

Appunti di

# Algoritmi

---

Rosario Terranova

Sommario	
Introduzione.....	2
Complessità degli algoritmi.....	2
Equazioni di ricorrenza.....	6
Algoritmi di ordinamento e loro complessità .....	8
Heap .....	15
Ordinamento in tempo lineare .....	19
Calcolo delle Probabilità .....	21
Mediane e statistiche d'ordine .....	21
Dizionari .....	21
Alberi.....	22
Programmazione dinamica .....	31
Grafi .....	33

Ultimo aggiornamento: 12/05/2015	Versione file: 1.65
----------------------------------	---------------------

## Introduzione

Nel mondo dell'informatica è di enorme importanza lo studio e la risoluzione di un problema computazionale, ovvero un problema di natura informatica dove l'utente cerca di fare una data operazione in un tempo più veloce possibile; esso è specificato da una data relazione tra input (istanze) e output (soluzioni).

Es. Sorting (problema dell'ordinamento) di una sequenza di numeri disordinati, che ha tali numeri come input e la sequenza di numeri ordinati come output. Lo strumento per risolvere tale problema è un algoritmo.

Un **ALGORITMO** è una procedura computazionale ben definita (ad ogni istante è possibile determinare ciò che l'algoritmo sta eseguendo) che produce uno o più valori (output) in funzione di uno o più altri valori (input) per risolvere problemi computazionali. È una sequenza di comandi elementari ed univoci, quindi un procedimento descrivibile finitamente.

INPUT → ALGORITMO → OUTPUT

Un algoritmo non può essere scomposto in comandi più semplici.

## Complessità degli algoritmi

Un aspetto fondamentale che va affrontato nello studio degli algoritmi è la loro **efficienza**, cioè la quantificazione delle loro esigenze in termini di tempo e di spazio, ossia tempo di esecuzione e quantità di memoria richiesta. Questo perché: i calcolatori sono molto veloci, ma non infinitamente veloci; la memoria è economica e abbondante, ma non è né gratuita né illimitata.

Per determinare l'efficienza degli algoritmi a prescindere dal tipo di calcolatore usiamo il **COSTO COMPUTAZIONALE**, inteso come il numero delle operazioni elementari e la quantità di spazio di memoria necessario in funzione della dimensione dell'input.

Es. Il computer V (veloce) effettua  $10^9$  operazioni al secondo; il computer L (lento) ne fa  $10^7$  al secondo. Bisogna ordinare  $n = 10^6$  numeri interi. L'algoritmo IS (insertion sort) richiede  $2n^2$  operazioni, mentre MS (merge sort) ne richiede  $50n \log n$ .

$$V(IS) = \frac{2(10^6)^2 \text{ istruzioni}}{10^9 \text{ istruzioni al secondo}} = 33 \text{ minuti} \quad L(MS) = \frac{50 \cdot 10^6 \log 10^6 \text{ istruzioni}}{10^7 \text{ istruzioni al secondo}} = 1,5 \text{ minuti}$$

Possiamo notare che indipendentemente dall'aumento di velocità dei computer, l'efficienza degli algoritmi è un fattore di importanza cruciale.

La **COMPLESSITÀ** di un algoritmo è l'impiego di risorse necessario dell'intero sistema di calcolo per la risoluzione di un particolare tipo di problema. È valutato secondo le diversi componenti che cooperano in un sistema di calcolo:

- Complessità temporale: quanto richiede l'esecuzione di un algoritmo
- Complessità spaziale: quanta occupazione di memoria richiede l'algoritmo
- Complessità di I/O: quanto tempo richiede l'acquisizione di informazioni dalle periferiche
- Complessità di trasmissione: efficienza di I/O di un algoritmo rispetto a più computer

Ci occuperemo della complessità più importante, ovvero della **COMPLESSITÀ TEMPORALE**. Dato che uno stesso problema può essere risolto in più modi diversi, cioè con algoritmi diversi, lo scopo di questa complessità è trovare quello più efficiente in termini di tempo di calcolo. I fattori che influenzano il tempo di esecuzione sono l'algoritmo scelto, la dimensione dell'input e la velocità della macchina. Il modello di calcolo di riferimento è quello di un processore RAM in cui le istruzioni sono eseguite una alla volta.

La **FUNZIONE DI COMPLESSITÀ TEMPO** lega la dimensione dell'input al tempo che l'algoritmo impiega su di esso. Dato un algoritmo  $A$ , tipicamente  $T_A$  indica la sua funzione di complessità.

Es. Data una costante  $c$  e  $T_A(n) = c \cdot n$ , consideriamo due input di dimensione  $n_1$  e  $2n_1$  e appliciamoli alla funzione trovando  $T_A(n_1) = c \cdot n_1$  e  $T_A(2n_1) = c \cdot 2 \cdot n_1 = 2 \cdot T_A(n_1)$ . Abbiamo dimostrato che un input più grande impiega più tempo per essere portato a compimento.

Per poter valutare la **COMPLESSITÀ DEGLI ALGORITMI**, essi devono essere formulati in modo che siano chiari, sintetici e non ambigui. Si adotta il cosiddetto **pseudocodice**, che è una sorta di linguaggio di programmazione “informale” nell’ambito del quale:

- Istruzioni elementari (operazioni aritmetiche, lettura/scrittura variabili, stampa, ecc.) hanno complessità 1
- L’istruzione *if* ha complessità 1 per la verifica + il max delle complessità delle istruzioni *then* e *else*.
- Le istruzioni iterative hanno una complessità pari alla somma delle complessità massime di ciascuna delle iterazioni. Se tutte le iterazioni hanno la stessa complessità massima, allora la complessità dell’iterazione è pari al prodotto della complessità massima di una singola iterazione per il numero di iterazioni.
- La complessità dell’algoritmo è pari alla somma delle complessità delle istruzioni che lo compongono.

Es. Semplice ciclo while

i=0;	1 assegnamento	Il numero totale di passi base è dato dalla somma dei valori $1 + n + 1 + 1n = 2 + 2n$ quindi $T(n) = c(2 + 2n) = O(n)$ ovvero l’algoritmo ha complessità lineare.
while (i<n)	n+1 test del ciclo	
i=i+1;	1n assegnamenti interni	

Es. Funzione Trova\_Max (A è un vettore)

max = A[1]	1 assegnamento	$T(n) = \Theta(1) + (n - 1) \Theta(1) + \Theta(1) = \Theta(n)$ complessità lineare.
for(i=2;i<n;i++)	n-1 iterazioni + 1 assegnamento	
if (A[i]>max)	1 test	
max = A[i];	1 assegnamento	
print max;	1 stampa	

Di seguito la **SCALA DELLE COMPLESSITÀ TIPICHE** in ordine dalla più veloce alla più lenta.

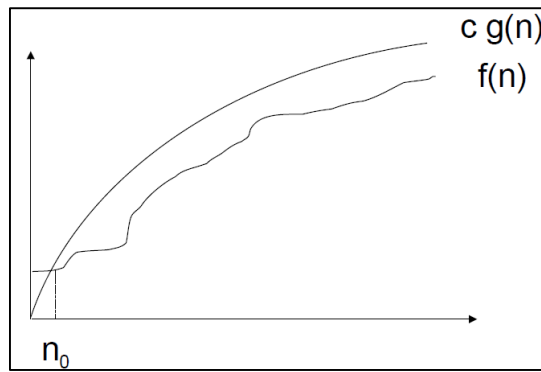
complessità \ dim. input	10	$10^3$	$10^6$
costante - $O(1)$	1 $\mu$ sec	1 $\mu$ sec	1 $\mu$ sec
logaritmica - $O(\lg n)$	3 $\mu$ sec	10 $\mu$ sec	20 $\mu$ sec
lineare - $O(n)$	10 $\mu$ sec	1 msec	1 sec
n lg n (pseudolineare) - $O(n \lg n)$	33 $\mu$ sec	10 msec	20 sec
quadratica - $O(n^2)$	100 $\mu$ sec	1 sec	10 <sup>12</sup> sec
cubica - $O(n^3)$	1 msec	10 <sup>9</sup> sec	10 <sup>18</sup> sec
esponenziale - $O(2^n)$	10 msec	10 <sup>301</sup> sec	10 <sup>301030</sup> sec

Per caratterizzare il tasso di crescita del tempo di esecuzione di un dato algoritmo viene usata la **COMPLESSITÀ COMPUTAZIONALE ASINTOTICA**. Esso è uno strumento che valuta l’efficienza dell’algoritmo così da permetterne il confronto con algoritmi diversi che risolvono lo stesso problema. Siamo interessati a risultati di tipo asintotico, ossia validi per *grandi* dimensioni dell’input. Abbiamo tre tipi di notazione:

**NOTAZIONI ASINTOTICA O (O GRANDE - LIMITE SUPERIORE):** sia  $g: \mathbb{N} \rightarrow \mathbb{N}$  (una funzione dall’insieme dei numeri naturali allo stesso insieme)

$$O(g(n)) = \{f(n): \exists c > 0 \wedge n_0 \in \mathbb{N} : 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

Quindi diremo che  $O(n)$  è il limite asintotico superiore della funzione  $f(n)$ , ossia il costo computazionale massimo dell’algoritmo nel quale esso impiegherà più tempo per essere risolto (caso peggiore).



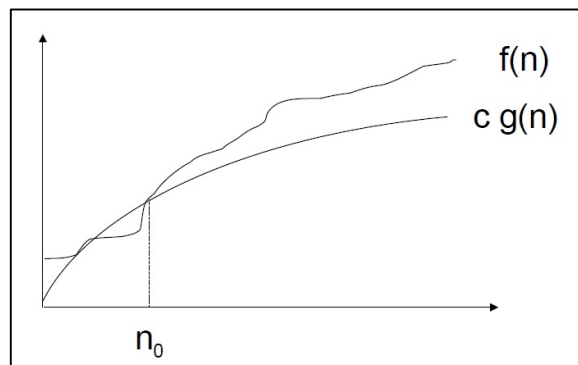
Es. Sia  $f(n) = 3n + 3$ .  
 $f(n)$  è un  $O(n^2)$  in quanto posto  $c = 6, cn^2 \geq 3 + 3 \forall n \geq 1$   
 $f(n)$  è un  $O(n)$  in quanto  $cn \geq 3 + 3 \forall n \geq 1$  se  $c \geq 6$

È facile convincersi che, data una funzione  $f(n)$ , esistono infinite funzioni  $g(n)$  per cui  $f(n)$  risulta un  $O(g(n))$ . Ci occorre determinare la più piccola funzione  $g(n)$  tale che  $f(n)$  sia  $O(g(n))$ . Quindi nella funzione sopra  $f(n)=3n+3$  è  $O(n)$ .

**NOTAZIONI ASINTOTICA  $\Omega$  (OMEGA GRANDE - LIMITE INFERIORE):** sia  $g: \mathbb{N} \rightarrow \mathbb{N}$

$$\Omega(g(n)) = \{f(n): \exists c > 0 \wedge n_0 \in \mathbb{N} : f(n) \geq cg(n) \geq 0 \forall n \geq n_0\}$$

Quindi diremo che  $\Omega(n)$  è il limite asintotico inferiore della funzione  $f(n)$ , ossia il costo computazionale minimo dell'algoritmo nel quale esso impiegherà meno tempo per essere risolto (caso migliore).



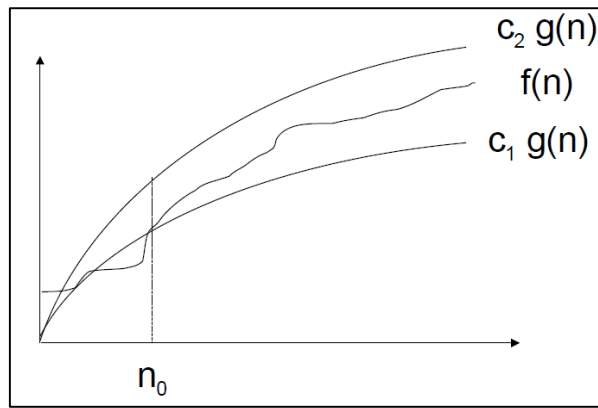
Es. Sia  $f(n) = 2n^2 + 3$ .  
 $f(n)$  è un  $\Omega(n)$  in quanto posto  $c = 1, 2n^2 + 3 \geq cn \forall n$   
 $f(n)$  è un  $\Omega(n^2)$  in quanto  $2n^2 + 3 \geq cn^2 \forall n$  se  $c \leq 2$

Si prende sempre il più grande  $g(n)$ , quindi nella funzione sopra  $f(n)=2n^2+3$  è  $\Omega(n^2)$ . In effetti  $O(g(n))$  e  $\Omega(g(n))$  sono insiemi di funzioni, e dire " $f(n)$  è un  $O(g(n))$ " oppure " $f(n) = O(g(n))$ " ha il significato di " $f(n)$  appartiene a  $O(g(n))$ ". Tuttavia, poiché i limiti asintotici ci servono per stimare con la maggior precisione possibile la complessità di un algoritmo, vorremmo trovare, fra tutte le possibili funzioni  $g(n)$ , quella che più si avvicina a  $f(n)$ . Per questo cerchiamo la più piccola funzione  $g(n)$  per determinare  $O$  e la più grande funzione  $g(n)$  per determinare  $\Omega$ .

**NOTAZIONI ASINTOTICA  $\Theta$  (THETA GRANDE - LIMITE STRETTO):**  $O$  e  $\Omega$  sono insiemi di funzioni. Per trovare la più piccola  $O(g(n))$  e la più grande  $\Omega(g(n))$  usiamo  $\Theta(g(n))$ .

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) = \{f(n): \exists c_1, c_2 > 0 \text{ ed } n_0 \in \mathbb{N} : \forall n \geq N, c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$

Quindi diremo che  $\Theta(n)$  è il limite asintotico stretto della funzione  $f(n)$ , ossia il costo computazionale medio dell'algoritmo nel quale esso impiegherà un tempo medio per essere risolto (caso medio).



Es. Sia  $f(n) = 3n + 3$ .  
 $f(n)$  è un  $\Theta(n)$  ponendo  $c_1 = 3, c_2 = 4, n_0 = 3$

Di seguito alcune regole dell'**ALGEBRA DELLA NOTAZIONE ASINTOTICA**:

- Le costanti moltiplicative si possono ignorare.
- Es. Trovare il limite asintotico stretto per  $f(n) = 3n2^n + 4n^4$   
 $3n2^n + 4n^4 = \Theta(n)\Theta(2^n) + \Theta(n^4) = \Theta(n2^n) + \Theta(n^4) = \Theta(n2^n)$
- Le notazioni asintotiche commutano con l'operazione di somma.
- Es. Trovare il limite asintotico stretto per  $f(n) = 2^{n+1}$   
 $2^{n+1} = 2^n \cdot 2 = \Theta(2^n)$
- Le notazioni asintotiche commutano con l'operazione di prodotto.
- Es. Trovare il limite asintotico stretto per  $f(n) = 2^{2n}$   
 $2^{2n} = \Theta(2^{2n})$

I limiti asintotici superiore e inferiore possono essere stretti oppure no.

- $2n^2 = O(n^2)$  è stretto  $2n^2 = \Omega(n^2)$  è stretto
- $2n = O(n^2)$  non è stretto  $2n = \Omega(n^2)$  non è stretto

I limiti non stretti si possono indicare con le notazioni  $o(g(n))$  e  $\omega(g(n))$ , i quali si chiamano rispettivamente **LIMITE O (o) PICCOLO** e **LIMITE OMEGA ( $\omega$ ) PICCOLO**. Data una coppia di funzioni  $f(n)$  e  $g(n)$  si ha:

- $o(g(n)) = f(n) : \forall c > 0 \exists n_0 > 0 : 0 \leq f(n) < cg(n) \forall n \geq n_0$   
 ◦ proprietà  $f(n) = o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
- $\omega(g(n)) = f(n) : \forall c > 0 \exists n_0 > 0 : 0 \leq cg(n) < f(n) \forall n \geq n_0$   
 ◦ proprietà  $f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$

Di seguito un elenco di **NOTAZIONI STANDARD** e **FUNZIONI COMUNI** utilizzate nello studio degli algoritmi.

- $\lfloor x \rfloor = (\text{massimo intero} \leq x)$  conosciuto come *floor*
- $\lceil x \rceil = (\text{minimo intero} \geq x)$  conosciuto come *ceiling*
- $\forall x \in \mathbb{R}: x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- $\forall n \in \mathbb{N} \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$
- $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$
- $e^x \geq 1 + x$
- $|x| \leq 1 \rightarrow 1 + x \leq e^x \leq 1 + x + x^2$
- $\lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n = e^x$
- $|x| < 1 \rightarrow \ln(1 + x)$
- $x > -1 \rightarrow \frac{x}{1+x} \leq \ln(1 + x) \leq x$

- $a > 0 \rightarrow \lim_{n \rightarrow \infty} \frac{\log^b n}{n^a} = 0 \rightarrow \log^b n = o(n^a)$
- $\log_b a = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a b}$
- $n! = o(n^n)$
- $n! = \omega(2^n)$
- $\sum_{i=1}^n i \frac{n(n+1)}{2} = \Theta(n^2)$
- $\sum_{i=1}^n i^2 \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$
- $\sum_{i=0}^n x^i \frac{x^{n+1}-1}{x-1}$

### Equazioni di ricorrenza

Le equazioni di ricorrenza sono necessarie per valutare la complessità di un algoritmo ricorsivo. Tali algoritmi hanno una funzione di costo anch'essa ricorsiva che deve essere riformulata in modo che non sia più ricorsiva, altrimenti il costo asintotico non può essere quantificato. Tale riformulazione si fa con le equazioni di ricorrenza.

Es. Un algoritmo che prende un vettore  $n$  ed esegue un test (o un if, operazione di complessità costante), se tale test non è soddisfatto effettua una chiamata ricorsiva su un vettore di  $(n-1)$  elementi. La complessità di tale algoritmo si esprime con

$$\begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

Dove il primo addendo è la chiamata ricorsiva, mentre il secondo è tutto quello che viene eseguito al di fuori di questa chiamata; infine il termine di sotto è il caso base.

Esistono alcuni metodi utili per risolvere le equazioni di ricorrenza, di seguito elencati quelli principali.

**METODO DI SOSTITUZIONE:** si applica solo se si è in grado di indovinare la soluzione.

- Si ipotizza una soluzione per l'equazione di ricorrenza data
- Si verifica se essa funziona usando l'induzione matematica per trovare le costanti

Es. Prendiamo l'algoritmo ricorsivo di sopra e ipotizziamo la soluzione  $T(n) = cn$  per una costante  $c$ . Sostituendo ottengo  $T(n) = cn = c(n-1) + \Theta(1)$ , mentre includendo il caso base ho

$$\begin{cases} T(n) = T(n-1) + c \\ T(1) = d \end{cases}$$

Dove  $c$  e  $d$  sono due costanti fissate entrambe  $= \Theta(1)$  ma con nome diverso per differenziarle.

Ipotizziamo ora la soluzione  $T(n) = O(n)$ , ossia  $T(n) \leq kn$  dove  $k$  è una costante che va ancora determinata.

Sostituiamo dapprima la soluzione ipotizzata nel caso base. Si ottiene:  $T(1) \leq k$ ; poiché sapevamo che  $T(1) = d$ , la disuguaglianza è soddisfatta se e solo se  $k \geq d$ . Se sostituiamo la soluzione nella formulazione ricorsiva dell'equazione di ricorrenza otteniamo invece:  $T(n) \leq k(n-1) + c = kn - k + c \leq kn$ . L'ultima disuguaglianza è vera se e solo se  $k \geq c$ .

L'ultima disuguaglianza è vera se e solo se  $k \geq c$ . La nostra soluzione è dunque corretta per tutti i valori di  $k$  tali che  $k \geq c$  e  $k \geq d$ . Poiché un tale valore di  $k$  esiste sempre, una volta fissati  $c$  e  $d$ , possiamo concludere che  $T(n)$  è un  $O(n)$ .

Per rendere il nostro risultato stretto, ipotizziamo ora la soluzione  $T(n) = \Omega(n)$ , ossia  $T(n) \geq hn$  dove  $h$  è una costante che va ancora determinata. In modo assolutamente analogo al caso  $O$ , sostituiamo dapprima la soluzione ipotizzata nel caso base ottenendo  $h \leq d$ , e poi nella formulazione ricorsiva dell'equazione di ricorrenza ottenendo invece:

$T(n) \geq h(n-1) + c = hn - h + c \geq hn$  vera se e solo se  $h \leq c$ . Deduciamo dunque che  $T(n)$  è un  $\Omega(n)$  poiché è possibile trovare un valore  $h$  ( $h \leq c$ ,  $h \leq d$ ) per il quale si ha  $T(n) \geq hn$ .

Dalle due soluzioni:  $T(n) = O(n)$  e  $T(n) = \Omega(n)$ , si ottiene ovviamente che  $T(n) = \Theta(n)$ .

**METODO ITERATIVO (O TELESOPING):** consiste nell'espandere la ricorrenza sino ad esprimerla come somma di termini dipendenti da  $n$ . In generale le funzioni del tipo  $T(n) = T(n-1) + f(n)$  si risolvono con la seguente formula

$$T(n) = \sum_{i=1}^n f(i)$$

Es.  $T(n) = T(n-1) + \Theta(1)$  in questo caso srotoliamo ripetutamente la parte destra dell'equazione e otteniamo

$$T(n) = T(n-1) + \Theta(1) \rightarrow T(n) = T(n-2) + \Theta(1) + \Theta(1) \rightarrow T(n) = T(n-3) + \Theta(1) + \Theta(1) + \Theta(1) \rightarrow \text{ecc.}$$

fino ad ottenere  $T(n) = n \Theta(1) = \Theta(n)$ . Come si vede, otteniamo una soluzione identica a quella trovata col metodo di sostituzione.

Es.  $T(n) = T(n-1) + n \rightarrow T(n) = \sum_{i=1}^n f(i) = \sum_{i=1}^n i = \frac{n(n+1)}{2}$

Es.  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

- $n = b^m$  ovvero  $m = \log_b n$
- $S(m) = T(b^m) \rightarrow S(m) = aS(m-1) + f(b^m)$
- $R(m) = \frac{S(m)}{a^m} \rightarrow R(m) = R(m-1) + \frac{f(b^m)}{a^m}$
- $R(m) = \sum_{i=0}^m \frac{f(b^i)}{a^i}$
- $T(n) = a^m \sum_{i=0}^m \frac{f(b^i)}{a^i}$

Es.  $T(n) = 3T\left(\frac{n}{2}\right) + n$

- $n = 2^m$  ovvero  $m = \log_2 n$
- $R(m) = \sum_{i=0}^m \frac{2^i}{3^i} \geq 3$
- $S(m) \leq 3 \cdot 3^m$
- $T(n) \leq 3 \cdot 3^{\log_2 n} = O(n^{\log_2 3})$

**METODO DELL'ALBERO DI RICORSIONE:** la sua costruzione è una tecnica per rappresentare graficamente lo sviluppo della complessità generata dall'algoritmo stesso, in modo da poterla valutare con più facilità. Essi sono particolarmente utili nell'applicazione del metodo iterativo. Ogni nodo dell'albero è relativo alla soluzione di un problema di una certa dimensione.

Es. Deriviamo l'albero dalla ricorrenza  $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2)$  costruendo la radice coi suoi figli:

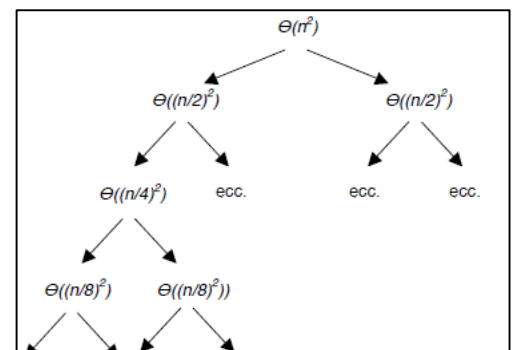
- Nella radice c'è tutta la parte dell'algoritmo che non è ricorsiva
- Si aggiunge un figlio per ogni chiamata ricorsiva effettuata
- Si itera il procedimento su ciascuno dei figli fino ai case base

Una volta completato l'albero, la complessità è data dalla somma delle complessità di tutti i livelli (cioè le "righe" in cui sono disposti i nodi) di cui è costituito l'albero.

- Livello 1:  $\Theta(n^2)$
- Livello 2:  $2\Theta\left(\frac{n}{2}\right)^2 = \Theta\left(\frac{n^2}{2}\right)$
- Livello 3:  $4\Theta\left(\frac{n}{4}\right)^2 = \Theta\left(\frac{n^2}{4}\right)$
- ...
- Livello i:  $2^{i-1}\Theta\left(\frac{n}{2^{i-1}}\right)^2 = \Theta\left(\frac{n^2}{2^{i-1}}\right)$

Il contributo totale è quindi:

$$\sum_{i=1}^{\log n+1} \Theta\left(\frac{n^2}{2^{i-1}}\right) = n^2 \sum_{j=0}^{\log n} \Theta\left(\frac{1}{2^j}\right) = \Theta(n^2)$$



**METODO DEL TEOREMA MASTER (O TEOREMA PRINCIPALE):** fornisce soluzioni per le equazioni di ricorrenza della forma

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

dove  $a \geq 1, b > 1$  sono delle costanti ed  $f(n)$  è una funzione asintoticamente positiva. Data l'equazione sopra, valgono le seguenti proprietà:



1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  per qualche  $\epsilon > 0$ , allora  $T(n) = \Theta(n^{\log_b a})$
2. Se  $f(n) = O(n^{\log_b a})$ , allora  $T(n) = \Theta(n^{\log_b a} \lg n)$
3. Se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  per qualche  $\epsilon > 0$ , e se  $af\left(\frac{n}{b}\right) \leq cf(n)$  per qualche costante  $c < 1$  e  $n$  sufficientemente grande, allora  $T(n) = \Theta(f(n))$

**Osservazione:** il caso due può essere generalizzato come segue. Se  $f(n) = \Theta(n^{\log_b a} \lg^k n)$ , con  $k \geq 0$ , allora  $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ .

Quanto sopra, in ciascuno dei tre casi vengono confrontati fra loro la sola parte  $f(n)$  e  $n^{\log_b a}$ . Informalmente, il teorema principale ci dice che vince il maggiore fra  $f(n)$  e  $n^{\log_b a}$ , ossia la complessità è governata dal maggiore dei due:

- Se il più grande dei due è  $n^{\log_b a}$ , siamo nel caso 1 e la complessità è  $\Theta(n^{\log_b a})$ ;
- Se sono uguali, siamo nel caso 2 e si moltiplica  $f(n)$  per un fattore logaritmico;
- Se il più grande dei due è  $f(n)$ , siamo nel caso 3 e la complessità è  $\Theta(f(n))$ .

Es.  $T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n)$   $a = 9, b = 3, f(n) = \Theta(n), n^{\log_b a} = n^{\log_3 9} = n^2$ .  
Poiché  $f(n) = \Theta(n^{\log_3 9 - \epsilon})$  con  $\epsilon = 1$ , siamo nel caso 1 per cui  $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Es.  $T(n) = T\left(\frac{2}{3}\right) + \Theta(1)$   $a = 1, b = \frac{3}{2}, f(n) = \Theta(1), n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ .  
Poiché  $f(n) = \Theta(n^{\log_b a})$ , siamo nel caso 2 per cui  $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$

Es.  $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$   $a = 3, b = 4, n^{\log_b a} = n^{\log_4 3}$ .  
Poiché  $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon}) \forall \epsilon \leq 1 - \log_4 3$  e inoltre  $af\left(\frac{n}{b}\right) = 3 \cdot \frac{n}{4} \log \frac{n}{4} \leq \frac{3}{4} n \lg n$  con  $c = \frac{3}{4}$ , siamo nel caso 3 per cui  $T(n) = \Theta(n \lg n)$ .

### Algoritmi di ordinamento e loro complessità

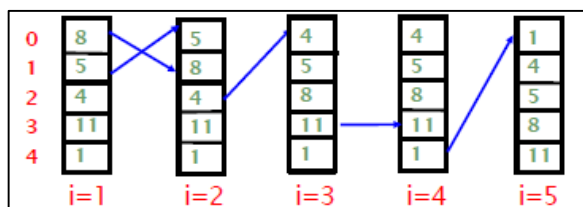
Gli ordinamenti costituiscono una classe molto importante tra gli algoritmi di uso comune. Essi consentono di riordinare una grande quantità di dati in base ad alcune chiavi (per esempio numeri o stringhe).

Il confronto tra la complessità degli algoritmi di ordinamento avviene valutando due importanti proprietà (indipendenti dalla macchina), che sono il numero dei confronti tra le chiavi (#confronti) e il numero di spostamenti di dati (#spostamenti). Essi possono non coincidere.

Poiché la complessità dello stesso algoritmo può variare a seconda della proprietà che si considera, la scelta dell'algoritmo da usare va valutata in relazione alla dimensione dei dati ed al tipo di chiavi.

**INSERTION SORT:** ordina gli elementi considerando un elemento alla volta, lo inserisce in un sottogruppo che viene costruito già ordinato e per ogni nuovo elemento viene ricercata la posizione all'interno della parte ordinata, slittando gli elementi per creare uno spazio libero.

- *passo 1:* si confrontano i primi 2 elementi e si spostano se in ordine inverso;
- *passo i:* dal terzo elemento in poi, si seleziona l'elemento di posto  $i$  e si inserisce nella giusta posizione nel sottoarray ordinato dal posto 0 al posto  $i-1$  slittando gli elementi per creare uno spazio libero.



Il vantaggio dell'ordinamento per inserimento è che l'array viene ordinato solo quando è realmente necessario. Lo svantaggio è che l'algoritmo non si accorge degli elementi che sono nella posizione corretta questi potrebbero venire spostati dalle loro posizioni per poi ritornarvi successivamente e questo può dar luogo a spostamenti ridondanti.

Pseudocodice	Costo	Numero di esecuzioni
Funzione Insertion_Sort(A[1..n])		
1     for j = 2 to n do	$c_1 = \Theta(1)$	$n$
2         x ← A[j]	$c_2 = \Theta(1)$	$n - 1$
3         i ← j - 1	$c_3 = \Theta(1)$	$n - 1$
4         while ((i > 0) and (A[i] > x))	$c_4 = \Theta(1)$	$\sum_{j=2}^n t_j$
5             A[i+1] ← A[i]	$c_5 = \Theta(1)$	$\sum_{j=2}^n (t_j - 1)$
6             i ← i - 1	$c_6 = \Theta(1)$	$\sum_{j=2}^n (t_j - 1)$
7         A[i+1] ← x	$c_7 = \Theta(1)$	$n - 1$

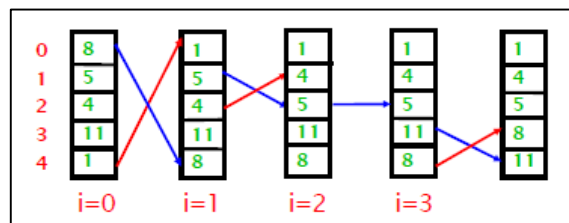
Nella terza colonna  $t_j$  denota il numero di volte che il test del while viene eseguito per quel valore di j.

- ❖ Caso medio:  $T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$
- ❖ Caso migliore:  $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$   
Esso si verifica quando il numero di spostamenti da effettuare è minimo, cioè quando il vettore è già ordinato (in senso non decrescente); in questo caso  $T(n)$  è lineare ( $n$ )
- ❖ Caso peggiore:  $T(n) = \frac{1}{2}(c_5 + c_6 + c_7)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$   
Esso si verifica quando il numero di spostamenti da effettuare è massimo, cioè quando il vettore di partenza è ordinato all'incontrario (in senso decrescente); in questo caso  $T(n)$  è quadratico ( $n^2$ )

Essendo diverse le complessità asintotiche del caso peggiore e del caso migliore, diremo che questo algoritmo è *input sensitive*.

**SELECTION SORT:** l'algoritmo seleziona di volta in volta il record con chiave minima (oppure massima), spostandolo nella posizione corretta.

- *passo 1:* il record con chiave più bassa viene selezionato e scambiato con l'elemento nella prima posizione;
- *passo 2:* tra i record rimanenti, si cerca quello di chiave minima e si scambia con il record in seconda posizione;
- *passo i:* tra i record dalla posizione i alla posizione n-1, si cerca quello di chiave minima e si scambia con il record in posizione i.



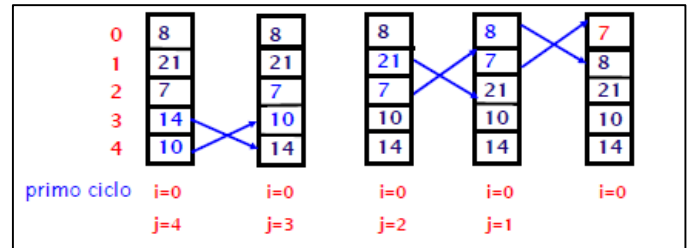
Poiché ogni elemento è spostato massimo una volta, questo algoritmo è da preferire quando si devono ordinare insiemi di record molto grandi con chiavi piccole.

Pseudocodice	Costo	Numero di esecuzioni
Funzione Selection_Sort(A[1..n])		
1     for i = 1 to n - 1 do	$c_1 = \Theta(1)$	$n$
2         m ← i	$c_2 = \Theta(1)$	$n - 1$
3         for j = i + 1 to n do	$c_3 = \Theta(1)$	$\sum_{i=1}^{n-1} (n - i + 1)$
4             if (A[j] < A[m])	$c_4 = \Theta(1)$	$\sum_{i=1}^{n-1} (n - i)$
5             m ← j	$c_5 = O(1)$	$\sum_{i=1}^{n-1} (n - i)$
6         Scambia A[m] e A[i]	$c_6 = \Theta(1)$	$n - 1$

La complessità di questo algoritmo non dipende in alcun modo dalla distribuzione iniziale dei dati nel vettore, cioè resta la stessa sia che il vettore sia inizialmente ordinato o no. Essa è:

$$T(n) = c_{1n} + (c_2 + c_3 + c_6)(n - 1) + (c_3 + c_4 + c_5)\Theta(n^2) = \Theta(n) + \Theta(n^2) + \Theta(n^2)$$

**BUBBLE SORT:** confronta elementi consecutivi ( $j$  e  $j-1$ ) iniziando da destra, scambiandoli se non li trova in ordine. Al termine del primo ciclo viene così trovato il minimo, che galleggia in cima all'array. Al  $i$ -esimo passaggio viene trovato il  $i$ -esimo elemento più piccolo, posizionandolo all'  $i$ -esimo posto.



Pseudocodice	Costo	Numero di esecuzioni
Funzione Bubble_Sort(A[1..n])		
1     for i = 1 to n do	$c_1 = \Theta(1)$	$n + 1$
2         for j = n downto i + 1 do	$c_2 = \Theta(1)$	$\sum_{i=1}^n (n - i + 1)$
4             if (A[j] < A[j - 1])	$c_3 = \Theta(1)$	$\sum_{i=1}^n (n - i)$
6                 Scambia A[j] e A[j - 1]	$c_4 = O(1)$	$\sum_{i=1}^n (n - i)$

In questo caso la complessità è  $T(n) = c_1(n + 1) + c_2n + (c_2 + c_3 + c_4)\frac{(n-1)n}{2} = \Theta(n) + \Theta(n^2) = \Theta(n^2)$

**MERGE SORT:** algoritmo ricorsivo che adotta una tecnica algoritmica detta *divide-et-impera*, ovvero:

- il problema principale si divide in sottoproblemi di dimensione minore (*divide*);
- i sottoproblemi si risolvono ricorsivamente (*impera*);
- le soluzioni dei sottoproblemi si fondono per ottenere un'unica soluzione al problema complessivo (*combina*).

La complessità di un algoritmo divide-et-impera è  $T(n) = aT\left(\frac{n}{b}\right) + D(n) + C(n)$  dove  $a$  è il numero dei sottoproblemi,  $n/b$  la dimensione di ciascun problema,  $c$  la soglia al di sotto della quale non c'è ricorsione,  $D(n)$  il tempo per divide il problema,  $C(n)$  il tempo per combinare le soluzioni dei sottoproblemi.

Il merge sort quindi funziona nel seguente modo:

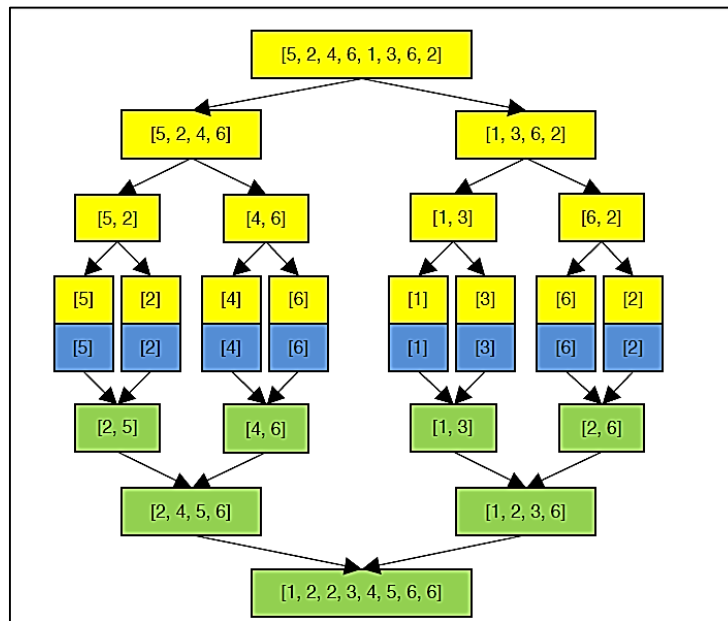
- *passo 1 (divide)*: la sequenza di  $n$  elementi viene divisa in due sottosequenze di  $n/2$  elementi ciascuna;
- *passo 2 (impera)*: le due sottosequenze vengono ordinate ricorsivamente, ma se sono ancora grandi vengono divise a sua volta;
- *passo base*: la ricorsione termina quando la sottosequenza ha solo un elemento;
- *passo i (combina)*: le due sottosequenze ormai ordinate di  $n/2$  elementi ciascuna vengono fuse in un'unica sequenza ordinata di  $n$  elementi.

```

Funzione Merge_sort (A: vettore; indice_primo, indice_ultimo: intero)
    if (indice_primo < indice_ultimo)
        indice_medio ← ⌊(indice_primo + indice_ultimo) / 2⌋
        Merge_sort (A, indice_primo, indice_medio)
        Merge_sort (A, indice_medio + 1, indice_ultimo)
        Fondi (A, indice_primo, indice_medio, indice_ultimo)
    return

```

Es. Funzionamento del merge sort su un problema di 8 elementi



- caselle gialle: operazioni di suddivisione del vettore in sottovettori tramite le chiamate ricorsive
- caselle azzurre: indicano i casi base
- caselle verdi: chiamate della funzione *Fondi* le quali restituiscono porzioni di vettore ordinate sempre più grandi

```
Funzione Fondi (A: vettore; indice_primo, indice_medio, indice_ultimo: intero)
1   i ← indice_primo;
2   j ← indice_medio + 1;
3   k ← 1;
4   while ((i ≤ indice_medio) and (j ≤ indice_ultimo))
5       if (A[i] < A[j])
6           B[k] ← A[i]
7           i ← i + 1
8       else
9           B[k] ← A[j]
10          j ← j + 1
11         k ← k + 1
12     while (i ≤ indice_medio) //il primo sottovettore non è terminato
13         B[k] ← A[i]
14         i ← i + 1
15         k ← k + 1
16     while (j ≤ indice_ultimo) //il secondo sottovettore non è terminato
17         B[k] ← A[j]
18         j ← j + 1
19         k ← k + 1
20     ricopia B[1..k-1] su A[indice_primo..indice_ultimo]
21     return
```

La complessità di questo algoritmo è  $\Theta(n)$  per la fusione e  $2T\left(\frac{n}{2}\right)$  per la chiamata ricorsiva, quindi la complessità totale è  $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$  la quale ricade nel secondo caso del teorema master, la cui soluzione è  $T(n) = \Theta(n \log n)$ .

*Osservazione:* nonostante il Merge Sort funzioni in tempo  $O(n \log n)$  mentre l'Insertion Sort in  $O(n^2)$ , i fattori costanti sono tali che l'Insertion Sort è più veloce del Merge Sort per valori piccoli di  $n$ . Quindi, ha senso usare l'Insertion Sort dentro il Merge Sort quando i sottoproblemi diventano sufficientemente piccoli. Si può quindi modificare il caso base del merge sort come segue:

```
Funzione Merge_Insertion (primo, ultimo, dim)
  if dim > k
    medio ← ⌊(primo+ultimo)/2⌋
    Merge_Insertion (primo, medio, medio-primo+1)
    Merge_Insertion (medio+1, ultimo, ultimo-primo)
    Fondi(primo, medio, ultimo)
  else Insertion
```

La cui complessità è  $O = (\log n)$ .

**QUICK SORT:** algoritmo basato anche sul paradigma divide-et-impera.

- *Divide:* nella sequenza di  $n$  elementi in input viene selezionato un elemento detto **pivot**; la sequenza viene divisa in due sottosequenze dove la prima ha elementi minori o uguali al pivot, la seconda maggiori o uguali;
- *Passo base:* la ricorsione procede fino a quando le sottosequenze sono costituite da un solo elemento;
- *Impera:* le due sottosequenze vengono ordinate ricorsivamente;
- *Combina:* non occorre poiché la sequenza {elem.minori\pivot\elem.maggiori} è già ordinata

Lo pseudocodice dell'algoritmo quicksort è il seguente:

```
Funzione Quick_sort (A: vettore; indice_primo, indice_ultimo: intero)
  if (indice_primo < indice_ultimo)
    then
      indice_medio ← Partiziona (A, indice_primo, indice_ultimo)
      Quick_sort (A, indice_primo, indice_medio)
      Quick_sort (A, indice_medio + 1, indice_ultimo)
  return
```

Invece quello della funzione partizionamento è il seguente:

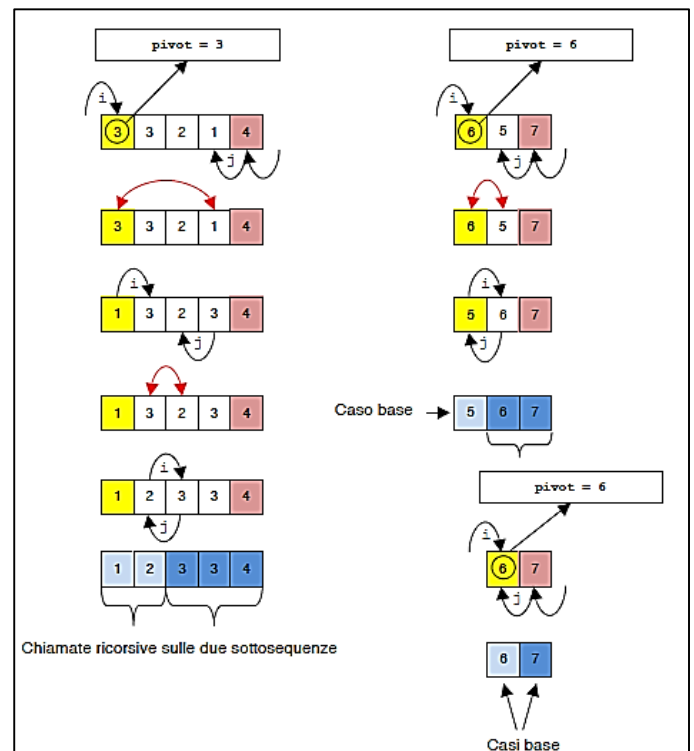
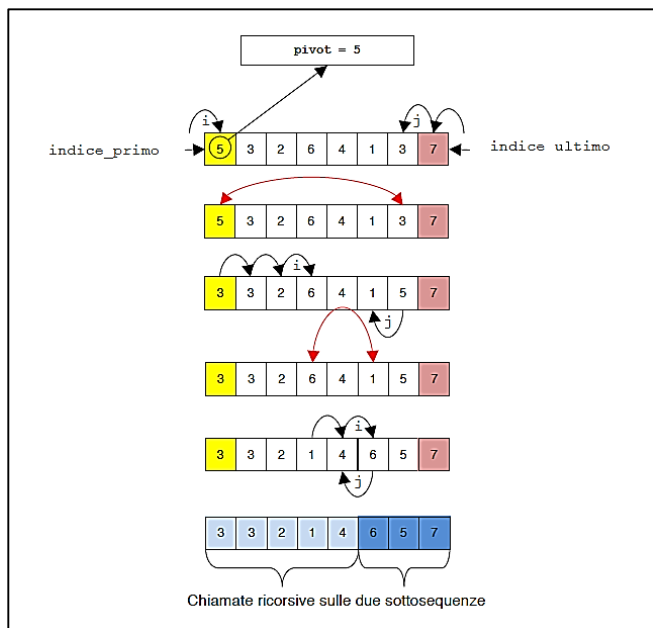
```

Funzione Partiziona (A: vettore; indice_primo, indice_ultimo: intero)

1  pivot ← A[indice_primo] //scelta arbitraria
2  i ← indice_primo - 1;
3  j ← indice_ultimo + 1
4  while true
5      repeat
6          j ← j - 1
7          until A[j] ≤ pivot
8      repeat
9          i ← i + 1
10         until A[i] ≥ pivot
11         if (i < j)
12             scambia A[i] e A[j]
13         else
14             return j

```

La **funzione partizionamento** sceglie come pivot il primo valore dell'array e inserisce due indici,  $i$  all'inizio dell'array e  $j$  alla fine; all'interno del while,  $j$  viene decrementato e  $i$  viene incrementato fino a che  $j \leq pivot$  e  $i \geq pivot$ ; in tal caso vengono scambiati, le due regioni vengono estese e il procedimento si ripete all'interno del while terminando solo quando i due indici coincidono.



La complessità del partizionamento è  $\Theta(n)$ . Il partizionamento suddivide la sequenza di  $n$  elementi in due sottosequenze di dimensioni  $k$  e  $(n-k)$  con  $1 \leq k \leq n-1$ ; in particolare delle due sequenze è costituita di un solo elemento ogniqualevolta il valore del pivot è strettamente minore o strettamente maggiore di tutti gli altri elementi della sequenza. L'equazione di ricorrenza del quicksort va dunque espressa come segue:

$$T(n) = T(k) + T(n - k) + \Theta(n)$$

- *Caso peggiore*: Il caso peggiore è quello in cui, ad ogni passo, la dimensione di uno dei due sottoproblemi da risolvere è 1. In tale situazione l'equazione di ricorrenza è  $T(n) = T(n - 1) + \Theta(n)$  al quale applicando il metodo iterativo otteniamo  $\Theta(n^2)$ .
- *Caso migliore*: quello in cui, ad ogni passo, la dimensione dei due sottoproblemi è identica. In tale situazione l'equazione di ricorrenza è  $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$  che ha soluzione  $\Theta(n \log n)$ .
- *Caso medio*: valutato nell'ipotesi che il valore del pivot suddivida con uguale probabilità  $1/n$  la sequenza da ordinare in due sottosequenze di dimensioni  $k$  ed  $(n - k)$ , per tutti i valori di  $k$  compresi fra 1 ed  $(n - 1)$ . La complessità diviene  $T(n) = \frac{1}{n} \left[ \sum_{k=1}^{n-1} (T(k) + T(n - k)) \right] + \Theta(n)$  che attraverso vari passaggi algebrici e con il metodo di sostituzione diventa  $T(n) = \Theta(n \log n)$ .

La scelta del pivot è indifferente nel caso questa scelta sia equiprobabile (evento che ha la stessa probabilità di verificarsi di un altro), ma a volte ciò non accade, ad esempio quando i valori in input sono poco disordinati, e le prestazioni dell'algoritmo degradano. Per ovviare a tale inconveniente si possono adottare delle tecniche volte a *randomizzare* la sequenza da ordinare, cioè volte a disgregarne l'eventuale regolarità interna. Esistono due tipi di tecniche di randomizzazione dell'algoritmo quicksort, le quali mirano a rendere l'algoritmo indipendente dall'input, e quindi consentono di ricadere nel caso medio con la stessa probabilità che nel caso di ipotesi di equiprobabilità del pivot:

- 1) *QuickSort Randomizzato*: prima di avviare l'algoritmo di ordinamento, all'input viene applicata una permutazione casuale degli elementi;
- 2) *Partizionamento Randomizzato*: il partizionamento sceglie casualmente il pivot dalla sequenza anziché quello più a sinistra.

RANDOMIZED-PARTITION( $A, p, r$ )	RANDOMIZED-QUICKSORT( $A, p, r$ )
1 $i = \text{RANDOM}(p, r)$	1 <b>if</b> $p < r$
2 <b>exchange</b> $A[r]$ with $A[i]$	2 $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
3 <b>return</b> PARTITION( $A, p, r$ )	3 $\text{RANDOMIZED-QUICKSORT}(A, p, q - 1)$
	4 $\text{RANDOMIZED-QUICKSORT}(A, q + 1, r)$

Di seguito una tabella con la **COMPLESSITÀ DEGLI ALGORITMI DI ORDINAMENTO** stilata tra i più famosi.

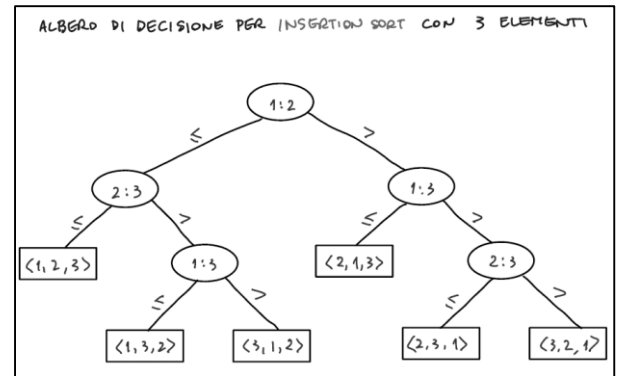
Nome	Migliore	Medio	Peggior
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bucket sort	$O(n+m)$	$O(n+m)$	$O(n+m)$
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion sort	$O(n)$	$O(n + d)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Radix sort	$O(n \cdot k/s)$	$O(n \cdot k/s)$	$O(n \cdot k/s)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Shell sort	—	—	$O(n^{1.5})$

Un **ALBERO DI DECISIONE** è uno strumento che stabilisce un limite di complessità al di sotto del quale nessun algoritmo di ordinamento basato su confronti fra coppie di elementi può andare (fissato a  $\Omega(n \lg n)$ ). Esso è un albero binario completo che rappresenta i confronti fra elementi che vengono effettuati da un particolare algoritmo di ordinamento che opera su un input di una data dimensione.

Vengono rappresentate tutte le strade che la computazione di un algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.

Ogni test ha due soli esiti essendo booleani, quindi l'albero di decisione ha queste proprietà:

- è un albero binario che rappresenta tutti i possibili confronti dell'algoritmo; ogni nodo interno (nodo senza foglie) ha esattamente due figli;
- ogni nodo interno (indicato con  $i:j$ ) rappresenta un singolo confronto, e i due figli del nodo sono relativi ai due possibili esiti del test;
- ogni foglia rappresenta una possibile soluzione al problema data dalla permutazione degli elementi in ingresso;
- gli altri aspetti dell'algoritmo che non siano i test vengono ignorati.



Eseguire l'algoritmo corrisponde a tracciare un *cammino* dalla radice dell'albero alla foglia che contiene la soluzione; la discesa è governata dagli esiti dei confronti che vengono via via effettuati. La *lunghezza* (ossia il numero degli archi o rami) di tale cammino rappresenta il numero di confronti necessari per trovare la soluzione. Dunque, la lunghezza del percorso più lungo dalla radice ad una foglia (*altezza* dell'albero binario) rappresenta il numero di confronti che l'algoritmo deve effettuare nel caso peggiore.

Ora, dato che la sequenza di ingresso può avere una qualunque delle sue permutazioni come soluzione, l'albero di decisione deve contenere nelle foglie tutte le permutazioni della sequenza in ingresso, che sono  $n!$  per un problema di dimensione  $n$ . D'altra parte, un albero binario di altezza  $h$  non può contenere più di  $2^h$  foglie. Quindi l'altezza  $h$  dell'albero di decisione di qualunque algoritmo di ordinamento basato su confronti deve essere tale per cui:

$$2^h \geq n! \rightarrow h \geq \log n!$$

Grazie all'approssimazione di Stirling sappiamo che  $n! \approx \Theta(\sqrt{nn^n})$  e quindi  $\log n! = \Theta(\log \sqrt{nn^n}) =$

$\Theta(\log n^{n+\frac{1}{2}}) = \Theta\left(\left(n + \frac{1}{2}\right) \log n\right) = \Theta(n \log n)$ . Dunque  $h \geq \Theta(n \log n)$ , per cui si deduce il seguente *teorema*: la complessità di qualunque algoritmo di ordinamento basato su confronti è  $\Omega(n \log n)$ .

### Heap

Un heap (o heap binario) è una struttura dati che si basa su un albero binario quasi completo (tutti i suoi livelli tranne l'ultimo sono completi, quindi i nodi sono addensati a sinistra) con la proprietà che la chiave su ogni nodo è maggiore o uguale alla chiave dei suoi figli (ordinamento verticale). Ne consegue che il valore massimo è nella radice.

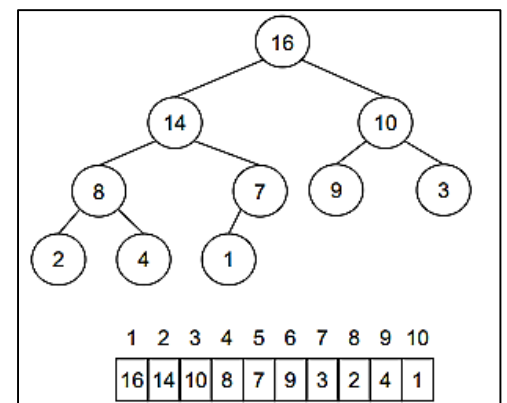
Gli heap sono **IMPLEMENTATI NELLA MEMORIA** del calcolatore come array, la radice viene memorizzata nell'elemento di indice 1 (il primo del vettore), i due figli dell'elemento in posizione  $i$  vengono memorizzati negli elementi in posizione  $2i$  (figlio sinistro) e  $2i+1$  (figlio destro). Indicheremo con *heap\_size* il numero complessivo di nodi nell'heap.

- L'array è riempito a partire da sinistra; se ha più elementi di *heap\_size* allora gli elementi  $\text{indice} > \text{heap\_size}$  non fanno parte dell'heap;
- Ogni nodo dell'albero binario corrisponde ad uno ed uno solo elemento dell'array  $A$ ;
- $A[1]$  è la radice;
- $A[2i]$  e  $A[2i+1]$  sono i figli sinistro e destro di  $A[i]$ ;
- $A[i/2]$  è il padre di  $A[i]$ .

Con questa implementazione, la proprietà di ordinamento verticale implica che per tutti gli elementi (tranne per  $A[1]$ , poiché è la radice e non ha genitore) vale la regola

$$A[i] \leq A[\text{parent}(i)]$$

Se per ogni  $i$ ,  $A[i] \geq \max(A[2i], A[2i+1])$ , allora l'array viene detto **MAX-HEAP** (Il massimo è la radice, il minimo una delle foglie). Se per ogni  $i$ ,  $A[i] \leq \min(A[2i], A[2i+1])$ , allora l'array viene detto **MIN-HEAP** (Il minimo è la radice, il massimo una delle foglie).





L'**ALTEZZA DI UN HEAP** è il numero di archi del cammino semplice più lungo dalla radice ad una foglia dell'heap. L'altezza di un heap con  $n$  elementi è  $\lfloor \log n \rfloor$  (dunque la complessità è  $\Theta(\log n)$ ).

Livello	#nodi ( $n_i$ )
0	1
1	2
2	$2^2$
...	...
$h-1$	$2^{h-1}$
$h$	$1 \leq n_h \leq 2^h$

- *Dimostrazione.* Sia  $h$  l'altezza di un heap con  $n$  elementi. Si ha:  $n = \sum_{i=0}^h n_i = \sum_{i=0}^{h-1} 2^i + n_h = 2^h - 1 + n_h$  ora  $2^h \leq 2^h - 1 + n_h \leq 2^h - 1 + 2^h = 2^{h+1} - 1$  cioè  $2^h \leq n < 2^{h+1}$   $h \leq \log n < h + 1$  dunque  $h = \lfloor \log n \rfloor$ .
- *Lemma.* Se  $2^{\lfloor \log n \rfloor} \leq i \leq n$ , il nodo  $i$  è una foglia.
- *Dimostrazione.* Sia  $h$  l'altezza di un heap con  $n$  elementi; si osservi che i nodi a livello  $h = \lfloor \log n \rfloor$  sono foglie, questi hanno un indice  $i$  tale che  $n \geq i \geq \sum_{i=0}^{h-1} 2^i + 1 = 2^h - 1 + 1 = 2^h = 2^{\lfloor \log n \rfloor}$ .

Da tale struttura dati è nato **HEAP SORT**, un algoritmo di ordinamento avente complessità  $O(n \log n)$  anche nel caso peggiore, opera sul posto e opera sulla struttura dati heap poiché sfrutta al meglio le sue proprietà, il cui mantenimento è essenziale per il corretto funzionamento dell'algoritmo, e anche perché gestisce così le code di priorità. Si basa su tre componenti distinte:

1. *Heapify*: ripristina la proprietà di heap; ha tempo  $O(\log n)$  ;
2. *Build\_heap*: trasforma un vettore disordinato in un heap sfruttando Heapify; ha tempo  $O(n)$ ;
3. *HeapSort*: algoritmo di ordinamento vero e proprio che ordina il vettore sfruttando Build\_heap; richiede tempo  $O(n \log n)$

**HEAPIFY** è una funzione avente lo scopo di mantenere la proprietà di heap. Essa se trova un figlio maggiore della radice, scambia la radice col maggiore dei suoi due figli; questo scambio può rendere non più soddisfatta la proprietà di heap nel sottoalbero coinvolto nello scambio (ma non nell'altro), per cui è necessario riapplicare ricorsivamente la funzione a tale sottoalbero. In sostanza, la funzione muove il valore della radice lungo un opportuno cammino verso il basso finché tutti i sottoalberi le cui radici si trovano lungo quel cammino risultano avere la proprietà di heap.

```

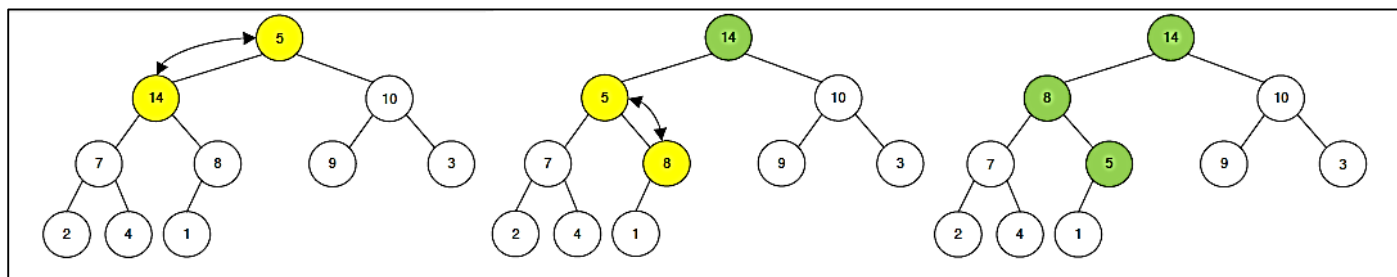
Funzione Heapify (A: vettore; i: intero)

    L ← left(i)
    R ← right(i)
    if ((L ≤ heap_size) and (A[L] > A[i]))
        indice_massimo ← L
    else
        indice_massimo ← i
    if ((R ≤ heap_size) and (A[R] > A[indice_massimo]))
        indice_massimo ← R
    if (indice_massimo ≠ i)
        scambia A[i] e A[indice_massimo]
        Heapify (A, indice_massimo)
    return

```

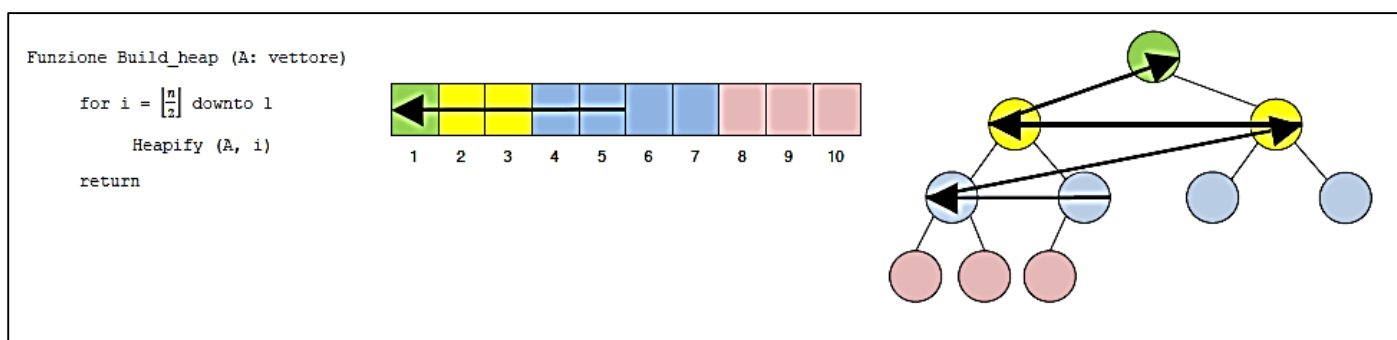
Nel caso peggiore, dato che i sottoalberi della radice non possono avere più di  $\frac{2n}{3}$  nodi, l'equazione di ricorrenza è  $T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$ , che ricade nel caso 2 del teorema principale ed ha soluzione  $T(n) = O(\log n)$ .

*Funzionamento di Heapify*: in giallo i nodi coinvolti nel mantenimento della proprietà di heap e in verde i nodi che sono stati sistemati definitivamente.



Ovviamente la funzione può essere richiamata con **Max-Heapify** per creare un Max-Heap, o con **Min-Heapify** per creare un Min-Heap.

**BUILD\_HEAP** usa la funzione Heapify per trasformare qualunque array di  $n$  elementi in uno heap. Esso lavora sugli elementi dell'albero che non sono foglie (poiché le foglie godono già della proprietà heap di un solo nodo) operando da destra verso sinistra dell'array, dal basso verso l'alto dell'albero per garantire che i sottoalberi di ogni nodo i siano heap prima che lavori sul nodo stesso.



$n/2$  è il massimo degli indici dei nodi interni.

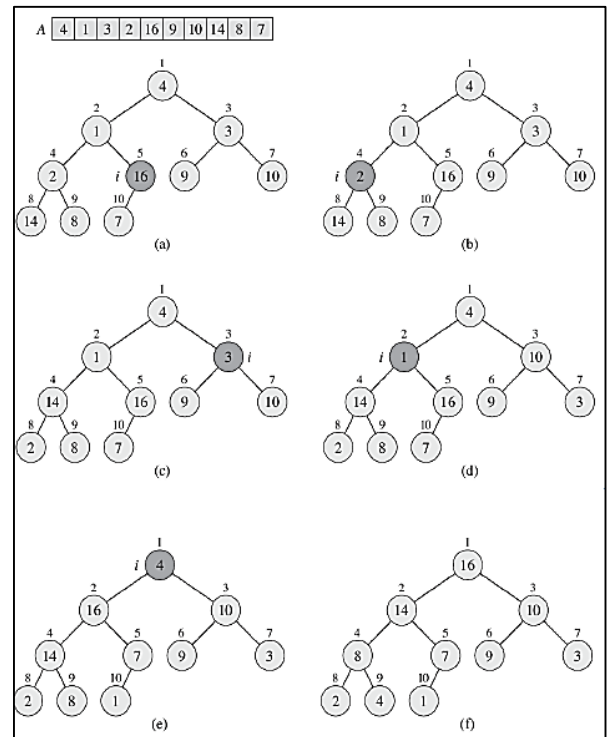
Es. Nella figura a destra, Build\_heap salta le foglie e viene richiamato Heapify sul valore 16, dove non scambia perché è maggiore del figlio 7; Heapify viene richiamato su 2 scambia col figlio maggiore 14; a salire viene Heapify su 3 e scambia col figlio maggiore 10 e così via fino alla radice.

La complessità del Build-Heap è  $O(n \log n)$

Proprietà: in un heap con  $n$  elementi ci sono massimo  $\left\lceil \frac{n}{2^{h+1}} \right\rceil$  nodi di altezza  $h$ .

**HEAP\_SORT** ordina un array seguendo questi passi:

1. Inizia trasformando l'array di dimensione  $n$  in un heap di  $n$  nodi mediante Build\_heap;
2. Prende il massimo del vettore che è in  $A[0]$  e lo mette alla fine in  $A[n]$  (che è la sua corretta posizione);
3. Riduce la dimensione dell'heap da  $n$  a  $n-1$  eliminando quindi l'ultimo elemento;
4. Ripristina la proprietà di heap sui residui  $n-1$  elementi con Heapify;
5. Scambia la nuova radice  $A[0]$  con il nuovo ultimo elemento  $A[n-1]$ ;
6. Riapplica il procedimento diminuendo man mano la dimensione dell'heap fino ad arrivare a 2.



Es. Un vettore composto da  $A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$

<p>1. Per comodità, il vettore ha già la proprietà di heap, non ha quindi bisogno di Build_heap</p>	<p>2. Inizia il primo scambio</p> <p>Il vettore diventa <math>A = [1, 14, 10, 7, 8, 9, 3, 2, 4, 16]</math></p>	<p>3. Rimuove l'ultimo elemento del vettore. Il vettore diventa <math>A = [1, 14, 10, 7, 8, 9, 3, 2, 4]</math></p>
<p>4. Ripristina la proprietà di heap partendo quindi dalla radice</p> <p>Il vettore diventa <math>A = [14, 8, 10, 7, 1, 9, 3, 2, 4]</math></p>	<p>5. Scambia la nuova radice 14 con il nuovo ultimo elemento 4</p>	<p>6. Riapplica il procedimento diminuendo man mano la dimensione dell'heap fino ad arrivare a 2, e riordinando così l'array.</p>

```

Funzione Heapsort (A: vettore)
    Build_heap(A)
    for i = n downto 2
        scambia A[1] e A[i]
        heap_size ← heap_size - 1
        Heapify(A, 1)
    return

```

La complessità di Heapsort è  $O(n \lg n)$  poiché Build\_heap ha complessità  $O(n)$  e ciascuna delle  $n-1$  chiamate di Heapify ha complessità  $O(\log n)$ . Quindi:

$$O(n) + O((n-1) \log n) = O(n \log n)$$

Una delle applicazioni più efficienti della struttura dati heap sono le **CODE DI PRIORITÀ**. Essa è una struttura dati variante della coda dove, come quest'ultima l'inserimento avviene anche ad un'estremità e l'estrazione avviene all'estremità opposta, però, a differenza della coda, nella coda con priorità la posizione di ciascun elemento non dipende dal momento in cui è stato inserito, ma dal valore di una determinata grandezza, detta priorità. Di conseguenza, gli elementi di una coda con priorità sono collocati in ordine crescente (o decrescente, a seconda dei casi) rispetto alla grandezza considerata come priorità.

Dunque una coda di priorità è una struttura dati che mantiene un array di elementi dove a ciascuno viene associato un valore chiamato "*chiave*". Quando un nuovo elemento avente  $key = x$  viene inserito in una coda con priorità (crescente) esso viene collocato come predecessore del primo elemento presente in coda che abbia  $key \geq x$ . Solo se la coda contiene solo elementi con priorità minore di quella dell'elemento che viene inserito, esso diviene l'elemento di testa della coda, il primo quindi che verrà estratto.

Nel caso degli heap, una coda a priorità può essere rappresentato da un albero binario completo, in cui la chiave del padre è sempre maggiore di quella dei figli. La struttura dati Heap gestisce una coda a priorità, dove viene estratto l'elemento a maggiore priorità (es. quello con chiave maggiore).

Dato una coda di priorità  $S$ , le operazioni supportate sono:

- INSERT ( $S, x$ ): inserisce  $x$  in  $S$
- MAXIMUM ( $S$ ): restituisce l'elemento di  $S$  con la chiave massima (maggiore priorità)
- EXTRACT-MAX( $S$ ): estrae da  $S$  l'elemento con la chiave più grande e lo restituisce
- INCREASE-KEY( $S, x, k$ ): aumenta il valore della chiave di  $x$  al nuovo valore  $k$  (con  $k$  non inferiore al valore corrente della chiave di  $x$ )

<b>HEAP-MAXIMUM(<math>A</math>)</b> 1 <b>return</b> $A[1]$ Complessità $O(1)$	<b>HEAP-EXTRACT-MAX(<math>A</math>)</b> 1 <b>if</b> $A.heap-size < 1$ 2 <b>error</b> "heap underflow" 3 $max = A[1]$ 4 $A[1] = A[A.heap-size]$ 5 $A.heap-size = A.heap-size - 1$ 6 <b>MAX-HEAPIFY</b> ( $A, 1$ ) 7 <b>return</b> $max$ Complessità $O(\log n)$	<b>MAX-HEAP-INSERT(<math>A, key</math>)</b> 1 $A.heap-size = A.heap-size + 1$ 2 $A[A.heap-size] = -\infty$ 3 <b>HEAP-INCREASE-KEY</b> ( $A, A.heap-size, key$ ) Complessità $O(\log n)$
---	--	---

Una delle applicazioni più comuni delle code di priorità è quella della schedulazione dei lavori su computer condivisi (per esempio per gestire le code di stampa). La coda di priorità tiene traccia del lavoro da realizzare e la relativa priorità. Quando un lavoro viene eseguito o interrotto, il lavoro con più alta priorità è selezionato da quelli in attesa utilizzando la procedura Extract-Max. Ad ogni istante un nuovo lavoro può essere aggiunto alla coda

### Ordinamento in tempo lineare

Anche se il teorema dell'albero di decisione stabilisce che non sia possibile ordinare  $n$  elementi in un tempo inferiore a  $\Omega(n \log n)$ , in realtà ciò è possibile da attuare purché l'algoritmo di ordinamento non sia basato sui confronti fra elementi, dato che, in tal caso, come abbiamo visto nessun algoritmo può scendere sotto tale limite. I seguenti tre algoritmi presentano operazioni diverse dai confronti per effettuare l'ordinamento.

**COUNTING SORT:** algoritmo basato sul conteggio, opera sotto l'assunzione che ciascuno degli  $n$  elementi da ordinare sia un intero di valore compreso tra  $[0 \dots k]$ . L'algoritmo prende in ingresso un vettore e restituisce un secondo vettore ordinato utilizzando un vettore di appoggio per l'elaborazione. Per ogni elemento  $i$  dell'insieme da ordinare si determinano quanti elementi sono ripetuti e tali valori si mettono sull'array temporaneo.

Es.  $A = [2, 5, 3, 0, 2, 3, 0, 3]$

1. il range di questo array è di  $[0 \dots 5]$ , costruiamo quindi l'array per ora vuoto  $aux = [aux[0], aux[1], aux[2], aux[3], aux[4], aux[5]]$ ;
2. Iniziamo a contare quante volte si ripete il valore 0 e inseriamo nel corrispettivo indice  $aux[0]$ , quindi  $aux = [2, aux[1], aux[2], aux[3], aux[4], aux[5]]$ ;
3. Stessa cosa con il valore 1, quindi  $aux = [2, 0, aux[2], aux[3], aux[4], aux[5]]$ ;
4. E così via fino a trovare quindi  $aux = [2, 0, 2, 3, 0, 1]$ ;
5. A questo punto è facile ricostruire il vettore A ordinato, scrivendo dapprima 2 occorrenze del valore 0, poi 0 occorrenze del valore 1, e così via, fino alla fine del vettore  $A = [0, 0, 2, 2, 3, 3, 3, 5]$ , che è il vettore ordinato.

```

CountingSort(A, B, k)
1 for i ← 1 to k
2 do   C[i] ← 0
3 for j ← 1 to length[A]
4 do   C[A[j]] ← C[A[j]] + 1
5 for i ← 2 to k
6 do   C[i] ← C[i] + C[i-1]
7 for j ← length[A] downto 1
8 do   B[C[A[j]]] ← A[j]
9       C[A[j]] ← C[A[j]] - 1

```

Quest'algoritmo ha una complessità  $\Theta(n + k)$ , dunque se  $k = O(n)$  allora l'algoritmo ordina  $n$  elementi in tempo lineare, cioè  $\Theta(n)$ . Esso è un algoritmo stabile, cioè i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input.

**RADIX SORT:** ordina rispetto ad una sequenza di chiavi; utilizza un procedimento controintuitivo per l'uomo, ma più facilmente implementabile. Esegue gli ordinamenti per posizione della cifra ma partendo dalla cifra meno significativa (unità, poi decine, poi centinaia, ecc.). Questo affinché l'algoritmo non si trova a dovere operare ricorsivamente su sottoproblemi di dimensione non valutabili a priori.

1° PASSO	2° PASSO	3° PASSO	4° PASSO	
253	10	5	5	5
346	253	10	10	10
1034	1034	127	1034	127
10	5	1034	127	253
5	346	346	253	346
127	127	253	346	1034

La sua complessità è di  $O(nk)$  dove  $n$  è la lunghezza dell'array di input e  $k$  è la media del numero di cifre degli  $n$  numeri.

**BUCKET SORT:** ordina valori numerici che si assume siano distribuiti uniformemente in un intervallo semichiuso  $(0, 1)$ . L'intervallo dei valori, noto a priori, è diviso in intervalli più piccoli, detti bucket (cesto). Ciascun valore dell'array è quindi inserito nel bucket a cui appartiene, i valori all'interno di ogni bucket vengono ordinati e l'algoritmo si conclude con la concatenazione dei valori contenuti nei bucket.

Prima parte dell'algoritmo: Divisione nei bucket.	Seconda parte dell'algoritmo: Ordinamento dei bucket e concatenazione.
<p>29 25 3 49 9 37 21 43</p> <p>0-9    10-19    20-29    30-39    40-49</p>	<p>0-9    10-19    20-29    30-39    40-49</p> <p>3 9 21 25 29 37 43 49</p>

```

BucketSort(array A, intero M)
  for i ← 1 to length[A] do
    // restituisce un indice di bucket per l'elemento A[i]
    bucket ← f(A[i], M)
    // inserisce l'elemento A[i] nel bucket corrispondente
    aggiungi(A[i], B[bucket])
  for i ← 1 to M do
    // ordina il bucket
    ordina(B[i])
  // restituisce la concatenazione dei bucket
  return concatena(B)

```

La complessità del bucket sort è lineare  $O(n + m)$ , dove  $m$  è il valore max nell'array, nel caso migliore, in quello peggiore è  $O(n^2)$  questo è possibile in quanto l'algoritmo non è basato su confronti.

### Calcolo delle Probabilità

Slide 05 Cantone

### Mediane e statistiche d'ordine

Slide 09 Cantone

### Dizionari

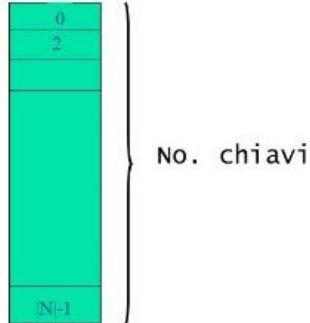
Tutte le strutture dati viste fino ad ora sono strutture dati lineari, con le quali è difficile rappresentare una gerarchia di dati dinamici. Un dizionario è una struttura dati che permette di gestire un insieme dinamico di dati tramite le tre sole operazioni INSERT, SEARCH e DELETE.

Le uniche strutture dati comuni che supportano in modo semplice tutte queste tre operazioni sono gli array, le liste concatenate e le doppie liste concatenate. Infatti, le code (con o senza priorità, inclusi gli heap) e le pile non consentono né la ricerca né l'eliminazione di un arbitrario elemento, mentre negli alberi l'eliminazione di un elemento comporta la disconnessione di una parte dei nodi dall'altra (cosa che può accadere anche nei grafi) e quindi è un'operazione che in genere richiede delle successive azioni correttive.

Di conseguenza, quando l'esigenza è quella di realizzare un dizionario, ossia una struttura dati che rispetti la definizione data sopra, si ricorre a soluzioni specifiche. Noi ne illustreremo tre: *tabelle ad indirizzamento diretto*, *tabelle hash* e (dopo un introduzione generale sugli alberi) *alberi binari di ricerca*.

Nel seguito, verrà dato  $U$  cioè l'insieme dei valori interi che le chiavi possono assumere;  $m$  cioè il numero delle posizioni a disposizione nella struttura dati;  $n$  cioè il numero degli elementi da memorizzare nel dizionario; infine che i valori delle chiavi degli elementi da memorizzare siano tutti diversi fra loro.

**TABELLE AD INDIRIZZAMENTO DIRETTO:** metodo che consiste nel ricorrere a un array nel quale ogni indice corrisponde al valore della chiave dell'elemento memorizzato o da memorizzare in tale posizione. Data l'ipotesi  $n \leq U = m$ , allora un array  $A$  di  $m$  posizioni assolve perfettamente il compito di dizionario, e per giunta con grande efficienza.

- Ogni chiave corrisponde a el. diverso dell'array
  - Può portare a spreco di memoria
  - Es.: 10000 studenti e matr.= No. decimale a 5 cifre
- 

```

Funzione Insert_Indirizz_Diretto (V: vettore; k: chiave)

    V[k] ← dati dell'elemento di chiave k
    return

Funzione Search_Indirizz_Diretto (V: vettore; k: chiave)

    return(dati dell'elemento di indice k)

Funzione Delete_Indirizz_Diretto (V: vettore; k: chiave)

    V[k] ← null
    return

```

Tutte e tre le operazioni hanno complessità  $\Theta(1)$ . Purtroppo potrebbe capitare che l'insieme  $U$  sia enorme con conseguente impraticabilità d'allocazione in memoria di un array  $A$  di sufficiente capienza, o che il numero delle chiavi da utilizzare sia molto più piccolo di  $U$  con conseguente spreco di memoria; Per queste ragioni si ricorre a differenti implementazioni dei dizionari.

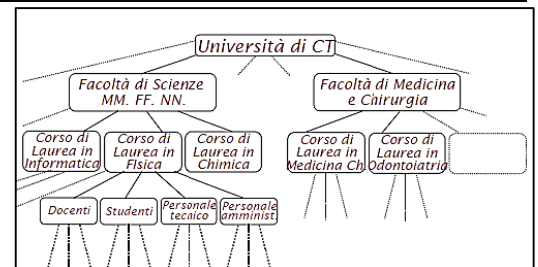
**TABELLE HASH:** utilizzate quando l'insieme  $U$  dei *possibili valori* che le chiavi possono assumere è molto grande (anche infinito), impossibile da allocare un array di  $U$  elementi, mentre l'insieme  $K$  delle chiavi da memorizzare è invece molto più piccolo di  $U$ ; richiede molta meno memoria rispetto all'indirizzamento diretto.

Slide 10 Cantone

### Alberi

Struttura dati ottimale per rappresentare delle gerarchie di dati. Fanno parte dei dizionari ed è una figura ispirata all'omonimo concetto esistente in natura, solo che in informatica tipicamente sono disegnati a testa in giù.

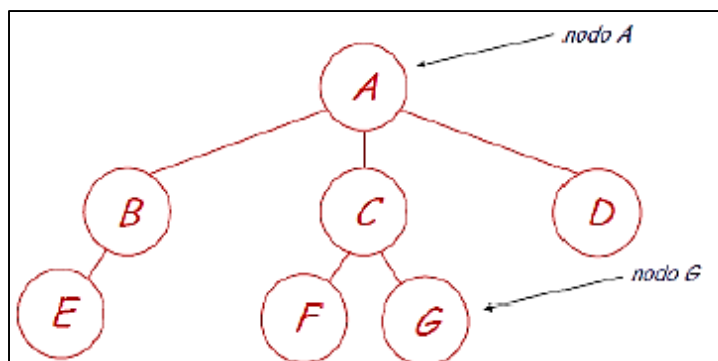
Lo scopo di una rappresentazione gerarchica dei dati è comprensibile a livelli di efficienza: le operazioni di ricerca, inserimento e cancellazione dei dati sono enormemente più veloci rispetto alla rappresentazione lineare.



Formalmente, un albero è un insieme (anche vuoto) di **nodi** (o *archi*) connessi mediante **rami** (o *vertici*). Ogni nodo (eccetto il nodo radice) è connesso tramite un ramo a un altro nodo che ne è il padre, e di cui rappresenta un figlio. Quindi la radice è l'unico nodo privo di padre.

Se  $T_1, T_2, \dots, T_n$  sono alberi (non vuoti) privi di nodi comuni, e  $X$  è un nodo, allora inserendo tutti  $T_1, T_2, \dots, T_n$  come sotto-alberi di  $X$ , si ottiene un unico albero.

Ogni nodo può contenere informazioni di vario tipo, rappresentate dalla sua **etichetta** (o chiave); spesso si indica un nodo mediante la sua etichetta.



Caratteristiche dei nodi: un nodo senza figli si chiama **foglia**; un nodo con almeno un figlio si chiama **nodo interno**; il nodo interno senza padre si chiama **radice**; due o più nodi con lo stesso padre si chiamano **fratelli**.

Es. Nell'albero sopra abbiamo:

- D,F,G,E sono delle foglie
- A,B,C sono nodi interni
- A è la radice
- B,C,D sono fratelli di padre A; e F,G sono fratelli di padre C

Cammino: sequenza di nodi e rami tali che ciascun nodo (tranne l'ultimo) sia padre del successivo. Esso è individuato dalla sequenza delle etichette dei suoi nodi.

Es. Nel seguente albero possiamo trovare i cammini (A,B,E), (A,C,F), (A,C,G), (A,D).

Lunghezza di un cammino: numero di rami coinvolti in un cammino da un nodo di partenza a un nodo di arrivo. Ogni nodo costituisce un cammino di lunghezza zero.

Es. Nell'albero sopra, il cammino (C,F) ha lunghezza 1, mentre il cammino (A,B,E) ha lunghezza 2. In generale, un cammino con K nodi ha lunghezza K-1.

Cammino caratteristico: cammino che esiste per ciascun nodo, il quale lo collega alla radice.

Es. Nell'albero accanto, (A,B,E) è il cammino caratteristico del nodo E, mentre (A,C,G) è il cammino caratteristico del nodo G.

Livello di un nodo: lunghezza del cammino caratteristico di un nodo.

Es. Nell'albero sopra, il nodo A ha livello 0; il nodo C ha livello 1; il nodo E ha livello 2.

Antenati e discendenti: se il cammino caratteristico di un nodo Y contiene un nodo X, diciamo che X è **antenato** di Y e Y è **discendente** di X.

Es. Nell'albero sopra, per trovare gli antenati di F dobbiamo prima trovare il suo cammino caratteristico che è (A,C,F), e poi possiamo dunque dichiarare A e C come antenati di F, mentre F è il loro discendente. I discendenti di A sono tutti nodi che partono da esso.

Altezza di un nodo: la lunghezza del più lungo cammino da un nodo ad una foglia si chiama altezza del nodo.

Tutte le foglie hanno altezza 0.

Es. Nell'albero sopra l'altezza di C è 1.

Altezza di un albero: si definisce come l'altezza della sua radice (quindi la lunghezza del cammino dalla radice alla foglia più distante), o il massimo livello delle sue foglie.

Es. L'albero sopra ha altezza 2.

Attraversamento di un albero: per attraversamento o visita di un albero si intende l'ispezione dei nodi dell'albero in modo che tutti i nodi vengano ispezionati una ed una sola volta. Quindi, un attraversamento di un albero definisce un ordinamento totale tra i nodi dell'albero in base alla loro posizione nell'albero, e NON in base alle loro etichette (che possono essere non ordinabili).

Trattandosi di ordinamento totale, ogni nodo ha un precedente e un successivo all'interno di un attraversamento.

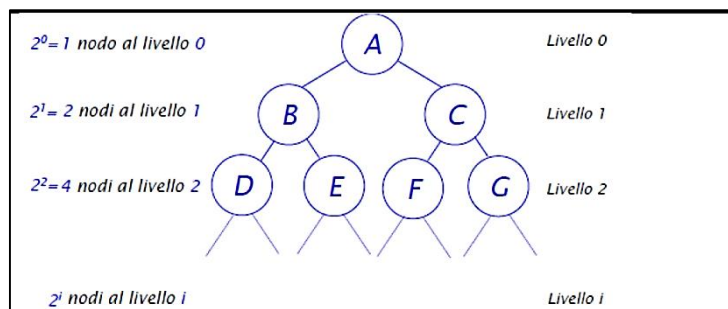
Es. Nell'albero di sopra i possibili attraversamenti sono:

- A,B,C,D,E,F,G
- B,C,D,E,F,G,A
- E,F,G,B,C,D,A ...

Ogni permutazione dei nodi definisce un diverso attraversamento. Certi attraversamenti procedono a salti e in pratica sono poco utili.

**ALBERO BINARIO**: un albero si dice binario se ogni nodo ha massimo due figli (detti rispettivamente *figlio sinistro* e *figlio destro*). In un albero binario, un nodo avente due figli si dice *pieno*. Un albero binario di altezza h si dice *completo* se tutti i nodi di livello minore di h sono pieni.





Consideriamo un albero binario completo avente altezza  $h$  ed  $n$  nodi. Possiamo esprimere  $n$  in funzione di  $h$  come  $n = 2^{h+1} - 1$ , e possiamo esprimere  $h$  in funzione di  $n$  come  $h = \log(n + 1) - 1$ .

**Attraversamento dell'albero binario PREORDER:** visitare la radice, attraversare ricorsivamente il sotto-albero sinistro della radice ed infine il sotto-albero destro della radice.

Es. Dall'albero binario sopra, attraversamento preorder: A, B, D, E, C, F, G

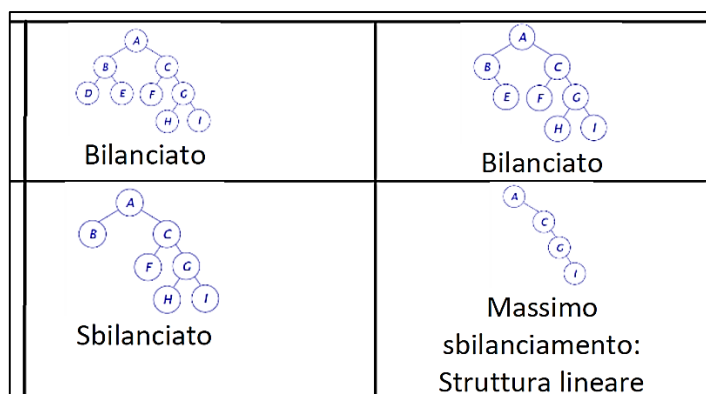
**Attraversamento dell'albero binario INORDER:** attraversare ricorsivamente il sotto-albero sinistro della radice, visitare la radice, e attraversare ricorsivamente il sotto-albero destro della radice.

Es. Dall'albero binario sopra, attraversamento inorder: D, B, E, A, F, C, G

**Attraversamento dell'albero binario POSTORDER:** attraversare ricorsivamente il sotto-albero sinistro della radice, il sotto-albero destro della radice, e infine visitare la radice.

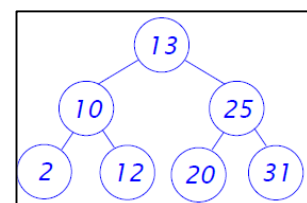
Es. Dall'albero binario sopra, attraversamento postorder: D, E, B, F, G, C, A

**Albero binario bilanciato:** un albero binario si dice bilanciato (in altezza) se, per ogni nodo, la differenza di altezza dei due sottoalberi è zero o uno. Un albero binario completo è bilanciato.

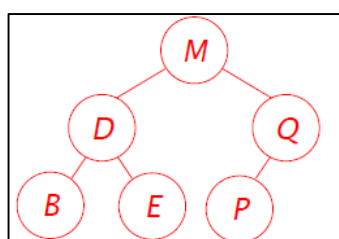


**ALBERO BINARIO DI RICERCA (BST):** un albero binario si dice di ricerca se, per ogni nodo, tutte le etichette del sottoalbero sinistro sono minori dell'etichetta del nodo, e tutte le etichette del sottoalbero destro sono maggiori dell'etichetta del nodo.

*tutti i figli sinistri  $\leq$  padre  $\leq$  tutti i figli destri*

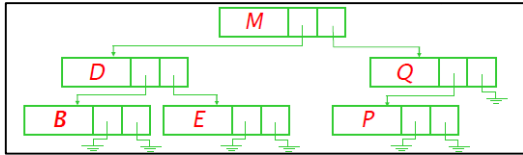


**Rappresentazione in memoria:** se il numero dei nodi di un BST è fissato a priori, allora lo si può rappresentare con un'implementazione statica tramite array. Ciascuna cella dell'array rappresenta un nodo ed è una struttura con 3 campi:



1. L'etichetta del nodo che la cella rappresenta
  2. L'indice della cella dell'array che rappresenta il nodo figlio sinistro
  3. L'indice della cella dell'array che rappresenta il nodo figlio destro
- La radice sta nella prima cella e -1 indica il figlio nullo.

5	2	-1	-1	4	-1	-1	-1	-1	3	1
M	E	Q	B	P	D					
0	1	2	3	4	5					



Si consideri il BST di sopra. Esso può essere implementato dinamicamente mediante una struttura in cui ogni nodo ha due riferimenti ai nodi figli.

Attraversamento simmetrico (Inorder): le chiavi di un BST possono essere facilmente elencate in maniera ordinata mediante un attraversamento simmetrico, il quale ha complessità  $T(n) = \Theta(n)$ .

INORDER-TREE-WALK( $x$ )

```

1  if  $x \neq \text{NIL}$ 
2    INORDER-TREE-WALK( $x.\text{left}$ )
3    print  $x.\text{key}$ 
4    INORDER-TREE-WALK( $x.\text{right}$ )

```

Operazioni su un BST:

- Operazioni di interrogazione: *Search*( $T, k$ ): cerca l'elemento con chiave di valore  $k$  in  $T$ ; *Minimum*( $T$ ): recupera l'elemento di valore minimo in  $T$ ; *Maximum*( $T$ ): recupera l'elemento di valore massimo in  $T$ ; *Predecessor*( $T, p$ ): recupera il precedente di  $p$  in  $T$ ; *Successor*( $T, p$ ): recupera il successore di  $p$  in  $T$ .
- Operazione di manipolazione: *Insert*( $T, k$ ): inserisce l'elemento di valore  $k$  in  $T$ ; *Delete*( $T, p$ ): elimina da  $T$  l'elemento di valore  $p$ .

Vedremo che tutte le suddette operazioni hanno complessità  $O(h)$  a seconda del bilanciamento dell'albero.

Ricerca: l'algoritmo per decidere se una chiave si trova in un BST avviene attraverso i seguenti controlli:

- Se il BST è vuoto restituisce null
- Se la chiave coincide con l'etichetta della radice, si restituisce la radice
- Se la chiave è minore dell'etichetta della radice, si esegue l'algoritmo sul sotto-albero sinistro della radice
- Se la chiave è maggiore dell'etichetta della radice, si esegue l'algoritmo sul sotto-albero destro della radice

Si osservi che ogni qual volta si scende verso destra o sinistra, automaticamente si decide di escludere l'altro l'intero sottoalbero.

### VERSIONE RICORSIVA

TREE-SEARCH( $x, k$ )

```

1  if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2    return  $x$ 
3  if  $k < x.\text{key}$ 
4    return TREE-SEARCH( $x.\text{left}, k$ )
5  else return TREE-SEARCH( $x.\text{right}, k$ )

```

### VERSIONE ITERATIVA

ITERATIVE-TREE-SEARCH( $x, k$ )

```

1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2    if  $k < x.\text{key}$ 
3       $x = x.\text{left}$ 
4    else  $x = x.\text{right}$ 
5  return  $x$ 

```

Dove LEFT è il puntatore al figlio sinistro, RIGHT è il puntatore al figlio destro, P è il puntatore al padre,  $x$  è l'albero,  $k$  è la chiave.

La funzione ricorsiva compie un numero costante di operazioni sulla radice dell'albero e poi richiama se stessa sulla radice di uno dei due sottoalberi, che al massimo ha altezza  $h - 1$ ; possiamo quindi scrivere:

$$T(h) = \Theta(1) + T(h - 1) = \Theta(h)$$

nel caso peggiore, che è quello della ricerca senza successo che termini in una foglia a distanza massima dalla radice.

Nel caso base, che si ha quando  $h=0$  (caso in cui la chiave cercata sia nella radice), il tempo di esecuzione è costante, cioè  $T(0) = \Theta(1)$ .

Per quanto riguarda la funzione iterativa, la complessità dipende dal numero di volte che viene eseguito il ciclo while. Nel caso peggiore esso viene eseguito un numero di volte pari all'altezza dell'albero e quindi la complessità è, come per la versione ricorsiva,  $O(h)$ .

Se la chiave si trova al livello 0, abbiamo 0 assegnamenti, a livello 1 abbiamo un assegnamento, a livello 2 abbiamo 2 assegnamenti, e così via; quindi il numero di assegnamenti dipende dalla posizione in cui si trova la chiave cercata.

Minimo e massimo: gli algoritmi usati per compiere quest'azione sono:

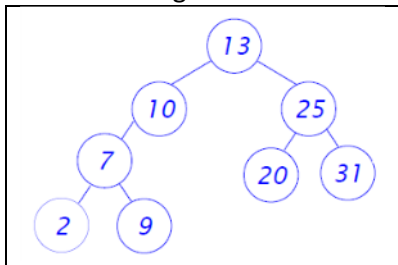
<u>MINIMO</u>	<u>MASSIMO</u>
<p>VERSIONE RICORSIVA</p> <pre> TREE-MINIMUM-RECURSIVE(x)   if x.left == NIL then     return x   else TREE-MINIMUM-RECURSIVE(x.left) </pre> <p>VERSIONE ITERATIVA</p> <pre> TREE-MINIMUM(x) 1 while x.left != NIL 2   x = x.left 3 return x </pre> <p>COMPLESSITA': <math>O(h)</math> (h ALTEZZA)</p>	<p>VERSIONE RICORSIVA</p> <pre> TREE-MAXIMUM-RECURSIVE(x)   if x.right == NIL then     return x   else TREE-MAXIMUM-RECURSIVE(x.right) </pre> <p>VERSIONE ITERATIVA</p> <pre> TREE-MAXIMUM(x) 1 while x.right != NIL 2   x = x.right 3 return x </pre> <p>COMPLESSITA': <math>O(h)</math> (h ALTEZZA)</p>

Predecessore e successore: dato un insieme totalmente ordinato, ha senso parlare di predecessore e di successore di un elemento dell'insieme.

Es. Dato l'insieme  $A=\{b,e,f,i,m,n,r,t,v,w,y\}$  e il suo elemento  $t$ :  $r$  è il predecessore in  $A$  di  $t$ ;  $v$  è il successore in  $A$  di  $t$ ; Il predecessore di  $b$  e il successore di  $y$  non sono definiti.

Dato un BST e una sua chiave che si trova in un nodo avente sotto-albero sinistro non vuoto, il predecessore della chiave (nell'insieme di etichette del BST) si trova nel nodo più a destra del sotto-albero sinistro del nodo contenente la chiave; dato un BST e una sua chiave che si trova in un nodo avente sotto-albero destro non vuoto, il successore della chiave (nell'insieme di etichette del BST) si trova nel nodo più a sinistra del sotto-albero destro del nodo contenente la chiave. Complessità  $O(h)$ .

Es. Nel seguente albero



Predecessore di 10 è 9;  
 Predecessore di 13 è 10;  
 Successore di 13 è 20;  
 Successore di 25 è 31.

Quindi, predecessore di una chiave è la chiave **massima** nel sotto-albero sinistro;  
 successore di una chiave è la chiave **minima** nel sotto-albero destro.

TREE-SUCCESSOR(x)	TREE-PREDECESSOR(x)
<pre> 1 if x.right != NIL 2   return TREE-MINIMUM(x.right) 3 y = x.p 4 while y != NIL and x == y.right 5   x = y 6   y = y.p 7 return y </pre>	<pre> 1 if x.left != NIL 2   return TREE-MAXIMUM(x.left) 3 y = x.p 4 while y != NIL and x == y.left 5   x = y 6   y = y.p 7 return y </pre>

Inserimento: la complessità asintotica dell'inserimento è analoga a quella della ricerca.

- Se il BST è vuoto, si inserisca la chiave in un nuovo nodo che sarà radice;
- Si ricerchi la chiave. Se si trova, si restituisca false
- Si determini la foglia che può essere padre del nuovo nodo
- Si innesti il nuovo nodo come figlio della foglia trovata

```

TREE-INSERT( $T, z$ )
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y \leftarrow x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x \leftarrow x.\text{left}$ 
7      else  $x \leftarrow x.\text{right}$ 
8   $z.p \leftarrow y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} \leftarrow z$  // tree  $T$  was empty
11 elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} \leftarrow z$ 
13 else  $y.\text{right} \leftarrow z$ 

```

L'algoritmo di inserimento visto non mantiene il bilanciamento del BST.

Cancellazione: per cancellare da un BST una certa chiave, mantenendo le proprietà di BST, abbiamo 3 casi:

	<p>Se <math>z</math> non ha figli viene semplicemente rimosso modificando il puntatore del padre al figlio <math>z</math> che verrà sostituito con con nil.</p>
	<p>Se <math>z</math> ha un solo figlio si estrae <math>z</math> e il figlio di <math>z</math> diventa il figlio del padre di <math>z</math>.</p>
	<p>Se <math>z</math> ha due figli si estrae il suo successore <math>y</math> che ha massimo un figlio e si sostituisce il contenuto di <math>z</math> con il contenuto di <math>y</math></p>

```

TREE-DELETE( $T, z$ )
1  if  $\text{left}[z] = \text{NIL}$  o  $\text{right}[z] = \text{NIL}$ 
2      then  $y \leftarrow z$ 
3      else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
4  if  $\text{left}[y] \neq \text{NIL}$ 
5      then  $x \leftarrow \text{left}[y]$ 
6      else  $x \leftarrow \text{right}[y]$ 
7  if  $x \neq \text{NIL}$ 
8      then  $p[x] \leftarrow p[y]$ 
9  if  $p[y] = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow x$ 
11     else if  $y = \text{left}[p[y]]$ 
12         then  $\text{left}[p[y]] \leftarrow x$ 
13         else  $\text{right}[p[y]] \leftarrow x$ 
14 if  $y \neq z$ 
15     then  $\text{key}[z] \leftarrow \text{key}[y]$ 
16      $\triangleright$  Se  $y$  ha altri campi, copia anche questi.
17 return  $y$ 

```

Il nodo  $y$  se  $z$  ha massimo un figlio diventa il nodo di input  $z$  stesso, oppure se  $z$  ha due figli diventa il successore di  $z$ .

Le operazioni Insert e Delete su un insieme dinamico possono essere eseguite in tempo  $O(h)$  utilizzando un BST di altezza  $h$ .

### Complessità asintotica in generale:

- Se il BST è completo avremo  $O(h)=O(\lg n)$ ;
- Se il BST è bilanciato avremo  $O(h)=O(\lg n)$ ;
- Se il BST è sbilanciato avremo  $O(h)=O(n)$ .

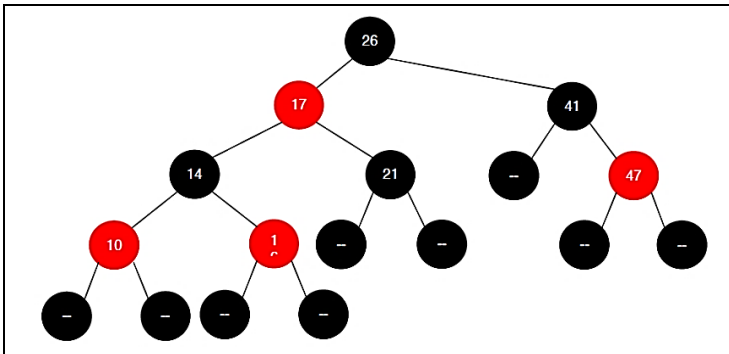
Pertanto, più l'albero è sbilanciato (caso peggiore), più la complessità si avvicina a quella della ricerca su una lista, ossia lineare. Più l'albero è bilanciato (caso migliore), più la complessità della ricerca si avvicina a quella logaritmica. Vedremo una variante dei BST che assicurano una complessità  $O(\lg n)$  anche nel caso peggiore (**alberi rosso-neri**).

**ALBERO ROSSO-NERO (RBT):** un RBT è una variante del BST i cui nodi hanno un campo aggiuntivo, il *colore*, che può essere solo rosso o nero, con lo scopo di garantire una complessità di  $O(\lg n)$ ; alla struttura si aggiungono foglie fittizie (che non contengono chiavi) per fare in modo che tutti i nodi "veri" dell'albero abbiano esattamente due figli. Esso soddisfa queste proprietà:

1. Ogni nodo è rosso o nero;
2. La radice è nera;
3. Ogni foglia è nera;
4. Se un nodo è rosso, i suoi figli sono entrambi neri;
5. Ogni cammino da un nodo a ciascuna delle foglie contiene lo stesso numero di nodi neri (proprietà di bilanciamento).

Inoltre, per la proprietà di BST, ogni nodo ha due figli tranne le foglie, se non ce li ha vengono messe delle foglie fittizie.

Es.



Questo albero rispetta tutte e sei le proprietà di un RBT.

Il colore viene determinato proprio in base a queste regole.

**ALTEZZA NERA DI UN NODO ( $bh(x)$ ):** col termine b-altezza di x (black height di x,  $bh(x)$ ) si indica il numero di nodi neri sui cammini da un nodo x (non incluso) alle foglie sue discendenti (è uguale per tutti i cammini). La b-altezza di un RBT è la b-altezza della sua radice (una delle proprietà del RBT).

Es. Nel RBT di sopra,  $bh(21)=1$ ,  $bh(26)=2$

**LEMMA 1:** Il sottoalbero radicato in un qualsiasi nodo x contiene almeno  $2^{bh(x)} - 1$  nodi interni.

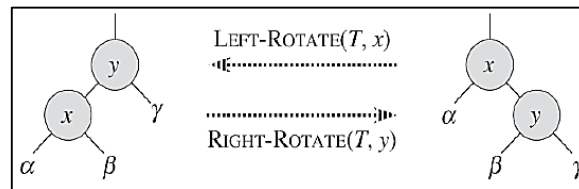
**LEMMA 2:** un RBT con n nodi interni ha massimo un'altezza di  $2 \lg(n + 1)$ . Questo lemma crea la proprietà che:

- Nessun cammino di un nodo di un RBT è più lungo del doppio di qualsiasi altro nodo.

Conseguenza immediata di questo lemma è che le operazioni su un RBT (ricerca, predecessore, minimo, ecc.) possono essere realizzate su un RBT ottenendo i tempi di esecuzione  $O(\lg n)$ , dato che su un comune BST i tempi sono  $O(h)$ .

**OPERAZIONI:** in un RBT con n nodi interni le operazioni SEARCH, MINIMUM, MAXIMUM, PREDECESSOR e SUCCESSOR possono essere implementate nel tempo  $O(\lg n)$ . Naturalmente l'esigenza di mantenere le proprietà di RBT implica che, dopo un inserimento o una cancellazione, la struttura dell'albero possa dover essere riaggiustata, in termini di colori assegnati ai nodi e collocazione dei nodi nell'albero. A tal fine sono definite apposite operazioni, dette **rotazioni**, che permettono di ripristinare in tempo  $O(\lg n)$  le proprietà del RBT dopo un inserimento o un'eliminazione.

**ROTAZIONE:** hanno lo scopo di preservare le proprietà di un RBT a seguito di inserimenti e cancellazioni.



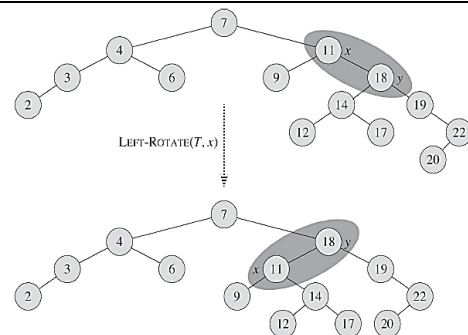
- **LEFT-ROTATE(T,x):** il figlio destro y di x diventa padre di y, e contemporaneamente il figlio sinistro  $\beta$  di y diventa figlio destro di x;
- **RIGHT-ROTATE(T,y):** il figlio sinistro x di y diventa padre di x, e contemporaneamente il figlio destro  $\beta$  di x diventa figlio sinistro di y;

**LEFT-ROTATE(T, x)**

```

1  y = x.right           // set y
2  x.right = y.left      // turn y's left subtree into x's right subtree
3  if y.left != T.nil
4      y.left.p = x
5  y.p = x.p             // link x's parent to y
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x            // put x on y's left
12 x.p = y

```



## INSERIMENTO:

**RB-INSERT(T, z)**

```

1  y = T.nil
2  x = T.root
3  while x != T.nil
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == T.nil
10     T.root = z
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
14 z.left = T.nil
15 z.right = T.nil
16 z.color = RED
17 RB-INSERT-FIXUP(T, z)

```

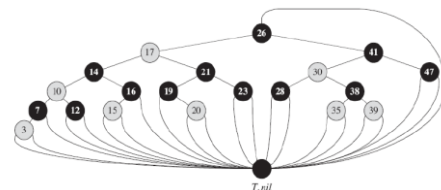
**TREE-INSERT(T, z)**

```

1  y = NIL
2  x = T.root
3  while x != NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z // tree T was empty
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z

```

In questo algoritmo, al posto del valore NIL per indicare la foglia fittizia abbiamo utilizzato il nodo T.NIL, che si comporta come una sentinella, la quale ha valori irrilevanti, ma risulterà possibile assegnare ad esse dei valori importanti.



**RB-INSERT-FIXUP(T, z)**

```

1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK // case 1
6              y.color = BLACK // case 1
7              z.p.p.color = RED // case 1
8              z = z.p.p // case 1
9          else if z == z.p.right
10             z = z.p // case 2
11             LEFT-ROTATE(T, z) // case 2
12             z.p.color = BLACK // case 3
13             z.p.p.color = RED // case 3
14             RIGHT-ROTATE(T, z.p.p) // case 3
15     else (same as then clause
16         with "right" and "left" exchanged)
17     T.root.color = BLACK

```

La chiamata **RB-INSERT-FIXUP(T,z)** serve a ripristinare la proprietà che vieta due nodi rossi padre-figlio.

Se T è vuoto, z è la radice e dunque il suo colore deve essere nero.

Complessità:  $O(\lg n)$

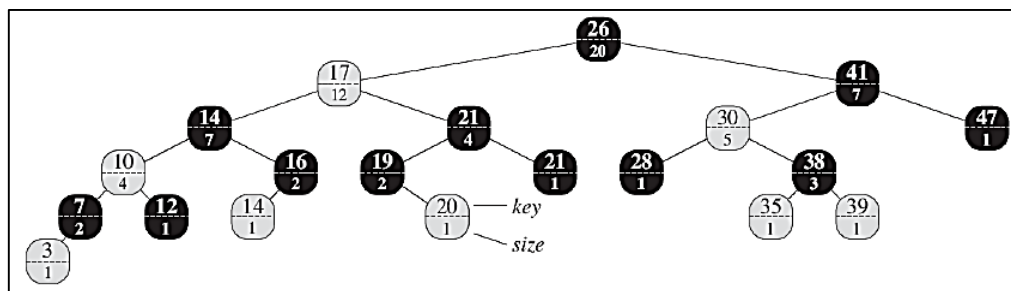
**CANCELLAZIONE:** la procedura per cancellare un nodo da un albero rosso-nero si basa sulla procedura tree-delete (e quindi su transplant).

<pre> RB-DELETE(<math>T, z</math>) 1  <math>y = z</math> 2  <math>y\text{-original-color} = y.\text{color}</math> 3  if <math>z.\text{left} == T.\text{nil}</math> 4      <math>x = z.\text{right}</math> 5      RB-TRANSPLANT(<math>T, z, z.\text{right}</math>) 6  elseif <math>z.\text{right} == T.\text{nil}</math> 7      <math>x = z.\text{left}</math> 8      RB-TRANSPLANT(<math>T, z, z.\text{left}</math>) 9  else <math>y = \text{TREE-MINIMUM}(z.\text{right})</math> 10     <math>y\text{-original-color} = y.\text{color}</math> 11     <math>x = y.\text{right}</math> 12     if <math>y.p == z</math> 13         <math>x.p = y</math> 14     else RB-TRANSPLANT(<math>T, y, y.\text{right}</math>) 15         <math>y.\text{right} = z.\text{right}</math> 16         <math>y.\text{right}.p = y</math> 17     RB-TRANSPLANT(<math>T, z, y</math>) 18     <math>y.\text{left} = z.\text{left}</math> 19     <math>y.\text{left}.p = y</math> 20     <math>y.\text{color} = z.\text{color}</math> 21 if <math>y\text{-original-color} == \text{BLACK}</math> 22     RB-DELETE-FIXUP(<math>T, x</math>) </pre>	<pre> RB-DELETE-FIXUP(<math>T, x</math>) 1  while <math>x \neq T.\text{root}</math> and <math>x.\text{color} == \text{BLACK}</math> 2      if <math>x == x.p.\text{left}</math> 3          <math>w = x.p.\text{right}</math> 4          if <math>w.\text{color} == \text{RED}</math> 5              <math>w.\text{color} = \text{BLACK}</math> 6              <math>x.p.\text{color} = \text{RED}</math> 7              LEFT-ROTATE(<math>T, x.p</math>) 8              <math>w = x.p.\text{right}</math> 9          if <math>w.\text{left}.color == \text{BLACK}</math> and <math>w.\text{right}.color == \text{BLACK}</math> 10             <math>w.\text{color} = \text{RED}</math> 11             <math>x = x.p</math> 12         elseif <math>w.\text{right}.color == \text{BLACK}</math> 13             <math>w.\text{left}.color = \text{BLACK}</math> 14             <math>w.\text{color} = \text{RED}</math> 15             RIGHT-ROTATE(<math>T, w</math>) 16             <math>w = x.p.\text{right}</math> 17             <math>w.\text{color} = x.p.\text{color}</math> 18             <math>x.p.\text{color} = \text{BLACK}</math> 19             <math>w.\text{right}.color = \text{BLACK}</math> 20             LEFT-ROTATE(<math>T, x.p</math>) 21             <math>x = T.\text{root}</math> 22         else (same as then clause with "right" and "left" exchanged) 23     <math>x.\text{color} = \text{BLACK}</math> </pre>
<pre> RB-TRANSPLANT(<math>T, u, v</math>) 1  if <math>u.p == T.\text{nil}</math> 2      <math>T.\text{root} = v</math> 3  elseif <math>u == u.p.\text{left}</math> 4      <math>u.p.\text{left} = v</math> 5  else <math>u.p.\text{right} = v</math> 6  <math>v.p = u.p</math> </pre>	

Dunque, su tutte le operazioni su un RBT la maggior parte del loro codice serve per gestire i vari casi di sbilanciamento che si possono presentare durante la fase di risistemazione dell'albero, e si risolvono usando propriamente la rotazione.

**ALBERO PER STATISTICHE D'ORDINE:** le potenzialità delle strutture dati fondamentali (liste, alberi, ecc.) possono essere aumentate con opportune modifiche alle strutture dati stesse, memorizzando opportune *informazioni aggiuntive*, come ad esempio gli alberi per statistiche d'ordine. Esso è un RBT che supporta anche l'operazione di ricerca di un elemento con un dato **rango**. Quindi oltre agli attributi *key*, *color*, *p*, *left* e *right* si memorizzerà in ciascun nodo *x* anche un attributo *x.size* che contiene il numero di nodi (interni) nel sottoalbero radicato in *x* (*x* compreso). Per ogni nodo interno *x* vale l'identità  $x.\text{size} = x.\text{left}.\text{size} + x.\text{right}.\text{size} + 1$

Es.



<pre> OS-SELECT(<math>x, i</math>) 1  <math>r = x.\text{left}.\text{size} + 1</math> 2  if <math>i == r</math> 3      return <math>x</math> 4  elseif <math>i &lt; r</math> 5      return OS-SELECT(<math>x.\text{left}, i</math>) 6  else return OS-SELECT(<math>x.\text{right}, i - r</math>) </pre>	<pre> OS-RANK(<math>T, x</math>) 1  <math>r = x.\text{left}.\text{size} + 1</math> 2  <math>y = x</math> 3  while <math>y \neq T.\text{root}</math> 4      if <math>y == y.p.\text{right}</math> 5          <math>r = r + y.p.\text{left}.\text{size} + 1</math> 6      <math>y = y.p</math> 7  return <math>r</math> </pre>
<pre> OS-SELECT(<math>T.\text{root}, i</math>) </pre> <p>COMPLESSITA': <math>O(\lg n)</math></p>	<p>COMPLESSITA': <math>O(\lg n)</math></p>

Nel caso dell'inserimento, durante la fase di discesa per innestare il nuovo nodo *z* si incrementa di un'unità il valore dell'attributo *size* dei nodi incontrati; nel caso della cancellazione invece si segue il cammino semplice dalla posizione originale del nodo *y* fino alla radice decrementando di un'unità il valore dell'attributo *size*.



La programmazione dinamica è una tecnica di progettazione di algoritmi basata sulla divisione del problema in sottoproblemi. A differenza degli algoritmi divide-et-impera, questi algoritmi si applicano anche quando i sottoproblemi non sono indipendenti, ma la soluzione di tali problemi richiede la stessa procedura. Dunque questi algoritmi risolvono ciascun sottoproblema una sola volta e memorizzano la soluzione in una tabella per evitare di calcolare nuovamente la soluzione ogni qual volta si ripresenta lo stesso problema.

Scopo della programmazione dinamica è risolvere i problemi di ottimizzazione utilizzando sottostrutture ottimali.

**SOTTOSTRUTTURA OTTIMALE:** significa che la soluzione ottimale al sottoproblema può essere utilizzata per trovare la soluzione ottimale dell'intero problema. In generale, è possibile risolvere un problema con una sottostruttura ottimale utilizzando un processo a tre passi:

1. Suddividere il problema in sottoproblemi più piccoli.
2. Risolvere questi problemi in modo ottimale, utilizzando in modo ricorsivo questo processo a tre passi.
3. Usare queste soluzioni ottimali per costruire una soluzione ottimale al problema originale.

I sottoproblemi sono, essi stessi, risolti suddividendoli in sotto-sottoproblemi, e così via, finché non si raggiungano alcuni casi semplici, facili da risolvere.

**SOTTOPROBLEMI SOVRAPPONIBILI:** dire che un problema ha sottoproblemi *sovrapponibili* equivale a dire che gli stessi sottoproblemi sono utilizzabili per risolvere diversi problemi più ampi. Per esempio, nella Successione di Fibonacci,  $F_3 = F_1 + F_2$  e  $F_4 = F_2 + F_3$ ; il calcolo di ciascun numero implica il calcolo di  $F_2$ . Siccome sia  $F_3$  che  $F_4$  sono necessari per il calcolo di  $F_5$ , un approccio ingenuo al calcolo di  $F_5$  può condurre a calcolare  $F_2$  due o più volte. Ciò si applica ogniqualevolta si presentino problemi sovrapponibili: un approccio ingenuo potrebbe sprecare del tempo ricalcolando la soluzione ottimale a sottoproblemi già risolti.

**MEMOIZZAZIONE:** per evitare che il problema sopra accada, occorre salvare la soluzione ai problemi già risolti. Quindi, se in seguito si necessita di risolvere lo stesso problema, è possibile riprendere e riutilizzare la soluzione già calcolata. Questo approccio è chiamato *memoizzazione* (non memorizzazione, anche se questo termine può risultare adeguato). Essendo sicuri che una particolare soluzione non sarà riutilizzata, è possibile liberarsene per risparmiare spazio. In alcuni casi, è possibile calcolare in anticipo delle soluzioni a sottoproblemi, sapendo che successivamente saranno necessarie.

**APPROCCI:** la programmazione dinamica generalmente si basa su uno dei due seguenti approcci:

- **Top-down:** Il problema è spezzato in sottoproblemi, questi sono risolti, e la soluzione è ricordata, nel caso sia necessario risolverli ancora. Si tratta di una combinazione di ricorsività e memorizzazione.
- **Bottom-up:** Sono risolti innanzitutto tutti i sottoproblemi che possono essere necessari, e successivamente utilizzati per costruire la soluzione a problemi più ampi. Questo approccio è leggermente migliore come dimensione degli stack e numero di chiamate alle funzioni, ma talvolta risulta non intuitivo prefigurarsi tutti i sottoproblemi necessari alla soluzione di un dato problema.

**ESEMPIO SUCCESSIONE DI FIBONACCI:** una implementazione di una funzione che cerchi l' $n$ -mo membro della Successione di Fibonacci, basato direttamente sulla definizione matematica:

```
function fib(n)
  if n = 0 or n = 1
    return n
  else
    return fib(n - 1) + fib(n - 2)
```

Notare che se si chiama `fib(5)`, si produce un albero di chiamate che chiama più volte ricorsivamente la medesima funzione sullo stesso valore:

```
fib(5)
fib(4) + fib(3)
(fib(3) + fib(2)) + (fib(2) + fib(1))
((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
```



In particolare,  $\text{fib}(2)$  è stato calcolato interamente per due volte. In esempi più estesi sono ricalcolati molti più valori di  $\text{fib}$ , o sottoproblemi, portando ad un algoritmo a tempi esponenziali. Ora, si supponga di avere un semplice oggetto **mappa**,  $m$ , che mappi ogni valore  $\text{fib}$  calcolato al proprio risultato, e si modifichi la funzione al fine di utilizzarlo ed aggiornarlo. La funzione risultante richiede solo un tempo  $O(n)$  invece che un tempo esponenziale; Questa tecnica di salvataggio dei valori già calcolati è chiamata memoizzazione.

```
var m := map(0 → 1, 1 → 1)
function fib(n)
  if map m does not contain key n
    m[n] := fib(n - 1) + fib(n - 2)
  return m[n]
```

Approccio *top-down*, visto che in primo luogo il problema è stato diviso in sottoproblemi, poi sono stati calcolati e salvati i valori.

```
function fib(n)
  var previousFib := 0, currentFib := 1
  repeat n - 1 times
    var newFib := previousFib + currentFib
    previousFib := currentFib
    currentFib := newFib
  return currentFib
```

Approccio *bottom-up*, prima si calcola il più piccolo valore di  $\text{fib}$ , poi si costruiscono i valori più grandi a partire da questo. Anche questo metodo richiede un tempo  $O(n)$ , visto che contiene un ciclo che si ripete  $n - 1$  volte:

In entrambi questi esempi, si è calcolato  $\text{fib}(2)$  solo una volta, e poi lo si è utilizzato per calcolare sia  $\text{fib}(4)$  che  $\text{fib}(3)$ , invece di calcolarlo ogni volta che questi ultimi valori vengono calcolati.

**ALGORITMO GREEDY:** è un algoritmo che cerca di ottenere una soluzione ottima da un punto di vista globale attraverso la scelta della soluzione più golosa ad ogni passo locale. Questa tecnica consente, dove applicabile (infatti non sempre si arriva ad una soluzione ottima), di trovare soluzioni ottimali per determinati problemi in un tempo polinomiale.

Es. Il problema "Dai il minor numero di monete di resto utilizzando monete da 100, 10, 1 eurocent" è un problema risolvibile tramite un algoritmo di tipo greedy: ad ogni passo viene controllato il resto ancora da dare e si aggiunge la moneta con valore maggiore possibile. Quindi per dare un resto di 112 eurocent la macchina farà cadere in sequenza una moneta da 100, poi 10, poi 1, e infine ancora 1 eurocent (per un totale di 4 monete).

Strategia greedy: riassunto, tale strategia consiste in 4 passi:

1. Verificare la proprietà della sottostruttura ottima
2. Verificare che esiste sempre una soluzione ottima che include la scelta greedy
3. Verificare che dopo la scelta greedy il problema iniziale è ricondotto ad un sottoproblema dello stesso tipo la cui soluzione ottima può essere combinata con la scelta greedy per dare luogo ad una soluzione ottima del problema iniziale

Sia la programmazione dinamica che la strategia greedy utilizzano entrambe la proprietà della sottostruttura ottima.

Slide 15 Cantone

**CODIFICA DI HUFFMAN:** algoritmo usato per la compressione di dati, basato sul principio di trovare il sistema ottimale per codificare stringhe basato sulla frequenza relativa di ciascun carattere.

La codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun simbolo, che esprime il carattere più frequente nella maniera più breve possibile. Questa tecnica funziona creando un albero binario di simboli:

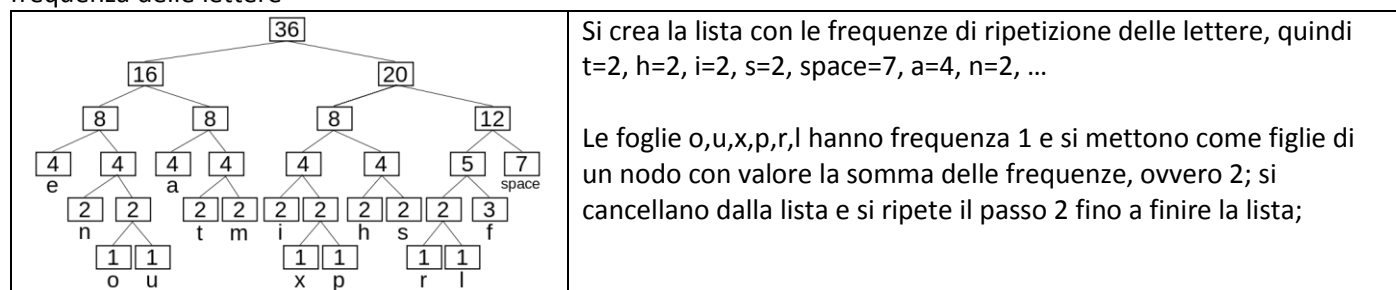
1. Ordina i simboli, in una lista, in base al conteggio delle loro occorrenze.
2. Ripete i seguenti passi finché la lista non contiene un unico simbolo:
  - a. Prende dalla lista i due simboli con la frequenza di conteggio minore. Crea un albero di Huffman che ha come "figli" questi due elementi, e crea un nodo di "genitori"

```
HUFFMAN(C, f)
m := |C|
Q := make_queue(C, f)
for i := 1 to m-1 do
  - SI ALLOCA UN NUOVO NODO INTERNO z
  left[z] := x := EXTRACT_MIN(Q)
  right[z] := y := EXTRACT_MIN(Q)
  f[z] := f[x] + f[y]
  INSERT(Q, z, f)
return EXTRACT_MIN(Q)
```

*COMPLESSITÀ:*  
 $(2n-1)$  EXTRACT\_MIN  $O(\log n)$   
 $(m-1)$  INSERT  $O(\log n)$   
 BUILDHEAP  $O(n)$   
 **$O(n \log n)$**

- b. Assegna la somma del conteggio delle frequenze dei figli ai genitori e li pone nella lista in modo da mantenere l'ordine.
  - c. Cancella il figlio dalla lista.
3. Assegna una parola codice ad ogni elemento basandosi sul path a partire dalla radice.

Es. Codifica di Huffman della frase "this is an example of a huffman tree" con rappresentazione binarie e indice di frequenza delle lettere



Tale codifica consente un fattore di compressione della stringa tra il 20% e il 90%.

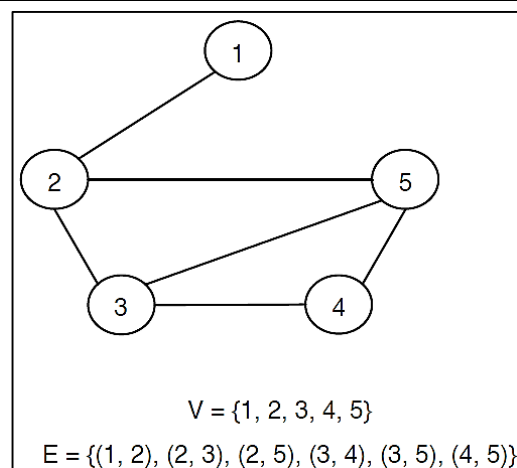
### Grafì

Indichiamo un grafo con  $G = (V, E)$ , dove  $V$  è l'insieme finito dei **nodi** (o vertici) ed  $E$  è l'insieme finito di **archi** (o spigoli) ciascuno dei quali connette due nodi in  $V$  detti estremi dell'arco.

Due vertici  $u, v$  connessi da un arco prendono nome di *estremi dell'arco*; l'arco viene anche identificato con la coppia formata dai suoi estremi  $(u, v)$ , ovvero parte entrante e parte uscente dal nodo.

L'intero  $n$  indica il numero di nodi ed  $m$  indica il numero di archi di cui è composto il grafo.

MASSIMO NUMERO DI ARCHI:  $\frac{n(n-1)}{2}$

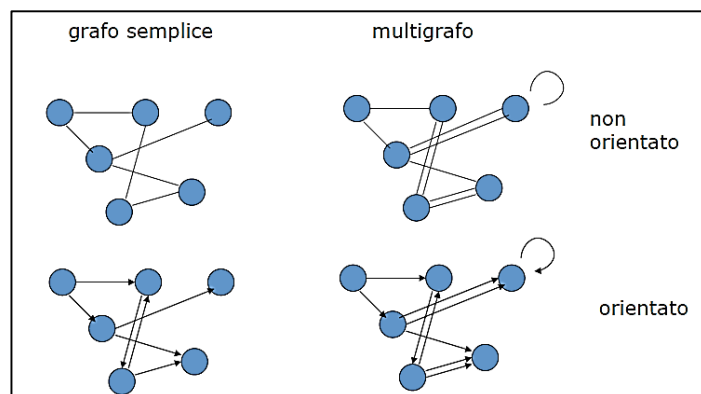


CAPPIO: un arco  $(u, v)$  con i due estremi coincidenti  $u=v$  è detto cappio.

ARCO MULTIPLO: un arco che compare più di una volta in  $E$  connettendo gli stessi due estremi.

GRAFO SEMPLICE: grafo che non contiene né cappi né archi multipli. Se un grafo è semplice possiamo identificare un arco con la coppia dei suoi estremi:  $e = (u, v) \in E$ . Quando  $e = (u, v) \in E$  diciamo che l'arco  $e$  è **incidente** o **adiacente** in  $u$  e in  $v$ . Se il grafo non è orientato la relazione di adiacenza è simmetrica.

MULTIGRAFO: grafo che contiene cappi e archi multipli.



GRAFO ORIENTATO: un grafo è orientato quando vi è un ordine tra i due estremi degli archi. Un grafo orientato (detto anche **digrafo**) è definito in modo analogo al grafo, ma gli archi sono delle coppie ordinate e sono detti **archi orientati**. Se il grafo è orientato la coppia  $(u, v)$  è ordinata; in questo caso diciamo che l'arco  $e$  esce da  $u$  ed entra in  $v$ .

GRADO  $\delta(v)$ : il numero di archi incidenti in un certo nodo  $v$  è detto grado di  $v$  o  $\text{degree}(v)$ . Se il grafo è orientato  $\delta(v)$  si suddivide in:

- Grado entrante  $\delta^-(v)$ : numero di archi entranti in  $v$  (in-degree);
- Grado uscente  $\delta^+(v)$ : numero di archi uscenti da  $v$  (out-degree).

**PASSEGGIATA:** una passeggiata in  $G$  dal nodo  $x$  al nodo  $y$  è una sequenza  $x_0, e_1, x_2, e_2, \dots, x_k, e_k$  in cui si alternano nodi ed archi del grafo con  $x_i \in V$  e  $e_i \in E$ , tale che per  $i = 1, 2, \dots, k$  si abbia  $\{x_i, x_{i+1}\} = e_i$ .

**LUNGHEZZA DELLA PASSEGGIATA:** è  $k$ , ovvero il numero dei suoi archi (incluse eventuali ripetizioni).

**CAMMINO:** un cammino da  $x$  a  $y$  è una passeggiata da  $x$  a  $y$  in cui non ci siano ripetizioni di archi: se  $i \neq j$  allora  $e_i \neq e_j$ . Formalmente, un cammino  $\pi$  di lunghezza  $k$  dal vertice  $u$  al vertice  $v$  in un grafo  $G = (V, E)$  è una sequenza di  $k+1$  vertici  $\pi = x_0, x_1, \dots, x_k$  tali che  $x_0 = u$ ,  $x_k = v$  e  $(x_{i-1}, x_i) \in E$  per  $i = 1, \dots, k$ .

Il cammino  $\pi = x_0$  ha lunghezza  $k = 0$ .

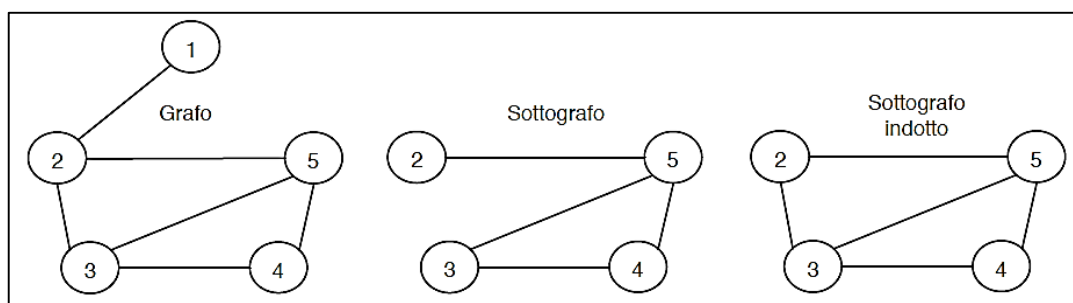
- Se  $k > 0$  e  $x_0 = x_k$  diciamo che il cammino è **chiuso**
- Un **cammino semplice** è un cammino i cui vertici  $x_0, x_1, \dots, x_k$  ( $k > 0$ ) sono tutti distinti, quindi non ci sono ripetizioni di nodi, con la possibile eccezione di  $x_0 = x_k$ , nel qual caso esso è un **ciclo**.
- Un cammino in cui primo e ultimo nodo coincidono si chiama **circuito**.
- Un ciclo di lunghezza  $k = 1$  è un cappio. Un grafo aciclico è un grafo che non contiene cicli.

**RAGGIUNGIBILITÀ:** Quando esiste almeno un cammino dal vertice  $u$  al vertice  $v$  diciamo che il vertice  $v$  è accessibile o raggiungibile da  $u$ . La relazione di raggiungibilità è perciò una relazione di equivalenza: le classi di questa relazione di equivalenza determinano una partizione dei nodi. Tutti i nodi di una stessa classe sono raggiungibili da ogni altro nodo della classe.

**CONNESSIONE:** un grafo non orientato si dice connesso se esiste almeno un cammino tra ogni coppia di vertici. Le componenti connesse di un grafo sono le classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità. Un grafo orientato si dice **fortemente connesso** se esiste almeno un cammino da ogni vertice  $u$  ad ogni altro vertice  $v$ . Le componenti fortemente connesse di un grafo orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di mutua raggiungibilità.

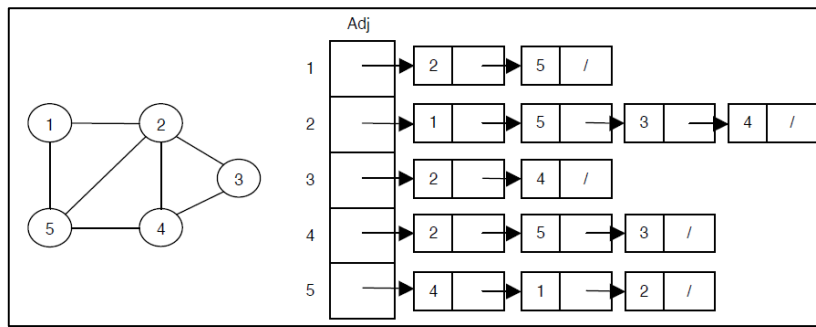
**COMPLETEZZA:** un grafo si dice **completo** se esiste un arco per ogni coppia di vertici. Il grafo completo con  $n$  vertici è denotato  $K_n$ . Un grafo  $G = (V, E)$  è **bipartito** se esiste una partizione  $(V_1, V_2)$  dell'insieme dei vertici  $V$ , tale che tutti gli archi hanno come estremi un vertice di  $V_1$  ed un vertice di  $V_2$ . Un grafo bipartito è completo ( $K_{n,m}$ ) se esiste un arco per ogni coppia (della bipartizione) di vertici.

**SOTTOGRAFO:** dato un grafo  $G = (V, E)$ , un sottografo di  $G$  è un grafo  $G' = (V', E')$  tale che  $V' \subseteq V$  e  $E' \subseteq E$ . Dato un sottoinsieme  $V'$  di  $V$ , si definisce **sottografo indotto** da  $V'$  il sottografo di  $G$  che ha come insieme dei vertici l'insieme  $V'$  e come insieme degli archi l'insieme di tutti gli archi di  $G$  che collegano nodi in  $V'$ . Un sottografo  $G' = (V', E')$  è **ricoprente** per  $G$  se  $V' = V$ .



**RAPPRESENTAZIONE IN MEMORIA DEI GRAFI:** vi sono diverse tecniche per rappresentare un grafo in memoria:

**LISTA DI ADIACENZA:** consiste in un vettore  $Adj[]$  grande quanto il numero di nodi ( $v$ ) del grafo, con  $Adj[v]$  che punta ad una lista contenente gli indici di tutti i nodi adiacenti a  $v$  in  $G$ .

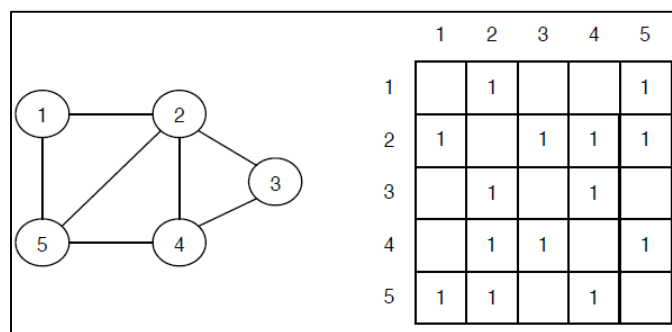


Nel caso di grafi non orientati, un arco  $(u,v)$  può apparire due volte, al contrario in quelli orientati l'arco  $(u,v)$  uscente da  $u$  ed entrante in  $v$  appare una sola volta.

**MATRICE DI ADIACENZA:** consiste in una matrice  $A$  di  $n^2$  numeri interi nella quale:

- $A[i,j]=1$  se e solo se l'arco  $(i,j) \in E(G)$ ;
- $A[i,j]=0$  altrimenti

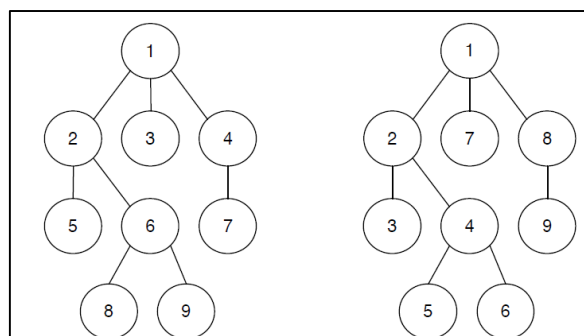
In altre parole, se l'arco esiste nella matrice verrà scritto 1 nel caso di nodi adiacenti, 0 zero in caso contrario.



**OCCUPAZIONE DI MEMORIA:** nelle rappresentazioni viste

- Liste di adiacenza  $\Theta(n + m)$
- Matrice di adiacenza  $\Theta(n^2)$

**VISITA DI GRAFI:** partendo da uno specifico vertice, detto *vertice di partenza*, e di solito identificato con  $s$ , si può desiderare di accedere a tutti i vertici che si trovano a una certa distanza da esso prima di accedere a quelli situati a distanza maggiore, e in questo caso si parla di **visita in ampiezza (BFS, Breadth-first search)**; oppure si vuole accedere ai vertici allontanandosi sempre di più dal vertice di partenza finché è possibile, e in questo caso si parla di **visita in profondità (DFS, depth-first search)**.



BFS a sinistra, DFS a destra

La visita in profondità è molto adatta ad essere realizzata in modo ricorsivo ma può essere definita in modo iterativo con l'ausilio di una pila. Viceversa, la visita in ampiezza è poco adatta a una implementazione ricorsiva e viene generalmente realizzata in modo iterativo con l'ausilio di una coda. Queste due visite sono concettualmente piuttosto semplici però, nel momento in cui si passa dagli alberi ai grafi ci sono alcune complicazioni in più: dato che nei grafi possono essere presenti dei cicli (assenti negli alberi) la visita deve essere in grado di "ricordarsi" se un determinato vertice  $v$  è stato già visitato, poiché è possibile che diversi cammini che originano nel vertice di partenza

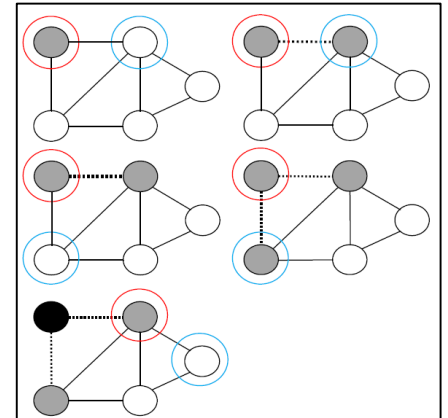
conducano a  $v$ , e in tal caso il vertice  $v$  sarà “scoperto” più volte. Per questa ragione nelle visite di grafi è necessario gestire alcune informazioni aggiuntive.

**VISITA IN AMPIEZZA (BFS):** dato un grafo  $G$  ed uno specifico vertice di partenza  $s$ , la visita in ampiezza (BFS, Breadthfirst search) si muove sistematicamente lungo gli archi di  $G$  per “scoprire”, e successivamente visitare, ogni vertice  $v$  raggiungibile da  $s$ .

La visita in ampiezza espande la “frontiera” fra vertici già scoperti e vertici non ancora scoperti, allontanandola uniformemente e progressivamente dal vertice di partenza. In particolare, la visita scopre tutti i vertici a distanza  $k$  dal vertice di partenza prima di scoprire anche un solo vertice a distanza  $k + 1$ . Viene costruito un albero breadth-first (un albero “scoperto”, che inizialmente consiste del solo vertice  $s$ , la sua radice) che, alla fine, contiene tutti i vertici raggiungibili da  $s$ .

Come accennato nel paragrafo precedente, è necessario tenere traccia dei vertici già visitati per evitare di visitarli più di una volta. A tale scopo i vertici vengono opportunamente **marcati**.

La scansione inizia marcando il vertice  $s$  d’inizio (vertice cerchiato di rosso nella figura) e continua cercando i vertici adiacenti al vertice già marcato  $u$ ; ogni volta che un nuovo vertice non marcato  $v$  viene scoperto (vertici cerchiati in azzurro nella figura), il vertice  $v$  viene marcato (reso grigio in figura) e l’arco  $(u, v)$  viene aggiunto all’albero breadth-first (arco tratteggiato nella figura). Non appena fra tutti gli adiacenti di  $u$  non vi è più alcun vertice bianco, il lavoro sul vertice  $u$  è terminato ( $u$  è reso nero in figura) e si prosegue passando ad un altro nodo marcato.



```

Funzione Breadth-first_search (G: grafo; q: puntatore a coda)
    Assegna a tutti i nodi v //inizializzazione
        marked[v] ← false
        pred[v] ← NULL
    scegli il nodo di partenza s //inizio da s
    marked[s] ← true
    Enqueue(q, s)
    finché la coda Q non è vuota
        u ← Dequeue(q) // u è il vertice su cui lavorare
        visita il vertice u //operazione generica
        per ogni vertice v nella lista di adiacenza di u
            if marked[v] = false then
                marked[v] = true
                pred[v] ← u
                Enqueue(q, v)
    return
    
```

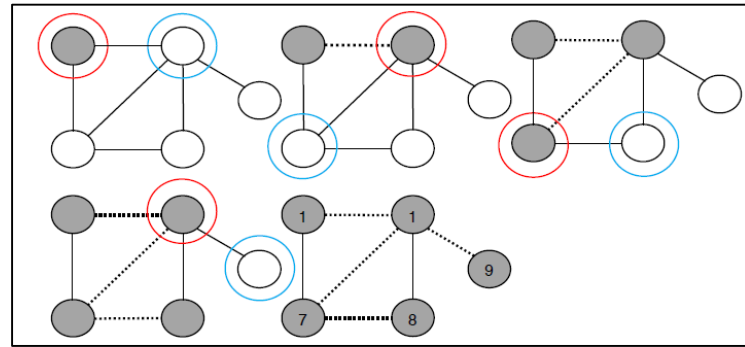
L’albero breadth-first si realizza mediante un campo aggiuntivo in ogni vertice, il campo predecessore: aggiungere l’arco  $(u, v)$  all’albero breadth-first significa memorizzare nel campo predecessore( $v$ ) l’indice  $u$ . Poiché ogni vertice viene marcato solo una volta, ogni vertice avrà un solo predecessore e questo garantisce l’assenza di cicli nell’albero breadth-first: infatti, se vi fosse un ciclo, almeno un vertice del ciclo avrebbe due predecessori.

L’algoritmo BFS è iterativo, e la complessità totale è  $O|V| + |E|$  data da inizializzazione + scansione liste adiacenza

**VISITA IN PROFONDITÀ (DFS):** dato un grafo  $G$  ed un vertice di partenza  $s$ , la visita in profondità (DFS, Depth-first search), a differenza di quella in ampiezza, esplora il grafo allontanandosi sempre più dal vertice di partenza finché è possibile, per poi ripartire da vertici più vicini a quello di partenza solo quando non è possibile allontanarsi ancora.

In particolare, se la visita in profondità muovendosi lungo l’arco  $(u, v)$  del grafo “scopre” il vertice  $v$ , proseguirà percorrendo uno degli archi uscenti da  $v$  e non percorrendo uno degli altri archi uscenti da  $u$ .

Il lavoro su un vertice  $u$  già marcato (quindi già scoperto – vertice cerchiato di rosso nella figura) consiste nella ricerca di un suo adiacente non marcato  $v$  (vertici cerchiati in azzurro nella figura). Il vertice  $v$  viene marcato (reso grigio in figura) e l'arco  $(u, v)$  viene aggiunto all'albero depth-first (arco tratteggiato nella figura). Il lavoro prosegue quindi sul vertice  $v$  anziché (come nella ricerca breadth-first) su un ulteriore adiacente di  $u$ . Si torna a lavorare su vertici precedenti solo quando non è più possibile proseguire il lavoro allontanandosi ulteriormente vertice di partenza.



```

Funzione Depth_first_search_ricorsiva (u: vertice)

    marked[u] ← true
    "visita u"
    "per ogni vertice v nella lista di adiacenza di u"
        if marked[v] = false then
            pred[v] ← u
            Depth_first_search_ricorsiva (v)
    return

```

L'algoritmo DFS è ricorsivo, e la complessità totale è

$$O(n) + \sum_{i=1}^n \text{degree}(v_i) = O(n) + O(m)$$

**Ordinamento topologico:** la ricerca in profondità si può usare per ordinare topologicamente un grafo orientato aciclico (detto anche DAG: Directed Acyclic Graph). Un ordinamento topologico di un grafo orientato aciclico  $G = (V, E)$  è un ordinamento (permutazione) dei suoi vertici tale che per ogni arco  $(u, v) \in E$  il vertice  $u$  precede il vertice  $v$ .

```

TP_DFS(G)
for v ∈ V[G] do
    color[v] ← bianco, p[v] ← nil
t ← 1
TP ← nil // lista concatenata vuota
for v ∈ V[G] do
    if color[v] = bianco then TP_DFSric(v, t)
return TP;

TP_DFSric(u, t)
color[u] ← grigio, d[u] ← t, t ← t + 1
for v ∈ Adj[u] do
    if color[v] = bianco then p[v] ← u, TP_DFSric(v, t)
color[u] ← nero, f[u] ← t, t ← t + 1
TP ← u+TP //inserimento in testa alla lista concatenata

```

La complessità è la stessa di DFS, ossia  
 $O(|V| + |E|)$

**Componenti fortemente connesse:** la ricerca in profondità si può usare anche per calcolare le componenti fortemente connesse di un grafo orientato. Una componente fortemente connessa (**cfc**) di un grafo orientato  $G = (V, E)$  è un insieme massimale di vertici  $U \subseteq V$  tale che per ogni  $u, v \in U$  esiste un cammino da  $u$  a  $v$  ed un cammino da  $v$  ad  $u$ . L'algoritmo per il calcolo delle componenti fortemente connesse si compone di tre fasi:

1. usa la ricerca in profondità in  $G$  per ordinare i vertici in ordine di tempo di fine  $f$  decrescente (come per l'ordinamento topologico):  $O(|V| + |E|)$
2. calcola il grafo trasposto  $G^T$  del grafo  $G$ :  $O(|V| + |E|)$
3. esegue una ricerca in profondità in  $G^T$  usando l'ordine dei vertici calcolato nella prima fase nel ciclo principale:  $O(|V| + |E|)$

Alla fine gli alberi della ricerca in profondità in  $G^T$  rappresentano le componenti fortemente connesse.

**Proprietà dei cammini:** Se due vertici  $u$  e  $v$  sono in una stessa cfc allora tale cfc contiene anche i vertici di tutti i cammini da  $u$  a  $v$ .

**Proprietà degli alberi:** una ricerca in profondità mette nello stesso albero tutti i vertici di una cfc.

**Grafo delle componenti:** dato un grafo orientato  $G$ , il grafo delle componenti fortemente connesse di  $G$  è il grafo orientato  $H$  avente come vertici le componenti fortemente connesse di  $G$  e un arco da una cfc  $C$  ad una cfc  $C'$  se e solo se in  $G$  vi è un arco che connette un vertice di  $C$  ad un vertice di  $C'$ .

**Proprietà:** Il grafo  $H$  delle componenti fortemente connesse di  $G$  è aciclico.