = \Omega(g(n)) \end{cases}$$

se e solo se appartengono alla stessa classe Θ.

4.4 Classi di complessità

Le tre classi Θ che si incontreranno maggiormente saranno

1. La classe **lineare** $\Theta(n)$;
2. La classe **log-lineare** $\Theta(n \cdot \log(n))$;
3. La classe **quadratica** $\Theta(n^2)$;

in cui si ha che

$$n = O(n \cdot \log(n)) \quad \text{e} \quad n \cdot \log(n) = O(n^2)$$

ovvero la complessità log-lineare è comunque poco più complessa rispetto a quella lineare, in quanto la crescita all'infinito è molto lenta.

Osservazione: Si osservi che la funzione

$$10^{10^{80}} n = \Theta(n)$$

anche se il suo coefficiente è estremamente elevato. Analogamentte, se si ha una funzione

$$2^{0,00000\ldots1} = \Theta(2^n)$$

nonostante la sua crescita è estremamente lenta rispetto ad una funzione esponenziale normale. Tuttavia, tale risultato non costituisce un'anomalia nella notazione Θ, in quanto tali tipologie di funzioni sono impossibili tra trovare nella vita reale, secondo una valenza empirica.

Si consideri il listato seguente:

Algorithm 8 Esempio listato 1

```
for i = 1 to k
    for j = 1 to k
        a_{i,j} = 0
    next i
next j
```

Naturalmente il numero di iterazioni di tale algoritmo è $k \cdot k = k^2$, così come il numero delle assegnazioni. Ma se il listato fosse stato quello proposto di seguito:

Algorithm 9 Esempio listato 2

```
s = 0
for i = 1 to k
    for j = 1 to k
        s = s + a_{i,j}
        a_{i,j} = 0
    next i
next j
s = s * s
```

allora il numero della assegnazioni è significativamente aumentato, passando da k^2 a $2 + 2k^2$. Tuttavia, la notazione Θ fa sì che in ogni caso la complessità sia rimasta pari a k^2, in quanto il numero di iterazioni, e quindi l'ordine, è **inalterato**.

Osservazione: Si osservi che nel caso dell'algoritmo **insertion-sort**, la complessità nel caso migliore è pari a $\Theta(n)$, in quanto il numero delle iterazioni è pari a $n-1$, mentre nel caso peggiore la complessità è $\Theta(n^2)$, in quanto le iterazioni sono $\frac{n \cdot (n-1)}{2}$. Per quanto concerne la complessità tipica, ovverosia la complessità media, dal punto di vista empirico si osserva che essa è dell'ordine $\Theta(n^2)$.
Per quanto concerne l'algoritmo di Euclide, il numero di iterazioni, nel caso peggiore era di

$$1 + 2 \cdot \log_2(\min(m, n))$$

per cui si ha che la complessità effettiva sia $O(\log(n))$, ovvero possibilmente inferiore ad una complessità logaritmica.

4.5 Merge-Sort

L'algoritmo **merge-sort** è un algoritmo la cui idea alla base è molto più complessa dell'algoritmo **insertion-sort**.
L'input, come di consueto, è dato da un **array** A di lunghezza $length(A) = n$, per cui si dovranno ordinare n numeri. Inoltre, due altri input sono p e r, i quali costituiscono delle posizioni dell'array (anche intermedie o coincidenti), tali che $1 \leq p \leq r \leq n$.
Di seguito si espone il listato:

Algorithm 10 Merge-sort

```
MERGE-SORT(A, p, r)
if p < r
    then q = ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Dove

$$\left\lfloor \frac{p + r}{2} \right\rfloor$$

prende il nome di *parte intera inferiore* della semisomma di p e r, ovvero viene eliminata la parte dopo la virgola.
Inizialmente l'algoritmo opera partendo ponendo $p = 1$ e $r = n$, ma successivamente le chiamate ricorsive fanno sì che p e r divengano delle posizioni intermedie, designate con q.
La procedura MERGE non è ricorsiva, ma presenta due cicli for in grado di ordinare due blocchi di numeri già ordinati. Tale procedura, infatti, inzia ad operare sue due insiemi di valori che sono già stati ordinati separatamente, con lo scopo di ricavare un unico insieme di valori ordinati, sfruttando l'ordinamento dell'insieme dei valori su cui si sta operando; per farlo si confrontano tutti i valori in testa agli insiemi ordinati, prendendo sempre il più piccolo tra i due che vengono confrontati, fino ad arrivare all'ultimo valore (necessariamente molto elevato, appositamente inserito come "finecorsa") che viene appositamente posto alla fine in modo da segnalare il termine dell'insieme di valori; in particolare si ha che $p < q < r$, in cui i valori che stanno tra $[p - q]$ e $[q - r]$ sono già stati ordinati.
La procedura viene esposta di seguito:

Algorithm 11 Merge

1: MERGE(A, p, q, r)
2: $n_1 = q - p + 1$
3: $n_2 = r - q$
4: Create two new arrays L and R
5: **for** $i = 1$ **to** n_1
6:     **do** $L[i] = A[p + i - 1]$
7:         $R[j] = A[q + j]$
8: ...
9: $L[n_1 + 1] = +\infty, R[n_2 + 1] = +\infty$
10: $i = 1, j = 1$
11: **for** $k = p$ **to** r
12:     **do if** $L[i] \leq R[j]$
13:         **then** $A[k] = L[i]$
14:             $i = i + 1$
15:         **else** $A[k] = R[j]$
16:             $j = j + 1$

In cui le iterazioni coinvolte, naturalmente, fino alla riga 10 sono

$$n_1 + n_2 = q - p + 1 + r - q = r - p + 1$$

ovverosia il numero delle posizioni che vanno da p a r.
Mentre dalla riga 10 in poi il numero di iterazioni è, ovviamente, ancora una volta $r - p + 1$; pertanto il numero di iterazioni complessive è pari al doppio di $r - p + 1$, quindi la complessità è dell'ordine

$$\Theta(r - p + 1)$$

Esempio: Si consideri il seguente array

$$\boxed{4}\ \boxed{5}\ \boxed{3}\ \boxed{1}\ \boxed{7}\ \boxed{2}\ \boxed{0}\ \boxed{9}$$

in cui $n = 8$. Per l'ordinamento di tale array, si divide lo stesso in due parti e poi successivamente ogni parte in ancora due parti e nuovamente in due parti fino ad ottenere una divisione dell'ordinamento riducendosi all'unità, in cui i singoli valori saranno automaticamente ordinati, come mostrato di seguito:

$$\boxed{4}\ \boxed{5}\ \boxed{3}\ \boxed{1}\ \mid\ \boxed{7}\ \boxed{2}\ \boxed{0}\ \boxed{9}$$

e poi

$$\boxed{4}\ \boxed{5}\ \mid\ \boxed{3}\ \boxed{1}\ \mid\ \boxed{7}\ \boxed{2}\ \mid\ \boxed{0}\ \boxed{9}$$

ed infine

$$\boxed{4}\ \mid\ \boxed{5}\ \mid\ \boxed{3}\ \mid\ \boxed{1}\ \mid\ \boxed{7}\ \mid\ \boxed{2}\ \mid\ \boxed{0}\ \mid\ \boxed{9}$$

Dal momento che i blocchi da 1 sono già ordinati si procede a ad utilizzare la procedura *Merge* per ordinare i blocchi da 2, ottenenendo

$$\boxed{4}\ \boxed{5}\ |\ \boxed{1}\ \boxed{3}\ |\ \boxed{2}\ \boxed{7}\ |\ \boxed{0}\ \boxed{9}$$

e poi procedendo con i blocchi da 4, applicando la procedura *Merge* si ottiene

$$\boxed{1}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ |\ \boxed{0}\ \boxed{2}\ \boxed{7}\ \boxed{9}$$

e infine si ordina il blocco completo, applicando la procedura *Merge* si ottiene

$$\boxed{0}\ \boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{7}\ \boxed{9}$$

Questa, naturalmente, è una **esecuzione parallela**, mentre quella dell'algoritmo è **lineare**, in quanto va ad analizzare progressiamente ogni ramo di un albero binario, risalendolo progresivamente.

Osservazione: Tale algoritmo è **profondamente rigido**, in quanto il numero delle iterazioni che si devono eseguire è sempre lo stesso, indipendente dal caso migliore, peggiore e generale.
Questa non è una buona caratteristica, la quale rende ragionevole l'utilizzo di tale algoritmo quando i numeri sono inzialmente fortemente disordinati; se i numeri iniziali sono parzialmente ordinati è più conveniente impiegare l'algoritmo di insertion-sort.
La complessità di tale algoritmo, tuttavia, nel caso peggiore è **log-lineare** ed è estremamente ottima in quanto un algortimo di ordinamento generale, come il merg-sort (che non richiede una specifica struttura dei dati in ingresso) non è in grado di battere una complessità log-lineare.
Il principio di costruzione di tale algoritmo, nonché la tecnica impiegata per la risoluzione di tale problema, prende il nome di *Divide et impera* (o *Divide and conquer*).

16 Marzo 2022

Una componente fondamentale dello studio algoritmico è la **relazione di ricorrenza lineare**, alla base degli algoritmi che si fondano sulla **ricorsività**.
È noto che la sezione aurea si basa sulla suddivisione di un segmento in maniera armoniosa:

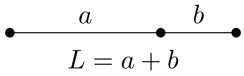

Figura 6: Sezione aurea

Rispettando la proporzione seguente

$$a + b : a = a : b$$

e ponendo $b = 1$ si ottiene che

$$a + 1 : a = a : 1$$

che dà vita all'equazione di secondo grado seguente (e ricordando che il prodotto dei medi è uguale al prodotto degli estremi):

$$a^2 - a - 1 = 0$$

che produce due soluzioni

$$a_{1,2} = \frac{1 \pm \sqrt{5}}{2}$$

in cui la soluzione più importante è quella naturalmente positiva, ovvero:

$$\phi = \frac{1 + \sqrt{5}}{2} \cong 1.62 \quad \text{e} \quad \psi = \frac{1 - \sqrt{5}}{2}$$

Esempio: Alla base dello studio della ricorsività vi é il *Liber Abaci* di Fibonacci, il quale è anche l'ideatore della sequenza di Fibonacci e delle straordinarie proprietà ad essa collegate; all'interno del Liber Abaci si espone anche il problema della crescita demografica dei conigli, alla base del quale vi è proprio una relazione di ricorrenza.
Per la visualizzazione del problema si parte dalla condizione iniziale che riguarda la popolazione dei conigli seguente:

$$F_1 = F_2 = 1$$

e si espone anche una legge che permette di calcolare l'evoluzione demografica dei conigli nel tempo, ossia la cosiddetta legge di Fibonacci, ovvero una **relazione di ricorrenza lineare**:

$$\boxed{F_n = F_{n-2} + F_{n-1}}$$

Naturalmente, nella risoluzione del problema, ciò che era maggiormente importante era determinare una formula chiusa, la quale è la seguente:

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$$

ma siccome $\psi < 0$, implode a 0, per cui con ottima approssimazione si può affermare che

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}} \cong \frac{\phi^n}{\sqrt{5}}$$

che sottolinea il carattere esponenziale della crescita dei conigli; tale risultato, inoltre, permette di ottenere anche l'evidenza seguente:

$$\lim_{n \to +\infty} \frac{F_{n+1}}{F_n} = \phi$$

Pertanto la relazione di ricorrenza lineare è accompagnata da una formula chiusa che, in questo caso, è una formula esatta (un risultato impressionante visto che, partendo da una relazione di ricorrenza, ottenere una formula ciusa non è banale).
In generale, infatti, un approccio sistematico ad una relazione di ricorrenza lineare permette di ottenere una formula chiusa, la quale, però, è tutt'altro che precisa, in quanto fornita in termini della notazione O-grande.

4.6 Master Theorem

Il **teorema maestro**, o il **teorema principale delle relazioni di ricorrenza**, si può applicare solamente in casi ben determinati (e particolarmente fortunati). In ogni caso, tale teorema inizia ad operare partendo dall'**espressione della condizione iniziale**

$$T(1) = \Theta(1)$$

ovvero $T(1)$ appartiene alla classe di equivalenza $\Theta(1)$, la quale è una definizione decisamente vaga (esistono due costanti positive tra cui $T(1)$ è compreso, ma null'altro), per cui non vale la pena di scriverla, giacché non interviene in nessuna maniera nella soluzione (benché sia fondamentale che vi sia, per decretare l'applicazione del teorema o meno).
La relazione, invece, alla base del teorema è la seguente

$$\boxed{T(n) = a_1 \cdot T\left(\frac{n}{\lfloor b \rfloor}\right) + a_2 \cdot T\left(\frac{n}{\lceil b \rceil}\right) + f(n)}$$

in cui $f(n)$ è una funzione nota. Tuttavia, la parte intera inferiore o superiore è ininfluente, per cui, posto $a = a_1 + a_2$, si preferisce la formula abbreviata

$$\boxed{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)}$$

imponendo che $a \geq 1$ e $b > 1$ (altrimenti non è possibile applicare il teorema considerato, cercando di ricorrere ad altri metodi).
Supponendo che le condizioni di cui sopra siano verificati, è possibile applicare il teorema: l'applicazione del teorema maestro prevede 3 differenti casi, a seconda della **natura della funzione $\boldsymbol{f(n)}$**:

- la funzione $f(n)$ è molto piccola
- la funzione $f(n)$ è giusta
- la funzione $f(n)$ è molto grande

Naturalmente, per confrontare tale funzione $f(n)$ se ne definisca una seconda, chiamata **funzione di confronto**:

$$\phi(n) = n^{\log_b(a)}$$

che assicura che la funzione sia crescente grazie alla base $b > 1$ del logaritmo. Pertanto si ha che il *master theorem* può essere applicato in questi tre differenti casi:

- La funzione $f(n)$ è **molto** più piccola della funzione $\phi(n)$, tuttavia **non è sufficiente** scrivere

  $$f(n) = O(\phi(n))$$

  per cui di fatto, rimane una sorta di "buco" in cui il teorema non si può applicare, ossia quando la funzione $f(n)$ è troppo piccola per rientrare nella seconda casistica, ma troppo grande per entrare nel primo caso.

- La funzione $f(n)$ rassomiglia alla funzione $\phi(n)$, ovvero

  $$f(n) = \Theta(\phi(n))$$

- La funzione $f(n)$ è **molto** più grande della funzione $\phi(n)$, che è una condizione che in algoritmica serve a poco, in quanto non basta affermare che

  $$f(n) = \Omega(\phi(n))$$

  per cui, ancora una volta, rimane un buco, determinato da funzioni che sono troppo grandi per rientrare nella seconda casistica, ma troppo piccole per per entrare nel terzo caso, per cui il teorema non si può impiegare.

Tuttavia, alla terza e ultima casistica si deve aggiungere anche un'ulteriore condizione che la funzione $f(n)$ deve soddisfare affinché si possa applicare il teorema; il problema, però, che si viene a determinare, è che la condizione in questione non è stobile rispetto alla classe di equivalenza Θ di appartenenza della funzione $f(n)$: è possibile che due funzioni $f(n)$ e $g(n)$ appartenenti alla stessa classe di equivalenza possano l'una soddisfare la condizione, la l'altra no, permettendo nel primo caso di applicare il teorema, nel secondo di non applicarlo.
Dal momento che in algoritmica la funzione $f(n)$ non si conosce bene giacché è nota solamente la classe Θ a cui appartiene, non è sempre possibile sapere se si può applicare il teorema o meno nel terzo caso.
Pertanto in algoritmica i casi più importanti sono il primo e il secondo; tuttavia, il caso che viene trattato nel dettaglio, però, è solo il secondo. Nel secondo caso è noto che $a \geq 1$, $b > 1$ e le due funzioni $f(n)$ e la funzione di confronto $\phi(n)$ appartengono alla stessa classe; da notare che vi è **transitività**, per cui il fatto di non conoscere $f(n)$ diviene irrilevante quando è nota la classe di equivalenza Θ di appartenenza della funzione $f(n)$: se $f(n)$ è equivalente a $\phi(n)$, qualunque funzione equivalente a $f(n)$ è equivalente a $\phi(n)$ per transitività.
Tramite procedimenti matematici omessi, si giunge alla soluzione matematica seguente

$$\boxed{T(n) = \Theta\left(f(n) \cdot \log(n)\right)}$$

in cui la base del logaritmo è ininfluente dal punto di vista della notazione Θ.

Osservazione: Nell'algoritmo **merge-sort**, la procedura **merge** permette di ordinare dei valori partendo da due insiemi di valori già ordinati, ottenendo un ultimo insieme perfettamente ordinato. La procedura *merge* non è una procedura ricorsiva, mentre l'algoritmo *merge-sort* si basa sul meccanismo della ricorsività, andando a scorrere un albero binario da sinistra verso destra, risalendolo progressivammente (con una esecuzione lineare e sequenziale, per nulla parallela).
L'algoritmo è assolutamente corretto, dal punto di vista esecutivo, è ciò è evidente, anche solo dal punto di vista empirico; tuttavia, ciò che non appare evidente è la complessità dell'algoritmo considerato.
Si ripropone di seguito, a tal proposito, il listato dell'algoritmo **merge-sort**:

Algorithm 12 Merge-sort

MERGE-SORT(A, p, r)
if $p < r$
    then $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
        MERGE-SORT(A, p, q)
        MERGE-SORT$(A, q+1, r)$
        MERGE(A, p, q, r)

Lo sforzo per ordinare n numeri, naturalmente, è dato dalla seguente semplice relazione

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

ovvero la somma dello sforzo per ordinare ciascuna metà più l'operazione di *merge* delle due metà, denotata dalla funzione $f(n)$, che, in forma più compatta, può essere scritta come segue

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

ed applicando tale formula a qualunque passo si ottiene, per esempio nel caso di $\left(\frac{n}{2}\right)$

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{4}\right) + f\left(\frac{n}{2}\right)$$

mentre nel caso di $\left(\frac{n}{4}\right)$ si ha

$$T\left(\frac{n}{4}\right) = 2 \cdot T\left(\frac{n}{8}\right) + f\left(\frac{n}{4}\right)$$

e via dicendo, fintantoché non si arriva a 1. Inoltre, per quanto concerne la procedura *merge*, partendo da un array iniziale A, si creavano ulteriori due array, uno di destra R, di dimensione

n_1, e uno di sinistra L, di dimensione n_2, e per farlo si necessitava di $n = n_1 + n_2$ iterazioni (ovviamente pari alla somma del numero di valori che compongono il primo e il secondo array di lavoro). Per l'ordinamento di tutti i valori, ancora una volta, si necessitava di $n = n_1 + n_2$ (sempre pari al numero totale di valori da confrontare) iterazioni, per un totale di $2n$ iterazioni: pertanto la complessità della procedura *merge* è dell'ordine $\Theta(n)$ (in quanto il coefficiente 2 è totalmente ininfluente per la notazione Θ).
Pertanto, il problema da risolvere tramite il *teorema maestro* è il seguente

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

in cui, evidente, se confrontata con la formula generale del *teorema maestro*, si evince che $a = b = 2$; inoltre si osservi che impiegare $\Theta(n)$ in luogo di $f(n)$ è una scorrettezza, giustificata dal fatto che non é nota la funzione precisa appartenente alla classe $\Theta(n)$. Per l'applicazione del *teorema maestro* è necessario considerare una funzione di confrono, come quella precedentemente introdotta

$$\phi = n^{\log_b(a)}$$

ma essendo $a = b = 2$ si ottiene che

$$\phi = n^{\log_2(2)} = n^1 = n$$

Ecco che allora la funzine $f(n)$ che non è nota, ma appartiene alla classe $\Theta(n)$ rassomiglia alla funzione di confronto; pertanto si può applicare il secondo caso del *teorema maestro* e anche la soluzione ad esso associata, ovvero

$$\boxed{T(n) = \Theta\left(f(n) \cdot \log(n)\right)}$$

che nel caso qui considerato, essendo $f(n) = \Theta(n)$ si traduce in

$$T(n) = \Theta(n \cdot \log(n))$$

ovvero la **complessità di merge-sort** è log-lineare, la quale è una complessità generale e **profondamente rigida**, in quanto le operazioni che vengono svolte sono sempre le stesso; pertanto tale complessità si mantiene costante sia nel caso migliore, sia nel caso peggiore, sia un quello medio.
Una complessità log-lineare è leggermente peggiore rispetto ad una complessità lineare, ma è comunque una complessità ottima: infatti, esiste un teorema che afferma che, asintoticamente, un **algoritmo di ordinamento generale** (ovvero un algoritmo che si propone di ordinare dei dati indipendemente da come essi vengono strutturati) è destinato ad avere una complessità nel caso peggiore **almeno log-lineare** e ad avere una complessità nel caso medio **almeno log-lineare**, che sono esattamente le prestazioni che merge-sort garantisce.
Particolarmente infruttuosa è la complessità di merge-sort nel caso migliore, la quale è sempre log-lineare, mentre nel caso dell'algoritmo insertion-sort è lineare. Questo permette di affermare che l'algoritmo merge-sort risulta essere particolarmente utile nel suo utilizzo quando i valori da ordinare non sono già ordinati, ma totalmente in disordine; nel caso fossero parzialmente ordinati, sarebbe più conveniente impiegare insertion-sort.

4.7 Heap-sort

La struttura di grafo più comunque è quella di **albero**; in un grafo semplice è sempre possibile costruire un percorso chiuso, in cui partendo da un primo vertice, utilizzando almeno un arco, solamente archi in un solo senso (ovvero un arco serve per andare solamente da a a b e non viceversa) e senza archi multipli (ovvero non è possibile che vi siano più archi che collegano a e b) si ritorna al vertice di partenza, come mostrato di seguito:

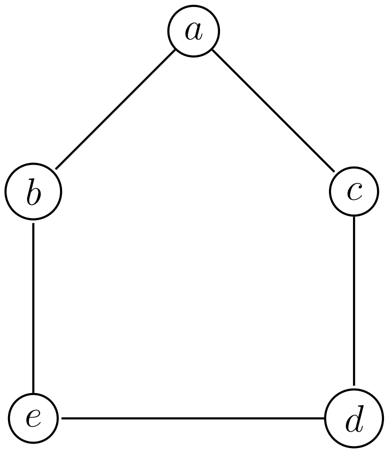

Figura 7: Esempio di grafo con percorso chiuso

La definizione di albero segue dall'assenza di percorso chiusi all'interno di un grafo semplice, come esposto nella definizione che segue:

ALBERO

Un grafo semplice **privo percorsi chiusi** prende il nome di **albero**.

Tuttavia, tale definizione è eccessivamente generica, in quanto molto spesso si preferisce operare con **alberi radicati** (rooted-tree), ovvero degli alberi in cui è stata opportunamente specificata la **radice**, ossia un vertice specifico.
Un esempio di **albero radicato**, con **radice** (root) il vertice a, che verrà considerato viene raffigurato nel seguito:

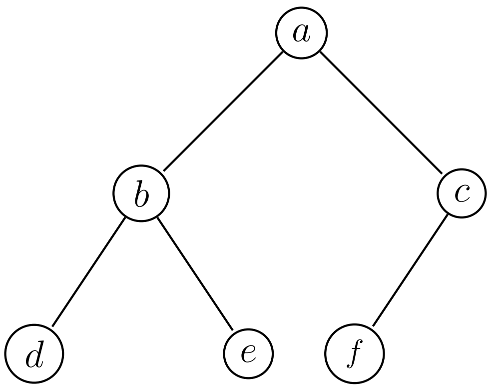

Figura 8: Esempio di albero radicato di radice a e profondità 2

Dove per **profondità** è da intendersi il livello più basso raggiunto dai vertici dell'albero. I vertici che non presentano figli prendono, invece, il nome di **foglie** (in questo albero le foglie sono i vertici d,e,f).
L'albero radicato che maggiormente verrà consideranto per la progettazione algoritmica è l'**albero binario**, in cui i figli di ogni vertice sono **al più 2**. Un albero binario ben strutturato prevede che vi siano n vertici e che l'albero sia **completo**, ovvero vengono saturati tutti i livelli di diversa profondità dell'albero prima di raggiungere al livello successivo, come mostrato di seguito:

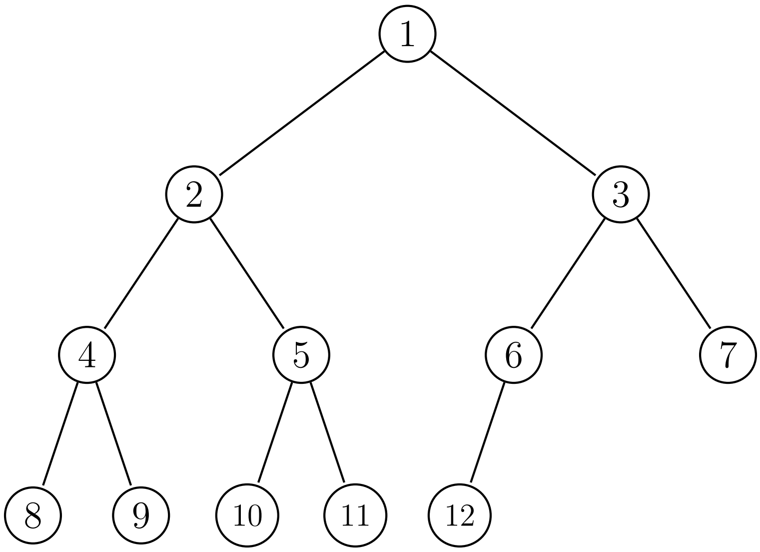

Figura 9: Esempio di albero binario completo

In cui la regola di completamento prevede di procedere da sinistra verso destra, completando ciascun livello prima di procedere in quello successivo, eccezion fatta per l'ultimo che, per ragioni di numero, non può essere completato. In un albero binario completo è facile capire la profondità dello stesso, semplicemente osservando che

$$\boxed{2^h \leq n < 2^{h+1}}$$

in cui h è la profondità dell'albero, mentre n è il numero di vertici dell'albero. Pertanto si ottiene che

$$\boxed{h \leq \log_2(n) < h+1}$$

ovvero ciò significa che h e $\log_2(n)$ appartengono alla stessa classe Θ.
Dire che un albero è **super completo** significa che anche il livello delle foglie è pieno, e quindi n deve essere una potenza di 2 meno uno, ovvero $n = 2^h - 1$, con h l'altezza dello stesso. Inoltre è facile osservare come il numero di nodi in un albero **super completo** sia

$$2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{h-1} + 2^h = n$$

in cui h è la profondità dell'albero.

Osservazione: Si osservi la progessione geometrica seguente

$$1 + q + q^2 + q^3 + ... + q^k = \sum_{i=0}^{k} q^i$$

alla quale può essere associata una formula chiusa, come mostrato di seguito

$$(1 + q + q^2 + q^3 + ... + q^k) \cdot (q-1) = (q + q^2 + ... + q^{k+1} - 1 - q - q^2 - ... - q^k) = (q^{k+1} - 1)$$

e dividendo ambo i membri per $(q-1)$ si ottiene

$$(1 + q + q^2 + q^3 + ... + q^k) \cdot (q-1) = (q^{k+1} - 1) \longrightarrow \sum_{i=0}^{k} q^i = \frac{q^{k+1} - 1}{q - 1}$$

In cui, nel caso in cui la ragione $q = 2$, si ottiene che

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

Per cui è facile osservare che in un albero binario super completo, di altezza $h = k+1$, la somma di tutti i vertici di ogni livello è pari a $2^h - 1$, come già dimostrato. Inoltre, nel livello delle foglie vi sarà sempre un numero di vertici, pari a 2^h, superiore a tutto il resto dell'albero, in cui il numero di vertici è $2^h - 1$, ovvero

$$\#\text{numero vertici} \cong 2 \cdot \#\text{numero foglie}$$

ovvero, dal punto di vista della notazione Θ, il numero di foglie è confondibile con il numero di vertici.

Osservazione: Il numero di archi in un albero di k vertici è esattamente pari a $k - 1$ (mentre nel caso di un grafo semplice, i nodi possono essere tutti isolati, con numero di archi pari a 0, oppure possono essere tutti connessi, con un numero di archi dell'ordine $\Theta(n^2)$), per cui dal punto di vista della notazione Θ il numero di vertici e il numero degli archi sono confondibili.

L'algoritmo **heap-sort** è un algoritmo di ordinamento per cataste, ove per catasta è da intendersi un albero binario completo, non necessariamente super completo, arricchito di ulteriori proprietà; tale proprietà all'interno della logica dell'**heap-sort**, è la designazione dei vertici, assegnando un valore numerico a ciascun vertice, facendo attenzione a far sì che il numero di ogni padre sia maggiore o uguale di ogni suo figlio (e di conseguenza, per ovvie ragioni di costruzione, di tutti quelli nei livelli sottostanti), ottenendo una **catasta** (heap), come mostrato di seguito:

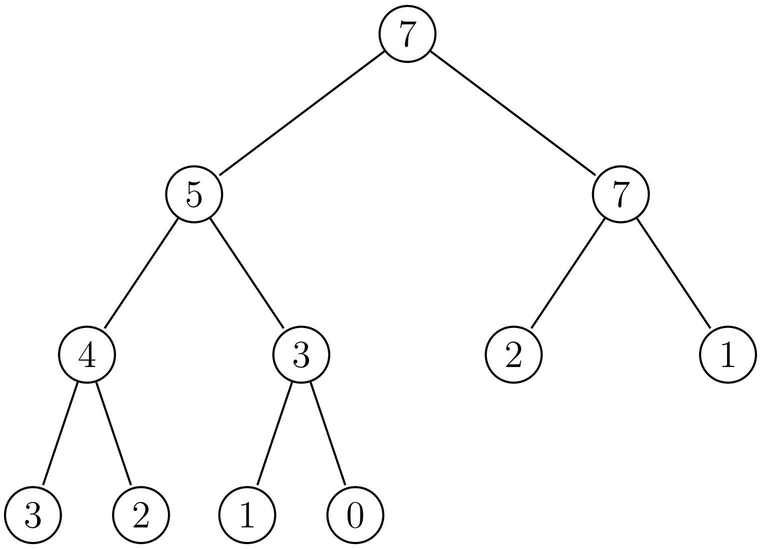

Figura 10: Esempio di catasta (heap)

All'interno della logica dell'**heap-sort** è possibile distinguere tra *maximum-heap* e *minimum-heap*: nel primo caso è il padre ad avere il valore numerico maggiore rispetto ai figli, nel secondo il genitore deve essere dominato da entrambi i figli.
Il primo algoritmo che si andrà a considerare prende il nome di **build-heap**, ovvero un algoritmo che è capace di creare una catasta che rispetta il vincolo del *maximum-heap* partendo da un albero in cui i nodi sono già pesati.

17 Marzo 2022

Si consideri un array, ovverosia una struttura dati assolutamente fondamentale in informatica, la quale presenta un numero di locazioni di memoria consecutive pari alla sua lunghezza $length(A) = n$.
Pertanto, scrivere

$$A[3]$$

signfica considerare il contenuto della 3 cella di memoria consecutiva memorizzata nell'array A; inoltre, si ha che lo sforzo per accedere ad una precisa locazione di memoria è dell'ordine $\Theta(1)$ (pari ad una sola operazione macchina), ovvero viene eseguito un **accesso diretto** (o **random access**, accesso casuale, o **accesso arbitrario**) alla cella di memoria richiesta, indipendemente dalla lunghezza dell'array.
Un array è, quindi, una struttura a random-access, ovvero ad accesso lineare, designato con una lettera maiuscola (come A) e di una lunghezza n. Tuttavia, la sua stuttura lineare si può interpretare come un albero binario (bidimensionale). Per esempio, l'array seguente

$$A = \underset{1}{\boxed{7}}\ \underset{2}{\boxed{8}}\ \underset{3}{\boxed{2}}\ \underset{4}{\boxed{9}}\ \underset{5}{\boxed{4}}\ \underset{6}{\boxed{1}}\ \underset{7}{\boxed{0}}\ \underset{8}{\boxed{5}}\ \underset{9}{\boxed{6}}$$

viene rappresentato tramite il seguente albero binario

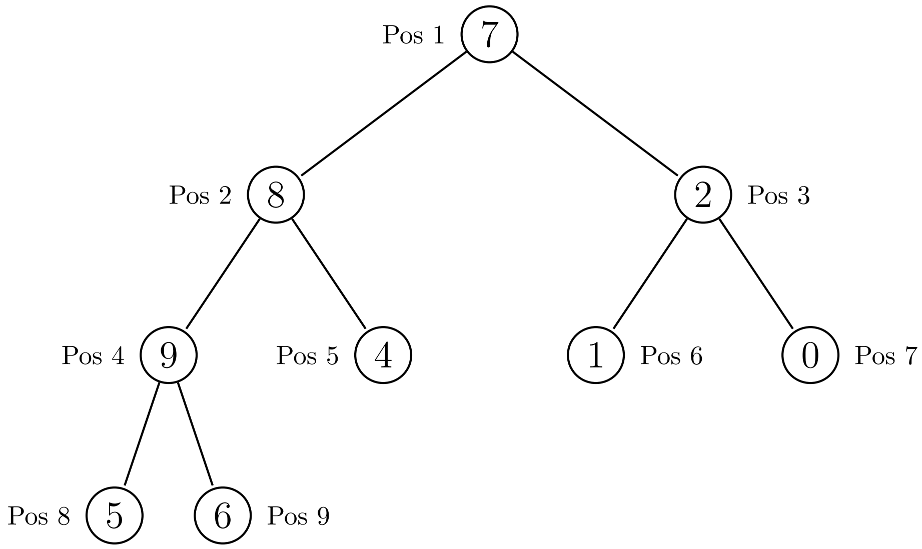

Figura 11: Esempio di catasta (heap)

La possibilità di considerare una struttura monodimensionale come un albero bidimensionale avviene tramite due algoritmi fondamentali (**heapify** e **buil-heap**), che adoperano le seguenti tre procedure:

Algorithm 13 Parent

PARENT(i)
  return $\left\lfloor \frac{i}{2} \right\rfloor$

Algorithm 14 Left

LEFT(i)
  return $2i$

Algorithm 15 Right

RIGHT(i)
  return $2i + 1$

In cui sono stati omessi degli opportuni controlli di validità dell'input i (come se $i = 1$ e si richiamasse la procedura Parent(1), la quale restituirebbe il valore 0 che è ovviamente corretto, o se non esiste un nodo sinistro/destro e si richiama la procedura Left/Right); questo in quanto si ha piena fiducia nell'utilizzo affidabile e consono di tali procedure; tramite esse, il calcolatore riesce ad interpretare una struttura monodimensionale come un albero bidimensionale.
La prima procedura che verrà impiegata nell'heap-sort (ordinamento per cataste) prende il nome di **heapify**, un sotto-algoritmo che verrà impiegato frequentemente dall'algoritmo **heap-sort**: una catasta, naturalmente, è una struttura ad albero in cui nei diversi nodi si trovano i numero che debbono essere ordinati (un ordinamento che poi si rifletterà nell'array di partenza).
Heapify opera su un sottoalbero dell'albero principale, la cui radice è uno dei nodi dell'albero primario individuato: ogni nodo, quindi, andrà ad individuare un sotto-albero (nel caso di una foglia, naturalmente il sottoalbero non esiste).
L'algoritmo **heapify**, tuttavia, viene utilizzato solamente quando il sottoalbero è noto a priori essere una catasta che rispetta il vincolo *maximum-heap*, eccezion fatta per la sottoradice del sottoalbero, in cui non è detto che venga rispettato tale vincolo.
Per comprendere il funzionamento di **heapify** si consideri il listato esposto di seguito, in cui si prende in considerazione il vertice di numero d'ordine i e il sottoalbero di A che presenta come sottoradice il nodo i:

Algorithm 16 Heapify

HEAPIFY(A, i)
l = LEFT(i), r = RIGHT(i)
if $l \leq$ heap-size$[A] \wedge A[l] > A[i]$
    then
        largest $= l$
    else
        largest $= i$
if $r \leq$ heap-size$[A] \wedge A[r] > A$[largest]
    then
        largest $= r$
if largest $\neq i$
    then
        Exchange: $A[i] \leftrightarrow A$[largest]
        HEAPIFY(A,largest)

in cui il termine "heap-size" fa riferimento alla dimensione, o altezza, del sottoalbero (ovvero al numero di nodi che costituiscono il sottoalbero).
La procedura esecutiva di heapify prevede dei controlli sequenziali: se esiste il figlio sinistro ($l \leq$ heap-size$[A]$) e si verifica che il valore del figlio sinistro è maggiore del nodo radice ($A[l] > A[i]$), ponendo il valore maggiore all'interno di una posizione di lavoro denominata *largest*.
Dopodiché si deve procedere alla valutazione anche del figlio destro, se esiste ($r \leq$ heap-size$[A]$), e si deve controllare se esso ha un valore maggiore sia del padre che del figlio sinistro ($A[r] > A[largest]$): se questo è il caso, allora all'interno di *largest* viene posta la posizione del figlio destro, ossia r.
Pertanto, ora, all'interno della posizione di lavoro *largest* vi è la posizione del padre, del figlio sinistro o del figlio destro, a seconda di chi abbia il valore massimo.
Naturalmente, ora, si procede con una modifica dell'albero solamente se all'interno di *largest* non vi è i, ossia la posizione del padre; se, infatti, vi fosse una posizione diversa da i significherebbe che uno dei due figli presenta un valore maggiore del padre e, quindi, il vincolo *maximum-heap* non sarebbe rispettato.
Quando si effettua uno scambio tra radice e figlio, ovviamente, è necessario ripetere la procedura **heapify** sul sottoalbero che presenta come sottoradice il figlio che aveva il valore maggiore, in quanto non è detto che taluno sia ancora una catasta.
Si consideri un sottoalbero come il seguente, in cui è possibile impiegare **heapify**, ovvero in cui la radice non è detto che domini i figli, ma il resto dell'albero rispetta il vincolo, come mostrato di seguito:

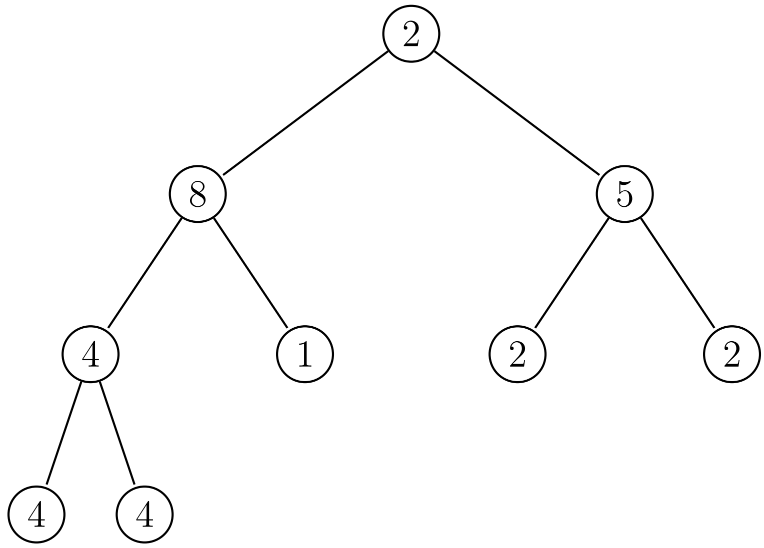

Figura 12: Esempio di sottoalbero al quale applicare heapify

Supponendo che il nodo radice di tale sottoalbero abbia il numero d'ordine $i = 3$, applicando l'algoritmo **heapify**, si considerano tre posizioni

$$\underset{l}{\boxed{6}}\ \underset{r}{\boxed{7}}\ \underset{\text{largest}}{\boxed{6 \text{ o } 3 \text{ o } 7}}$$

Naturalmente $l = 6$ in quanto la procedura LEFT(i) restituisce $2i$, mentre $r = 7$ in quanto la procedura RIGHT(i) restituisce $2i + 1$.
Operando secondo la procedura heapify, se il figlio sinistro esiste e il suo valore è maggiore di quello della sua radice, allora all'interno di *largest* andrà posto il valore $l = 6$, altrimenti il valore $i = 3$; Analogamente, se il valore del figlio di destra è maggiore di quello di sinistra e della radice, allora in *largest* andrà inserito il valore $r = 7$.
Nella posizione finale "largest", pertanto, vi può essere la posizione del padre, del figlio sinistro o del figlio destro; se nella posizione "largest" vi è un valore più grande della radice, avviene l'**exchange**, come in questo caso, ottenendo

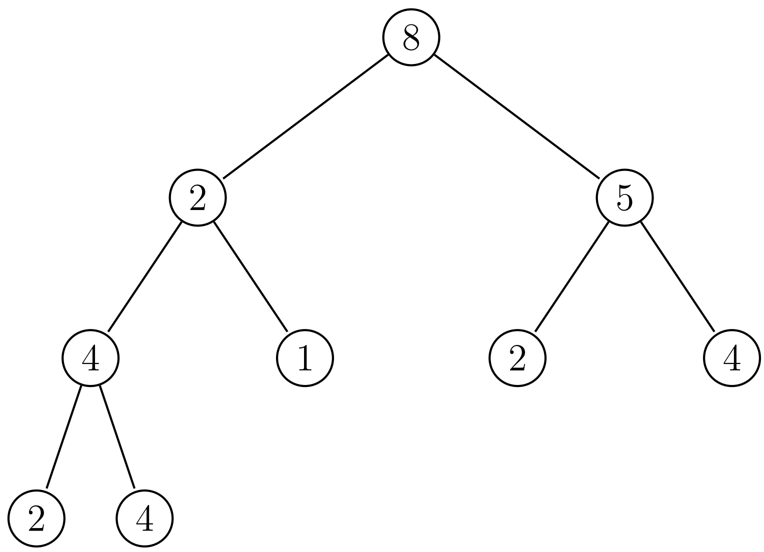

Figura 13: Esempio di sottoalbero al quale applicare heapify dopo un primo passo

Avendo operato tale sostituzione sul figlio di sinistra, è opportuno verificare il rispetto del vincolo del sottoalbero di sinistra (mentre il sottoalbero di destra non presenta anomalie, in quanto se prima rispettava il vincolo, lo rispetta anche ora). Si applica, quindi, ancora una volta heapify considerando come sottoradice il nodo avente numero d'ordine 6:

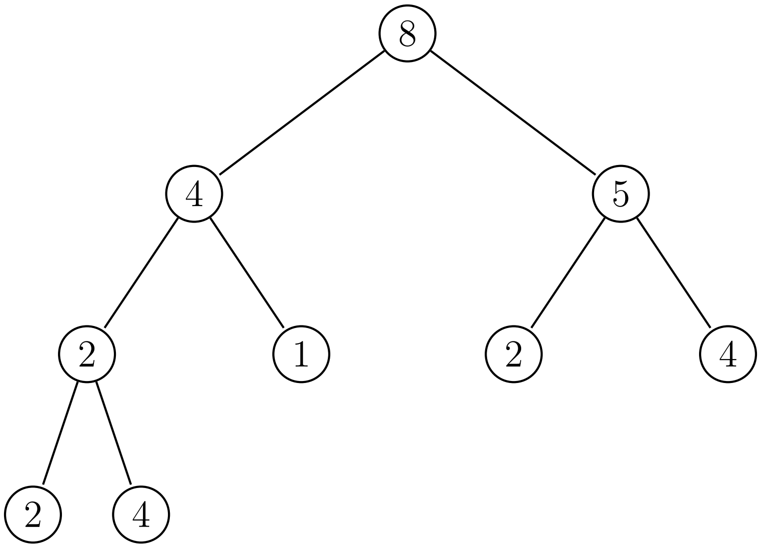

Figura 14: Esempio di sottoalbero al quale applicare heapify dopo un secondo passo

Ancora una volta, avendo scambiato la radice con uno dei figli, è necessario applicare heapify, questa volta considerando il sottoalbero avente sottoradice il nodo con numero d'ordine 12:

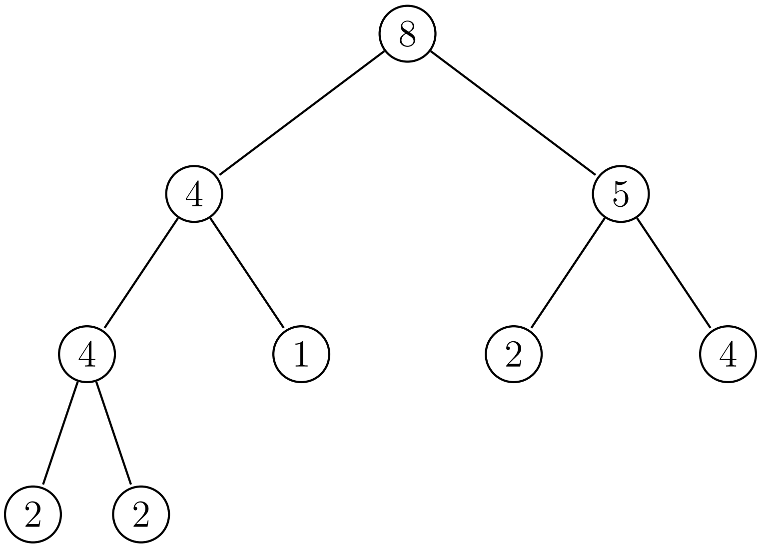

Figura 15: Esempio di sottoalbero al quale applicare heapify dopo un secondo passo

Avendo sostituito la radice con il figlio destro, è necessario applicare heapify ancora una volta, sul sottoalbero di destra; tuttavia, in questo caso, heapify non esegue alcuna operazione, giacché il sottoalbero di destra non esiste.
Ecco che l'albero inziale, ora, rispetta il vincolo di *maximum heap*.

Osservazione: Si osservi che non sono presenti cicli in tale algoritmo, in quanto la complessità è data dalla ricorsività; in particolare, il numero delle iterazioni è

$$\#\text{iterazioni} \leq h_i + 1$$

in cui h_i è l'altezza del sottoalbero (infatti, nel caso peggiore, si deve scandagliare un albero per ogni livello, più una chiamata finale per constatare che non vi è un sottoalbero del nodo figlio);

pertanto la complessità è, operando una forte maggiorazione, dell'ordine $O(\log(n))$, in cui n è il numero dei nodi dell'albero intero (avendo interpretato il sottoalbero come l'albero intero); infatti è noto che la profondità di un albero completo avente n nodi è circa $\log(n)$, per cui si ha che h e $\log(n)$ appartengono alla stessa classe Θ.
Si osservi che, nel caso in cui si consideri l'albero intero di partenza, con H la sua altezza complessiva, allora si ha che

$$\#\text{iterazioni} \leq H + 1$$

Tuttavia, gli algoritmi più importanti da dover considerare sono altri due: **heap-sort** e **build-heap**, l'ultimo del quale serve proprio per costruire una catasta partendo da un albero non catastizzato, come mostrato di seguito:

Algorithm 17 Build-Heap

```
BUILD-HEAP(A)
heap-size= length[A]
for i = ⌊n/2⌋ down to 1
    do HEAPIFY(A,i)
```

In cui

$$i = \left\lfloor \frac{n}{2} \right\rfloor$$

è il primo nodo con figli (in quanto è noto che il livello delle foglie, in un albero binario super completo, contiene più nodi di tutto il resto dell'albero) e che, quindi, potrebbe non rispettare il vingolo di *maximum-heap* (i nodi succcessivi, naturalmente, non possono avere di questo problema, visto che non presentano figli). Si consideri il caso seguente:

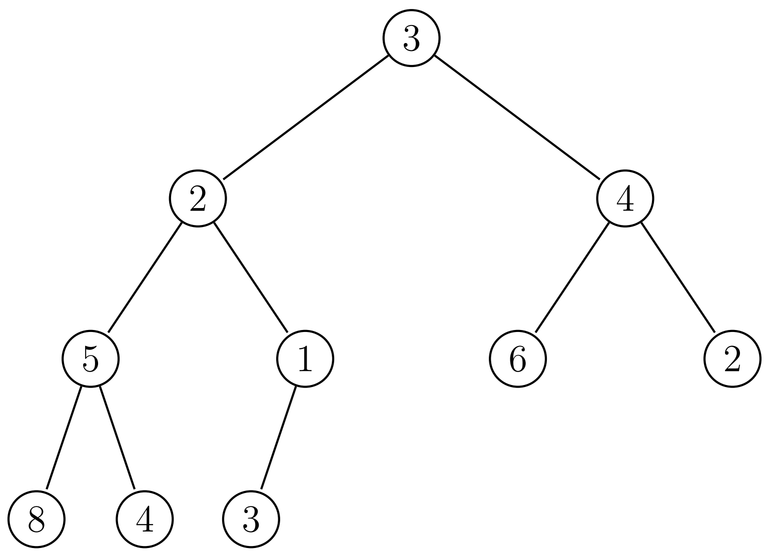

Figura 16: Esempio di albero da catastizzare

Quest'ultimo è un albero binario completo composto da 10 nodi. Pertanto il primo nodo che potrebbe avere figli è il noto 5, mentre le foglie non hanno alcun problema. A partire dal nodo avente numero d'ordine 5 si procede risalendo l'albero e applicando heapify, come mostrato di seguito, scambiando il nodo 5 con il nodo 10:

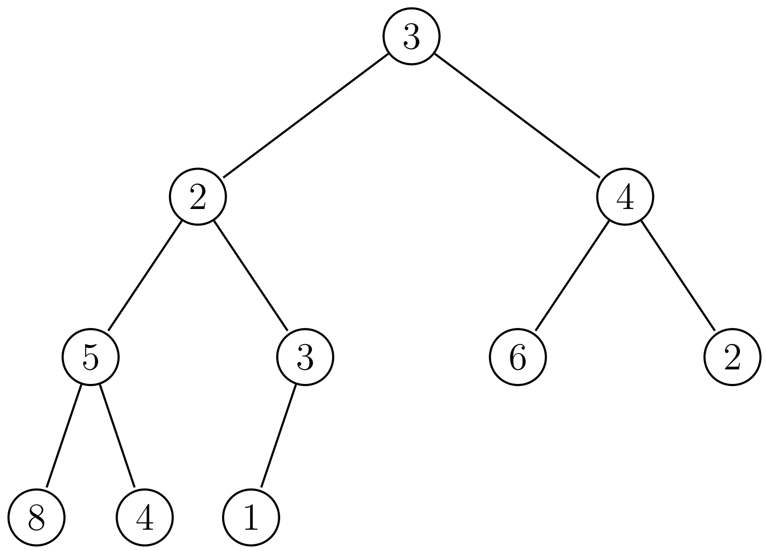

Figura 17: Esempio di albero da catastizzare dopo un primo passo

Risalendo progressivamente si considera il nodo precedente, ossia il 4 e si applica heapify, scambiando il nodo 4 con il nodo 8:

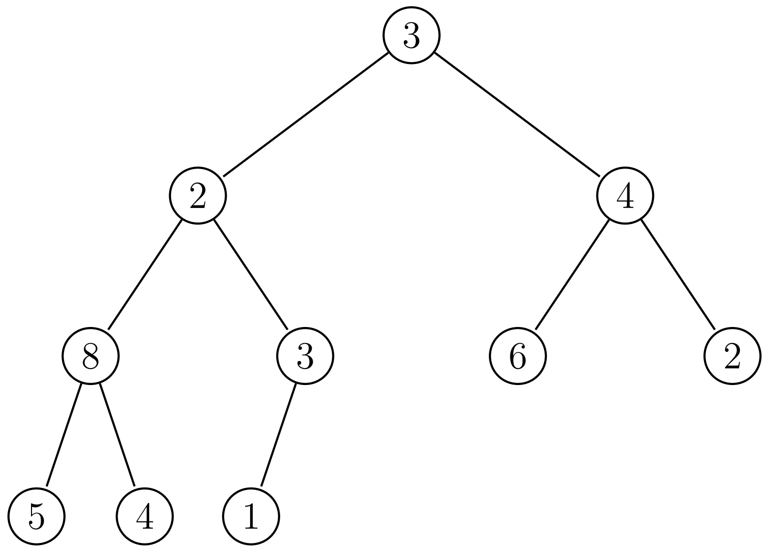

Figura 18: Esempio di albero da catastizzare dopo un secondo passo

Procedendo a ritroso si deve applicare nuovamente heapify, considerando il nodo precedente, ossia il 3 ed effettuando lo scambio tra il nodo 3 e il nodo 6:

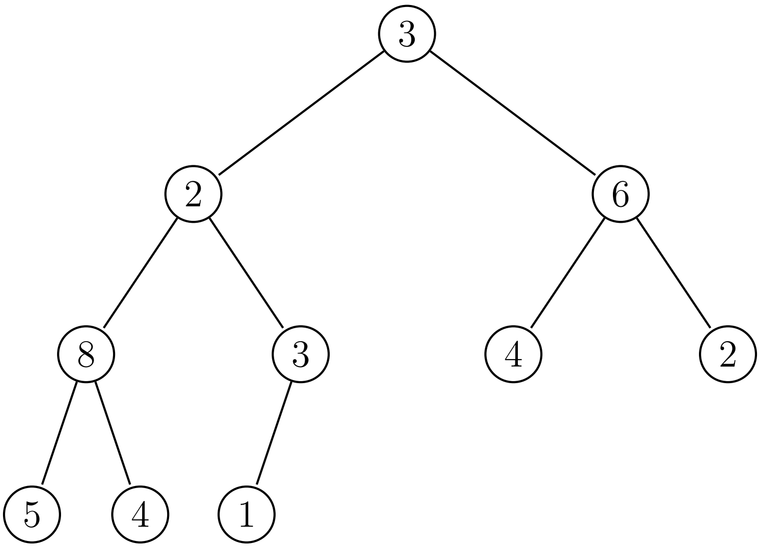

Figura 19: Esempio di albero da catastizzare dopo un terzo passo

Considerando ancora il nodo 2 si applica heapify e si scambia il nodo 2 con il nodo 4; tuttavia, heapify è ricorsivo e deve procedere operando sul sottoalbero avente sottoradice il nodo 4, effettuando uno scambio tra il nodo 4 e il nodo 8:

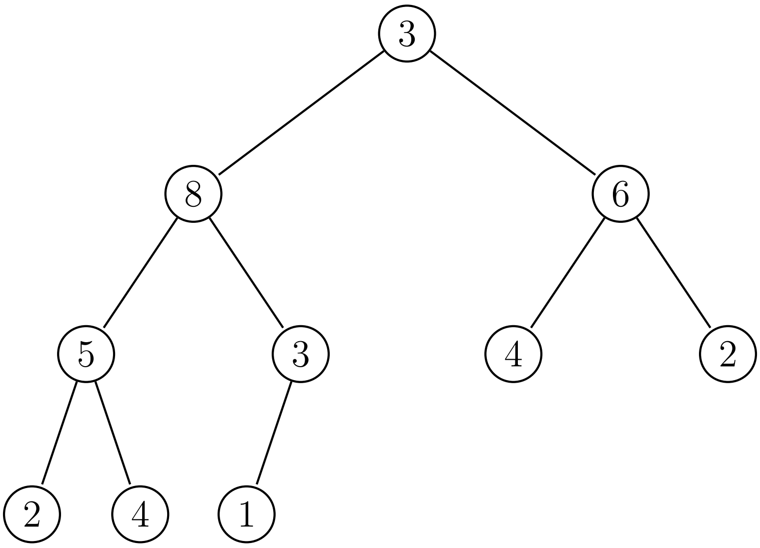

Figura 20: Esempio di albero da catastizzare dopo un quarto passo

Ed infine si applica heapify sul nodo 1, scambiandolo con il nodo 2; ancora una volta heapify deve scendere verso il basso, facendo rispettare il vincolo a tutti i sottoalberi sottostanti, scambiando il nodo 2 con il nodo 4 ed il nodo 4 con il nodo 9:

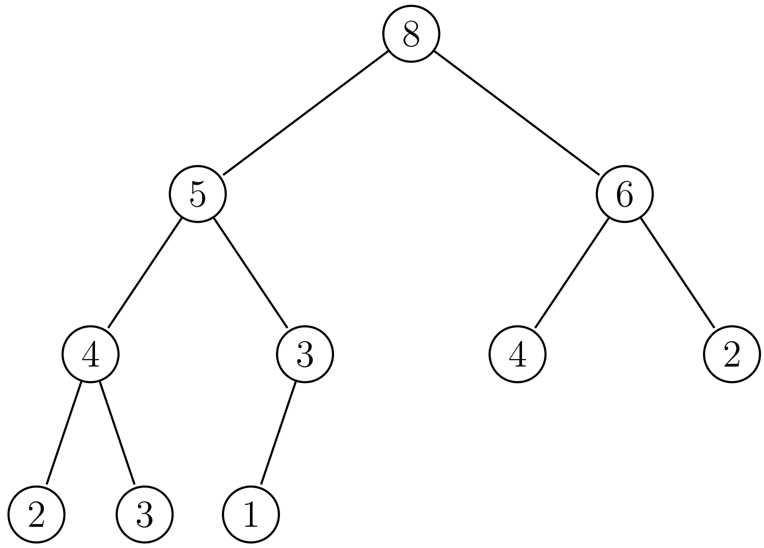

Figura 21: Esempio di albero da catastizzare dopo un quinto passo

Procedendo in questo modo, si procede alla creazione di una catasta, con un numero di iterazioni che dipende dalle chiamate ricorsive della procedura heapify; in particolare, si può osservare che il numero di nodi considerati da build-sort è approssimabile a

$$\frac{n}{2}$$

in quanto l'ultima metà è certamente esclusa, essendo i nodi delle foglie. Pertanto il ciclo for dell'algortimo ha un nuumero di iterazioni pari a $\frac{n}{2}$; tuttavia, in ognuna di tali iterazioni viene richiamata la procedura heapify, la quale è ricorsiva e presenta una complessità di $O(\log(n))$; da ciò si evince che la complessità dell'algoritmo **build-heap** è pari a

$$\boxed{O\left(\frac{n}{2} \cdot \log(n)\right) \cong O(n \cdot \log(n))}$$

Inoltre è immediato capire che siccome il numero dei nodi che bisogna visitare in **build-heap** è pari a $\frac{n}{2}$, indipendemente che heapify sia operativa o mena, si capisce che la complessità di **build-sort** è almeno lineare, ovvero

$$\boxed{\Omega\left(\frac{n}{2}\right) \cong \Omega(n)}$$

Quindi la complessità di **buil-heap** è almeno **lineare** e al più **log-lineare**; tuttavia, è sufficiente osservare che i nodi che la procedura heapify considera sono prossimi alle foglie (sempre perché considera la metà dei nodi in su ed è noto che se un albero binario è super completo, in livello delle foglie contiene più nodi di tutto il resto dell'albero), quindi i sottoalberi che vengono analizzati hanno tutti una lunghezza molto ridotta, che può abbassare la limitazione superiore assunta. Infatti, si può dimostrare che la complessità di calcolo di **build-heap** è lineare

$$\boxed{\Theta(n)}$$

e questo risultato, dal punto di vista dell'ordinamento, è totalmente irrilevante, anzi deve essere così affinché la complessità generale di altri algoritmi che ne fanno uso sia quantomeno accettabile.

Per conoscere come opera **heap-sort** si consideri l'albero binario seguente:

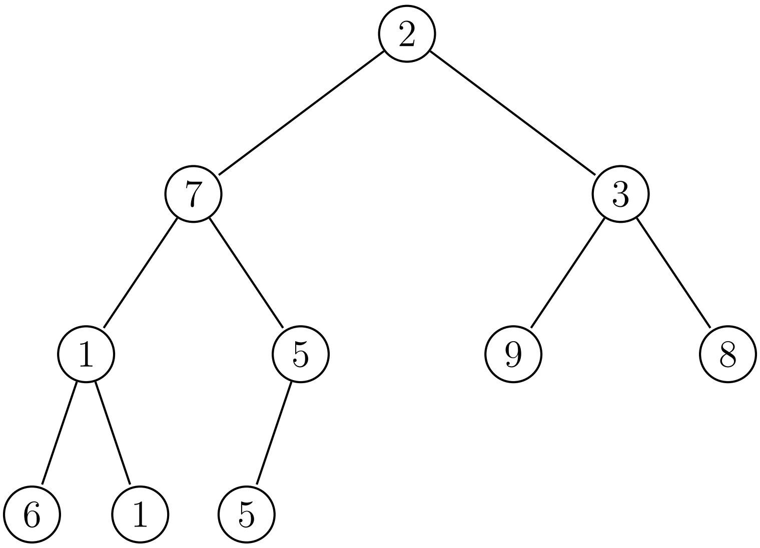

Figura 22: Esempio di albero da ordinare con heap-sort

Lavorando in loco, la prima operazione da eseguire su tale albero è quello di catastizzarlo, ottenendo una struttura completamente opposta rispetto a quella che si necesssita: sul nodo radice è presente il valore maggiore, e sull'ultima posizione il valore più piccolo, com'è evidente nel seguito:

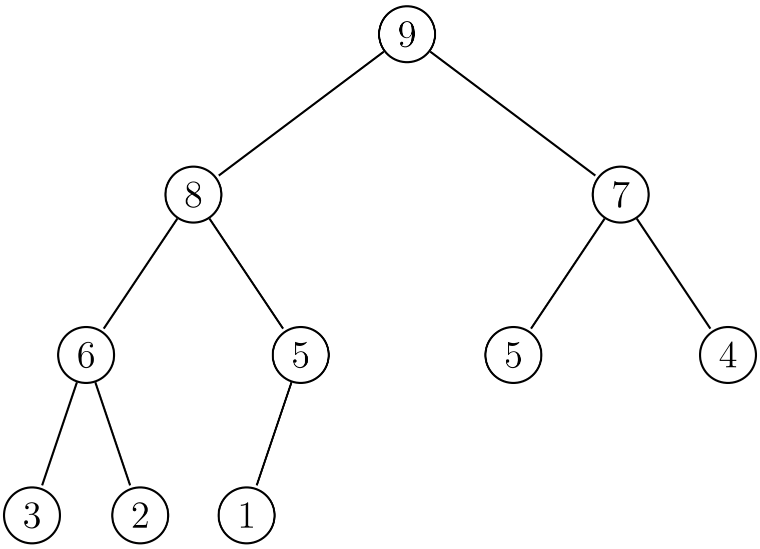

Figura 23: Esempio di albero catastizzato

Allora, secondo la logica heap-sort, si procede a sostituire il primo nodo con l'ultimo (effettuando un primo ordinamento); dal momento che l'ultimo nodo è un nodo foglia, non vi è problema che rispetti il vincolo *maximum-heap*, quindi si opera una cancellazione dell'ultimo nodo e dell'arco che lo congiunge con il nodo padre (heap-size[A] = heap-size[A] - 1).

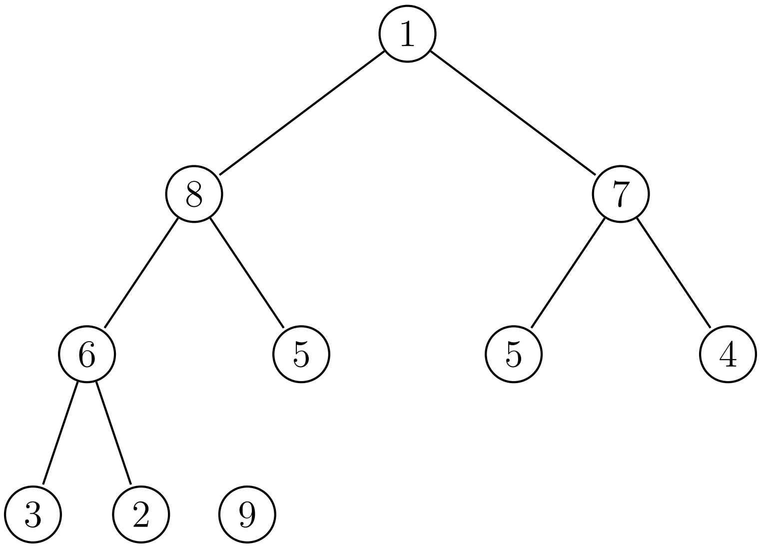

Figura 24: Esempio di albero da ordinare dopo un primo heap-sort

Ora si procede nuovamente a catastizzare, applicando heapify all'albero intero: da notare che è possibile applicare tale algoritmo in quanto l'albero intero è una catasta, essendolo già da prima; l'unico problema nel rispetto del vincolo, che giustifica l'applicazione di heapify, sta proprio nel nodo radice dell'albero intero:

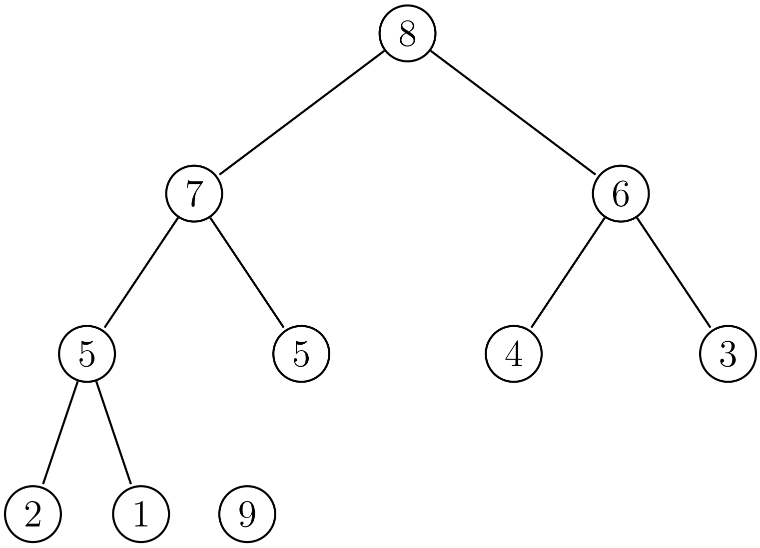

Figura 25: Esempio di albero catastizzato dopo un primo heap-sort

Ancora una volta si è ottenuta una catasta nella quale, dei valori rimasti, il maggiore si trova in prima posizione e il minore in ultima. Si procede, allora, a sostituire il primo nodo con l'ultimo, il quale viene eliminato insieme al suo collegamento con il nodo padre.

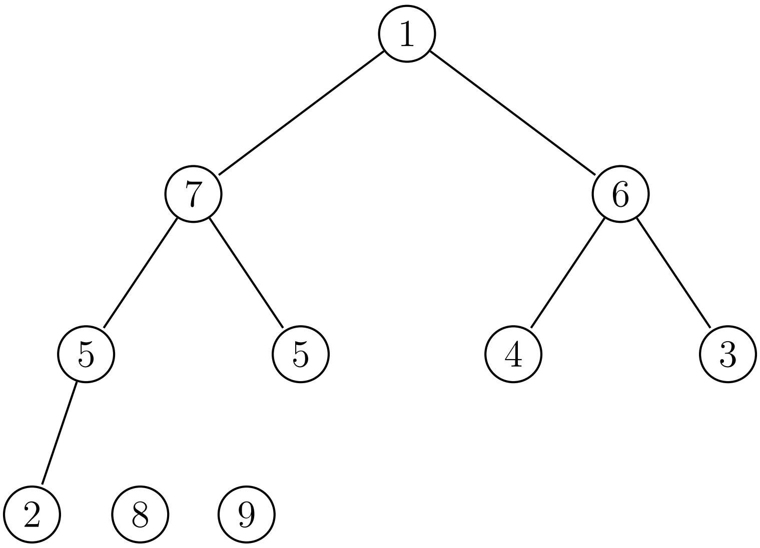

Figura 26: Esempio di albero da ordinare dopo un secondo heap-sort

Ecco che allora il penultimo nodo si trova perfettamente ordinato con gli altri; procedendo con un'operazione di heapify sull'albero intero, si ottiene ancora una volta una catasta che rispetta il vincolo e si ricomincia il procedimento daccapo, fino ad ottenere l'array perfettamente ordinato, come mostrato di seguito:

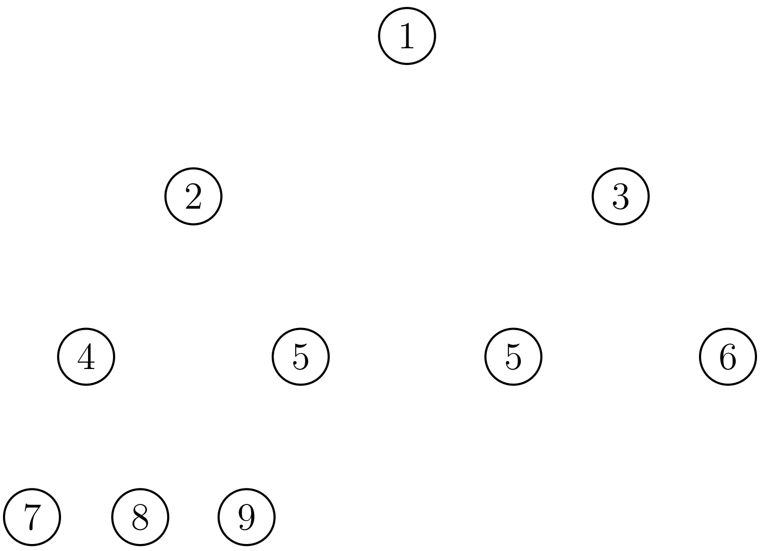

Figura 27: Esempio di albero ordinato dopo $n-1$ heap-sort

Di seguito si propone il listato dell'algoritmo **merge-sort**:

Algorithm 18 Heap-Sort

HEAP-SORT(A)
BUILD-HEAP(A)
for $i = length[A]$ **down to** 2
    do Exchange: $A[1] \leftrightarrow A[i]$
        heap-size[A] = heap-size[A] - 1
        HEAPIFY($A, 1$)

In questo caso il numero di iterazioni di ciclo for sono $n-1$ e sono tutte operazioni in serie. Per valutare la complessità di tale algoritmo si deve tenere presente la seguente regola:

COMPLESSITÀ COMPLESSIVA DI UN ALGORITMO MULTI-BLOCCO

Quando un algoritmo è costituito da diversi blocchi di istruzioni, ognuno avente una propria distinta complessità esecutiva, prevale sempre la **complessità maggiore**.

In particolare, in questo caso, la complessità di build-heap è $\Theta(n)$, mentre la complesità nel secondo blocco è $O(n \cdot \log(n))$, essendo la complessità di heapify $O(\log(n))$ e dal momento che tale procedura si richiama $n-1$ volte, applicandola su tutto l'albero e non su sottoalberi.
Pertanto si evince che la complessità dell'algoritmo **heap-sort** è al più log-lineare (ossia nel caso peggiore), ovvero

$$\boxed{O(n \cdot \log(n))}$$

mentre la complessità nel caso migliore è lineare, ovvero $\Theta(n)$. Richiamando il **teorema sugli algoritmi di ordinamento generale**, il quale afferma che un algoritmo di ordinamento di tipo generale, come **heap-sort**, è condannato ad avere una complessità nel caso peggiore almeno log-lineare e una complessità nel caso medio almeno log-lineare: questo significa che, siccome la complessità di **heap-sort** è al più log-lineare e tale risultato conferma che è almeno log-lineare, quindi, alla fine, la complessità dell'algoritmo **heap-sort** (nel caso peggiore e nel caso medio) è proprio **log-lineare**.

18 Marzo 2022

Si osservi che le operazioni che vengono svolte dall'algoritmo **heap-sort** sono **in serie**, pertanto devono essere valutate in maniera attenta per comprender la complesità dell'algortimo stesso.
In particolare, è noto che, dal punto di vista della notazione Θ, se vi sono due blocchi di istruzioni che presentano complessità diverse, ciò che determina la complessità finale dell'algoritmo è la complessità peggiore e più pesante fra le due date. Ciò si può dimostrare facilmente, considerando, a tal proposito, due funzioni $f(n)$ e $g(n)$, allora

$$f(n) + g(n) \ \text{ e } \ f(n) \vee g(n)$$

in cui con $f(n) \vee g(n)$ è da intendersi il massimo tra $f(n)$ e $g(n)$, appartengono alla stessa classe Θ, in quanto, logicamente, si ha che

$$f(n) \vee g(n) \leq f(n) + g(n) \leq 2 \cdot [f(n) \vee g(n)]$$

In particolare, si osserva che l'algoritmo **heap-sort** procede dapprima realizzando una catasta tramite il sotto-algoritmo **build-heap**, la cui complessità è lineare; dopo questa procedura si ottiene un albero che è ordinato al contrario (con i valori maggiori sui primi nodi e i valori minori sugli ultimi nodi); pertanto si procede a sostituire il primo nodo con l'ultimo dei rimanenti ed eliminando progressivamente l'ultimo nodo e il collegamento con i rimanenti (tramite l'istruzione heap-size = heapsize - 1), procedendo ad ordinare l'albero intero (a partire dalla radice, a differenza della procedura *build-heap*, in cui *heapify* veninva applicata a sottoalberi vicini alle foglie), tramite la procedura *heapify*, ad ogni passaggio.
Pertanto la complessità dell'algoritmo **heap-sort** è **log-lineare**, che **asintoticamente** è ottima, come avvalorato dal teorema sulla complessità di un algoritmo di ordinamento generale.

4.8 Quick-sort

L'algoritmo di ordinamento **quick-sort** è un algoritmo che, come merge-sort, lavora su posizioni intermedie da 1 a n, impiegando gli indici p e r, e opera in maniera ricorsiva.
Tuttavia, mentre in merge-sort, ad ogni chiamata, l'insieme dei valori da ordinare si dimezzava fino a considerare un solo valore, in quick-sort tale divisione esatta in due parti non si verifica, in quanto le divisioni sono irregolari.
Il primo sotto-algoritmo adoperato da quick-sort prende il nome **partition** (partizione), il quale considera come argomenti l'array A di lunghezza n e due posizioni intermedie p e r, con il vincolo che $p < r$ (vincolo imposto da quick-sort), in cui inzialmente $p = 1$ e $r = n$, come mostrato di seguito:

$$\begin{array}{ccccccccc} p &&&&&&&& r \\ \boxed{2} & \boxed{8} & \boxed{7} & \boxed{2} & \boxed{3} & \boxed{5} & \boxed{6} & \boxed{4} \end{array}$$

Tale array verrà diviso in 4 aree, in una delle quali (l'ultima, generalmente) è presente un **perno** (o **pivot**): nella prima area saranno contenuti valori $\leq$ del perno, nella seconda area saranno contenuti valori $>$ del perno, mentre nella terza area saranno contenuti valori che ancora con sono stati esaminati (e che quindi non è noto se siano maggiori o minori del perno): è chiaro che progressivamente le prime due aree andranno incrementandosi, mentre la terza area andrà via via riducendosi, fino a sparire con la fine dell'algoritmo. Pertanto, vi sono i confini delle due prime aree che sono mobili, chiamati i e j: il primo, i, che divide la prima e la seconda area, si muove in maniera irregolare; il secondo, j, che aumenta di un passo alla volta in modo meccanico.
Si osservi l'esecuzione di tale algoritmo: posto $p = 1$ e $r = n$, si ha $i = p - 1$ e $j = p$, come mostrato di seguito:

$$\begin{array}{ccccccccc} & p &&&&&&& r \\ | & \boxed{2} & \boxed{8} & \boxed{7} & \boxed{2} & \boxed{3} & \boxed{5} & \boxed{6} & \boxed{4} \\ i & j &&&&&&& \end{array}$$

e si procede a verificare se il primo elemento è più piccolo o più grande dell'ultimo elemento (ossia del perno), come nel caso seguente $2 \leq 4$; se questo è il caso, allora i si incrementa di 1 e si cambia $A[i] \leftrightarrow A[j]$ e si incrementa j di 1:

$$\begin{array}{cccccccccc} p &&&&&&&&& r \\ \boxed{2} & | & \boxed{8} & \boxed{7} & \boxed{2} & \boxed{3} & \boxed{5} & \boxed{6} & \boxed{4} \\ & i & j &&&&&&& \end{array}$$

Naturalmente in tale prima fase, lo scambio $A[i] \leftrightarrow A[j]$ è totalmente inutile, in quanto $i = j = 1$, ma avrà una validità significativa nel seguito.
Ora si confronta il j-esimo elemento con il pivot e si ha che $8 > 4$ e si incrementa j di uno, e così anche per l'iterazione successiva, in quanto $7 > 4$, giungendo alla configurazione seguente:

$$\overset{p}{\boxed{2}} \underset{i}{|} \boxed{8}\boxed{7}\underset{j}{\boxed{2}}\boxed{3}\boxed{5}\boxed{6}\overset{r}{\boxed{4}}$$

Tuttavia, quando $j = 4$, si ha che $2 \leq 4$, per cui si incrementa i di 1: tuttavia, adesso l'indice i punta ad una zona errata ($i = 2$ e quindi $A[i] = 8$), per cui si deve scambiare $A[i] \leftrightarrow A[j]$ ed incrementare j, ottenendo:

$$\overset{p}{\boxed{2}}\boxed{2} \underset{i}{|} \boxed{7}\boxed{8}\underset{j}{\boxed{3}}\boxed{5}\boxed{6}\overset{r}{\boxed{4}}$$

Confrontando il j-esimo elemento, ancora una volta $3 \leq 4$, per cui si incrementa i di 1 e si deve scambiare $A[i] \leftrightarrow A[j]$ ed incrementare j:

$$\overset{p}{\boxed{2}}\boxed{2}\boxed{3} \underset{i}{|} \boxed{8}\boxed{7}\underset{j}{\boxed{5}}\boxed{6}\overset{r}{\boxed{4}}$$

Le due ultime due iterazioni prevedono solo di incrementare j essendo $A[j]$ tutti valori maggiori del perno:

$$\overset{p}{\boxed{2}}\boxed{2}\boxed{3} \underset{i}{|} \boxed{8}\boxed{7}\boxed{5}\boxed{6} \underset{j}{|} \overset{r}{\boxed{4}}$$

Ora da p a i vi sono valori più piccoli del pivot, mentre da $i + 1$ a j vi sono valori maggiori del perno. Ora è sufficiente scambiare l'ultima posizione con la posizione $i + 1$ per porre il perno nella sua posizione corretta:

$$\overset{p}{\boxed{2}}\boxed{2}\boxed{3} \underset{i}{|} \boxed{4}\boxed{7}\boxed{5}\boxed{6} \underset{j}{|} \overset{r}{\boxed{8}}$$

Ecco che il **perno** è stato collocato proprio dove deve stare, all'interno dell'array: la sua posizione è definitiva e non verrà più cambiata, ed è questa la chiave di funzione dell'algoritmo **quick-sort**. Pertanto, ora, *partition* dovrà essere applicato da p ad i e da $i + 2$ a r, andando a collocare nella loro posizione esatta e definitiva tutti i perni; ora che l'array è stato partizionato, di seguito si espone il listato dell'algoritmo **partition**:

Algorithm 19 Partition

```
PARTITION(A,p,r)
x ← A[r]
i = p − 1
for j = p to r − 1
    do if A[j] ≤ x
        then i = i + 1
        Exchange: A[i] ↔ A[j]
Exchange: A[i + 1] ↔ A[r]
return i + 1
```

In cui, ovviaente $i + 1$ è la posizione definitiva del perno che viene restituita dal sotto-algoritmo *partition*.
Mentre di seguito si espone il listato dell'algoritmo **quick-sort**:

Algorithm 20 Quick-Sort

```
QICK-SORT(A,p,r)
if p < r
    then q = PARTITION(A,p,r)
    QUICK-SORT(A,p,q − 1)
    QUICK-SORT(A,q + 1,r)
```

In cui in q viene memorizzata la posizione definitiva del perno. Naturalmente ciò che permette di uscire dalla chiamata ricorsiva sono i tratti di lunghezza unitaria, per cui $p = r$.

Esempio: Si consideri l'array seguente:

3	4	1	2

Allora, nell'esecuzione dell'algoritmo **quick-sort**, si procede eseguendo dapprima la procedura *partition*, con $p = 1$ e $r = n$ e fissando come perno $A[r] = 2$.
Naturalmente, nelle prime due iterazioni, essendo $3 > 2$ e $4 > 2$, *partition* non opera alcun cambiamento all'array A, ed effettua esclusivamente l'incremento di j. Quando $j = 3$, allora si osserva che $1 \leq 2$, per cui si deve incrementare i di 1, scambiare $A[i] \leftrightarrow A[j]$ e incrementare j, ottenendo:

1	4	3	2

Arrivati all'ultima iterazione, bisogna eseguire lo scambio $A[i+1] \leftrightarrow A[r]$, ottenendo:

1	2	3	4

ecco che il perno 2 si trova nella sua posizione $q = 2$ definitiva, dalla quale non verrà più spostato. Ora bisogna applicare quick-sort da $p = 1$ a $q - 1 = 1$ e da $q + 1 = 3$ a $r = 4$; naturalmente, nel primo caso, quick-sort non effettua alcuna operazione, visto che $p = r = 1$; nel secondo caso si dovrà applicare *partition* con $p = 3$ e $r = 4$, pertanto, essendo $3 \leq 4$, *partition* dovrà incrementare i di 1, eseguire lo scambio $A[i] \leftrightarrow A[j]$ (ma essendo $i = j$, tale scambio è superfluo) ed incrementare j di 1. Terminata l'unica iterazione del ciclo, *partition* eseguirà lo scambio $A[i+1] \leftrightarrow A[r]$ (ma essendo $i + 1 = 4$ e $r = 4$, ancora una volta tale scambio è superfluo) per cui l'array diverrà

1	2	3	4

e *partition* terminerà, restituendo $q = 4$. Ancora una volta si dovrà applicare quick-sort, dalla posizione $p = 3$ e $r = 3$ e $p = 5$ e $r = 4$; naturalmente, in nessuno di tali casi $p < r$, per cui le chiamate ricorsive cessano, così come l'algoritmo quick-sort.

Osservazione: Per comprendere il numero di iterazioni che comporta l'utilizzo di quick-sort, non è possibile ragionare come con merge-sort, in cui venivano eseguite delle operazioni di divisione netta in due parti uguali ad ogni iterazione, in quanto con quick-sort non si ha la certezza che le partizioni su cui si andrà a lavorare siano proprio delle metà esatte.
Si consideri, a tal proposito, il caso di un ordinamento di n numeri che sono, in realtà, già ordinati:

1	2	3	4	$\cdots$	$n-1$	|	n

Naturalmente, in questo, caso, il ciclo for della procedura *partition* eseguirà un numero di iterazioni pari a $n - 1$, ossia da 1 a $r - 1 = n - 1$: il perno si trovava in ultima posizione all'inizio del ciclo e taluna sarà la sua posizione definitiva; dopo il primo quick-sort, si dovranno applicare due quick-sort, uno da $p = 1$ a $r = q - 1 = n - 1$ e un secondo da $p = q + 1 = n + 1$ a $r = n$: è ovvio che dei due lavorerà solamente il primo, mentre il secondo prevederà solo un controllo iniziale; tuttavia, la prima *partition* prevede un ciclo for di $n - 2$ iterazioni, ossia da 1 a $r - 1 = n - 2$, e dopoiché si dovranno eseguire due nuovi quick-sort, il primo da $p = 1$ a $r = q - 1 = n - 2$ e il secondo da $p = q + 1 = n$ fino a $r = n - 1$; ancora una volta il primo prevede $n - 3$ iterazioni, mentre il secondo un solo controllo, e così via, fino ad arrivare ad un quick-sort da 1 iterazione e 1 controllo.
Naturalmente, la complessità data dal numero di controlli che devono essere eseguiti nei secondi quick-sort è $\Theta(n)$, mentre la somma di tutte le iterazioni dovute all'esecuzione dei primi quick-sort è data da

$$n - 1 + n - 2 + n - 3 + ... + 1 = \frac{n \cdot (n+1)}{2} \cong n^2$$

Ecco che allora si evince come la complessità di tale algoritmo, nel caso *peggiore*, è, appunto **quadratica**, ovvero $\Theta(n^2)$.
Tuttavia, la complessità dell'algoritmo nel caso generale (o caso medio) è **log-lineare**, che quindi rende tale algoritmo **asintoticamente** ottimo. Inoltre anche la **complessità empirica** è log-lineare, in quanto l'utilizzo di tale codice regolarmente conferma come la complessità quadratica si riscontri solamente in casi particolari, quando i dati in input sono strutturati in un certo modo.

4.9 Randomize Quick-Sort

Per comprendere la complessità dell'algoritmo **quick-sort**, nel caso generale, bisogna considerare una variante aleatoria di tale algoritmo, la quale prende il nome di **randomize quick-sort**.
La probabilità di un evento può essere definita come il grado di fiducia che il senso comune attribuisce al verificarsi dell'evento, valutato su una scala $[0, 1]$: tuttavia, la casualità dell'evento riguarda non tanto l'evento in sé, ma l'osservatore che, non potendo conoscere il risultato dell'evento stesso, lo reputa casuale.
Il concetto di probabilità, in senso teorico, può essere suddivisa in **tre macroaree di interesse**:

- probabilità combinatoria;
- probabilità empirica;
- probabilità soggettiva o neo-bayesiana.

La probabilità, infatti, è strettamente connessa al calcolo combinatorio: pertanto, la probabilità di un evento è definibile come il rapporto fra il numero di casi favorevoli e il numero di casi possibili (ossia gli eventi elementari), sempre nell'ipotesi che gli eventi elementari siano tutti equiprobabili (che è un'ipotesi decisamente condizionante per fornire significato alla formula seguente):

$$\boxed{P(E) = \frac{\#\text{casi favorevoli}}{\#\text{casi possibili}}}$$

per cui il grado di fiducia è proporzionale al numero di casi favorevoli.
Dal punto di vista empirico (o statistico o oggettivo, in quanto fondata sull'esperimento), invece, si ha che

$$\boxed{P(E) = \lim_{n \to +\infty} \frac{\#\text{successi}}{n}}$$

ovvero il limite metafisico quando il numero degli esperimenti diverge all'infinito del rapporto tra il numero dei successi e il numero degli esperimenti (è ovvio che sia un limite metafisico, perché affinché la mera valutazione statistica fornita da un numero finito di esperimenti si trasformi in probabilità, il numero degli esperimenti deve divenire infinito, cosa che è materialmente impraticabile). È chiaro che anche tale definizione è fortemente limitante, in quanto affinché essia sia affidabile, dovrebbe richiedere che tutti gli esperimenti che vengono eseguiti siano una copia dello stesso esperimento astratto, ossia che si ripetano sempre nelle stesse condizioni (**decidendo in maniera** soggettiva quali fattori siano rilevanti e, pertanto, debbano essere mantenuti costanti nel corso della sperimentazione): eccco, allora, che tale elemento soggettivo eguaglia quello della definizione combinatoria, nella quale si richiedeva l'equiprobabilità degli eventi elementari.
La probabilità soggettiva o neo-bayesiana (o definettiana) deve la sua concezione, naturalmente, al reverendo **Bayes**, mentre la sua diffusione e formulazione a **Bruno de Finetti**, il quale è riuscito ad analizzare gli *odds* tramite delle fondamenta teoriche precise, definendo la probabilità come la misura di quanto l'uomo è propenso a scommettere.
Si possono, pertanto, interpretare i tre differenti concetti di probabilità in **modalità telescopica**:

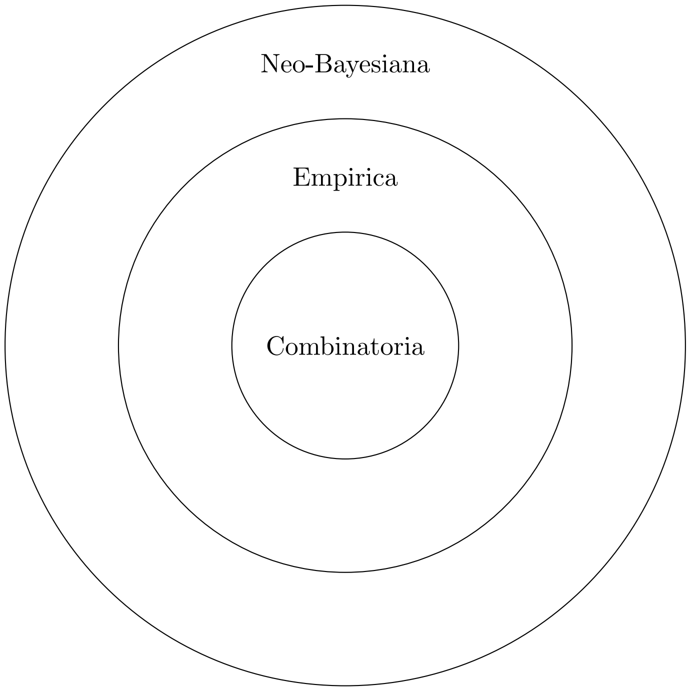

Figura 28: Interpretazione telescopica della probabilità

In cui è evidente come la probabilità combinatoria è contenuta nella probabilità statistica (o empirica) che sono contenute nella probabilità bayesiana, in quanto il suo campo di applicazione è enorme, notevolmente più esteso dei precedenti.

23 Marzo 2022