DEF: ALGORITMO RISOLUTORE:

DEF DI PROBLEMA:

Dato un problema P, e un insieme X di possibili ~~ott~~ input di P e un insieme Y di possibili output P, P è una funzione che associa ad ogni elemento $x \in X$ un elemento $y \in Y \rightarrow y = P(x)$

ALGO RISOLUTORE:

Dato un algoritmo A ed un problema P, con $P: X \rightarrow Y$.
Si dice che A è un algoritmo risolutore di P se per ogni $x \in X$, l'esecuzione di $A(x)$ restituisce un elemento $y \in Y$ tc $y = P(x)$ dopo un numero finito di passi.

DEF MIGLIORE:

insieme dei possibili input

insieme dei possibili output.

Dato un problema $P: I \rightarrow O$, un algoritmo A è risolutore di P se
$\forall i \in I \quad A(i) = P(i)$

CRITERI DI COSTO:

COSTO UNIFORME: considero ciascuna operazione come un singolo passo, senza considerare la dimensione dell'input.

COSTO LOGARITMICO: in questo modello per ciascuna operazione si paga un costo di $\log n$ dove n è la dimensione dell'input.

DEF DI ALGORITMO: Un algoritmo è un procedimento che risolve un determinato problema attraverso un numero finito di passi, chiari e non ambigui.

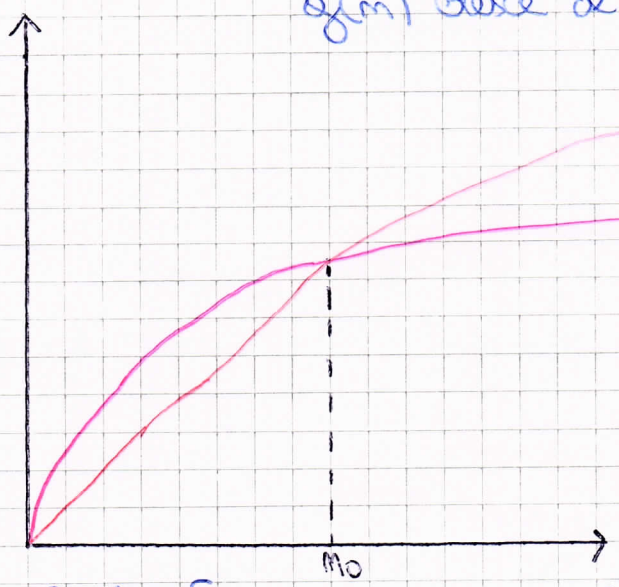
DEF ALGO OTTIMALE: Quando per un problema P, il costo dell'algoritmo $O(n)$ e $\Omega(n)$, UPPER BOUND e LOWER BOUND coincidono.

NOTAZIONE ASINTOTICA:

Data una funzione $f(n)$ definiamo:

- $O(f(n)) = \{g(n) \text{ tc } \exists c>0 \text{ e } n_0 \geq 0 \text{ tc } g(n) \leq c f(n), \forall n \geq n_0\}$

  $g(n)$ cresce al più come $f(n)$

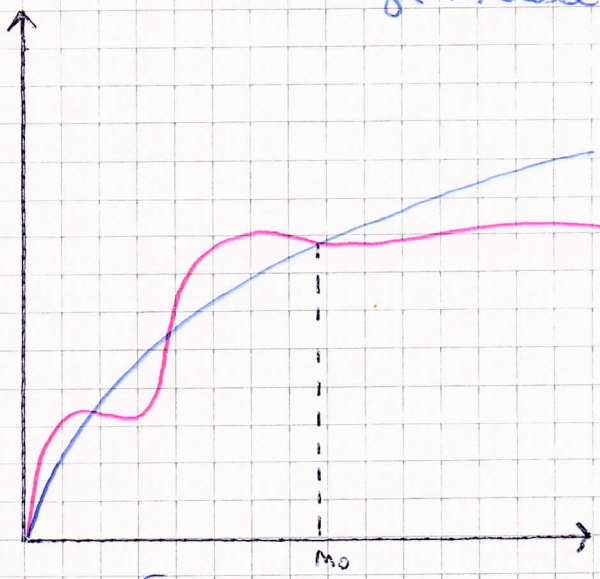


$O(n)$ è una delimitazione superiore

UPPER BOUND.

- $\Omega(f(n)) = \{g(n) : \exists c>0 \text{ e } n_0 \geq 0 \text{ tc } g(n) \geq c f(n), \forall n \geq n_0\}$

  $g(n)$ cresce almeno come $f(n)$

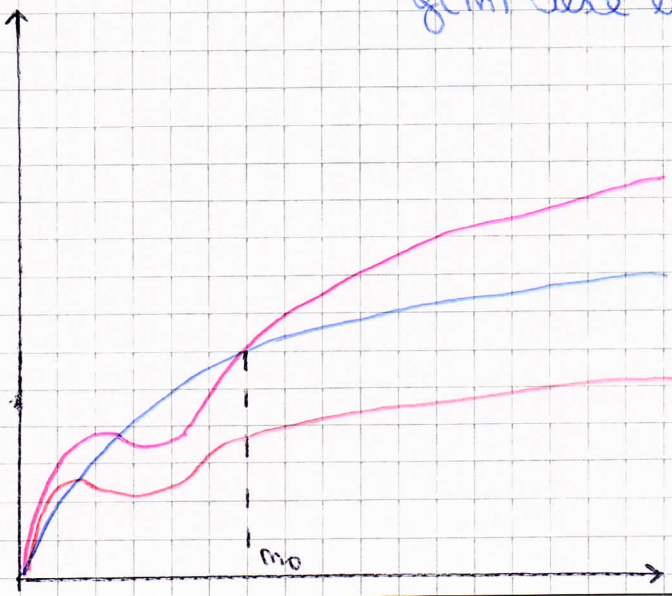


$\Omega(n)$ è una delimitazione inferiore

LOWER BOUND

- $\Theta(f(n)) = \{g(n) : \exists c_1, c_2 > 0 \text{ e } n_0 \geq 0 \text{ tc } c_1 f(n) \leq g(n) \leq c_2 f(n), \forall n \geq n_0\}$

  $g(n)$ cresce esattamente come $f(n)$



Caso peggiore, caso migliore e caso medio.

Il concetto è quello di analizzare il tempo di esecuzione di un algoritmo, fra tre diverse categorie d'istanze, a parità di dimensione.

Dato un istanza I di dimensione m, indichiamo con tempo(I) il tempo di esecuzione su quella istanza da parte dell'algoritmo A.

CASO PEGGIORE:

$$T^A_{WORST}(m) = \max\left(\{T^A(i) \mid i \in I \wedge |i| = m\}\right)$$

CASO MIGLIORE:

$$T^A_{BEST}(m) = \min\left(\{T^A(i) \mid i \in I \wedge |i| = m\}\right)$$

CASO MEDIO:

$$T^A_{AVG}(m) = \underset{\substack{\downarrow \\ \text{valore atteso}}}{E}\{T^A(i) \mid i \in I \text{ tle} \wedge |i| = m\}$$

Analisi negli algoritmi iterativi:

Determiniamo l'operazione ripetuta più volte (ISTRUZIONE DOMINANTE)

DEF: COMPLESSITÀ INTRINSECA Dato un problema P e una funzione g, si dice che P ha complessità intrinseca $\Omega(g)$ se tutti gli algoritmi risolutori di P hanno complessità $\Omega(g)$.

In altre parole, in qualunque modo risolvo il problema, meno di $\Omega(g)$ non posso andare.

TECNICA DELL'AVVERSARIO: Questa tecnica consiste nel supporre che un dato algoritmo, che risolve un dato problema ad un certo costo, possa essere "battuto" cioè dare in input dati d'ingresso fatti appositamente per mettere in crisi l'algoritmo.

ANALISI AMMORTIZZATA:

L'analisi ammortizzata studia le prestazioni medie di una sequenza di operazioni su una collezione dati, piuttosto che su una singola esecuzione.

Per caratterizzare ~~le medie~~ le prestazioni in media di un algoritmo parleremo di costo ammortizzato, che è definito

$$T_a(n) = \frac{T(n,k)}{k}$$

← tempo totale richiesto dall'algoritmo nel caso peggiore per tutte le k operazioni su istanze di dimensione n.

numero di operazioni fatte da una sequenza di un algoritmo.

DIVIDE ET IMPERA

~~La tecnica~~ Il principio su cui si basa la tecnica divide-et-impera consiste nel dividere i dati in ingresso in due o più sotto insiemi (DIVIDE), risolvere ricorsivamente il problema sui sottoinsiemi e poi ricombinare le soluzioni dei sotto ~~insiemi~~-problemi e ottenere la soluzione globale del ~~proprio~~ problema originario. (Impera)

TECNICA GOLOSA:

Questa tecnica viene utilizzata per risolvere problemi di ottimizzazione, ovvero problemi per cui dobbiamo trovare la migliore soluzione possibile.

Nella situazione generale avremo a che fare

- Un insieme C di candidati possibili (Insieme di possibili algo risolutori)
- Insieme dei candidati che sono stati ~~scartati~~ esaminati, tra cui distinguiamo quali sono stati scartati e quelli che sono stati selezionati per far parte di una soluzione parziale S.
- Una funzione ammissibile che verifica se un insieme di candidati fornisce una soluzione (anche non ottima) al problema.
- Una funzione ottimo che verifica ~~che~~ se C fornisce una soluzione ottima.
- Una funzione ~~obiettivo~~ selezione per estrarre un elemento dall'insieme dei candidati possibili non ancora esaminati.
- Una funzione OBIETTIVO che fornisce il valore di una soluzione.

Per risolvere il problema, occorre trovare un insieme di candidati che:

- è una soluzione
- Ottimizza il valore della funzione obiettivo.

ALGORITMI DI ORDINAMENTO CON DIVIDE-ET-IMPERA

```
void MergeSort (int [] V, int inf, int sup) {
        if(sup - inf) <= 0) return;
        int med = (inf + sup)/2;
        mergeSort(V, inf, med);
        mergeSort(V, med+1, sup);
Θ(n)    merge(V, inf, med, sup);
```

COMPLESSITÁ SPAZIALE

Ad ogni chiamata alloco ~~una~~ memoria pari allo spazio di allocazione di $\frac{n}{2}$ elementi.

$$S(n) = S\left(\frac{n}{2}\right) + \overset{\text{merge}}{b}$$

$$\vdots$$

$$S(1) = b$$

Quindi la complessità sarà $\Theta(\log n + n)$

COMPLESSITÁ TEMPORALE: $\Theta(n \log n)$ in tutti i casi.

Quick sort

```
void quicksort (int [] V, int inf, int sup) {
        if(sup <= 0) return;
        int med = ~~(sup inf + sup~~ partizione(V, inf, sup);
        quicksort(V, inf, med-1);
        quicksort(V, med, sup);
```

quando prendo come pivot il min o il max oppure se il vet è ordinato in modo decrescente

COMPLESSITÀ:

	CASO MIGLIORE	CASO PEGGIORE	CASO MEDIO
TEMPORALE	$\Theta(n \log n)$	$\Theta(n^2)$	$O(n \log n)$
SPAZIALE		$O(n)$	

Ricerca binaria

~~SPAZIALE~~: TEMPORALE:

Caso migliore $\Theta(1)$ Caso Peggiore $\Theta(\log n)$

SPAZIALE

CASO MIGLIORE CASO MIGLIORE

$\Theta(1)$ $\Theta(\log n)$

ALBERI:

- Gli alberi sono utili per i fini di ricerca, in quanto hanno complessità logaritmica
- Posso essere alberi ~~con n~~ i cui nodi possono essere n illimitati figli ~~nodi~~.

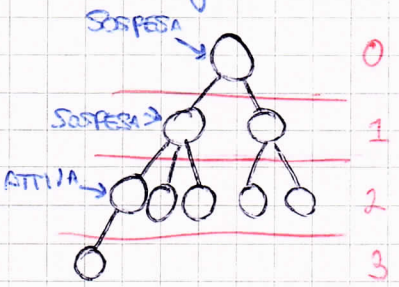
VISITA ANTICIPATA: / VISITA POSTICIPATA

TEMPORALE: $O(n)$ perché faccio al più una chiamata per ogni nodo.

$\Omega(n)$ perché faccio almeno una chiamata per ogni nodo.

per cui $\Theta(n)$

SPAZIALE è uguale al numero di chiamate sospese + quella attiva.

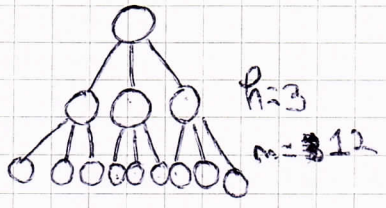


Da ciò segue che la complessità è pari all'altezza. L'altezza coincide con il maggior numero di chiamate ricorsive.

MOLTO IMPORTANTE:

Alberi con lo stesso numero di nodi, possono essere altezze diverse.

CASO BEST

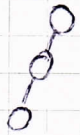


CSM($\Theta(\log n)$)

CSP($\Theta(n)$)

Albero Completo

CASO WORST



h=3
n=3

In un albero degenere il numero di nodi è uguale all'altezza.

Albero degenere.

VISITA PER LIVELLI:

SPAZIALE:

Dipende dall'altezza (spazio) nel caso peggiore, è lineare al numero di nodi, nel caso ~~peggiore~~ migliore è logaritmica nel n° di nodi.

$$CSP(\Theta(n)) \quad CSM(\Theta(\log n))$$

TEMPORALE:

Sarà uguale al numero di nodi, non c'è complente migliore o peggiore

$\Theta(n)$.

ALTEZZA:

COMPLESSITÀ TEMPORALE	BEST	WORST
RICERCA	$\Theta(1)$	$\Theta(h)$
ADD	$\Theta(1)$	$\Theta(h)$
REMOVE	$\Theta(1)$	$\Theta(h)$

```
int altezza (AB a) {
    if(a == null) return 0;
    else
    return 1+ Max.(altezza (a.sin), altezza(a.des));
```

$\Theta_T(n)$

CSM $\Theta(\log n)$ ← poiché conto le chiamate attive.

CSP $\Theta(n)$ ← conto il numero di nodi

ADD

Faccio la ricerca del padre in cui andrebbe inserito, se è già presente devo restituire il nodo in cui è già presente.

COMPLESSITÀ = COSTO RICERCA + INSERIMENTO (COSTANTE)

RIMOZIONE: h diminuirà di 1.

COMPLESSITÀ: COSTO RICERCA + ELIMINAZIONE (SE ESISTE)

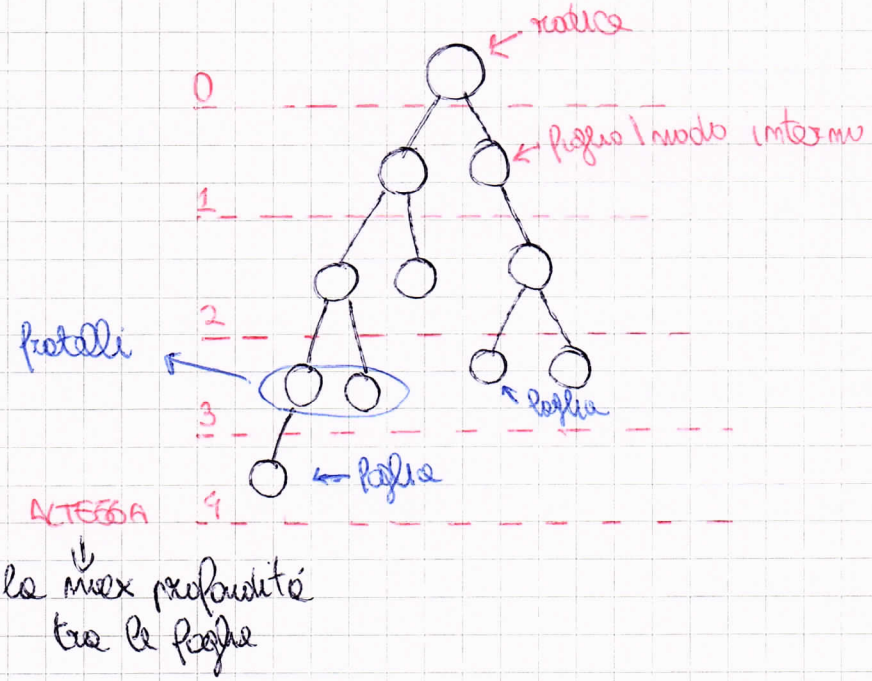
COMPLESSITÀ SPAZIALE: COSTANTE, SE CREO QUESTI METODI IN MODO ITERATIVO NON ALLOCO SPAZIO.

TEO SUGLI ALBERI

- In un ~~un~~ albero ogni nodo ha un solo padre, esclusa la radice. Ogni nodo può avere più figli, il cui numero è detto grado del nodo.
- Un nodo senza figli viene chiamato foglia.
- I nodi che non sono ne radice e ne foglia vengono detti nodi interni

La profondità è definita così:

- La radice ha profondità 0.
- Se un nodo ha profondità k, i figli avranno profondità k+1.

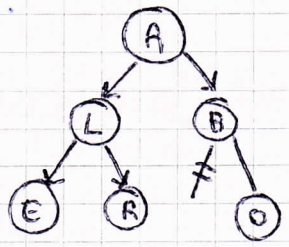


Questo è un albero binario in cui ogni nodo può avere al max 2 figli.
Non è completo poiché alcuni nodi hanno un solo figlio.

ABR:

Un ABR è un albero che soddisfa le seguenti proprietà:

- Il figlio ~~dx~~ Sx è minore del padre
- Il figlio Dx è maggiore del padre.



VISITE $\Theta(n)$ tutte.

PRE ORDINE ~~PROFONDITÀ~~ Si visita prima la radice, poi ricorsivamente il figlio Sx e dx ALBERO

SIMMETRICA Prima nel figlio Sx, poi radice, poi destro. ELRABO

POST-ORDINE Prima il figlio sinistro, poi destro e poi radice. ERLOBA

ALBERO AVL

Nel caso peggiore l'altezza di un ABR può essere proporzionale al numero di nodi, mentre se fosse bilanciato, avrebbe altezza logaritmica, migliorando l'efficienza. Per questo vengono usati gli alberi AVL.

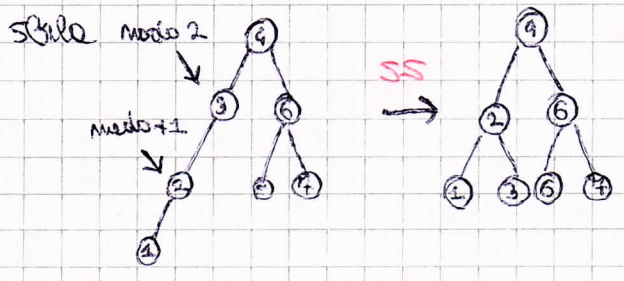
ALBERO BILANCIATO: Un albero è bilanciato se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di un'unità.

FATTORE DI BILANCIAMENTO: Il fattore di bilanciamento di un nodo v è la differenza tra l'altezza del sotto albero sinistro e quella del sottoalbero destro.

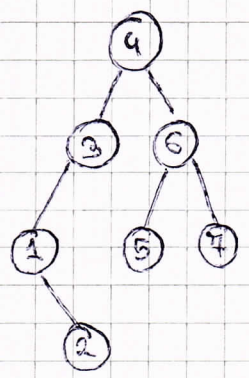
Per mantenere il bilanciamento di un albero AVL si operano delle rotazioni a seguito di operazioni di inserimento e rimozione.

Le operazioni di rotazione vengono effettuate sui nodi i cui fattori di bilanciamento, in valore assoluto è ≥ 2.

SINISTRA SINISTRA Quando un nodo a coeff. di bil +2 e il figlio sinistro un coefficiente pari a 0 o +1.



DESTRA SINISTRA:
Quando un nodo ha coefficiente -2 e il figlio sx coeff pari a -1 o 0.



PILA ESTENDIBILE:

La pila è una struttura dati realizzabile sia con strutture indicizzate, sia collegate.

boolean isEmpty();
void push(elem e); inserisce in cima
elem pop(); restituisce ed elimina in cima
elem getTop(); restituisce l'in cima.

Inserimento:

CASO 1: (ARRAY)
c'è spazio, quindi è $\Theta(1)$

CASO 2: (ARRAY)
Non c'è spazio, quindi devo ridimensionare, ovvero copiare gli elementi in un array nuovo. $\Theta(n)$

PILA E CODE:

Una pila è una struttura dati che segue una particolare politica di gestione:

LIFO LAST IN ~~LAST O~~ FIRST OUT.

- Gli inserimenti vengono fatti in cima (PUSH) $\Theta(1)$
- La rimozione è ~~a~~ è l'elemento in cima (POP) $\Theta(1)$

Una coda è una struttura dati che segue la politica FIFO First in First out.

Gli inserimenti avvengono in coda (enqueue) $\Theta(1)$

La rimozione avviene in testa (dequeue) $\Theta(1)$

Entrambe possono essere realizzate usando una struttura indicizzata che ~~collegata~~ collegate.

TABELLE HASH.

Sono strutture dati realizzate mediante vettori che permettono nel caso medio, di eseguire operazioni in tempo costante.

- Def:

FATTORE DI CARICO:

Il fattore di carico di un tabella hash è definito come il rapporto $\alpha = \frac{n}{m}$

elementi memorizzati

dim struttura dati.

FUNZIONE HASH:

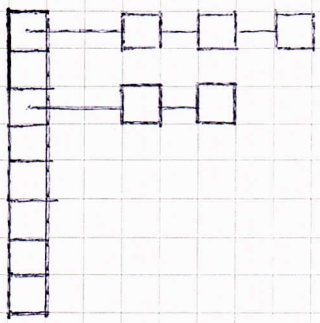
Una funzione hash è una funzione $h: U \rightarrow \{0, \dots, m-1\}$ che trasforma chiavi in indici di una tabella.

universo delle chiavi associabili

COROLLARIO:

Una funzione hash è perfetta se è iniettiva. ($\forall$ chiave è associato 1 solo indice)

Se una funzione hash non è perfetta potrebbe capitare che si verifichi una collisione, ossia si possono avere più chiavi diverse con valore associato iniziale. Per risolvere tale problema occorre adoperare strategie di risoluzioni quali la lista di overflow./array di BUCKET.



COUNTING SORT / BUCKET SORT

```
void countingSort(int [] a){
    int max = a[0];
    int min = a[0];
    int i=1;
    for(; i<a.length; i++){
        if(a[i] >max)
            max=a[i];
        else if(a[i] < min)
            min = a[i];
    }
    int [] a2 = new Int [max-min+1]; // array delle frequenze
    for(i=0; i<a2.length; i++)
        a2[i]=0;
    for(i=0; i<a.length; i++)
        a2[a[i]-min]++; // numero di volte che si è incontrato il valore.
    // ordinamento in base al contenuto dell'array C
    int k=0  // indice di a
    for(i=0; i<a2.length; i++){
        while(a2[i]>0){
            a[k]=i+min;
            k++;
            a2[i]--;
        }
    }
}
```

Calcolo il min e il max

inutile

elementi della hash table

$\Theta(n+k)$

↑ chiavi

HEAP

Un Heap è un albero completo fino al penultimo livello dove l'ultimo livello è riempito da sinistra verso destra.

Ha le seguenti proprietà:

- Nel caso di min-heap il padre ha associato una chiave minore o uguale a quelle dei propri figli.

- Nel caso di max-heap il padre ha associato una chiave maggiore o uguale a quelle dei propri figli.

Un Heap viene ~~realizzato~~ rappresentato utilizzando un vettore.

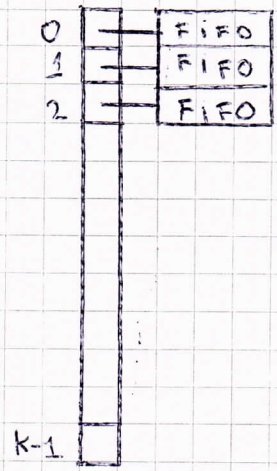
Per un generico nodo in posizione i si ha:

- Padre in $[i/2]$;
- Figlio sx $[i \cdot 2]$;
- Figlio Dx $[i \cdot 2 + 1]$;

Un heap con n nodi ha altezza $O(\log_2 n)$

CODE CON PRIORITÀ:

Una coda con priorità è un tipo di dato che permette di mantenere il minimo (o il massimo) in un insieme di chiavi su cui è definita una relazione d'ordine totale.



È un array ~~pari~~ di dimensione pari alle priorità.

RADIX SORT:

algoritmo di ordinamento di interi, Ordina i numeri per cifre ~~a posti~~ a partire da quella meno significativa.

Effettua un numero di passaggi pari alla quantità di cifre del numero.

Complessità $O(n \cdot n^{\circ}$ cifre elemento più grande$)$

RAPPRESENTAZIONE DEGLI INSIEMI TRAMITE STRUTTURE DATI:

insiemi	AVL	HASH BUONO	HASH CATTIVO	BIT SET
ins	$\lg n$	1		1
delete	$\lg n$	1		1
contains	$\lg n$	1		1
union	$n+m$	$n \times m$	$n+m$	n
intersect	$n+m$	n	$n \cdot m$	n
subtraction	$n+m$	n	$n \cdot m$	n

Tutto in Θ

n = n° di elementi in A

m = n° di elementi in B

AVL: Per le operazioni di union uso 2 iteratori sui due alberi, scorro gli elementi in ordine e uso una funzione tipo merge.

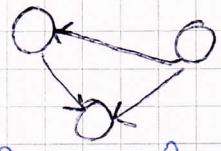
Sfrutto la funzione anche per intersect e subtraction

GRAFI:

Un grafo $G = (V, E)$ consiste di:

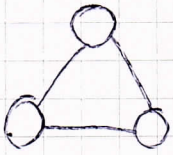
- un insieme V di vertici
- un insieme E di coppie di vertici, detti archi.

GRAFO ORIENTATO



Grafi in cui le relazione sono orientate.

GRAFO NON ORIENTATO



Grafo le cui relazioni non sono orientate.

GRADO DI ENTRATA: Il numero di archi ~~uscenti~~ entranti di un nodo

GRADO DI USCITA: Il numero di archi uscenti da un nodo.

CICLO: Percorso che parte da un nodo in cui da un certo punto del cammino arrivo di nuovo al nodo di partenza.

I grafi orientati che non contengono cicli sono detti DAG (Directed Acyclic Graph)

GRAFO NON ORIENTATO CONNESSO:

Se comunque presa una coppia di nodi i-j c'è un percorso che mi porta da i a j e vice versa.

GRAFO ORIENTATO FORTEMENTE CONNESSO:

Comunque presa una coppia di nodi $<i, j>$ ha un cammino da i verso j e vice versa.

Quindi se esiste un cammino da ogni vertice verso un altro vertice.

DEF PER GRAFI ORIENTATI:

① Grafi debolmente connesso: (PIÙ USATA)

Prendo un grafo orientato, tolgo le orientazioni, se questo è connesso l'originario è connesso debolmente

② (più forte)

Se comunque prendo una coppia di nodi $\langle x, y \rangle$ è debolmente connesso se il cammino da $x \to y$ oppure da $y \to x$.

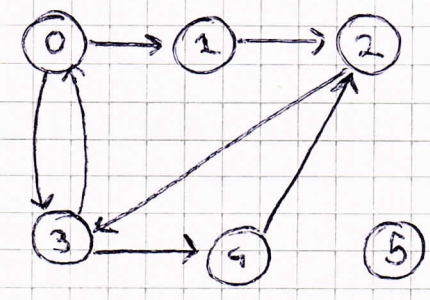
Quando un grafo è un albero?

Quando non è orientato ed è aciclico e connesso.

- Un albero libero è un grafo connesso con $m = n - 1$ archi
- Un albero radicato è un albero libero in cui è stata identificata la radice.

MATRICE DI ADIACENZA: (ORIENTATATO)

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$



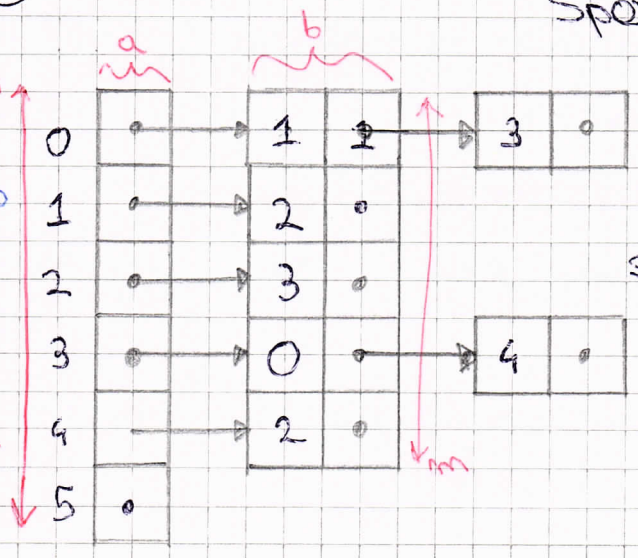
da

	0	1	2 (a)	3	4	5
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

Spazio $= m^2$ bit

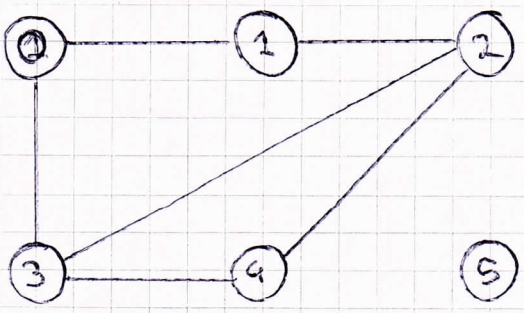
LISTE DI ADIACENZA

Se devo memorizzare una rete sociale, quindi molto grande, non conviene utilizzare una matrice di adiacenza, in quanto lo spazio è m^2 e inoltre è una matrice sparsa.



Spazio $= a \cdot n + b m$

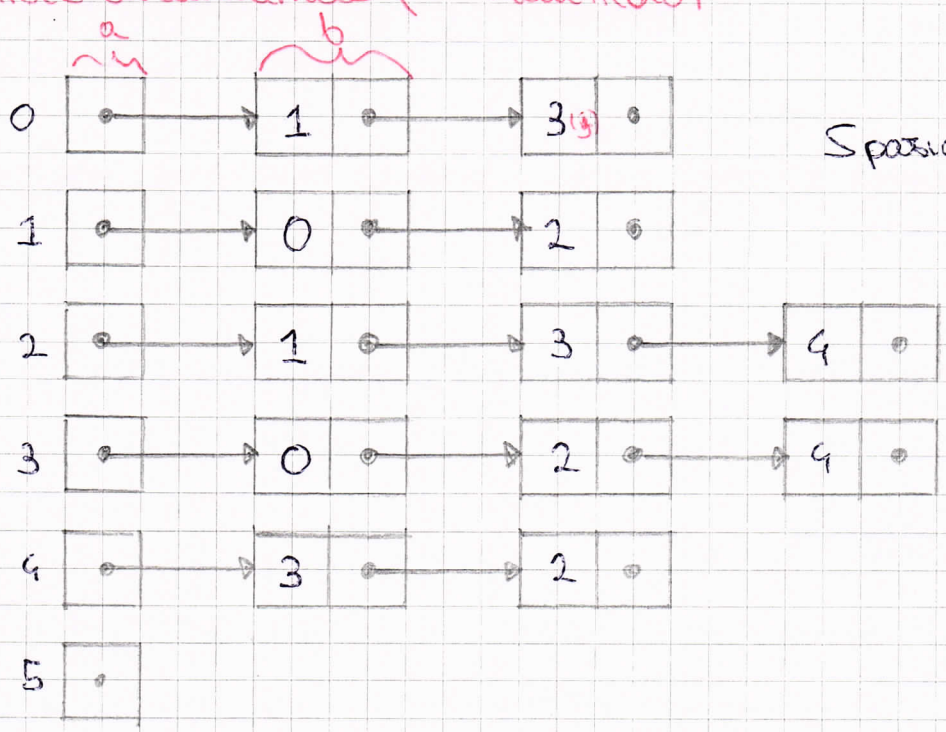
Matrice di Adiacenza (non orientato)



	0	1	2	3	4	5
0		1	0	1	0	0
1			1	1	0	0
2				1	1	0
3					1	0
4						0
5						

Spazio $= m^2 \vee m(m-1)/2$ bit

Lista di adiacenza (non orientato)



Spazio $= am + 2 \cdot bm$

GRAFI PESATI

MATRICE DI ADIACENZA: Al posto di 1, memorizzo il valore dell'arco.

LISTA DI ADIACENZA: Devo memorizzarlo nella struttura

Ricapitolando:

Matrice di Adiacenza:

- Spazio richiesto $O(n^2)$
- Verificare se u è adiacente a v tempo $O(1)$
- Iterare su tutti gli archi tempo $O(n^2)$
- Ideale per grafi densi.

Lista di adiacenza:

- Spazio richiesto $O(\overset{\text{nodi}}{n}+\overset{\text{archi}}{m})$
- Verificare se u è adiacente a v tempo $O(n)$
- Iterare su tutti gli archi richiede tempo $O(n+m)$
- Ideale per grafi sparsi

Come ottengo il numero di ~~nodi~~ archi? = (GRADO DI USCITA DEI NODI) SOMMA

VISITA IN AMPIEZZA: Visita dei nodi per livelli, prima si visita la radice, poi i nodi a distanza 1 dalla radice, poi a distanza due, ecc.

VISITA IN PROFONDITÀ Visita ricorsiva, per ogni nodo adiacente, si visita ricorsivamente tale nodo, visitando ricorsivamente i nodi adiacenti.
Possibili applicazioni: Ordinamento topologico, ciclicità, componenti connesse e componenti fortemente connesse.

VISITA IN AMPIEZZA

OBIETTIVI:

- Visitare i nodi a distanza crescente dalla sorgente (nodo di partenza)
- Calcolare il cammino più breve da r a tutti gli altri nodi
  numero di archi attraversati.
- Generare un albero contenente tutti i nodi raggiungibili da r, tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u.

PSEUDO CODICE:

```
void visitaInAmpiezza(Grafo G, Nodo r){
Coda C = new Coda();
C.enqueue(r); //inserisco r nella coda
boolean[] visitato = new Boolean[G.size];
```

inizializzo tutto il vettore a falso.

Tramite questo meccanismo marco quei nodi che ho visitato.

```
visitato[r] = true;
while(!C.isEmpty()){
    Nodo u = C.dequeue();
    {visito il nodo e faccio qualcosa}
    foreach v ∈ G.archi(u){ //nodi adiacenti a u.
        {visito l'arco (u, v)}
        if(!visitato[v]){
            visitato[v] = true;
            C.enqueue(v);
        }
```

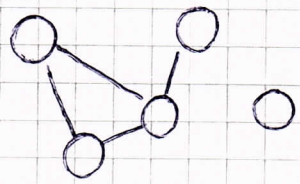
Complessità $\Theta(n + m)$

COMPONENTI CONNESSE E FORTEMENTE CONNESSE:

DEF: Un nodo v è raggiungibile da un nodo u se esiste un cammino da u a v.

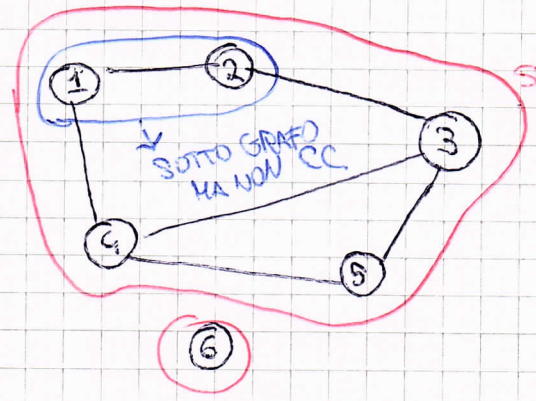
GRAFO CONNESSO E COMPONENTI CONNESSE

- Un grafo (non orientato) è connesso $\iff$ ogni suo nodo è raggiungibile da ogni suo altro nodo.



NON CONNESSO

- Un grafo $G' = (V', E')$ è una componente connessa di $G \iff G'$ è un sottografo connesso e massimale di G.



CICLO:

In un grafo orientato, un ciclo C di lunghezza $k \geq 2$ è una sequenza di nodi $u_0, u_1, \dots, u_k$ con ~~$0 \leq k$~~ tale che $u_0 = u_k$

Inoltre è detto semplice se tutti i suoi nodi sono distinti, ad esclusione del primo e dell'ultimo.

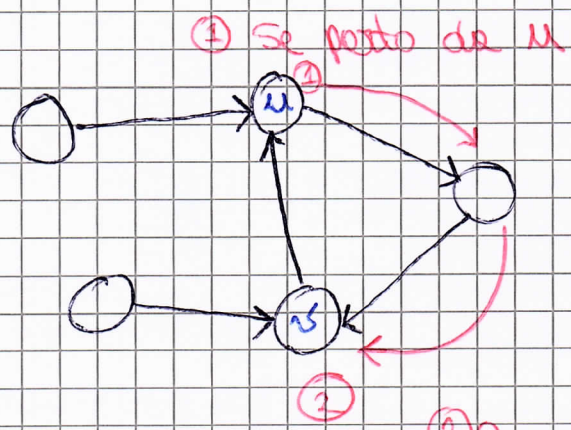
TEOREMA:

Un grafo orientato è aciclico $\Leftrightarrow$ (se e solo se) non esistono archi all'indietro nell'~~arco~~ grafo.

DIMOSTRAZIONE:

- Se: Se $\exists$ un ciclo, sia u il primo nodo del ciclo che viene visitato e sia ~~[illegible]~~ (v, u) un arco del ciclo. Il cammino che connette u e v sarà prima o poi visitato, e da v sarà scoperto l'arco all'indietro (v, u).
- Solo se: Se esiste un arco all'indietro (u, v), dove v è un ~~arco~~ antenato di u, allora esiste un cammino da v a u e un arco da u a v, ovvero un ciclo.

Ho un grafo così

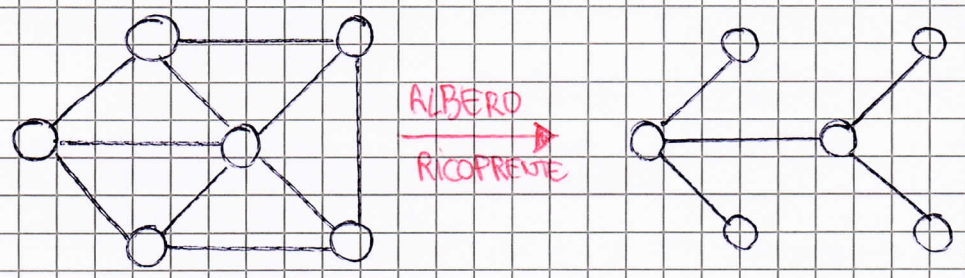


② Quando arrivo a v, se v è un antenato di u, allora c'è ciclo.

KRUSKAL:

L'algoritmo di Kruskal calcola un minimo albero ricoprente.

ALBERO RICOPRENTE: È un albero che contiene tutti i vertici del grafo e contiene un sotto insieme degli archi, cioè solo quelli necessari per connettere i vertici.



ALBERO RICOPRENTE MINIMO: È un albero ricoprente, che sommando nel quale sommando i pesi degli archi, si ottiene un valore minimo.

```
                 perché restituisce
Grafo        Kruskal (Grafo G) {
Archi Pesati [] archi = una lista di archi ordinati in modo ~~pesato~~ ordinato
int inseriti = 0;
Grafo ab = new Grafo (numero di nodi di G);
i = 0;
while ( i < ~~numero~~ archi.length AND inseriti < numero nodi G - 1) {
        lista l = visita in profondità (~~archi[i].~~ nodo iniziale.archi[i] )
        if (! ab.contiene (nodo.finale di archi[i])) {
                ab.add (archi[i]);
                inseriti++;
        }
        i++;
}
}
```

Ordinamento ha costo $\Theta(m \log m)$

Se il grafo è in lista $\Theta(m * n)$

MA $\Theta(m * m^2)$

DIJKSTRA

Trova il cammino minimo tra nodo u e v.

- Idea di base, non conoscendo il peso degli archi, assumiamo il loro costo infinito, tranne u, che varrà 0.
- Inserisco gli archi in una coda di priorità.
- I nodi saranno ordinati nella coda in base alla distanza minore

(A) Sono nel nodo di partenza. / nodo in cima

- Controllo i nodi adiacenti ed aggiorno il loro corrispettivo peso in nella coda di priorità. (di conseguenza si riordina.
- Dobbiamo anche indicare il predecessore di tale nodo.

- Metto s in una coda di scarto

(3) Prendo il primo nodo che è in cima alla coda di priorità

- Controllo gli adiacenti, non considerando quelli scartati / già visitati
- Tra nodo il nodo adiacente e aggiungo come costo nella coda

costo nodo attuale + adiacente

le modifiche non aggiorno

- indico anche i nuovi precedenti, colonnini a fianco aggiornamenti.
- Sposto nodo attuale nella ~~tb~~ coda scartati

Vai a (3) e ripeti

```
Grafo ~~Dist~~ Dijkstra(Grafo G, Nodo iniziale){
    for(∀ vertice di G) ~~valore~~ distanza = +∞;
    Coda prioritaria S; iniziale.distanza = 0
    ~~Distanza iniziale = 0~~;
    S.insert(iniziale, iniziale.distanza);
    while(!S.isEmpty){
        u = S.~~deq~~ estrai;
        for(∀ v adiacente a u){
            if(v.distanza = +∞){
                ~~v.distanza = dist(u,v)~~;
                v.padre = u;
                S.inserisci(v, v.distanza + dist(u,v));
            }
            else if(v.distanza ~~+dist~~ > dist(u,v) + v.distanza){
                ~~S.inserisci(v, v.distanza - dist(u,v))~~
                S.inserisci(v, v.distanza - (v.distanza + dist(u,v)));
                v.padre = u;
            }
        }
    }
    return vero;
```

ALGORITMO DI PRIM

Generalmente è un algoritmo di visita, che partendo da un nodo u, scelto arbitrariamente, raggiunge tutti nodi del grafo.

Analogamente agli algoritmi di visita, ad ogni iterazione, partendo da un nodo v, visita un nodo w (scelto secondo opportuni criteri) e pone l'arco (u, v) nell'insieme T, che al termine dell'esecuzione conterrà una soluzione ottima.

La differenza tra un algoritmo Dijkstra e che la scelta del prossimo nodo avviene mediante un concetto di priorità tra i nodi e che l'insieme Q dei nodi da visitare è gestito come una coda di priorità.

Sia R l'insieme dei nodi visitati, la priorità di un nodo $v \in Q$ è pari al minimo peso di arco che collega v in un nodo in R.

Durante un'iterazione l'algoritmo estrae da Q, il nodo u con priorità minima ed inserisce in T l'arco (v, u) che realizza il peso minimo tra tutti nodi che collegano u con il nodo in R.

Alla fine dell'iterazione si aggiornano le priorità dei nodi collegati con u.

~~Scegli $u \in S$ (Scegli $u \in S$ nodo $\neq s$)~~
~~for(nodi u adiacenti a u)~~
~~se arco da nodo precedente a $u = +\infty$~~
~~$N_{pr} = 0$ nodo aggiornato~~
~~S. inserisci (u)~~
~~while (S. isEmpty)~~
~~u = primo elemento in coda;~~
~~for(nodi adiacenti a u)~~
~~S. inserisci (v, valore(u, v))~~
~~finché il nodo in coda è s.~~
~~fine~~

M	V

```
prim (Grafo G, Nodo x) {
  // W(x,y) dare la distanza da x a y.
  CodeDiPriorità code;   Π vettore dei predecessori;
  for (∀ v ∈ G) {
        code.inserisci(v, ∞);
        Π[v] = null;
  }
  code.set(x, 0);   // setto la distanza 0.
  Set Q;  // insieme dei nodi da visitare;
  for (∀ v ∈ G t.c. v ≠ x)
        Q.add(v);
  while (Q ≠ 0)
        Nodo u = code.dequeue;
        for (∀ v adiacente u) {
              if (v ∈ Q AND W(u,v) < code.get(v))
                    Π[v] = u   // u è antenato di v;
                    code.set(v, W(u,v));
```

FLOYD

L'algoritmo di FLOYD il cammino minimo per tutte le coppie di un grafo pesato e orientato e contenente archi con costo negativo:

```
double[][] floyd() {

    double[] dist = [n,n];
    for( i=0; i<n; i++)
        for(j=0; j<n; j++)
            dist[i][j] = 0 se diagonale
                          altrimenti +∞

Scorro gli archi e controllo il peso:
for( ∀ arco del grafo) {

        dist[A.getIn()][A.getFin] = A.peso();

for( k=0, k ≤ k≤n → n-1 )
    for( i=0, i → n-1)
        for( j=0, j → n-1)
            if(dist[i][j] > dist[i][k] + dist[k][j])
                dist[i][j] = dist[i][k] + dist[k][j]

    return dist;

}
```

ALGO ALGO ALGO

Complessità varie:

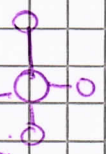
Viste alla lezione:

	TEMPORALE	SPAZIALE	
Anticipata:	$\Theta(n)$	$\Theta(\log n)$	CSP $\Theta(n)$
Posticipata:			
Livelli:			

- COUNTING SORT (BUCKETSORT): $\Theta(n)$ C.M $\Theta(n^2)$ C.P

- RADIX SORT $\Theta(n \cdot$ n° di cifre elemento più grande$)$

- COMPLESSITÀ MATRICE DI ADIACENZA $n^3 = n \cdot n^2$
  " LISTA DI ADIACENZA $2n$

- COMPL. TEMPORALE VISITA IN PROFONDITÀ GRAFI
  L.A $\Theta(n)$ CASO MIGLIORE $\Theta(1)$ (solo un nodo e no archi uscenti)
  M.A $\Theta(n^2)$ CM $\Theta(n)$
- COMPLESSITÀ SPAZIALE VISITA IN PROFONDITÀ
  CASO P $\Theta(n)$ in lista
  CASO M $\Theta(1)$ → SE GRAFO A STELLA 
- COMPLESSITÀ VISITA TEMP. VISITA IN AMPIEZZA GRAFI
  · L A $\Theta(n+m)$ M A $= \Theta(n^2)$
- COMPLESSITÀ SPAZIALE VISITA IN AMPIEZZA
  CM $\Theta(1)$ CP $\Theta(n)$
- COMP TEMPORALE ALGO DI VISITA GRAFO NON ORIENTATO
  M.A n^2 LA $n \cdot m$
  Pesa migliorarla in n^2
- PRIM $\Theta(m \log n)$ L.A OTTIMIZZATA
  altrimenti L.A = M.A $= \Theta(n^2)$
- DIJKSTRA $\Theta(m \log n)$

- KRUSKAL $\Theta(m \log m)$
- Union find $\Theta(m^2)$ ottimizzando con quick find $\Theta(m \log m)$
- Floyd spaziale $\Theta(m^2)$ temporale $\Theta(m^3)$
- Complessità dijkstra tra stringhe $\Theta(n+m)$
- FORD-FULKERSON $\Theta(m^2)$
- TEMPORALE QUICKSORT • $\Theta(m^2)$ PEGGIORE • $\Theta(m \log m)$ migliore
- TEMPORALE MERGSORT • $\Theta(m \log m)$

  SPAZIALE MERGSORT • $\Theta(\log m + m)$

```
package grafi;
import java.util.*;
public class Grafi {
	private Grafi() { }
	public static boolean eConnesso(Grafo<? extends Arco> g) {
		// Per grafi non orientati
		// Complessità pari al costo di visita del grafo
		List<Integer> l = g.depthFirstSearch(0);
		return l.size() == g.getN();
		/* Oppure:
		return numComponentiConnesse(g) == 1;
		*/
	}
	public static boolean eAciclico(Grafo<? extends Arco> g) {
		// Per grafi non orientati
		// Complessità:
		//	Lista -> O(n)
		//	Matrice -> O(n^2)
		if (g.getM() > g.getN() - 1) return false;
		return g.getM() == (g.getN() - numComponentiConnesse(g));
	}
	public static boolean eConnessoAciclico(Grafo<? extends Arco> g) {
		// Per grafi non orientati
		// Complessità:
		//	Lista -> O(n)
		//	Matrice -> O(n^2)
		return (g.getM() == g.getN() - 1) && eConnesso(g);
	}
	public static int numComponentiConnesse(Grafo<? extends Arco> g) {
		// Per grafi non orientati
		// Complessità pari al costo di visita del grafo
		boolean[] visitati = new boolean[g.getN()];
		for (int i = 0; i < g.getN(); i++) visitati[i] = false;
		boolean grafoVisitato = false;
		int nodoPartenza = 0, numComponenti = 0;
		while (!grafoVisitato) {
			List<Integer> l = g.depthFirstSearch(nodoPartenza); // oppure breadthFirstSearch
			numComponenti++;
			while (!l.isEmpty()) visitati[l.remove(0)] = true;
			while (nodoPartenza < g.getN() && visitati[nodoPartenza]) nodoPartenza++;
			if (nodoPartenza == g.getN()) grafoVisitato = true;
		}
		return numComponenti;
	}
	public static boolean eFortementeConnesso(Grafo<? extends Arco> g) {
		// Per grafi orientati
		// Complessità pari al costo di visita del grafo
		return numComponentiFortementeConnesse(g) == 1;
	}
```

Se un grafo è connesso
è aciclico se
$m = n - 1$;