APPUNTI
Algoritmi e Struttire Dati

Francesco De Lucchini
Università di Pisa
2022

1. Introduzione

Definizioni

Si definisce **algoritmo** una sequenza finita di istruzioni non ambigue finalizzata a risolvere un dato problema in un tempo finito, utilizzando eventualmente una quantità finita di memoria per i risultati intermedi. E' diverso da un **programma**, ossia la codifica di un dato algoritmo in un particolare linguaggio di programmazione

Caratteristiche di un algoritmo:

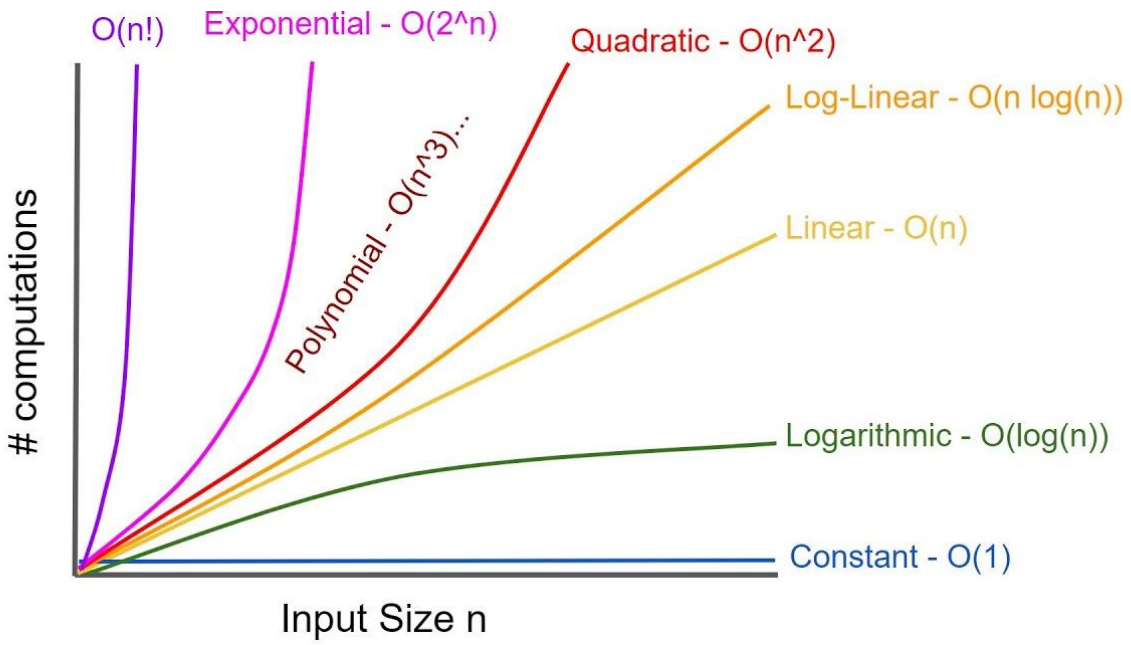
- Istanza — *gli input del problema*
- Dimensione dell'istanza — *quanti sono gli input (istanze diverse possono avere dim uguale)*
- Modello di calcolo — *di che tipo sono gli input*
- Complessità temporale — *numero (#) totale di operazioni che esegue*

In generale la complessità di un algoritmo è una funzione *(sempre positiva)* che alla dimensione di un problema associa il costo *(tempo, memoria…)* della sua risoluzione, è diversa dal **profiling**, tecnica di analisi basata sulla misurazione di indici prestazionali dipendenti dall'ambiente di esecuzione.

Per misurare l'efficienza di un programma senza dover aspettare l'esecuzione effettiva dei calcoli e indipendentemenete dall'hardware del calcolatore si **analizza asintoticamente la funzione complessità**, ossia quella funzione che all'aumentare della complessità dell'input associa una quantità *(tempo impiegato, memoria occupata…)* che determina l'efficienza del programma.

[*Notazione: con Very long si indica un tempo di esecuzione* $> 10^{25}$ *anni*]

Classi di complessità



NOTAZIONE *O* GRANDE

Rappresenta il limite asintotico **superiore**, una funzione $f(n)$ si dice di ordine $O(g(n))$ se da un certo indice in poi f sta sempre al di sotto o pari a g, in termini matematici se:

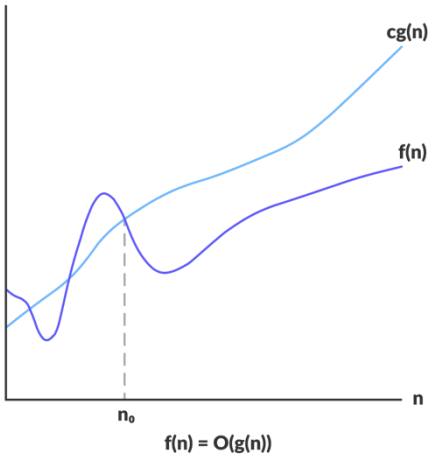
$$\exists\ v \in \mathbb{N}, \lambda > 0 \in \mathbb{R} :\ \forall\, n \geq v\ \ f(n) \leq \lambda\, g(n)$$

Esempi:

$$O(1) = \{\, k\ \forall k \in \mathbb{N}\,\}$$
$$O(n) = O(1)\ \cup\ \{\, n,\ 20n,\ 12 + 100n\,, \ldots\}$$
$$O(n^2) = O(n)\ \cup\ \{\, n^2,\ 20n^2,\ 25n^2 + 12n\,, \ldots\}$$

Inoltre:

- Per ogni costante positiva $\lambda\ \ O\big(f(n)\big) = O(\lambda f(n))$
- Se $f(n)$ è $O\big(g(n)\big)$ allora $f(n) + g(n)$ è $O\big(g(n)\big)$
- $f(n)g(n)$ è $O(f(n)g(n))$



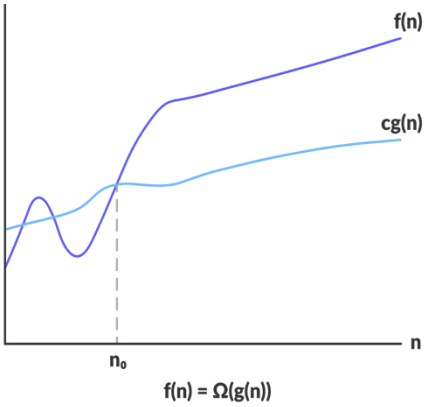
NOTAZIONE Ω GRANDE *(omega)*

Rappresenta il limite asintotico **inferiore**, una funzione $f(n)$ si dice di ordine $\Omega(g(n))$ se da un certo indice in poi f sta sempre al di sopra o pari a g, in termini matematici se:

$$\exists\ v \in \mathbb{N}, \lambda > 0 \in \mathbb{R} :\ \forall\, n \geq v\ \ f(n) \geq \lambda\, g(n)$$

Esempi:

$$\Omega(1) = \Omega(n)\ \cup\ \Omega(n^2)\ \cup\ \Omega(\ldots$$
$$2n^2 + 3n\ \text{ è }\ \Omega(\,n)$$

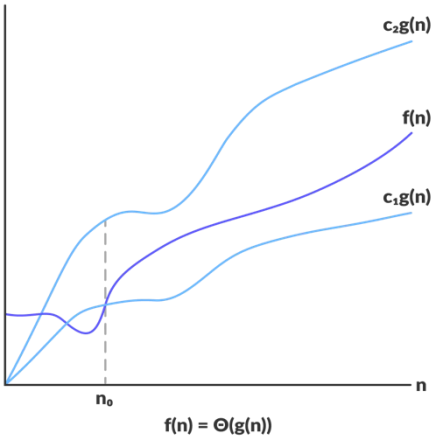


NOTAZIONE Θ GRANDE *(theta)*

Rappresenta il limite asintotico **stretto**, una funzione $f(n)$ si dice di ordine $\Theta(g(n))$ se $f(n)$ è $O(g(n)\ e\ f(n)$ è $\Omega(g(n))$, in altre parole quando hanno lo stesso ordine di complessità, in termini matematici: $\exists\ v \in \mathbb{N}, \lambda, \mu > 0 \in \mathbb{R} :\ \forall\, n \geq v\ \ \mu g(n) \leq f(n) \leq \lambda g(n)$

Esempi:

$$2n^2 + 3n\ \text{ è }\ \Theta(n^2)$$



2. Valutazione dei programmi ricorsivi

Substitution method *(Risoluzione matematica per induzione)*

```
int fact(int x) {
  if (x == 0) return 1;
  else return x*fact(x-1);
}
```

O(1) O(1) ⇒ T(0) = a → costante

O(1) O(m-1) ⇒ T(m) = b + T(m-1)

Risolvo la relazione:

$T(0) = 0b + a$
$T(1) = 1b + a$
$T(2) = 2b + a$
$\vdots$
$T(m) = mb + a$

⇒ T(m) è O(m)

Recurrence tree method *(Risoluzione grafica)*

Questo metodo consiste nel disegnare l'albero delle chiamate ricorsive, sommando il lavoro compiuto ad ogni livello dell'albero e moltiplicandolo per la lunghezza dell'albero, in modo da ottenere la complessità asintotica dell'algoritmo

Master theorem *(Risoluzione tramite schemi di ricorrenza noti)*

Supponiamo un algoritmo basato sul ***divide et impera***, la sua relazione di ricorrenza sarà del tipo:

$T(n) = d$	$se\ n \leq m$	*Caso/i base costante*
$T(n) = hn^k + aT(\,^n/_b\,)$	$se\ n > m$	*Caso generico (con $h > 0$)*

Questa relazione di ricorrenza è nota e risolve nel modo seguente:

$T(n)$ è $O(n^k)$	$se\ a < b^k$
$T(n)$ è $O(n^k \log(n))$	$se\ a = b^k$
$T(n)$ è $O(n^{\log_b a})$	$se\ a > b^k$

Supponiamo un algoritmo con una **relazione di ricorrenza lineare**:

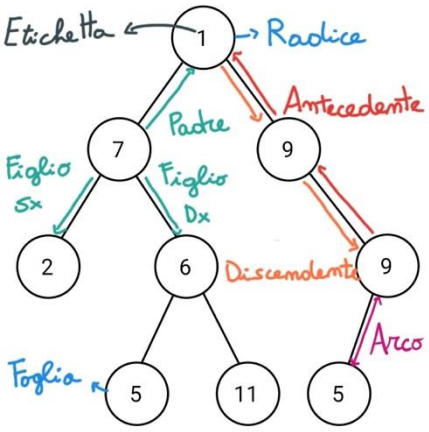
$T(0) = d$	*Caso/i base costante*
$T(n) = hn^k + aT(\,n-1\,)$	*Caso generico (con $h > 0$)*

Questa relazione di ricorrenza è nota e risolve nel modo seguente:

$T(n)$ è $O(n^{k+1})$	$se\ a = 1$	***Polinomiale***
$T(n)$ è $O(a^n)$	$se\ a \neq 1$	***Esponenziale***

3. Strutture dati

Alberi binari



Etichetta di un nodo

- È un campo informativo
- Più nodi possono avere la stessa etichetta

Livello di un nodo

- Equivale al numero antecedenti
- Un albero binario vuoto ha livello -1
- La radice ha livello 0

Livello di un albero binario

- Equivale al massimo livello di un nodo
- Equivale al cammino più lungo tra la radice e una foglia

Un albero binario si dice:

- **Bilanciato** se i nodi di tutti i livelli tranne quelli dell'ultimo hanno due figli
- **Quasi bilanciato** se è bilanciato fino al penultimo livello
- **Completo** se ogni nodo possiede esattamente due figli oppure non ne possiede (foglia)
- **Degenere** se tutti i nodi escluse le foglie hanno solamente figli o tutti a sinistra o tutti a destra, concettualmente rappresenta una lista.

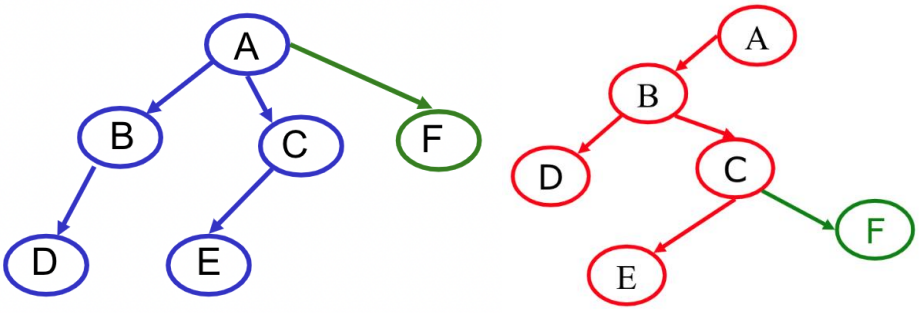
Un albero binario con S foglie ha <u>almeno</u> $\log_2 S$ *livelli*

Un albero binario bilanciato con K livelli ha <u>esattamente</u> $2^{(k+1)} - 1$ *nodi di cui* 2^k *sono foglie*

Un albero binario completo ha tanti nodi interni quante foglie meno una

Alberi generici

Sono alberi in cui ogni nodo può avere infiniti figli, che tra di loro si dicono **fratelli**. Un albero generico viene memorizzato in un albero binario tramite la **tecnica figlio-fratello**, secondo la quale il primo figlio viene memorizzato a sinistra dell'albero binario e il primo fratello a destra.



Alberi binari di ricerca

Sono alberi binari tali che:

- L'etichettà è sufficiente per stabilire un ordine tra i nodi
- Ogni figlio sinistro ha etichetta minore o uguale al padre
- Ogni figlio destro ha etichetta strettamente maggiore del padre

Heap

Uno heap è un'**albero binario quasi bilanciato in cui i nodi dell'ultimo livello sono addossati a sinistra** (*non ha buchi*) e in ogni sottoalbero l'etichetta della radice è maggiore o uguale (*max heap*) oppure minore o uguale (*min heap*) di quella di tutti i discendenti.

Questa struttura dati è particolarmente indicata per implementare una coda con priorità

Alberi di decisione

Sono alberi binari che **rappresentano algoritmi basati su confronti:**

- Ogni arco rappresenta una decisione (si/no)
- Ogni foglia rappresena una soluzione per un dato input
- Ogni cammino radice-foglia rappresenta un esecuzione dell'algoritmo (sequenza di confronti)
- L'altezza dell'albero corrisponde al numero massimo di confronti che esegue l'algoritmo

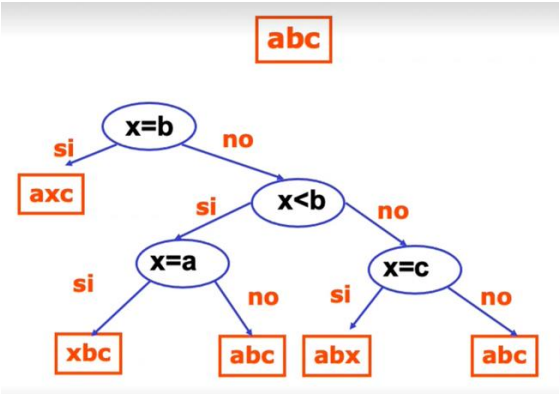
Un algoritmo che risolve un problema che ha S soluzioni ha un albero di decisione con almeno S foglie, posso infatti arrivare alla stessa soluzione in più modi diversi e dunque ripetere la foglia. Tra tutti gli alberi di decisione che rappresentano gli algoritmi che risolvono un problema:

- L'albero con il percorso di lunghezza massima (*ossia l'altezza*) minore rappresenta il limite inferiore del caso peggiore
- L'albero con la lunghezza dei percorsi media minore rappresenta il limite inferiore del caso medio

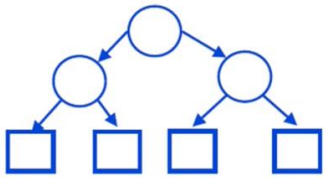
Limite inferiore è la complessità del migliore algoritmo teorico per risolvere un certo problema e si calcola facendo il logaritmo in base 2 del numero di soluzioni, è teorico perché non è detto che esista effettivamente, ad esempio la ricerca di un elemento in un insieme *(senza presupposti sull'ordinamento)* ha n possibili soluzioni, tuttavia il limite inferiore è $log(n)$.

Un altro esempio può essere il limite inferiore per algoritmi di ordinamento basati sul confronto: le soluzioni possibili sono $n!$, dunque limite inferiore è $log(n!)$ ossia $nlogn$, dunque per esempio il merge sort è ottimo.

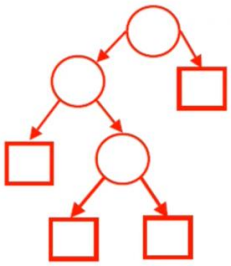
Esempio di albero di decisione: ricerca binaria



Albero ottimo *(limite inferiore)* a sinistra



cammino max :2
cammino medio: 2



cammino max : 3
cammino medio: 2,25

Hash

Una **funzione hash** è una funzione che mappa un valore in input ad un valore in output ottenuto effettuando operazioni matematiche sul dato in ingresso. Particolari funzioni hash possono essere programmate per restituire sempre valori compresi tra un certo range, ad esempio potrei programmare una funzione hash per associare ad ogni valore di un array il suo indice nell'array, in questo modo per cercare un elemento mi basterebbe calcolarne il suo hash.

Perché questo funzioni il numero di elementi possibili da memorizzare deve essere minore o uguale alla dimensione dell'array, in altre parole la funzione deve essere **iniettiva**, con questa premessa è possibile cercare un elemento in un array con il solo costo di calcolo della funzione hash, molto spesso costante ossia trascurabile.

Metodo di hash ad **ACCESSO DIRETTO**

L'indice nell'array di un certo valore è il valore stesso, si usa quindi un array di booleani senza dover memorizzare effettivamente l'elemento. È senza dubbio il metodo più efficiente ma risulta utilizzabile solamente se l'intervallo di possibili elementi è ridotto.

Metodo di hash ad **INDIRIZZAMENTO APERTO**

In realtà anche per richieste molto semplici, come memorizzare una parola di pochi caratteri, si genera un numero enorme di possibili valori, dunque è impossibile allocare preventivamente un array più lungo di tutte le possibilità e di conseguenza sarà inevitabile il verificarsi di **collisioni**, ossia elementi diversi avranno lo stesso valore della funzione hash e dunque lo stesso indice.

Mi accorgo di aver ottenuto una collisione se cercando un valore trovo il suo posto occupato da un altro, in questo sfortunato caso posso applicare varie strategie, la più nota è la **scansione lineare**, ossia la ricerca nelle posizioni successive fino a trovare il valore richiesto o trovare una posizione vuota, scorrendo l'array in modo circolare.

Il difetto di questo approccio è che **una volta inserito un valore non posso più cancellarlo**, questo perché se dovessi eliminarlo sovrascrivendo il valore di default, tutti i valori in collisione vengono persi, inoltre questo approcio può causare agglomerati, ossia blocchi di elementi adiacenti con indirizzi hash diversi non nel posto in cui dovrebbero essere, i quali aumentano la probabilità di effettuare confronti e dunque il tempo di ricerca.

Per consentire la cancellazione di un elemento senza preoccuparmi delle collisioni invece di mettere il valore di default per l'elemento libero scrivo un altro **valore di default dedicato ai blocchi cancellati**, in questo modo viene modificato solo l'inserimento per poter scrivere nuovi elementi su questo valore di default.

La scansione lineare può anche essere modificata per cercare non nella posizione strettamente successiva nella cella successiva sommata o moltiplicata per una costante o per se stessa (***scansione quadratica***), la diversa lunghezza del passo di scansione riduce gli agglomerati, **ma è necessario controllare che la scansione visiti ogni posizione dell'array.**

Generalmente il tempo medio di ricerca, ossia il numero medio di confronti, dipende da:

- **Fattore di carico α**

  Rappresenta il numero medio di elementi per posizione, si ottiene tramite il rapporto tra possibili valori e dimensione dell'array (*sempre <= 1*), **minore è meglio è**.

- **Tecnica di scansione**

  La scansione quadratica si rivela essere migliore della lineare che è minore o uguale a $1/(1-\alpha)$

$\frac{n}{M}$	scansione lineare	scansione quadratica
10%	1.06	1.06
50%	1.50	1.44
70%	2.16	1.84
80%	3.02	2.20
90%	5.64	2.87

- **Uniformità della funzione hash**

  Probabilità di ottenere una collisione

Nota: Per garantire sempre prestazioni accettabili è necessario ristrutturare l'array di tanto in tanto

Metodo di hash a **CONCATENAZIONE**

Invece di memorizzare un array di valori **memorizzo un array di puntatori** dove ogni puntatore equivale ad una lista di elementi che collidono su quel particolare indice, questo consente di **evitare del tutto gli agglomerati** e riduce la dimensione dell'array necessaria, il **fattore di carico sarà sicuramente maggiore o uguale ad uno**.

Generalmente il tempo medio di ricerca (e anche di cancellazione) è $O(\alpha)$ perché per ogni lista in media ho α elementi, mentre il tempo di inserimento, considerando un inserimento in testa, è costante. Il problema principale di questo metodo è che **viene sprecata memoria per salvare i puntatori**, nel caso di strutture complesse questa dimnesione è irrilevante ma nel caso di interi si occupa il doppio della memoria.

In conclusione possiamo dire che il principale utilizzo delle funzioni hash è quello di generalizzare un array, creando un **dizionario** che permette l'accesso ai suoi elementi tramite una chiave e non tramite un indice.

4. Programmazione dinamica

La programmazione dinamica è un modo di programmare che viene spesso usato per risolvere problemi di ottimizzazione, quando non è possibile applicare il metodo del divide et impera perché non si sa con esattezza **quali** sottoproblemi risolvere, allora si risolvono **tutti** i sottoproblemi e si conservano i risultati ottenuti per poi arrivare ad una soluzione ottima del problema globale.

Un problema può essere risolto tramite la programmazione dinamica se presenta due caratteristiche:

- **Sottostruttura ottima**
  Una soluzione ottima del problema globale contiene la soluzione ottima dei sottoproblemi

- **Sottoproblemi comuni**
  Il problema può esssere risolto tramite un algoritmo ricorsivo, che eventualmente risolve lo stesso sottoproblema più volte

Esempio di programmazione dinamica
Sottosequenza comune più lunga di due sequenze di caratteri

Una sottosequenza è una sequenza ottenuta cacellando alcuni elementi e mantenendo nello stesso ordine gli altri, dobbiamo trovare la più lunga sequenza che è sottosequenza di entrambe. Diamo una definizione ricorsiva del problema:

Siano a e b sequenze e sia $L(i,j)$ la lunghezza delle PLSC di $a_1, \dots, a_i$ e $b_1, \dots, b_i$, allora:

- $L(0,0) = L(i,0) = L(0,j) = 0$
  La PLSC tra due sequenze di cui almeno una è vuota è chiaramento zero
- $L(i,j) = L(i-1, j-1) + 1 \quad \text{se} \quad a_i = b_j$
  Se due caratteri sono uguali allora la lunghezza della PLSC è la PLSC che avevo prima più uno
- $L(i,j) = \max\left[L(i,j-1), L(i-1,j)\right] \quad \text{se} \quad a_i \neq b_j$
  Se due caratteri sono diversi allora la lunghezza della PLSC è la massima tra la PLSC con a_i ma senza b_i e quella senza a_i ma con b_i

Una volta definito il problema, per vedere se è possibile applicare la programmazione dinamica, è necessario verificare le due condizioni:

- La condizione di sottostruttura ottima è verificata perché se *X* è una PLSC *(soluzione ottima di un sottoproblema)*, sicuramente *X* sarà prefisso di una PLSC tra *a* e *b* completi *(soluzione ottima del problema globale).*
- La condizione di sottoproblemi comuni è verificata perché per costruzione il problema può essere risolto tramite un algoritmo ricorsivo *(complessità esponenziale in minimo tra n ed m)*.

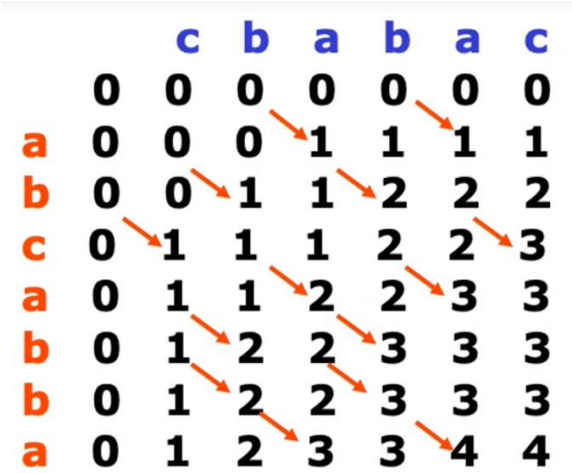
Si può dunque procedere applicando la programmazione dinamica, ossia, invece di chiamare ricorsivamente i casi base, posso costruire iterativamente tutti i casi possibili, cioè tutti i $L(i,j)$, partendo dai valori di i e j più piccoli, salvando i risultati in una matrice, per poi arrivare a calcolare la soluzione al problema globale $L(n,m)$ sfruttando i risultati intermedi.

Esempio di esecuzione dell'algoritmo per individuare le lunghezze numeriche delle PLSC

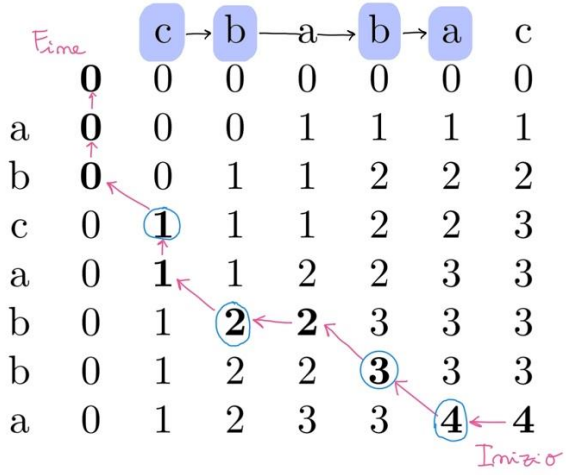
*Complessità O(n*m)*

Esempio di esecuzione dell'algoritmo per trovare le PLSC in base alla tabella delle sequenze *(algoritmo greedy come vedremo)*

Complessità O(n+m)



		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1
b	0	0	1	1	2	2	2
c	0	1	1	1	2	2	3
a	0	1	1	2	2	3	3
b	0	1	2	2	3	3	3
b	0	1	2	2	3	3	3
a	0	1	2	3	3	4	4



		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1
b	0	0	1	1	2	2	2
c	0	1	1	1	2	2	3
a	0	1	1	2	2	3	3
b	0	1	2	2	3	3	3
b	0	1	2	2	3	3	3
a	0	1	2	3	3	4	4

La matrice delle lunghezze si costruisce per colonne seguendo le semplici regole:

- Se $a_i = b_j$ prendo il valore nella diagonale precedente *(in alto a sinistra)* incrementato di uno
- Se $a_i \neq b_j$ prendo il massimo tra il valore sopra e quello a sinistra

Le PLSC si ricavano, partendo dall'utlimo elemento della matrice, seguendo le semplici regole:

- Se $a_i = b_j$ vado nella diagonale precedente *(in alto a sinistra)*
- Se $a_i \neq b_j$ vado nel massimo tra il valore sopra e quello a sinistra

5. Algoritmi greedy *(avidi, golosi)*

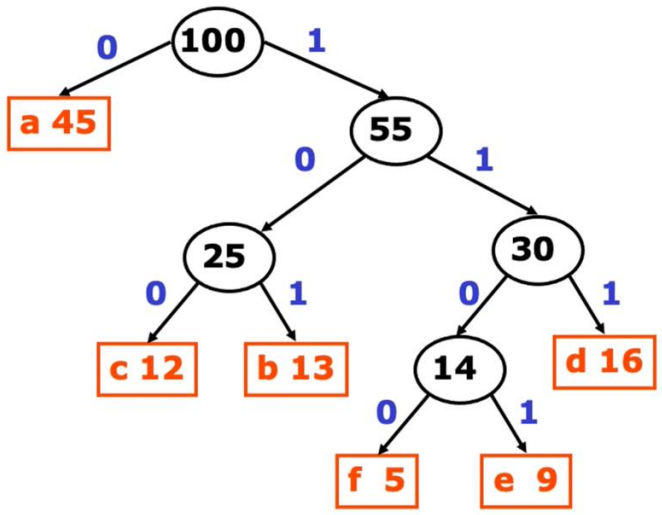
Sono algoritmi progettati per **risolvere in maniera ottimale un problema la cui soluzione si ottiene mediante una sequenza di scelte**, ad esempio il riempiemento ottimale del portabagagli di una macchina data una serie di oggetti. Per ogni scelta l'algoritmo deve prendere la strada che al momento sembra migliore, questa scelta dovrebbe sempre essere in accordo con la scelta globale, ossia non si dovrebbero perdere alternative che potrebbero rivelarsi migliori in seguito.

Algoritmo di Huffman *(greedy)*

Considerando un alfabeto di caratteri ed un codice binario assegnato a ciascun carattere, il problema è trovare il codice binario per ogni carattere che comprima al massimo un testo dato in input, sapendo la frequenza di ciascun carattere all'interno del testo.

Come primo passo bisogna scegliere se assegnare un codice di lunghezza fissa o variabile, chiaramente il secondo è più efficiente ma per non ottenere ambiguità è necessario tenere conto di una semplice regola: nessun codice può essere prefisso di un altro codice.

In questo modo ogni codice prefisso è unico e può essere facilmente rappresentato tramite un albero binario, l'algoritmo di Huffman descrive come creare questo albero nel modo più efficiente possibile.



L'algoritmo di Huffman inizia con una foresta di alberi di un solo nodo dove la radice è la frequenza e il nodo è il carattere con tale frequenza, ad ogni passo vengono fusi i due alberi con radice minore in un albero unico avente come radice la somma delle due.

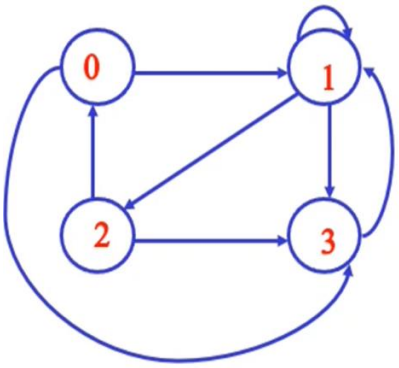
La creazione di questo albero binario può avvenire attraverso estrazioni da un **min heap** dei vari sottoalberi, in questo preciso caso la complessità è $O(nlog(n))$

6. Grafi

Grafi orientati

Un grafo è un insieme $G(N, A)$ di **nodi** N ed **archi** A, in particolare si parla di grafo **orientato** se un arco collega in modo ordinato due nodi, in un grafo di questo tipo il numero massimo di archi è N^2.

Si definisce **cammino** una sequenza di nodi collegata da archi orientati, e si definisce **ciclo** un cammino che inizia e termina nello stesso nodo, un grafo è detto **aciclico** se non contiene cicli.

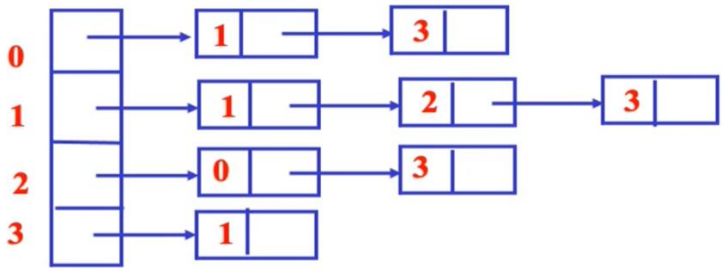


Primo metodo di rappresentazione in memoria: **liste di adiacenza**

Questo metodo è conveniente se il grafo è sparso, ossia ha pochi collegamenti

Viene definito un array con dimensione uguale al numero di nodi

Ogni elemento nell'array rappresenta la lista di un nodo con i suoi successori



Nota: nulla vieta che un nodo sia successore di se stesso *(come ad esempio 1)*

Primo metodo di rappresentazione in memoria: **matrice di adiacenza**

Questo metodo conviene se il grafo è denso, ossia ha molti collegamenti

Viene definita una matrice quadrata di dimensione $N * N$

L'elemento di indici *i j* della matrice vale uno se esiste un arco dal nodo *i* al nodo *j*, zero altrimenti

	0	1	2	3
0	0	1	0	1
1	0	1	1	1
2	1	0	0	1
3	0	1	0	0

Ovviamente questi metodi funzionano se il nome dei nodi degli archi va da *0* ad *n*, qualora avessi ad esempio nodi etichettati con dei caratteri dovrei creare un **vettore per mappare ogni indice all'etichetta di un nodo**. È anche possibile **etichettare gli archi**, ossia dare un peso ai collegamenti tra nodi, nel caso di liste di adiacenza questo valore andrà nella struttura del nodo, nel caso di matrice di adiacenza questo valore andrà al posto del valore di default uno.

Grafi non orientati

Si parla di grafo **non orientato** se gli archi non hanno un verso, in un grafo di questo tipo il numero massimo di archi è $n(n-1)/2$.

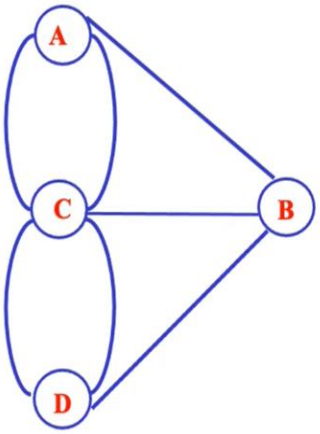
Due nodi si dicono **adiacenti** se collegati da un arco, si definisce **cammino** una sequenza di nodi adiacenti, e si definisce **ciclo** un cammino che inizia e termina nello stesso nodo senza ripetizioni, un grafo non orientato è detto **connesso** se esiste un cammino fra due nodi qualsiasi del grafo.

In memoria un grafo non orientato può essere visto come un grafo orientato dove ogni arco è presente in entrambi i versi, nel caso volessi rappresentarlo con una matrice, questa sarebbe sempre simmetrica.

Multi-grafi

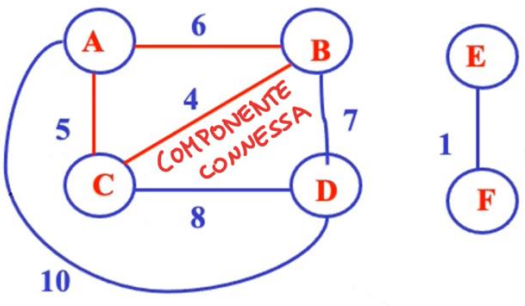
Un multi-grafo (*orientato o meno*) è un insieme $G(N, A)$ di **nodi** N ed **archi** A dove tra due nodi è possibile stabilire più archi, possono risultare utili per esprimere, ad esempio, i diversi collegamenti con le relative velocità tra due città.

Un multi grafo è detto **hamiltoniano** se esiste un ciclo che tocca tutti i nodi una ed una sola volta, mentre è detto **euleriano** se esiste un ciclo che tocca tutti gli archi una ed una sola volta (esiste se e solo se da ogni nodo parte un numero pari di archi).



Minimo albero di copertura

Si definisce **componente connessa** un qualsiasi sottografo connesso, questa si dice **massimale** se nessun nodo della componente connessa appartiene ad un'altra componente connessa, ossia devo prendere l'intero sottografo connesso



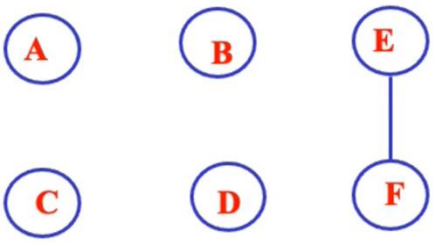
Si definisce **albero di copertura** l'unione aciclica di tutte le componenti connesse massimali, supponendo che gli archi siano pesati, questo albero è detto **minimo** se la somma dei pesid degli archi è minore di quella di tutti gli altri possibili alberi di copertura.

Algoritmo di Kruskal

Dato un grafo non orientato con archi pesati produce il suo minimo albero di copertura

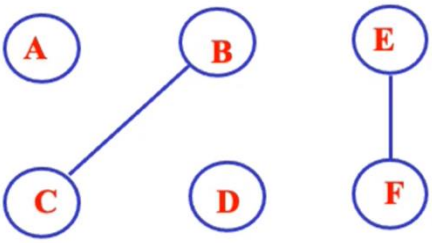
Ordina gli archi del grafo in ordine crescente Per ogni arco: Se connette due componenti non connesse: Includi l'arco nel minimo albero di copertura Connetti le componenti

Esempio di esecuzione *(sul grafo nella parte del minimo albero di copertura)*



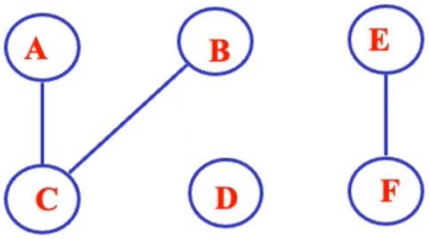
(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A} {B} {C} {D} {E, F}



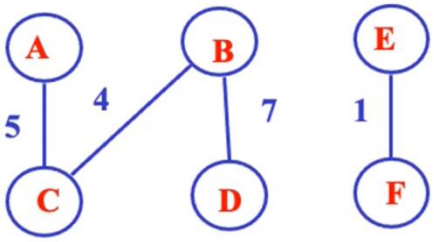
(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A} {B, C} {D} {E, F}



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A, B, C} {D} {E, F}



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

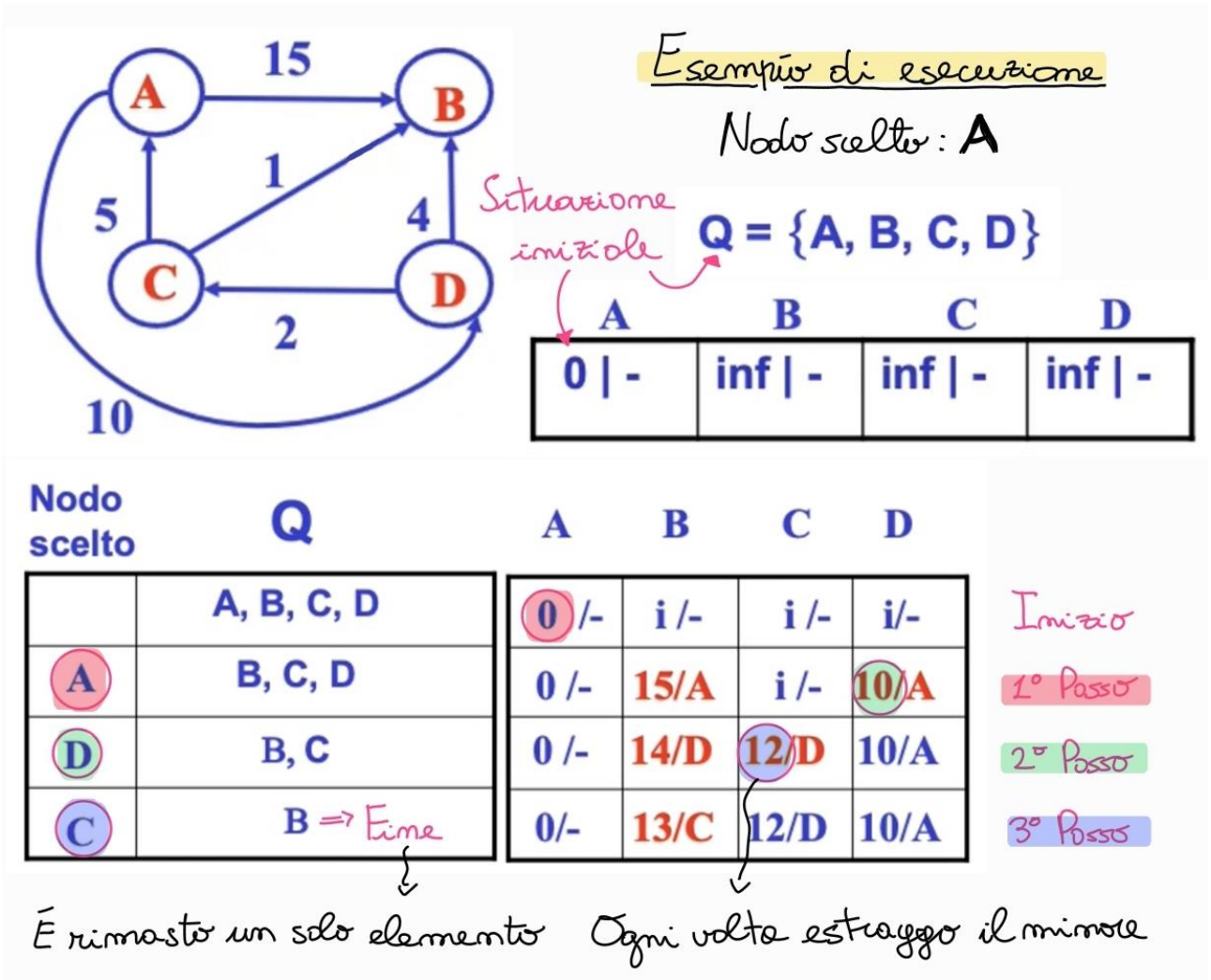
{A, B, C, D} {E, F}

Algoritmo di Djikstra *(greedy)*

Dato un grafo orientato con archi pesati produce il cammino minimo da un nodo di partenza verso tutti gli altri nodi

Crea una lista con tutti i nodi del grafo ed una tabella distanza-predecessore inizialmente vuota
Per ogni nodo della lista, selezionando sempre quello con distanza minore nella tabella
    Se l'arco tra esso ed un suo diretto successore è minore di come riportato in tabella la aggiorno

La complessità dell'algoritmo, implementato tramite un **min-heap**, è $O(N(\log N + A/N \log N))$
Perché ad ogni ciclo (N) estraggo la radice minore (logN) e al peggio la aggiorno (logN) A/N volte



Output finale dell'algoritmo

da A a B: A->D->C ->B lung=13

da A a C: A->D->C lung=12

da A a D: A->D lung=10

<u>Esempio di applicazione della teoria dei grafi</u>: algoritmo di ranking di un pagina web

Supponiamo che la rete sia rappresentata tramite un grafo orientato e supponiamo di voler calcolare la rilevanza di una pagina *P* all'interno della rete, potremmo calcolarla sommando, per ogni link che porta alla pagina *(arco nel grafo che porta a P)*, la rilevanza della pagina da cui parte il link diviso il numero di link che escono da tale pagina: $R(P) = \sum_{Link\ Q \to P} R(Q)/|Q|$.

*<u>Nota</u>: Bisogna fare attenzione nel caso di **nodi pozzo**, ossia di nodi da cui non parte alcun arco*

7. NP Completezza

La teoria della NP completezza si usa per classificare un insieme di problemi difficili di tipo **decisionale**, la cui risposta è si oppure no (*se un problema non è di tipo decisionale posso sempre riformularlo*). Normalmente un problema decisionale ha complessità minore dello stesso problema riforumlato in modo discorsivo, dunque se il problema decisionale risulta difficile a maggior ragione lo sarà il problema discorsivo.

Introduciamo gli **algoritmi non deterministici**, ossia una speciale categoria di algoritmi che ad ogni esecuzione con input uguale possono potenzialmente produrre output diversi, questi algoritmi non sono implementabili nella realtà e hanno la capacità di **fare sempre la scelta migliore** in un tempo costante.

- **P** rappresenta tutti i problemi decisionali **facili**, ossia risolubili in tempo polinomiale con un algoritmo deterministico

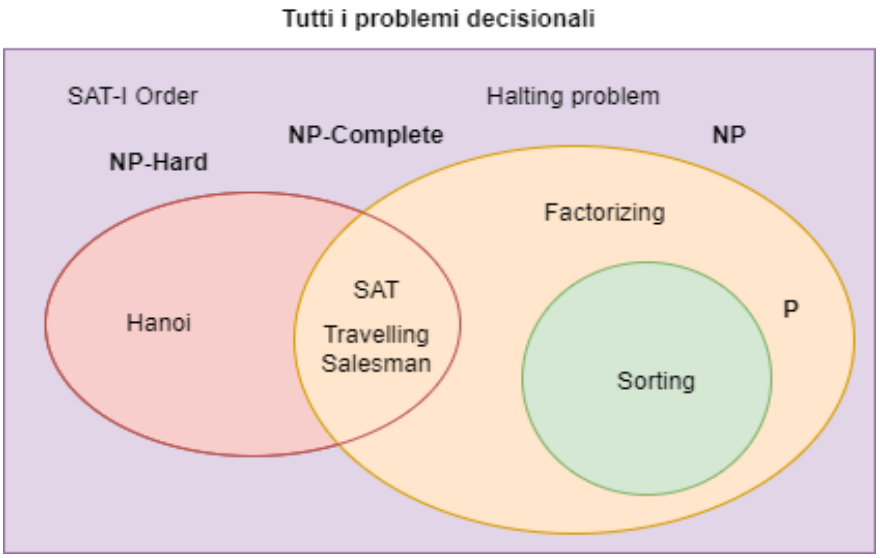
- **NP** rappresenta tutti i problemi decisionali che **potrebbero essere difficili**, ossia per i quali non abbiamo un algoritmo deterministico ottimo che li risolva in tempo polinomiale ma che in futuro potrebbero averlo. Questi sono risolubili in tempo polinomiale solo con un algoritmo non deterministico e comprendono anche tutti i problemi decisionali la cui soluzione può essere verificata in un tempo polinomiale da un algoritmo deterministico.

Un problema si dice **riducibile** *(indicato con $\leq$)* in tempo polinomiale se ogni sua soluzione può essere ottenuta deterministicamente in tempo polinomiale da una soluzione di un altro problema.

Teorema di Cook
Ogni problema NP è riducibile ad un problema SAT,
di conseguenza SAT è il più difficile dei problemi NP

- **NP-Complete** rappresenta i più difficili problemi in *NP,* ossia quei problemi la cui soluzione può essere usata per risolvere un qualsiasi problema *NP*, un problema si dice **NP completo** se è NP e SAT è riducibile al problema, dunque un problema NP completo è difficile tanto quanto SAT e può essere usato per dimostrare la NP completezza di un altro problema.

- **NP-Hard** rappresenta i problemi decisionali per i quali è stato dimostrato che il migliore algoritmo attuale e futuro è almeno di complessità esponenziale, inoltre anche la sola verifica è almeno esponenziale.



Una domanda ancora irrisolta: è P uguale ad NP ? È possibile che essere veloci a riconoscere le soluzioni comporti che è possibile trovare velocemente tali soluzioni ? … Per rispondere dovrei trovare la soluzione in tempo polinomiale di un qualsiasi problema NP completo.

Template

Sono costrutti che permettono di scrivere codice universale, indipendente dal tipo che verrà utilizzato.

-- Funzioni template

```
template <class type, ...> → Possibile anche utilizzare più tipi
type myMax(type arg1, type arg2) {
    return (arg1 > arg2) ? arg1 : arg2;
} → Si suppone che per il tipo type sia stato ridefinito l'operatore >

myMax(1, 1); → Tipo del template dedotto dal tipo dei parametri
myMax(1.0, 1); → ERRORE: Solo match perfetti, no conversioni implicite
myMax<double>(1.0, 1); → OK: Tipo speficato dall'utente
```

-- Template con parametri noti

I parametri noti devono sempre essere **espliciti <...>** e devono sempre essere **espressioni costanti o letterali**.

```
template <int n, class type> → n è un argomento di un tipo noto
void fun(type arg){ cout << arg + n; }

fun<10>(.5); → Tipo del template dedotto dal tipo dei parametri, il tipo corretto che vede il compilatore è fun<10, double>(double)
```

-- Template con membri statici

```
template <class type>
void fun(type arg){
    static int counter;
    counter++;
    cout << counter << ' ' << arg;
}

fun(100); // 1 100
fun(100); // 2 100
fun(101); // 1 101 → Dimostra che ogni funzione con parametri di template diversi è vista in modo completamente diverso dal compilatore
```

Nota: Per questo motivo non posso più separare la definizione e la dichiarazione in due unità di compilazione diverse, deve essere tutto nella stessa, altrimenti il compilatore non sa che cosa fare.

Derivazione semplice

La derivazione o ereditarietà consente di trasmettere un insieme di caratteristiche comuni da una classe **base** ad una **derivata** senza che ciò comporti una duplicazione del codice, offrendo allo stesso tempo la flessibilità necessaria per adattare o estendere il comportamento a casi d'uso specifici, un'oggetto di una classe derivata ha infatti tutti i campi della classe base più i suoi particolari.

-- Modalità di derivazione

```
Class Derivata : [public, protected, private] Base { ... };
```

Public In generale indica l'accessibilità dei campi di una classe al di fuori della stessa, in questo contesto indica che tutti i campi public della base rimangono public nella derivata.

Protected In generale indica l'accessibilità dei campi di una classe alle sue derivate, in questo contesto indica che tutti i campi public della base diventano protected nella derivata.

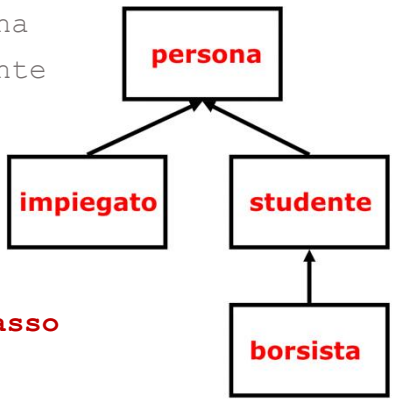
Private In generale indica l'accessibilità dei campi di una classe solamente a se stessa, in questo contesto indica che utti i campi public o protected della base diventano private nella derivata.

-- Compatibilità tra i tipi

Un oggetto o un puntatore ad oggetto di un tipo può essere convertito in un supertipo (classe base) o in un puntatore a supertipo, ma non vale il viceversa. **Nella conversione i campi derivati non scompaiono, vengono solo resi inaccessibili alla classe base.**

 # Esempi corretti (conversioni dal basso verso l'alto)

```
persona = studente;  Conversione studente → persona
persona = borsista;  Conversione borsista → persona
studente = borsista; Conversione borsista → studente
...
Persona *pers = new Studente(...);
Persona *pers = &borsista;
```



 # Esempi sbagliati

```
studente = persona; → Conversione dall'alto al basso
studente = impiegato; → Tipi diversi
...
Studente *stud = &persona; → Conversione dall'alto al basso
```

-- Regole di costruzione e distruzione

Quando un oggetto di classe derivata viene costruito **prima avviene la chiamata al costruttore della classe base** *(prima ancora vengono costrutiti i suoi oggetti)*, **poi a quello della classe derivata** *(prima ancora vengono costrutiti i suoi oggetti)*. Nel caso la classe base abbia più costruttori, il costruttore di una classe derivata **deve richiamarne uno nella lista di inizializzazione,** a meno che la classe base non ne abbia uno di default, la distruzione avviene in ordine inverso a come specificato.

```
Base B;
Derivata D;

Output:
  Nuova base → Classe base di B
  Nuova base → Classe base di D
  Nuova base → Oggetto base in D
  Nuova derivata → Classe derivata D

// quantiBase = 3
// quantiDerivata = 1
```

```
Class Base {
    static int quantiBase;
    Base(){
        cout << "Nuova base"
        ++quantiBase;
    };
};

Class Derivata : public Base {
    Base  ogg;
    static int quantiDerivata;
    Derivata(){
        cout << "Nuova derivata"
        ++quantiDerivata;
    };
};
```

Nota: **distruggere un oggetto di una classe derivata usando un puntatore alla sua classe base provoca un comportamento idefinito**, per ovviare a questo problema è necessario utilizzare un distruttore virtuale, che verrà chiamato automaticamente in base al tipo puntato a tempo di esecuzione.

Polimorfismo

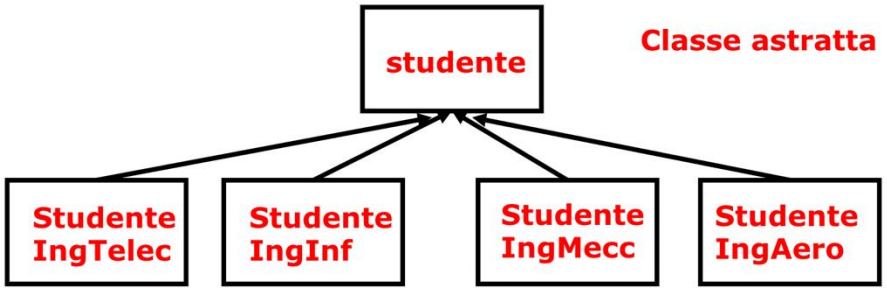
-- Funzioni virtuali

Una funzione virtuale, preceduta dalla keyword **virtual**, è una particolare funzione membro che si prevede venga ridefinita nelle classi derivate, una volta dichiarata virtuale una funzione lo rimane per tutte le classi derivate. La particolarità di queste funzioni è che la scelta di quale delle diverse implementazioni da chiamare avviene dinamicamente a tempo di esecuzione, **in base all'oggetto effettivamente puntato.**

```
Studente *stud = new borsista
stud->chisei() → Stampa studente se il metodo chisei() non è virtuale
stud->chisei() → Stampa borsista se il metodo chisei() è virtuale
```

-- Classi astratte

Una classe astratta è uno scheletro sulla quale le classi derivate si baseranno, una classe è definita astratta se presenta almeno una funzione virtuale pura, ossia posta uguale a 0, non è dunque possibile istanziare oggetti di classe astratta ma **solamente puntatori.**




```
Class Studente { ...
    virtual void stampa()=0; → Funzione virtuale pura (Studente è astratta)
    ... };

Studente *studenti[3]; → Puntatore ad uno studente generico
studenti[0] = new StudenteIngInf(001);
studenti[1] = new StudenteIngMecc(002);
studenti[2] = new StudenteIngAero(003);

studenti[0]->stampa(); → Ingegneria Informatica
studenti[1]->stampa(); → Ingegneria Meccainca
studenti[2]->stampa(); → Ingegneria Aerospaziale
```

Gestione delle eccezioni

Le eccezioni sono errori a runtime (ad esempio una divisione per zero), ossia situazioni anomale non rilevabili dal compilatore, queste possono essere gestite preventivamente dal programmatore traimte vari metodi:

-- Metodo semplice: tipi predefiniti come eccezioni

```
try { // Codice che potrebbe creare eccezioni
    if (y == 0) throw "Divisione per zero"; → Gestisco l'eccezione
    int div = x/y;

    if (x < 0) throw -1; → Non per forza errori del linguaggio, magari
    if (y < 0) throw -2;   voglio che sia sempre positivo il numero div
}
catch (const char* e) { cout << e << endl; } → Catturo l'eccezione
catch (int e) {
    if (e == -1) cout << "Numeratore negativo";
    if (e == -2) cout << "Denominatore negativo";
}
catch (...) {} } → Clausola catch generica, cattura qualsiasi eccezione
```

Se viene lanciata un'eccezione l'esecuzione del blocco try si interrompe e il programma va alla ricerca di un catch che corrisponda al tipo dell'eccezione da gestire, ignorando il codice in mezzo, per poi riprendere ad eseguire il codice dopo il catch, oppure terminare con errore nel caso non venisse trovato un catch opportuno. I catch vengono esaminati nell'ordine in cui compaiono, possono anche trovarsi in blocchi diversi e devono avere un **match perfetto** con il tipo dell'eccezione da gestire (nessuna conversione implicita: ad esempio catch int ignora un throw double).

-- Metodo medio: classi come eccezioni

Vediamo subito che questo metodo di gestire le eccezioni è molto primitivo, sarebbe meglio avere eccezioni più complete e dettagliate, per questo è possibile lanciare una classe come eccezione:

```
class Errore {
int codice;
public:
    Error(int n) : codice(n) {};
    void dettagli() {...};
};
```

```
try {
    ...
    throw Error(45000);
}
catch (Error &e) { e.dettagli(); }
```

-- Metodo avanzato: classi astratte come eccezioni

Immaginiamo di dover gestire gli errori di una struttura dati a pila (stack), un modo avanzato per farlo è:

```
class StackExc { → Classe astratta di base
protected:
    const char *prefix = "Errore nello stack: ";
    virtual void details() = 0;
};

class StackFull : StackExc { → Eccezione derivata per lo stack pieno
    const char *error = "Stack pieno";
public:
    void details() override { cout << prefix << error; };
};

class StackEmpty : StackExc { → Eccezione derivata per lo stack vuoto
    const char *error = "Stack vuoto";
public:
    void details() override { cout << prefix << error; };
};


int main() {
    try {
        ... // Codice
        throw new StackEmpty();
    }

    catch (StackEmpty *e) {
        e->details();
        delete e;
    }
}
```