Costi algoritmo:

- Il tempo di esecuzione, o costo computazionale, è espresso in funzione della dimensione dell'input.
- Il caso medio è il valore atteso del costo, ma per quantificarlo occorrerebbe conoscere la distribuzione probabilistica di ciascun caso; per questo, nella valutazione del costo, si tiene sempre presente il caso peggiore (post case scenario).

- Si definisce ***upper bound*** di un algoritmo la funzione f(n) di un algoritmo il cui costo di esecuzione è O(f(n)). Ne esiste uno sul tempo e uno sullo spazio.
- Si definisce ***lower bound*** di un algoritmo la funzione f(n) di un algoritmo il cui costo di esecuzione è Ω(f(n)). Ne esiste uno sul tempo e uno sullo spazio.
- Si definisce ***upper bound*** di un problema la funzione f(n) di un algoritmo il cui upper bound è O(f(n)) nel caso peggiore dell'implementazione migliore.
- Si definisce ***lower bound*** di un problema la funzione f(n) di un algoritmo che consiste nello stimare il lower bound del caso peggiore dell'implementazione migliore, considerando tutte le varianti implementative dell'algoritmo.

- Il lower bound si indica con **Ω**, upper bound si indica con **O**, la crescita equivalente si indica con **θ,** e indica l'equivalenza asintotica di due funzioni.

PQ (Code di Priorità)	Worst Case	Best Case	In place
Insertion Sort	$\theta(n^2)$	$\theta(n)$	Yes
Selection Sort	$\theta(n^2)$	$\theta(n^2)$	Yes
Heap	$\theta(n\log(n))$	$\theta(n\log(n))$	Yes
Merge Sort	$\theta(n\log(n))$	$\theta(n\log(n))$	No
Quick Sort	$\theta(n^2)$		

	Find	Insert	Remove	Find All	Predecessor	Min	Max
Array Disordinato	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Array Ordinato	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(\log(n)) + k$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$
Lista Disordinata	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Lista Ordinata	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$
Heap	$\theta(n)$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$
BST	$\theta(h)$	$\theta(h)$	$\theta(h)$	$\theta(h)$	$\theta(h)$	$\theta(h)$	$\theta(h)$
AVL	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$

$$h = \text{altezza dell'albero}$$

Alberi n – ari:

- Un albero è un modello astratto di struttura gerarchica, la cui definizione ricorsiva è: ogni nodo è un albero; ogni albero ha un numero qualsiasi di sottoalberi.
- ***La profondità*** di un nodo è il numero di antenati *(padri)* che ha.
- ***L'altezza*** è la profondità massima di un albero, corrispondente alla profondità della *foglia* più lontana dalla radice.
- **Non** c'è metodo di aggiunta di nodi;

- ***La visita*** (o attraversamento) in ***pre-ordine*** di un albero è la visita sistematica di tutti i nodi dell'albero, ciascuno visitato una volta sola. Si parte dalla cima e poi si visitano i discendenti.

Esempio:

```
void PreOrder(nodo) {
    if(nodo != NULL) {
        visita(nodo);
        PreOrder(nodo->sinistra);
        PreOrder(nodo->destra);
    }
}
```

- **La visita** in ***in-order*** si esplora prima il sottoalbero sinistro poi si visita il nodo corrente ed infine si passa al sottalbero destro. l'algoritmo esplora i rami di ogni sottoalbero fino ad arrivare alla foglia più a sinistra dell'intera struttura, solo a questo punto si accede al nodo. Terminata la visita del nodo corrente si procede poi con l'esplorazione del sottoalbero a destra, visitando sempre i nodi a cavallo dell'esplorazione del sottoalbero sinistro e quello destro.

Esempio:

```
void InOrder(nodo)
{
  if ( nodo == NULL ) return;
  InOrder( nodo->sinistra );
  visita( nodo );
  InOrder( nodo->destra );
  return;
}
```

- **La visita** in ***post-ordine*** ha uno schema opposto: quindi visita l'albero a partire dal fondo. L'algoritmo può essere una funzione di stampa o di altro tipo, a seconda delle esigenze.

Esempio:

```
void PostOrder(nodo)
{
  if ( nodo == NULL ) return;
  PostOrder( nodo->sinistra );
  PostOrder( nodo->destra );
  visita( nodo );
  return;
}
```

Pre-order	In-order	Post-order
θ(n)	θ(n)	θ(n)

- Gli alberi binari hanno diverse proprietà associate alle sue caratteristiche: n nodi, e nodi esterni, i nodi interni (n = e + i), h altezza. Il minimo numero di foglie di un albero binario è 1. Il massimo numero si registra in corrispondenza di alberi completi (con tutte le foglie sullo stesso livello). Il totale delle foglie è 2^h. Il totale dei nodi è $2^h - 1$. Quindi $h + 1 = (\log(n + 1) - 1)$.

- la rappresentazione di un qualsiasi albero di una qualsiasi dimensione:
  - un elemento
  - il puntatore al primo figlio
  - il puntatore al prossimo fratello

I nodi si possono memorizzare in un array, ipotizzando che parta da 1. I nodi vengono allocati nelle posizioni $2i, 2i + 1$. Quindi si può navigare nell'albero usando semplicemente gli indici degli array.

N.B.

Lo svantaggio è che si creeranno dei buchi esponenziali all'interno della struttura, a meno che l'albero non sia completo: in tal caso la rappresentazione è ottimale.

Code di priorità(PQ):

- © L'ADT (Tipi di dato astratto) ***coda di priorità*** gestisce una collezione di elementi di cui ogni entry è formata da una coppia $< key, value >$. La chiave indica il "privilegio" d'ingresso in coda; infatti, le code di priorità non sempre seguono la politica FIFO.

- © L'inserimento avviene tramite $insert(k, x)$ e per la rimozione implementiamo $removeMin()$, che rimuove l'entry col privilegio più alto (cioè con la chiave di valore minimo). Se esistono più elementi di questo tipo, se ne rimuove uno qualsiasi.

	Insert	Remove
coda con priorità ordinata (Insertion sort)	$O(n)$	$O(1)$
coda con priorità non ordinata (Selection sort)	$O(1)$	$O(n)$

- © Nel ***Selection sort*** la coda viene ordinata in base alla selezione degli elementi; il costo è $O(n^2)$ poiché per cercare la chiave minima si impiega $O(n)$, e per inserire l'elemento serve ancora $O(n)$. Il costo decresce mano a mano che la coda viene ordinata.

- © Nell'***Insertion sort*** si controlla sempre dove vada a finire l'elemento inserito. Il costo è sempre $O(n^2)$, poiché una "passata" serve per il controllo e una per l'inserimento.

- © ***Selection sort e insertion sort*** possono essere modificati per renderli ***in place***. L'Insertion sort in place viene realizzato spostando via via gli elementi con priorità maggiore. È per vantaggioso il fatto che gli elementi già ordinati vengano saltati in una data implementazione, quindi il costo minimo è $O(n)$.

Heap

- Esistono 2 tipi di heap=max-heap e min-heap

- **Min-Heap** -> la radice ha ***key*** più piccola
  - La chiave in un nodo non può essere maggiore delle chiavi che si trovano nei sottoalberi; e l'albero deve essere completo.

- Il ***min***() ha costo $O(1)$. Inserendo un nuovo minimo, può avere luogo una violazione dell'ordine.

- **Max-Heap**->la radice ha la ***key*** più grande
  - La chiave in un nodo non può essere minore delle chiavi che si trovano nei sottoalberi; e l'albero deve essere completo

  - Il ***max***() ha costo $O(1)$. Inserendo un nuovo massimo, può avere luogo una violazione dell'ordine.

- Gli algoritmi di gestione delle heap, ***l'upheap*** e il ***downheap***, ripristinano la priorità corretta. L'estrazione e ripristino dell'heap avviene in entrambi i casi in tempo $\theta(\log n)$

- Ogni coppia di valori di un dato nodo è una entry. Anche nel caso delle heap è possibile una ***rappresentazione tramite array***, seguendo la regola numerica in base alla quale ogni figlio ha posizione $2i$ e $2i + 1$ rispettivamente per il figlio sinistro e il figlio destro del nodo i. Nel caso dell'heap, la rappresentazione tramite array è pressoché ottimale, considerata la struttura.

- Ordinare un heap ***(HeapSort)*** secondo il **PQSort (Priority Queue Sort),** di tipo ***Insertion o Selection***, richiede un tempo pari a $O(n \log n)$. Tecnicamente il logaritmo sarebbe $\log(n!)$, considerato che gli inserimenti in una pila vuota hanno costi leggermente variabili, ma asintoticamente il costo è quasi uguale.

- ***Estrazione ed inserimento*** sono due operazioni opposte: laddove per l'inserimento c'è il caso migliore, di costo $O(1)$, per l'estrazione si avrebbe il caso peggiore $O(n \log n)$, e viceversa. Si può implementare anche l'HeapSort in place.

- Due heap della ***stessa*** **altezza e forma** possono essere ***fusi*** con un nodo aggiuntivo. Quest'ultimo diviene la radice e ha per figli le radici degli altri due alberi; dopo l'inserimento, di tempo costante, si esegue un sort sugli elementi. Se l'albero di sinistra è più piccolo di quello di destra, li si scambia prima di fonderli.

- La costruzione ***bottom-up*** parte da una rappresentazione in array di un heap, probabilmente disordinato. L'ultimo livello del caso in esame ha, supponiamo, 4 heap. Ogni sub-heap viene fuso con un nodo aggiuntivo che è il parent, e si prosegue a riordinarli. Il processo si ripete fino a risalire all'heap ordinato.

- Con $O(n)$ si può ordinare l'array tramite ***minHeap, maxHeap*** e altro algoritmo in place. Si può estendere la classe ***minHeap*** quando ne importa, senza modificarli, i comandi. Inserimento ed estrazione costano al più $O(\log(n))$.

- L'implementazione tramite array ha il difetto di limitare a priori la taglia della heap. Ogni volta che occorre ampliare la dimensione dell'array bisogna allocarne uno nuovo e riempirlo con tutti gli elementi già inseriti. Viene definita analisi ammortizzata poiché raddoppiando la dimensione dell'array ci si assicura che l'ampliamento della memoria avvenga molto raramente, permettendo di trascurarne i costi rispetto all'ottimizzazione dei metodi implementati.

Heap	Costo

Insert	$\theta(\log(n))$
Remove	$\theta(\log(n))$
Insert First	$\theta(1)$
Heap-Sort	$\theta(n\log(n))$
Bottom-up - downheap	$\theta(n)$

Mappa:

- Le mappe sono un tipo di dato astratto che rappresenta coppie <key, value> per le quali è possibile fare inserimento, cancellazione e ricerca.

- **Non** sono ammesse key uguali per due elementi distinti.

- I metodi principali delle mappe sono $\boldsymbol{get(k), put(k,v), e\ remove(k)}$. Per preservare la proprietà di unicità della chiave, il metodo put modifica chiavi eventualmente già esistenti. In quel caso restituisce l'elemento vecchio dopo averlo modificato.

- Le mappe si possono implementare attraverso ***liste concatenate bidirezionali non ordinate***. Usa due variabili di riferimento ai nodi (***previous*** *e* ***next***) e due per chiave e valore. Anche se non c'è ordine, sussiste comunque il problema del controllo della presenza della chiave inserita: non si può inserire in testa o in coda senza verificare che la chiave non ci sia già. Questo comporta che le liste non ordinate abbiano un ***costo d'inserimento*** minimo pari a $\theta(n)$,come le operazioni di ***ricerca*** e ***remove***.

Tabelle Hash:

- Strutture di dati tramite le quali è possibile implementare mappe. Ogni chiave k nelle tabelle è, idealmente, associata biunivocamente alla posizione della casella corrispettiva (la struttura è analoga agli array). **(Associazione diretta)**

- Organizzare le chiavi nell'insieme K che le contiene richiede una regola che determini l'associazione univoca delle chiavi a una posizione precisa dell'array/insieme che le contiene. Questo processo di ***shuffling*** deve dare, a parità di input, gli stessi risultati. Tale funzione si chiama ***funzione di hashing.*** In linea ideale dovrebbe essere possibile avere tutte le
operazioni in tempo costante $\theta(1)$, ma occorre prima risolvere i ***problemi di collisione***: ***è possibile che una tabella hash associ una chiave a una posizione in cui c'è già una chiave.***

- la funzione di hashing la trasforma in un numero, dopodiché lo "comprime", ad esempio attraverso l'operazione $h(x) = x * \mathrm{mod}(N(x))$. La parte più complicata è nella trasformazione della chiave in un numero. ***La funzione di hashing*** ha il compito di fare una buona operazione di ***shuffling***.

- Il ***costo*** della funzione di hashing non deve dipendere dalla dimensione della tabella, o scadrebbe nel lineare.

- Il ***separate checking*** consiste nell'associare ad ogni cella di array una lista collegata di entry mappate in quella stessa cella. ***Non*** bisogna comunque inserire la stessa volta due chiavi: ad ogni operazione di inserimento segue un'operazione di controllo mirata a individuare se l'elemento associato a una data chiave è già presente.
Questa soluzione è ottimale se solo alcune caselle sono abitate da liste, quindi nel caso peggiore le prestazioni sono pari a $O(n)$ in inserimento; non c'è grande vantaggio rispetto

alla lista unica. La soluzione diventa più conveniente se la tabella si sovradimensiona, riducendo così la probabilità che si formino liste lunghe e minimizzando di conseguenza la casistica peggiore.

- ***L'open addressing*** si fonda sulla filosofia in base alla quale un elemento che andrebbe in una casella occupata viene invece posizionato in un'altra casella. È una famiglia di algoritmi. Il più semplice di essi è il ***linear probing***, che consiste nel sistemare il dato nella prima casella libera successiva a quella data, ripartendo dalla prima se si raggiunge l'ultima. il costo, nel caso peggiore, è $O(n)$.

- il problema del ***clustering primario*** (agglomerazione): se due *caselle adiacenti* sono occupate, la probabilità di collisione aumenta; infatti, la probabilità di collisione della seconda casella è pari alla sua probabilità di collisione intrinseca, più la probabilità di collisione della casella precedente. Questo comporta l'inserimento di più elementi e la conseguente creazione di zone molto "popolate". Per questo si introduce il ***load factor***, che indica la percentuale della pienezza della struttura.

- Il problema sopraggiunge quando bisogna *eliminare un elemento*: usando ***l'open addrerssing*** bisogna scandire l'intero array per notare se l'elemento c'è. Un modo per evitare che succeda è usare ***un array di booleani*** che registri la presenza o meno di collisioni. Per far tornare a false la variabile è necessario sapere che non ci sarà più rimozione.

- Altro algoritmo è il ***quadratic probing***. Tramite quadratic probing la distanza tra due chiavi compete, perché aumento con la distanza dal primo punto. Non risolve il ***clustering secondario*** (dato da f = f).

- Nel caso di ***hashing doppio*** il metodo di sfasamento è $(i + jd(k)) * mod(N)$. Risolve sia il clustering primario che quello secondario. Il calcolo della funzione hashing, di per sé l'unica componente costosa dell'algoritmo. Quando la tabella è molto piccola, la soluzione primaria, il *linear probing*; quando è più grande (centinaia di entry) si usa il *quadratic probing*, ma non è ottimizzato, perché per esempio a load factor >= 0.7.

Dizionario:

- I dizionari violano il vincolo dell'unicità della chiave.

- L'algoritmo $find(k)$ presenta un problema: come fare a scegliere tra due chiavi uguali? Facile, implementare anche $findAll()$, che li tova tutti. Quindi l'inserimento ***non*** comporta più una ricerca: $O(1)$.

- I ***dizionari ordinati*** sono dizionari in cui ogni elemento è legato agli altri da una relazione d'ordine totale.

	get(x)	**put(x)**	**remove(x)**	**findAll()**
Lista non ordinata	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Array non ordinato	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Array ordinato	$O(\log(n))$	$O(n)$	$O(n)$	$O(\log(n) + k)$

Binary Search Tree (BST):

- Alberi che memorizzano chiavi (o entry <key, value>) nei nodi interni secondo la proprietà: dato un nodo e i suoi **due figli**, quello *sinistro* contiene solo ***key inferiori*** e quello *destro* contiene solo ***key superiori***

- Le operazioni supportate sono ***find e findAll***; la prima trova una qualsiasi entry associata a una data key, la seconda le trova tutte. Il costo dell'algoritmo ***findAll*** è $O(h + k)$, ove h è l'altezza e k è la numerosità delle key presenti per il valore v.

- Le operazioni di cancellazione sono immediate nel caso di zero figli e un figlio, ma nel caso di due figli occorre definire il ***successore*** come la minima tra le chiavi >=, e il ***predecessore*** come la massima tra le chiavi <=.

- Calcolando la media delle altezze di tutti i nodi viene restituita una media logaritmica. Questo implica che il costo atteso di ciascuna operazione è logaritmico, ma non è una garanzia. L'altezza di un BST varia tra n e logn;
  Per ovviare a questo inconveniente si fa *una range query*, o interrogazione d'intervallo: dati in input [a, b] vengono restituiti i nodi contenenti una key appartenente all'intervallo. Le range query si basano sulle visite in-order.

- Se ho albero quasi completo altezza BST è $O(\log(n))$ altrimenti $O(n)$, questo implica che su gli alberi non possiamo formulare una relazione d'ordine ma si utilizzano altre tecniche basate sull'altezza dell'albero.

- Rimozione 3 casi:
  - Nodo è foglia:
    - Il puntatore che punterà alla chiave k dovrà essere messo a NULL, ma bisognerà tenere in una variabile d'appoggio il padre.
  - Nodo ha figlio dx o sx:
    - Se K ha figlio sinistro vuol dire che si trova nel sottoalbero sinistro del padre di K.
    - Se K ha figlio destro vuol dire che si trova nel sottoalbero destro del padre di K
    - In entrambi i casi usiamo la tecnica Bypass in cui il padre di K NON punterà più a K ma al figlio di K in modo tale da togliere K.
  - Nodo ha 2 figli:
    - Non posso usare la tecnica del bypass ma devo trovarmi il predecessore/successore di K, che è il massimo tra i più piccoli/minimo tra i più grandi.
    - Il predecessore lo metto al posto di K e noto che a destra di K è soddisfatta la condizione di albero di ricerca e anche a sinistra di K è soddisfatta la condizione. Poi rimuovo K, senza fare poi ulteriori controlli e scambi tra i puntatori.

	get	put	remove
BST	$O(h)$	$O(h)$	$O(h)$

AVL:

- Albero binario di ricerca ma con una proprietà di bilanciamento: le altezze dei sottoalberi dei figli di un nodo possono differire al più di 1

- Il fattore di bilanciamento (FDB) è l'altezza del sottoalbero destro meno quella del sinistro, compresa in [-1, +1]. Un fattore diverso implica che l'albero non è bilanciato.

- Albero di Fibonacci è un particolare albero AVL che ammette fattore du

- Gli AVL sono alberi di ricerca bilanciati. Nelle operazioni d'inserimento a volte si rende necessario il ribilanciamento, che avviene a opera di rotazioni. Ne esistono di quattro tipi: sx-sx, sx-dx, dx-dx, dx-sx. Occorre tenere conto del fatto che con ciascuna operazione, potenzialmente, si potrebbe sbilanciare l'intera struttura che lo collega alla radice.

- La rotazione DD avviene in corrispondenza dell'aggiunta di un nodo a un sottoalbero AVL dal lato destro, così come la rotazione SS avviene per il sinistro. Nel caso del destro, ogni nodo di fattore di bilanciamento pari a +1 dello stesso ramo del nodo inserito viene sbilanciato, poiché tale fattore diventa +2. Gli sbilanciamenti vanno corretti al più al nodo "grandparent", poiché la rotazione così effettuata preserva l'altezza originaria dell'albero, bilanciando automaticamente anche gli altri nodi.

- Anche nelle ***rotazioni doppie*** la proprietà di preservazione dell'altezza si conserva. Se l'inserimento è a sinistra di un nodo, la rotazione avviene prima verso destra (il genitore del nodo inserito diventa il suo figlio destro, e il progenitore del nodo inserito diventa il suo figlio sinistro) e poi verso sinistra (il nodo più in basso e più esterno viene sostituito al suo genitore, che diventa suo figlio), e viceversa.

- Per controllare l'altezza o il fattore di bilanciamento di un dato albero, è sufficiente controllare il suo albero genitore con tutti i relativi progenitori, che la memorizzano come variabile d'istanza.

	get	put	remove	rotazione
AVL	O(log(n))	O(log(n))	O(log(n))	O(1)

Merge Sort:

- Ordinamento merge sort fa parte della categoria dei ***divide et impera***, la cui idea di base è la suddivisione in regioni più piccole per poi organizzare localmente. **NON** è un algoritmo ***in-place***.

- Il merge sort divide quindi il problema in due o più *sottoproblemi* equamente ripartiti, itera il procedimento fino a risolverli, e poi unifica i risultati ottenuti. Ad esempio, un problema di ordinamento su array si riduce in ultima analisi a un albero di chiamate ricorsive che giunge a ordinare la singola casella dell'array e unificare le caselle così ordinate.

- Il costo del merge sort è O(nlog(n)) il che lo profila come uno dei migliori algoritmi di ordinamento esistenti. In caso di elementi dispari in numero, l'albero binario delle ricorsioni ovviamente non è completo, ma risulta in ogni caso bilanciato.

- L'idea fondamentale del merge si basa sul continuo *confronto tra le due sequenze ordinate*: ogni volta viene preso l'elemento minore tra i due alla testa delle sequenze ordinate, e viene anche rimosso dalla sequenza da cui è stato estratto e messo nell'insieme unione, il cui indice di scorrimento viene aumentato. Una volta esauriti gli elementi di uno dei due insiemi, un ciclo while completa le operazioni di inserimento prendendo tutti gli elementi rimanenti dell'altro insieme.

- *La suddivisione degli elementi* avviene in ***preordine***, mentre la *fusione* avviene in ***postordine***. I costi del merge sort derivano dalla suddivisione in albero ricorsivo per la parte

logaritmica, più l'ordinamento delle due metà per la parte lineare, risultando nell'atteso $O(nlog(n))$

N.B.

- Si noti che le equazioni di ricorrenza sono modellate su *algoritmi ricorsivi*, quindi ***non*** esistono per algoritmi ciclici. Inoltre, non sono definite per gli alberi se non per alberi binari completi o altrimenti omogenei, perché le due ripartizioni devono essere omogenee.

Quick Sort:

- Quick sort è implementato sempre secondo il ***divide et impera***: sceglie un elemento casuale, chiamato **pivot**, e partiziona l'insieme in elementi *maggiori, minori e uguali*. Dopo di questo, ordina ricorsivamente e fondo le sequenze ordinate. In tal modo, l'unico insieme totalmente ordinato è quello al centro, il gruppo di elementi uguali al pivot.

- L'efficienza di questo algoritmo dipende dalla ***scelta del pivot:*** se si riesce a partizionare l'insieme in due parti pressoché uguali, si minimizzano le chiamate ricorsive, altrimenti massimizzate se il pivot scelto è il minimo o il massimo dell'insieme. Il caso peggiore è $O(n^2)$ e quello migliore è $O(nlog(n))$

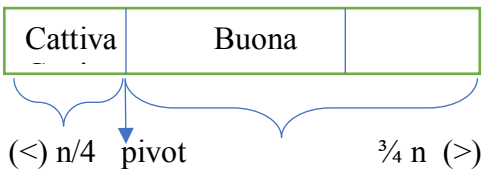
- Il primo passo è il partizionamento dell'insieme. Tutti *gli elementi maggiori vengono accodati all'insieme, mentre tutti gli elementi minori vengono aggiunti in testa.* Anche il quick sort può essere rappresentato come albero binario.

- Se numero i livelli pari ad n, per partizionarli farò un lavoro pari ad n e quindi il costo della quick è quadratico. Ma se il numero di livelli dell'albero è logaritmico per ogni livello spendo al massimo n, quindi il costo sarà $O(nlog(n))$.

- Un modo per evitare il caso peggiore è scegliere tre elementi a caso della sequenza, confrontarli e scegliere il valore mediano dei tre. In tal modo si è sicuri che il pivot scelto non sarà né il minore né il maggiore. Sebbene in informatica si sia portati a fare analisi di caso peggiore, nel caso del quick sort è pessimistico pensarlo: le probabilità del caso peggiore sono ridottissime.

- Il pivot può essere il risultato di una *buona scelta* o di una *cattiva scelta*: nel primo caso, la dimensione di ciascun insieme non supera i 3/4 degli elementi del sottoinsieme d'origine. Quindi, a ogni passo, la probabilità di fare una buona scelta è 1/2.

- ¾ perché se prendo ad esempio il minimo o massimo della parte buona, avrò che n/4 è parte cattiva e ¾ n è parte rimanente



- È possibile effettuare il ***quick sort in-place***: al posto di usare strutture d'appoggo, si usano indici per scandire ad esempio l'array dall'inzio e dalla fine. Appena scelto il pivot, lo si scambia col primo elemento per semplicità. Anche gli elementi delle due metà che dovrebbero trovarsi nell'altra metà si scambiano. Successivamente il pivot viene reinserito a metà tra le due sequenze.

- Il ***lower bound*** non è sempre semplice da determinare: una delle tecniche di base è l'albero binario di ricerca. Il numero di foglie dell'albero è n!, tutti i possibili ordinamenti di tutti gli elementi.

- Dato che l'altezza dell'albero è almeno pari al logaritmo delle foglie, che sono la metà dei nodi, log(n/2) <= h, il che significa che l'altezza minima dell'albero risulta essere è $O(n\log(n))$ che è il lower bound degli algoritmi di ordinamento. Lo stesso procedimento si può ripetere per gli algoritmi di ricerca, provando che il loro lower bound è $O(\log(n))$.

Quick Select:

- Si basa sullo stesso principio del quickSort: prende un elemento a caso nell'insieme (pruning del pivot) e divide il resto degli elementi in maggiori e minori.

- L'analisi prosegue nel sottoinsieme dato: *il pivot ha una sua posizione nell'insieme*, quindi si sa se, a partire da esso, occorre andare a sinistra (insieme dei minori) o a destra (insieme dei maggiori). Proseguendo nei minori, si cerca sempre l'elemento k; proseguendo nei maggiori, si ***sottraggono a k la cardinalità degli elementi minori e uguali***, e si cerca l'elemento k'.

- Nel caso peggiore l'algoritmo richiede $O(n^2)$ ma il caso medio è lontano dal caso peggiore. Le buone chiamate del metodo sono quelle che, come nel quickSort, *suddividono l'insieme in porzioni entrambi grandi almeno 1/4 del totale*, il che avviene con probabilità 1/2.

- $T(n) = T\left(\frac{3}{4}n\right) + 2n = T\left(\left(\frac{3}{4}\right)^2 n\right) + 2\frac{3}{4}n + 2n = T\left(\left(\frac{3}{4}\right)^2 n\right) + \left(\frac{3}{4} + 1\right)2n =$
  $T\left(\left(\frac{3}{4}\right)^k n\right) + 2n\sum_{i=0}^{k-1}(\frac{3}{4})^i$

  $k = \log_{\frac{3}{4}} n$

  $T(1) + 2n\sum_{i=0}^{\infty}(\frac{3}{4})^i\ (Serie\ geometrica) \leq T(1) + 8n \quad \in O(n)$ [lower bound]

Insiemi:

- Operazioni più importanti da essi supportate sono ***l'unione e l'intersezione***. Hanno un criterio di ordinamento arbitrario.

- L'operazione di ***unione*** si basa sullo stesso principio del merge di ordinamento; l'unica differenza è che *all'incontro di due elementi uguali ne aggiunge **uno solo***. L'algoritmo di intersezione fa esattamente il contrario.

- Rappresentando gli *insiemi disgiunti come* **liste**, ciascuna lista ha una targhetta con il nome dell'insieme ed ogni elemento della lista ha puntatore all'insieme. Lo spazio utilizzato è $O(n)$

- Se concateno due liste dovrò scegliere tra una o l'altra targhetta e in base alla mia scelta, dovrò aggiornare i puntatori della lista che non ho scelto, ovviamente in modo tale che quei puntatori puntino al nome dell'insieme nuovo che ho scelto

- La scelta dell'etichetta viene fatta furbamente, cioè sceglierò l'etichetta che andrà a far aggiornare i minor numero di puntatori possibili.

- Gli elementi non possono essere uniti un numero indefinito di volte: il costo di aggiornare il suo puntatore e l'arrivo in un insieme grande il doppio di quello precedente determinano il fatto che, dopo il decimo reinserimento, l'elemento giunga a n. Giungervi costa al più $O(\log(n))$.

- **MakeSet**:
  - Creazione di una lista con la targhetta , ma crea un insieme con un solo elemento e infatti ha costo $O(1)$

- Rappresentando *insieme disgiunti come* **alberi:**
  - Nodi contengono riferimento al genitore
  - Radice contiene riferimento a sé stessa
  - Elemento della radice è usato come nome o rappresentante dell'insieme
  - *Unione*: basta assegnare al riferimento presente nella radice di uno dei due alberi, la posizione dell'altro sottoalbero della radice dell'altro
  - *Find*: basta seguire i riferimenti, partendo dal nodo assegnato, fino ad arrivare a un nodo che punta a se stesso
  - **Union-by-size**:
    - L'insieme con più elementi viene attaccato,cioè a ciascuna unione, si fa puntare la radice dell'albero con meno nodi a quella dell'albero con più nodi
  - **Union-by-rank**:
    - La radice con rank(altezza sottoalbero) minore viene attaccata a quella con rank maggiore cosi da mantenere altezza $O(\log(n))$.

- Il calcolo delle prestazioni per gli insiemi è complesso e fa riferimento alla funzione di Ackerman che, in ultima analisi, riconduce i costi di ciascuna operazione a costi lineari.

- L'implementazione di questi algoritmi è flessibile: come si possono realizzare con alberi, si possono realizzare con array, oppure alberi basati su array. Non è necessaria alcuna struttura specifica.

Grafi:

- G(V,E)
  - V insieme di nodi, vertici
    - Es. $u, v \in V$
  - E insieme degli archi
    - Es. $\{u, v\} \in E$

- ***Tipi*** di Grafo:
  - *Semplice* – Non diretto
    - Non orientato
    - No cappi
    - No archi multipi
  - *Orientato* – Diretto
    - Orientato
  - *Competo*
    - *Grafo semplice* che ha massimo numero di archi
      - Es. Kn= grafo completo di n nodi
    - Numero di archi= $\frac{n(n-1)}{2}$

- ***Percorso Grafo non orientato(Path)***: sequenza nodo-spigolo fino ad arrivare al nodo stabilito

- *Percorso semplice*: si passa una sola volta per quel nodo,cioè non ritorno su quel nodo, tutti vertici e archi sono distinti
- *Percorso chiuso*: primo e ultimo nodo sono gli stessi
- *Percorso non semplice*: passo più volte per uno stesso nodo, cioè posso ripassare su stesso nodo, ma senza visitarlo, verificando solo se è già stato visitato

↺ ***Proprietà grafo non orientato***:
- *Grado del nodo*: numero di spigoli(archi) coincidenti sul nodo
  - N.B.
    - Arco coincide su 2 nodi
    - 2 nodi sono adiacenti ad 1 arco
  - $\sum_{v_i \in V} deg(v_i) = 2m$
    - $|V| = cardinalità\ dei\ veritici = n$
    - $|E| = cardinalitàdeglispigoli = m$
- $m \leq n(n-1)/2$
  - Ciascun vertice ha grado al più $(n-1)$

↺ ***Visita grafo:***
- DFS – profondità
  - Basa sulla pila
    - ricorsione
- BSF – lunghezza
  - Basa sulla coda
    - ciclo

↺ ***Rappresentare grafo:***
- *Lista adiacenze*
  - Ciascun nodo ha una lista di nodi adiacenti ad esso
    - Liste adiacenza comportano una ridondanza dal punto della segnatura, posso ripetere più volte nodo
  - Lo spazio che occupo per rappresentare il grafo è pari $\theta(n+m)$
- *Matrice adiacenza*
  - Ad ogni vertice è associata chiave intera (indice)
  - Righe e colonne della matrice sono appunto nodi e verifichi se sono adiacenti
  - Nel posto (i, j) della matrice si trova un **1** se e solo se esiste nel grafo un arco che va dal vertice *i* al vertice *j*, altrimenti si trova uno **0.**
  - La matrice è simmetrica è la *diagonale* è sempre vuota
  - Lo spazio che occupo per rappresentare il grafo è pari $\theta(n^2)$

N vertici,m spigoli	**Lista adiacenze**	**Matrice Adiacenze**
Space	$O(n+m)$	$O(n^2)$
Archi-incidenti-su-nodo(v) ***(incidentEdges(v))***	$O\big(deg(v)\big)$	$O(n)$
Test-Adiacenza ***(areAdjacent(v,w))***	$O\left(min\big(deg(v), deg(w)\big)\right)$	$O(1)$
Aggiungi-vertice ***(inserVertex(o))***	$O(1)$	$O(n^2)$
Aggiungi-Arco ***(insertEdge(v,w,o))***	$O(1)$	$O(1)$

Rimuovi-Vertice (removeVertex(v))	$O(deg(v))$	$O(n^2)$
Rimuovi-Arco (removeEdge(e))	$O(1)$	$O(1)$

- ***Considerazioni:***
  - ***Test-Adiacenza (areAdjacent(v,w))*** **:**
    - *Matrice* è $O(1)$ perché basta che visito la cella
  - ***Space:***
    - *Lista:* Sapendo che per rappresentare n nodi per ciascun nodo ha una lista che ha al su inteno n elementi pari al grado del nodo
    - *Matrice:* Essendo matrice nxn occupa esattamente n^2
  - Se inserisci o rimuovi conviene la lista di adiacenza invece per quanto riguarda il test di adiacenza conviene la matrice ma la matrice ha il lato negativo di occupare uno spazio in forma quadratica

- ***Sottografi:***
  - $\mathbf{G' = (V', E')}$ $\mathbf{V' = V}$ **,** $\mathbf{V' \subseteq V}$ **,** $\mathbf{E' \subseteq V'x\ V, E' \subseteq E}$
  - Vertici del sottografo sono un sottoinsieme di quelli del grafo
  - Spigoli del sottografo sono un sottoinsieme di quelli del grafo
  - *Sottografo ricoprente* è una sottografo di che contiene ***tutti*** i vertici del grafo

- ***Connettività:*** grafo è connesso se ***esiste*** un percorso fra ogni coppia di vertici ,cioè se scelgo un vertice sx e uno dx ci sta un percorso che li connette

- ***Componente connessa:*** è un sottografo del grafo connesso e massimale
  - ***Se grafo è connesso:*** ha esattamente uno componente connessa

- ***Albero:*** grafo *non orientato connesso e aciclico*
  - ***Albero ricoprente (Spanning Tree):*** un albero che contiene tutti i nodi del grafo, ma degli archi ne contiene soltanto un sottoinsieme, cioè solo quelli necessari per connettere tra loro tutti i vertici con uno e un solo percorso.

- ***Foresta:*** grafo *non orientato aciclico*
  - ***Componenti connesse:*** alberi
  - ***Foresta ricoprente(Spanning Forest):*** una foresta che contiene tutti i nodi del grafo, ma degli archi ne contiene soltanto un sottoinsieme, cioè solo quelli necessari per connettere tra loro tutti i vertici con uno e un solo percorso.
  - ***Se grafo è connesso*** lo ***Spanning Tree*** coincide con lo ***Spanning Forest***

- ***Visita in profondità grafo non orientato (Vedi Pseudocodice dopo):***
  - *Driver*
    - Serie di inizializzazioni a causa del fatto che se grafo ha un ciclo mi porto a rivisitare un nodo, quindi ho bisogno di introdurre un'etichetta per marcare il nodo
    - Setto nodi e archi – UNEXPLORED
    - Scelgo vertice UNEXPLORED e inizio la visita con DFS
  - *DFS*
    - Chiama su un grafo a partire da un vertice e a partire da quest'ultimo visita il visitabile, cioè tutte le componenti connesse che contiene il nodo visitato v Marco subito etichetta di v come VISITED e controllo quindi per tutti archi incidenti/nodo adiacente se un arco è stato visitato

- Se UNEXPLORED
    - Vedo vertice w, opposto di v,cioè l'altro nodo appartenente arco su cui è collegato v
    - Se w è UNEXPLORED
        - Marco DISCOVERY e richiamo ricorsivamente sul nodo w
    - Se w è VISITED
        - Marco BACK
- Numero di volte che chiamo la DFS è pari al numero di componenti connesse che ho e per ogni nodo avrò al più deg(nodo) chiamate ricorsive

- *Costo Visita*: $O(n+m)$ [Lista di adiacenza]
- *Quest'algoritmo **NON** è un algoritmo in cui vado a calcolarmi il percorso **più breve** all'interno dal grafo ma un percorso **qualsiasi***

↺ ***Ricerca Percorsi grafo non orientati(DFS)(Vedi Pseudocodice dopo):***

- Parti da un nodo v e arrivi in z
- Usando la pila Stack
    - Setto nodo visitato e lo metto nella Stack
- Se v=z ritorno gli elementi della pila
- Se no scandisco tutti archi incidenti del vettore v
    - Prendo vertice opposto w
    - Se w è UNEXPLORED
        - Setto arco DISCOVERY e metto arco nella Stack e chiamo ricorsivamente la funzione sul nodo w
        - Se poi nelle chiamate ricorsive arrivi ad un nodo che non ti va bene lo elimini cercando poi di vedere se su un altro nodo trovi percorso se ancora non trovi nulla elimini l'arco dalla pila e continui
    - Se è VISITED
        - Setto BACK
- Rimuovi vertice dalla coda, dopo ave visitato tutti i suoi archi incidenti

↺ ***Ricerca Cicli grafo non orientato(DSF)(Vedi PseudoCodice dopo):***

- Stesso principio della ricerca percorsi in cui vado sempre ad utilizzare la stack

↺ ***Grafi diretti(digrafi):***

- Tutti spigoli(archi) sono orientati
- Numero archi $m \leq n(n-1)$ perché non ho più deg(nodo) avrò archi uscenti e archi entranti
    - Somma di tutti archi uscenti=m
    - Somma di tutti archi entranti=m
    - *lista di archi uscenti* la quale è obbligatoria
    - *lista di archi entranti **non*** è obbligatoria
- **In-degree=0** se *vertice sorgente,* no archi entranti
- **Out-degree=0** se *vertice pozzo*, no archi uscenti

↺ ***DFS dei digrafi:***

- Simile a quella dei grafi non orientati ma con opportune modifiche
- Ci sono 4 tipi di spigoli: DISCOVERY,BACK,FORWARD,CROSS
    - *Archi BACK* chiudono ciclo, cioè arco esplorato che porta ad un arco già visitato precedentemente,cioè porta ad un suo antenato.

- *Archi FORWARD*,arco esplorato che porta in avanti, cioè porta un vertice a un suo discendente.
- *Archi CROSS*, arco esplorato che porta ad vertice che non è ne suo antenato ne suo discendente.
- *Archi DISCOVERY*, arco che porta vertice non esplorato.

↺ ***Connettività debole:***
- *Digrafo è debolmente connesso* quando grafo semplice sottostante è connesso
- *Non* implica la connettività debole.

↺ ***Connettività forte:***
- *Digrafo è fortemente connesso* se esiste un cammino da ogni vertice verso un altro vertice oppure se per qualunque coppia di vertici {u,v} esiste un percorso orientato che va da u a v e un altro che va da v ad u.
- *Implica* la connettività debole.
- *Verifica connettività forte* utilizzando DFS(o D-DFS)
  - Un grafo è fortemente connesso se e solo se il suo trasposto è fortemente connesso
    - ❖ Per le liste di adiacenza il grafo trasposto ha come lista degli archi entranti quella degli archi uscenti del grafo.
    - ❖ Per la matrice di adiacenza basterà scambiare gli indici e ottengo la matrice di adiacenza del grafo traso
  - ***Condizione necessario per la connettività forte:*** *avendo un vertice v devono esistere percorsi da v verso tutti gli altri nodi*
    - ❖ Dopo questa verifica restituisco 2 informazioni

  1. Se ho visitato tutti i nodi
  2. Nodo di partenza v
     - ❖ Se visitato tutti i nodi utilizzo grafo trasposto, verifico che se il trasposto è fortemente connesso allora anche il grafo è fortemente connesso
     - ❖ Quindi vuol dire che eseguo una DFS partendo dallo stesso nodo di partenza v e restituisco le solite 2 informazioni

  1. Se ho visitato tutti i nodi
  2. Nodo di partenza v
     - ❖ ***Se verificato che no visitato tutti i nodi del grafo trasposto allora grafo è fortemente connesso***
- *Costo D-DFS*: $O(n+m)$ [Liste di adiacenza]
- *Costo D-DFS*: : $O(n^2)$ [Matrice di adiacenza]
  - Perché trovare adiacenti consta n per riga e n per colonna

↺ ***Grafo sparso:*** numero archi m $\in O(n)$

↺ ***Grafo denso:*** numero archi $m \in O(n^2)$

↺ ***Componenti fortemente connesse digrafi:***
- Sottografo indotto fortemente connesso e massimale(più grande sottografo che puoi avere),cioè non posso aggiungere altro nodo e avere ancora un grafo fortemente connesso
- ***Sottografo indotto:***
  - grafo che ha massimo numero di archi che appartengono al sottografo originale
- *Operazioni per verificarla*
  - D-DFS(G) prendi nota delle etichette di uscita

- D-DFS(G^T) con condizione che il "driver" considera vertici secondo ordinamento inverso delle etichette prese
- *Costo* $O(n+m)$ perché facciamo solamente 2 DFS

↺ ***Poset:*** insieme parzialmente ordinato su cui ho definito una relazione binaria che è una relazione d'ordine

↺ ***Dag:*** grafo diretto e aciclico

↺ ***Ordinamento topologico(Vedi Pseudocodice dopo):*** da una relazione d'ordine parziale a una totale
- Numerazione dei vertici tale che se esiste spigolo (u,v) allora u< v

↺ ***Th***. Un digrafo ammette ordinamento topologico se e solo se è un DAG

↺ ***Th.*** Ogni DAG ha almeno una sorgente e un pozzo
- Dim.
  - Supponiamo che ciò non sia vero quindi vuol dire che tutti i nodi hanno archi di ingresso e archi di uscita, questo vuol dire che dopo che avrò visitato al più di n nodi andrò a visitare un nodo che ho già visitato, questo determina un ciclo ciò implica una contraddizione
- Sorgente e pozzo possono non essere unici

↺ ***Chiusura transitiva:***
- Digrafo G^* ha gli stessi vertici del grafo G
- Se G ha percorso da u a v allora G^* ha uno spigolo orientato da u a v
- Fornisce quindi informazioni esplicite di raggiungibilità in un digrafo
- Calcolo chiusura 2 modi:
  1. *Eseguire nDFS da ogni vertice*
     - ❖ Costo $O\big(n(n+m)\big)$+ $O(n^2)$ [l'ultimo riguarda aggiunta degli archi]
       - Se è *grafo sparso* è $O(n^2)$
       - Se è *grafo denso* è $O(n^3)$
  2. *Tecnica "programmazione dinamica"*
     - Algoritmo di ***Floyd-Warshall(Vedi Pseudocodice dopo)***
       - Costo $O(n^3)$

↺ ***Diametro grafo:*** distanza più grande che esiste tra due nodi

↺ ***BFS(Vedi Pseudocodice dopo):***
- *Driver:*
  - Inizializzazione
- *BFS:*
  - Esplorazione dato nodo, in cui si privilegiano i nodi vicini a quello da cui si comincia ad esplorare, Si costruiscono k liste per ogni distanza (ma non sono obbligatorie)
  - *Non* è funzione ricorsiva ma è regolata da un ciclo while che verifica finché lista di quella lunghezza non è vuota, mi costruisco nuova lista vuota della distanza successiva.
  - Mi scandisco elementi(vertici) della lista della distanza corrente e per tutti gli archi incidenti a quel v vertice
    - ❖ Se arco è UNEXPLORED

- Vado a considerarmi il nodo w (opposto di v)
  - Se e è UNEXPLORED
    - Setto arco DISCOVERY e w VISITED e inserisco w nella lista della lunghezza successiva a quella corrente
  - Setto arco CROSS e continuo ciclo, l'etichetta CROSS non è la stessa di quella del DFS ma indica che arco *non* appartiene allo spanning tree e *non* interessa se è CROSS o FORWARD
- L'ultima lista non potrà aggiungere nulla perché ormai avrà visitato tutti i nodi e quindi sarà NULL e uscirà dal ciclo
- È impossibile che un vertice NON venga scandito perché vi è il ciclo while, inoltre non è possibile che venga visitato 2 volte a causa della marcatura
- Costo $O(n+m)$ dato dal grado del vertice, questo perché il fatto che scandisco tutti i vertici implica che scandisco tutti gli archi

o Se si tratta di un grafo orientato o no non ha differenza perché non poniamo un'etichettatura precisa

↺ ***Semplificazione BFS:***

o 1 sola lista gestita come coda
- Perdita informazione dell'indice della lista che indica la distanza massima
  - ❖ Soluzioni
    - *Nodo fantoccio:*
      - Inizializzo a zero e la metto in coda insieme al nodo iniziale, come un contatore, appena lo estraggo lo incremento e lo rimetto subito in coda perché vuol dire che è finito un livello
    - *2 code*:
      - Simile esercitazione sugli alberi

↺ ***Grafi pesati:*** ogni arco ha associato un valore numerico detto *peso* dell'arco

↺ ***Problema cammini minimi:***

o *Shortest Path(SP)*
- Dato un grafo e 2 vertici trovare percorso minimo che collega i due vertici

o *Single Source Shortest Path(SSSP)*
- Dato un grafo e un nodo trovare la distanza minima tra esso e tutti gli altri nodi

o *All Pairs Shortest Path (APSP)*
- Dato un grafo trovare distanza minima di tutte coppie di vertici nel grafo
- Soluzione(caso non grafo pesato)
  - ❖ Faccio n volte la BFS e mi trovo n alberi dei cammini minimi
  - ❖ Costo $O\big(n(n+m)\big)$
    - Grafo denso costo $O(n^3)$
    - Grafo sparso costo $O(n^2)$

↺ ***Proprietà cammini minimi:***

o Un sottocammino di un cammino minimo è un cammino minimo

o È sempre possibile descrivere insieme dei cammini minimi a partire da un nodo attraverso un albero con radice s. La dimensione dell'albero è lineare, pari al numero di nodi
- Oppure
  - ❖ L'insieme dei cammini minimi da un vertice di partenza a tutti gli altri vertici forma un albero

- ***Problema rappresentazione del grafo***
  - Lista di archi del grafo e ciascun arco del grafo contiene informazione sui due vertici e il *peso* dell'arco

- ***Problema lunghezza percorso:***
  - Prima era numero degli archi, ora è la somma dei *pesi* degli archi

- ***Algoritmo di Dijkstra(Vedi Pseudocodice dopo):***
  - Si ispira alla BFS e l'obbiettivo è fare *SSSP*
  - Ho un nodo di partenza (router in telecomunicazioni) ed a partire da esso mi raccolgo percorsi ottimali per arrivare agli altri nodi
  - Man mano che visito costruisco una "nuvola" che ha al suo interno i nodi visitati. L'inizializzazione della "nuvola" è la sorgente. L'algoritmo farà quindi (n-1) passi, ne sceglie 1, lo processa, capisce percorso minimo e lo mette nei raggiunti("nuvola"). Quindi quest'algoritmo e si strutturerà in modo tale da avere albero dei percorsi minimi.
  - Per ogni nodo gestisco informazione che è una *stima* della distanza da S (nodo partenza) è zero e quella degli altri nodi è ∞
  - Ad ogni passo, visito nodo, metto nella nuvola e vedo archi uscenti, in particolare vedo se posso modificare la stima e in caso aggiorno la stima dei nodi, per poi prendere un nuovo nodo scelto in base alla stima più bassa.
  - Costo $O\big((m+n)log(n)\big)$

- ***La stima è sempre esatta***
  - *Dim*.
    - Supponiamo per assurdo che la stima <u>*non*</u> sia esatta e se la distanza vera è *Minore* vuol dire che raggiungo quel determinato nodo (MIN) con un altro percorso
    - Ma in quest'altro percorso il nodo precedente a quello che sto considerando sta nella "nuvola" o fuori?
      - Supponiamo che nodo precedente sia nella nuvola ma questo vuol dire che il percorso che va fino a MIN è *esatto* perché a questo punto Min-Nodoprecedente ha percorso minimo
      - Supponiamo che nodo precedente sia fuori dalla "nuvola" ma questo vuol dire che <u>*non*</u> avrei scelto quel determinato nodo precedente ma che quindi ne avrei preso un altro questo porta ad una contraddizione che implica che la stima è <u>*sempre*</u> esatta

- ***Rilassamento arco:***
  - aggiornamento percorso migliore, prendo distanza minima tra quelle che arrivano al nodo $\mathrm{d(z)} \leftarrow \min\{\mathrm{d(z)}, \mathrm{d(u)} + \mathrm{peso(arco)}\}$

- ***Algoritmo di Bellam-Ford(Vedi Pseudocodice dopo):***
  - Vale più di Dijkstra
  - Consente di contemplare archi con peso negativo
  - Non usa heap
  - Prende tutti archi vedo se arrivo a un determinato nodo ad una distanza che magari è cambiata
  - Avrò per forza un grafo *orientato* se ho peso negativo perché se ho un grafo *semplice* non avendo un verso potrò percorrere quell'arco infinite volte. Grafo non orientato, ogni arco può essere considerato come un ciclo

- Altro ciclo che esegue volta in più corpo del ciclo precedente perché se eseguo n-iterazioni verrà dire che avrò avuto un ciclo negativo, cioè ho fatto un altro miglioramento delle stime usando peso negativo
- Costo $O(m * n)$, m archi, (n-1) iterazioni
- Se archi *non* sono negativi Bellam-Ford non mi conviene, distrugge efficienza

↻ ***Algoritmo APSP – Floyd Warshall (Vedi Pseudocodice dopo)***

	Grafo sparso	***Grafo denso***
Floyd Warshall	$O(n^3)$	$O(n^3)$
Dijkstra	$O(n\log(n))$	$O(n^2 log(n))$
Bellam-Ford	$O(n * m)$	$O(n^2 * m)$

Pseudo-Codici:

Selection sort: ***(trova il più piccolo nell'array e se non è quello corrente lo scambia)***

```
procedure SelectionSort(a: array);
  for i = 1 to n - 1
    posmin ← i
    for j = (i + 1) to n
      if a[j] < a[posmin]
        posmin ← j
    if posmin != i
        tmp ← a[i]
        a[i] ← a[posmin]
        a[posmin] ← tmp
```

Il ciclo interno è un semplice test per confrontare l'elemento corrente con il minimo elemento trovato fino a quel momento (più il codice per incrementare l'indice dell'elemento corrente e per verificare che esso non ecceda i limiti dell'array).
Lo spostamento degli elementi è fuori dal ciclo interno: ogni scambio pone un elemento nella sua posizione finale quindi il numero di scambi è pari a N-1 (dato che l'ultimo elemento non deve essere scambiato). Il tempo di calcolo è determinato dal numero di confronti.

L'ordinamento per selezione effettua $N(N-1)/2$ confronti e, nel caso peggiore/migliore/medio $\theta(n-1)$ scambi.

La complessità di tale algoritmo è dell'ordine di $\theta(n^2)$

MergeSort:

```
function mergesort (a[], left, right)
    if left < right then
        center ← (left + right) / 2
        mergesort(a, left, center)
        mergesort(a, center+1, right)
        merge(a, left, center, right)
```

```
function merge (a[], left, center, right)
    i ← left
    j ← center + 1
    k ← 0
    b ← array temp size= right-left+1

    while i ≤ center and j ≤ right do
       if a[i] ≤ a[j] then
          b[k] ← a[i]
          i ← i + 1
       else
         b[k] ← a[j]
         j ← j + 1
       k ← k + 1
    end while

    while i ≤ center do
       b[k] ← a[i]
       i ← i + 1
       k ← k + 1
    end while

    while j ≤ right do
       b[k] ← a[j]
       j ← j + 1
       k ← k + 1
    end while

    for k ← left to right do
       a[k] ← b[k-left]
```

Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti: la sequenza viene divisa (*divide*) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda).

Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo (*impera*). Le due sottosequenze ordinate vengono fuse (*combina*). Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata.

Opera da un insieme A e lo divide in sotto insiemi (A1,A2) fino ad arrivare all'insieme contenente un solo elemento, per poi riunire le parti scomposte. (*approccio Top-down*).

L'algoritmo Merge Sort, per ordinare una sequenza di n oggetti, ha complessità temporale $T(n) = \theta(n \log(n))$ sia nel caso medio che nel caso pessimo. Infatti:

- la funzione *Merge* qui presentata ha complessità temporale $\theta(n)$
- *Mergesort* richiama sé stessa due volte, e ogni volta su (circa) metà della sequenza in input

Equazione di ricorrenza: $\mathrm{T(n)} = 2\mathrm{T}\left(\frac{n}{2}\right) + cn$

Insertion Sort: ***(confronta elemento i-esimo con precedente e scambia se i è più piccolo)***

```
function insertionSort (array a, int n)
      for i = 1 to n
        tmp ← a[i]
         for j = i-1 to 0 and a[j] > tmp
                a[j+1] ← a[j]
         a[j+1] ← tmp
```

L'algoritmo solitamente ordina la sequenza sul posto. Si assume che la sequenza da ordinare sia partizionata in una sottosequenza già ordinata, all'inizio composta da un solo elemento, e una ancora da ordinare. Alla k-esima iterazione, la sequenza già ordinata contiene k elementi.

In ogni iterazione, viene rimosso un elemento dalla sottosequenza non ordinata (scelto, in generale, arbitrariamente) e *inserito* (da cui il nome dell'algoritmo) nella posizione corretta della sottosequenza ordinata, estendendola così di un elemento.

Per fare questo, un'implementazione tipica dell'algoritmo utilizza *due indici:* uno punta all'elemento da ordinare e l'altro all'elemento immediatamente precedente. Se l'elemento puntato dal secondo indice è maggiore di quello a cui punta il primo indice, i due elementi vengono scambiati di posto; altrimenti il primo indice avanza.

Il procedimento è ripetuto finché si trova nel punto in cui il valore del primo indice deve essere inserito. Il primo indice punta inizialmente al secondo elemento dell'array, il secondo inizia dal primo. L'algoritmo così tende a spostare man mano gli elementi maggiori verso destra.

HeapSort:

Esempio di MaxHeap:

```
function HeapSort(A: array)
   BuildMaxHeap(A)
   for i=n downto 2
        do scambia A[1]<->A[i]
           heapsize[A]<->heapsize[A]-1
        MaxHeapify(A,1)

function BuildMaxHeap(A: array)
    n=A.lenght
    for i=n/2 downto 1
        do MaxHeapify(A,i)

function MaxHeapify(A,i)
    l <-left[i]
```

```
    r <-right[i]
    if(l<= A.lenght and A[l]>A[i])
        then massimo <-l
        else massimo <-i
    if(r<=A.lenght and A[r]>A[massimo])
        then massimo <-r
    if(massimo ≠ i)
        then scambia A[i]<->A[massimo]
               MaxHeapify(A,massimo)
```

Esempio MinHeap:

```
function HeapSort(A: array)
   BuildMinHeap(A)
   for i=n downto 2
        do scambia A[1]<->A[i]
           heapsize[A]<->heapsize[A]-1
        MinHeapify(A,1)

function BuildMinHeap(A: array)
    n=A.lenght
    for i=n/2 downto 1
        do MinHeapify(A,i)

function MinHeapify(A,i)
    l <-left[i]
    r <-right[i]
    if(l<= A.lenght and A[l]<A[i])
        then minimo <-l
        else minimo <-i
    if(r<=A.lenght and A[r]<A[massimo])
        then minimo <-r
    if(minimo ≠ i)
        then scambia A[i]<->A[minimo]
               MinHeapify(A,minimo)
```

L'algoritmo che ordina in senso *crescente* inizia creando uno heap decrescente. Per ogni iterazione si copia la radice (primo elemento dell'array) in fondo all'array stesso, eseguendo uno scambio di elementi. L'algoritmo poi ricostruisce uno heap di n-1 elementi spostando verso il basso la nuova radice, e ricomincia con un altro scambio (tra il primo elemento dell'array e quello in posizione n-1), eseguendo un ciclo che considera array di dimensione progressivamente decrescente.

A questo punto è molto semplice calcolare la complessità computazionale dell'algoritmo: O(nlogn).

QuickSort:

Vettore A, la funzione riceve i parametri p e q che rappresentano gli indici del sottovettore sul quale si opera la partizione

QUICKSORT NO IN PLACE

```
Algorithm partition(S, p)
Input sequence S, position p of pivot
Output subsequences L, E, G of the elements of S less than, equal to,or
greater than the pivot, resp.
L, E, G ← empty sequences
x ← S.remove(p)
E.insertLast(x)
while ¬S.isEmpty()
    y ← S.remove(S.first())
    if y < x
       L.insertLast(y)
    else if y = x
       E.insertLast(y)
    else { y > x }
       G.insertLast(y)
return L, E, G
```

Partizioniamo la sequenza di input come segue: rimuoviamo ogni elemento y da S e inseriamo y in L, E o G, sulla base del risultato del confronto con il pivot x.

Ogni inserimento e rimozione è all'inizio o alla fine della sequenza e quindi richiede tempo $O(1)$
Quindi, il passo di partizione del quick-sort richiede tempo $O(n)$

QUICKSORT IN PLACE

```
Algorithm inPlaceQuickSort(S, l, r)
Input sequence S, ranks l and r
Output sequence S with the elements of rank between l and r rearranged in
increasing order

if l ≥ r
   return
i ← a random integer between l and r
x ← S.elemAtRank(i)
(h, k) ← inPlacePartition(x)
inPlaceQuickSort(S, l, h - 1)
inPlaceQuickSort(S, k + 1, r)
```

Quick-sort può essere implementato sul posto,nella fase di partizione, usiamo operazioni di riordino degli elementi della sequenza di input in modo tale che gli elementi minori del pivot hanno rank minore di h, gli elementi uguali al pivot hanno rank tra h e k , gli elementi maggiori del pivot hanno rank maggiore di k

Le chiamate ricorsive considerano elementi con rank minori di h n elementi con rank maggiore di k

Analisi del costo qui possiamo dividerla nel caso peggiore,medio,migliore:

- Peggiore:
  - Scelgo pivot come massimo o minimo
  - La mia relazione di ricorrenza è $T(n) = T(n-1) + cn \in O(n^2)$
- Medio:
  - l numero medio di confronti tra elementi del vettore di ingresso eseguiti dall'algoritmo, che dipende dall'altezza dell'albero, infatti se ho un' altezza logaritmica il mio costo sarà $\theta(n\log(n)$ perché per ogni livello spendo al massimo n
- Migliore:
  - QuickSort è un algoritmo di ordinamento per confronto e non potrà ma avere meno di $\Omega(n\log(n)$ confronti

QuickSelect:

```
Algorithm partition(S, p)
Input sequence S, position p of pivot
Output subsequences L, E, G of the elements of S less than, equal to, or
greater than the pivot, resp.

L, E, G ← empty sequences
x ← S.remove(p)   E.insertLast(x)
while ¬S.isEmpty()
    y ← S.remove(S.first())
    if y < x    L.insertLast(y)
    else if y = x     E.insertLast(y)
    else { y > x }    G.insertLast(y)
return L, E, G
```

QuickSelect è un algoritmo randomizzato ricorsivo che trova il k-esimo elemento di un array disordinato di grandezza n eseguendo $O(n^2)$ confronti nel caso peggiore e $O(n)$ nel caso atteso. Si basa sulle tecniche del QuickSort.

Rimuoviamo ogni elemento y da S e Inseriamo y in L, E or G, sulla base del risultato del confronto con il pivot x. Ogni inserimento e rimozione è all'inizio o alla fine della sequenza, e quindi richiede tempo O(1) Quindi, la partizione di QuickSelect richiede tempo O(n)

© Se k < N1 dove N1 è la dimensione dell'array che contiene gli elementi più piccoli del pivot scelto
  - Cerco nell'insieme A1,k

- © Se k=N1 cerco A1,k
- © Sek>N1
  - Sto cercando il k-N1- esimo, vuol dire che k sfora la dimensione di A1 quindi non è più il k-esimo ma k'=k-N1

La relazione di ricorrenza $T(n) = T(\frac{3}{4}n) + 2n \quad \in \quad \theta(n)$

Insiemi:

UNIONE

```
Union(x, y )
Link(FindSet(x),FindSet(y))

Link(x,y)
if rank(x)>rank(y)
    then p[y] ← x
    else p[x] ← y
      if rank[x]==rank[y]
         then rank[y] ← rank[y]+1
```

Unisce gli insiemi A e B in un unico insieme, di nome A, e distrugge i vecchi insiemi A e B

FIND

```
FindSet (int x)  //p -> parent
if p[x]!=x
   then p[x] ← FindSet(p[x])
return p[x];
```

Restituisce il nome dell'insieme contenente l'elemento x

MAKESET

```
MakeSet(array parent, x)
p[x] ← x
rank(x) ← 0
```

Crea il nuovo insieme {x}

DFS:

```
Algorithm Driver(G)
Input grafo G
Output etichettatura degli spigoli di G come tree-edge e back-edge
for all u ∈ G.vertices()
```

```
    setLabel(u, UNEXPLORED)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
for all  v ∈ G.vertices()
    if  getLabel(v) = UNEXPLORED
        DFS(G, v)

DFS(G,v)
Input grafo G vertice iniziale v di G
Output etichettatura degli spigoli di G nella componente connessa di v
come tree-edge e back-edge
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
        else
            setLabel(e, BACK)
```

n numero di verici, m numero di archi

Una visita DFS di un grafo G visita tutti i vertici e gli spigoli di G e determina se G è connesso inoltre calcola le componenti connesse di G e calcola una foresta ricoprente di G.

Visita DFS permette di determinare numero di componenti connesse pari al numero di chiamate e inoltre ricostruisce la foresta ricoprente(Spanning forest).

Costo del Driver:

- primo ciclo ha costo $\theta(n)$ perché scandisco tutti i vertici
- secondo ciclo ha costo $\theta(m)$ perché scandisco tutti gli archi
- terzo ciclo scandisco di nuovo tutti i vertici
    - Minimo avrò $\Omega(n)$ se chiamo su nodo isolato
    - Max molte chiamate a DFS quindi $\theta(n+m)$

Costo del DFS:

- Primo ciclo
    - Minimo 0 [no nodi adiacenti]
    - Max deg(chiamatericorsive)
- Notiamo come è impossibile che un nodo ***non*** venga *visitato* e ovviamente ***non*** posso visitarlo 2 volte perché esso risulterà EXPLORED se *richiamato* quindi nodo verrà visitato **esattamente** una volta
- Siccome visito tutti i nodi ho componenti del costo pari alla somma dei gradi di tutti i nodi, ***2m***
- Il resto delle operazioni è tutto pari a valori costanti

- Costo totale: $\theta(n+m)$
- N.B
  - La dimensione dell'input è (n + m) quindi quest'algoritmo è lineare per la dim dell'input.

PercosoDFS:

```
Algorithm pathDFS(G, v, z)
setLabel(v, VISITED)
S.push(v)
if v = z
  return S.elements()
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            S.push(e)
            pathDFS(G, w, z)
            S.pop(e)
        else
          setLabel(e, BACK)
S.pop(v)
```

Usiamo una pila stack S per tener traccia del percorso fra il vertice iniziale e quello corrente. Non appena viene incontrato il vertice destinazione z, restituiamo il percorso contenuto nella pila.

RicercaCicliDFS

```
Algorithm cycleDFS(G, v)
setLabel(v, VISITED)
S.push(v)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
    w ← opposite(v,e)
    S.push(e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            cycleDFS(G, w)
            S.pop(e)
        else
            T ← new empty stack
            repeat
                 o ← S.pop()
                 T.push(o)
                 until o = w
```

```
                return T.elements()
S.pop(v)
```

Usiamo uno stack S per tener traccia del percorso fra il vertice iniziale e quello corrente. Appena si incontra un back-edge (v, w), restituiamo il ciclo presente nella porzione di stack che va dalla cima al vertice w.

Ordinamento topologico(basato su DFS)

```
Algorithm topologicalDriver(G)
Input DAG G
Output ordinamento topologico di G
n ← G.numVertices()
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
for all v ∈ G.vertices()
    if  getLabel(v) = UNEXPLORED
        topologicalDFS(G, v)

Algorithm topologicalDFS(G, v)
Input grafo G e vertice iniziale v di G
Output etichettatura dei vertici di G nella componente connessa di v
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            topologicalDFS(G, w)
        else
            { e è uno spigolo forward o cross } etichetta v con il numero n
n ← n - 1 // side effect!
```

Sequenza di abbandoni del nodo è una possibile implementazione per un algoritmo di ordinamento topologico, cioè primo che abbandono riceve come numero n, il secondo n-1 e cosi via, questo si realizza nel caso di appunto DFS diretta. Costo è sempre $O(n+m)$

Floyd-Warshall

```
Algorithm FloydWarshall(G)
Input digrafo G
Output la chiusura transitiva G* di G
```

```
i ← 1
for all v ∈ G.vertices()
    denota v come vi
    i ← i + 1
G_0 ← G
for k ← 1 to n do
    G_k ← G_{k−1}
    for i ← 1 to n (i ≠ k) do
        for j ← 1 to n (j ≠ i, k) do
            if G_{k−1}.areAdjacent(v_i, v_k) ∧ G_{k−1}.areAdjacent(v_k, v_j)
            if ¬G_k.areAdjacent(v_i, v_j)
                G_k.insertDirectedEdge(v_i, v_j, k)
return G_n
```

Numera i vertici 1,2,..,n. Considera percorsi che usano come vertici intermedi solo i vertici numerati da 1,2,..k dove k ha indici compresi tra i e j. Calcola una serie di digrafi $G_0, .., G_n$

$G_0 = G_n$. G_k ha uno spigolo orientato (v_i, v_j) se G ha un percorso orientato da v_i da v_j con vertici intermedi nell'insieme $\{v_i, .., v_k\}$

Per definizione G_n = G* Nella fase k, viene calcolato il digrafo G_k a partire da G_{k-1} Tempo di esecuzione $O(n^3)$ se sono adiacenti viene eseguito in $O(1)$ (ad esempio, usando una matrice di adiacenza)

BFS

```
Algorithm Driver(G)
Input graph G
Output labeling of the edges and partition of the vertices of G
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
       BFS(G, v)

Algorithm BFS(G, s)
L_0 ← new empty sequence
L_0.insertLast(s)
setLabel(s, VISITED)
i ← 0
while ¬L_i.isEmpty()
   L_{i+1} ← new empty sequence
   for all v ∈ Li.elements()
```

```
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
                w ← opposite(v,e)
                if getLabel(w) = UNEXPLORED
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    L_{i+1}.insertLast(w)
                else
                    setLabel(e, CROSS)
    i ← i +1
```

Una visita BFS di un grafo G,visita tutti i vertici e gli archi di G e determina se G è connesso infine calcola le componenti connesse di G e calcola una spanning forest di G. Costo è lo stesso della DFS $O(n+m)$

Dijkstra

```
Algorithm DijkstraDistances(G, s)
Q ← new heap-based priority queue
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
        {relax edge e}
        z ← G.opposite(u,e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z,r)
            Q.replaceKey(getLocator(z),r)
```

Coda di priorità che memorizza i vertici fuori della nuvola, chiave: distanza, elemento: vertice

1. Devo inizializzare n stime quindi $O(n)$

2. Creare min-heap e inserire gli elementi quindi $O(nlog(n))$

3. Ciclo che fa n operazioni, le n estrazioni dall' heap $O(nlog(n))$

4. Cambiamento stime, supponiamo che deve sempre aggiornare le stime, quindi ogni volta levo nodo e quindi costo aggiornamento stime sarà al massimo deg(nodo). Per tutti i nodi $O(m)$ ma aggiornamento di stima mi costa log volte up-heap di rimessa apposto, vuol dire $O(mlog(n))$

$$\text{Costo totale} = O(n) + O(nlog(n)) + O(nlog(n)) + O(mlog(n)) = O((m+n)log(n))$$

Caso grafo connesso n viene assorbito perché l'inserimento dell'heap non sarà n dato che è connesso perché io prima vedevo il caso peggiore cioè quello dove nessun nodo è connesso e quindi faccio n log n inserimenti, di conseguenza il costo è $O(mlog(n))$

Grafo pesato con Dijkstra è un *errore concettuale* perché costo è peggiore.

Non funziona per archi con peso negativo perché arco incidente con peso negativo potrebbe alterare le distanze dei vertici già nella nuvola, l'algoritmo di Bellam-Ford invece funziona per grafi con archi di peso negativo.

BellamFord

```
Algorithm BellmanFord(Grapg G, Vertex source)
for all v ∈V(G)
    dist[v]=infinite
    pred[v]=null
dist[source]=0
for i from 1 to |V(G)|-1
    for all (u,v) ∈E(G)
        w=weight(u,v)
        if(dist[u]+w<dist[v])
            dist[v]=dist[u]+w
            pred[v]=u
for all (u,v)∈E(G)
    w=weight(u,v)
    if(dist[u]+w<dist[v])
        error "Graph contains negative-weight cycle"
return dist[], pred[]
```

Costo $O(n * m)$

ASPS – Floyd Warshall

```
Algorithm FloydWarshall(Graph G)
dist = new n×n matrix of distances
   // (all infinite)
next = new n×n matrix of vertex indices
   // (all zeros)
for all (u,v)∈E(G)
    dist[u][v]=w(u,v)
    next[u][v]=v
```

```
for k from 1 to n
    for i from 1 to n
        for j from 1 to n
            if(dist[i][j]>dist[i][k]+dist[k][j])
                dist[i][j]=dist[i][k]+dist[k][j]
                next[i][j]=next[i][k]

Algorithm Path(Vertex u, Vertex v)
if(next[u][v]==0) return []
   path=[u]
   while(u!=v)
        u=next[u][v]
        path.append(u)
return path
```

Restituisce 2 matrici

- Dist
  - All'inizio ha distanze tutte ∞
  - Contiene le distanze fra le coppie dei vertici
- Next
  - All'inizio per tutti gli archi ha distanze zero
  - Contiene nella cella [i][j] indice del nodo successore di nello SP da i a j

Sfrutta fatto che un sottopercorso di un percorso ottimale è ancora ottimale. Algoritmo ha 3 cicli tutti che vanno da 1 a n e in quello più interno fa confronto con la distanza di dist[i][j] con dist[i][k]+dist[k][j] cioè vede se conviene passare per k. Nel caso fosse vero modifica dist e next

Algoritmo Path ricostruisce dalla matrice Next lo SP da u a v

Costo pari a $O(n^3)$ dato dai 3 cicli for

Teorema: La complessità temporale di un qualsiasi algoritmo di ordinamento per confronto è pari a $\Omega(n\log(n))$, dove *n* è il numero di elementi da ordinare.

Dim. (2 modi per dimostrarlo)

1)

Si vuole dimostrare che in un algoritmo **confronti e scambi** la complessità è $\Omega(n\log(n))$. Data in input una sequenza $n_1, n_2, n_3 \dots n_n$ di n elementi, l'azione dell'algoritmo si può rappresentare come un albero binario, per ogni sequenza di ingresso ci sarà un cammino all'interno dell'albero, questo perché si ha una permutazione delle sequenze, infatti il numero di permutazioni possibili su n elementi sono ***n!*** combinazioni che corrispondono al numero di foglie dell'albero.

Date due permutazioni distinte esse identificano diversi cammini all'interno dell'albero.

n! è il numero di foglie nell'albero di decisione dove n è il numero di elementi da ordinare.

Date ***n!*** foglie ed essendo l'albero binario l'altezza dell'albero sarà:

$\mathrm{h(albero)} \leq \log_2 n$

L'altezza dell'albero corrisponde al numero di confronti, elemento indicativo del tempo di esecuzione dell'algoritmo. Nel caso peggiore, ossia quando si arriva al fondo dell'albero(foglia), si avrà una complessità pari a

$\log_2 n!$

Per ultimare la dimostrazione si utilizza la formula di Stirling sull'approssimazione di $n!$

$$\text{numero di confronti } \geq (\frac{1}{2}\log_2 2\pi + \frac{1}{2}\log_2 \mathrm{n} + \mathrm{nlog_2 n} - \mathrm{nlog_2 e}) \sim \mathrm{nlog_2 n}$$

che a livello asintotico corrisponde a Ω(nlog(n)).

2)

Sapendo che albero di decisione rappresenta i confronti eseguiti da un algoritmo su un dato input e che ogni foglia corrisponde ad una delle possibili permutazioni implica che ci sono ***n!*** permutazioni e quindi l'albero deve contenere ***n!*** foglie.

Inoltre sappiamo che l'altezza di un albero di decisione è il limite inferiore su tempo di esecuzione e poiché vi sono ***n!*** foglie,l'altezza sarà almeno $\log_2 n!$

$$\log_2 n! \geq log(\frac{n}{2})^{\frac{n}{2}} = (\frac{n}{2}) log(\frac{n}{2})) \sim (\mathrm{nlogn})$$

Ogni algoritmo di ordinamento basato su confronti deve eseguire in tempo Ω(nlog(n)).