**Rappresentazione di grafi:**
**Si discuta quali rappresentazioni di grafo sono convenienti per applicazioni che devono usare *poco spazio*, e si motivi la risposta.**

Per rappresentare grafi usando poco spazio è necessario usare una *lista di adiacenza*, ad ogni nodo i del grafo è associata una lista dei nodi adiacenti, verrà occupato *spazio  O(n+m)*
Non conviene usare una matrice di adiacenza perchè lo spazio che occupa è maggiore, ovvero O(n^2)

**Rappresentazione di grafi:**
**Si discuta quale rappresentazione di grafo conviene usare per un'applicazione nella quale si deve spesso determinare quali siano tutti gli archi entranti in un nodo dato in un grafo orientato e si motivi la risposta**

L'implementazione migliore per tale richiesta è quella di una ***lista di adiacenza*** basata sugli archi entranti in un nodo. Tale rappresentazione è migliore perché la complessità di tale operazione è lineare sul grado entrante del nodo, se si usasse una matrice sarebbe lineare sul numero di nodi presenti nel grafo. Questo discorso vale indipendentemente dal fatto che il grafo sia denso o sparso, ma se fosse denso la complessità computazionale si avvicinerebbe a quella della matrice di adiacenza.

**Visite di grafi:**
**Si dica come si determina se un grafo non orientato è connesso, e su cosa si basa la dimostrazione di correttezza.**

Per determinare che un grafo è connesso, è necessario che l'algoritmo di visita raggiunga tutti i nodi del grafo partendo da un nodo, la sorgente.

L'algoritmo di visita, svolgendo le sue operazioni, crea un albero quindi non deve contenere cicli e deve essere connesso.

Pseudocodice;
```
{
        S={sorgente}
        finchè è possibile scelgo un arco (u,v) tale che u appartiene a S, v non appartiene aS
        S=S U {v}
        E' = E' U {(u,v)}
}
```

A ogni iterazione all'insieme S viene aggiunto un nodo, al massimo fino a quando S è uguale a V cioè l'ordine del grafo.
Oppure si ferma prima se il grafo non è connesso.
Se G(grafo) è connesso allora tutti i nodi sono stati visitati.

**Base:**
S=1 → (S,E') è un albero  Per E'={} è verificato

**Ipotesi Induttiva:**
S=K → (S di K, E' di K) è un albero // S sono i nodi scoperti = K ed E' sono gli archi utilizzati per visitare K nodi

**Passo Induttivo:**
S=K+1 → (S di K+1, E' di K+1) è un albero

Nello psudocodice S(K+1) corrisponde a S(k) U {v}

**Visite di grafi:**
**Per determinare se un grafo NON orientato ha un ciclo si deve usare una vista DFS o una BFS? Si motivi la risposta**

IN UN grafo non orientato, per scoprire se si ha un ciclo, è possibile usare tutte e due le tipologie di ricerca BFS e DFS, infatti tutte e due ogni volta che "scoprono" un nodo lo segnano come "scoperto, quindi durante la ricerca se scopro un nodo (escludendo il padre ad ogni iterazione) che è già visitato hanno trovato un ciclo! Stesso discorso non vale se il grafo è orientato in questo particolare caso, si può solo usare una DFS.

==========

Per determinare se un grafo NON orientato ha un ciclo o meno si possono utilizzare entrambe le visite, dato che basta tenere traccia dei padri dei nodi. In un grafo NON orientato esiste un ciclo se durante la visita di un nodo v se ne scopre un altro già visitato che non sia suo padre.

**Visita di grafi:**
**Si consideri un grafo non pesato. Ragionando sulla definizione di distanza, dire se è vero che la distanza di un nodo v da un nodo s è sempre minore o uguale della lunghezza del cammino da s a v in un albero di visita la cui radice è s. Spiegare il perché della proprio affermazione.**

**LEMMA:**

Dg(s,v) <= liva(v)
Il cammino minimo (distanza) nel grafo può essere minore o uguale al livello del nodo all'interno di un qualsiasi albero di visita

Questa affermazione è corretta: la distanza tra i nodi v e s è minore o uguale alla lunghezza del cammino da s a v in un albero di visita la cui radice è s. Questo perché nel caso di visita BFS la distanza coincide con il cammino dato che la BFS, strutturando l'albero a livelli, ci permette di ritrovare il cammino più breve mentre nel caso della DFS non c'è certezza che la distanza ritornata sia la minore dato che svolge la sua visita in profondità.

Nel caso di un grafo non orientato connesso dove vi sono 4 nodi così connessi(4;0 1;0 3;1 2;3 2): si cercasse la distanza tra 0 e 1 la ricerca BFS ritornerebbe la distanza = cammino analizzando prima i nodi 1 e 3 e successivamente il nodo 2 mentre la DFS ritornerebbe una lunghezza del cammino maggiore uguale dato che vi è la possibilità che segua la visita in questo modo 0-3-2-1

**Visita di grafi:**
**Cosa hanno in comune tutti gli algoritmi di visita di grafi? Cosa garantisce la terminazione di un algoritmo di visita? Si discuta il costo della visita generica**

Tutti gli algoritmi di visita hanno in comune che considerano solamente gli archi che hanno almeno un nodo che non è ancora stato scoperto, perchè nel caso in cui si consideri l'arco u,v e u appartiene ai nodi scoperti a anche v appartiene ai nodi scoperti, non viene preso in considerazione perchè nel caso si formerebbe un ciclo. Quindi tutti gli algoritmi di visita hanno in comune che devono raggiungere tutti i nodi raggiungibili dalla sorgente.
L'algoritmo termina quando tutti i nodi raggiungibili dal punto iniziale, sorgente, sono stati raggiunti e scoperti, quindi non è possibile visitare nessun altro arco perché nel caso si formerebbe un ciclo.

Pseudocodice:

```
{
        S={sorgente}
        finché è possibile scelgo arco(u,v) dove u appartiene a S e v non appartiene a S
        S = S U {v}
        E' = E U {v}
}
```

Il costo della visita generica è O(n+m) perché visito i nodi e gli archi, ed ho al massimo n nodi ed m archi.

**BFS:**
**Si consideri l'albero di visita ottenuto con una visita di ampiezza di un grafo orientato. Si dica quali tipi di archi aggiuntivi (non presenti nell'albero di visita) possono essere presenti nel grafo e quali non possono esserci( ad esempio archi trasversali, archi all'indietro, ecc... ) Si motivi la risposta**
Effettuando su un grafo orientato una visita BFS, gli archi che NON risultano possibili nell'albero di visita sono gli archi in avanti (e trasversali in avanti). Questo per il fatto che la

strategia usata dall'algoritmo di visita in ampiezza prende in considerazione prima tutti i nodi vicini ad un nodo dato.

**BFS:**
**Si consideri l'albero di visita ottenuto con una visita di ampiezza di un grafo NON orientato. Si dica quali tipi di archi aggiuntivi (non presenti nell'albero di visita) possono essere presenti nel grafo e quali non possono esserci( ad esempio archi trasversali, archi all'indietro, ecc... ) Si motivi la risposta**

Una visita BFS su un grafo NON orientato genera un albero BFS che NON può avere archi in avanti o all'indietro, mentre i trasversali sono possibili. Questo è dovuto alla strategia della visita in ampiezza, che operando a "livelli" visita man mano i vicini del nodo preso in considerazione, perciò non sono possibili questi tipi di archi perché i nodi sarebbero stati scoperti prima.

**BFS:**
**Nella dimostrazione del fatto che "Un nodo è al livello k dell'albero di visita BFS se e solo se ha distanza k dalla sorgente" qual'è l'ipotesi induttiva e come la si usa?**

**Ipotesi induttiva:**
Dg(s,v) = h  quindi LivA(v) = h

**Tesi Induttiva:**
Dg(s,v) = h+1 quindi LivA(v)=h+1

**Supponiamo che Dg(s,v) = h+1**
Per il teorema della sottostruttura ottima se Dg(s,u) = h allora Dg(s,v)=h+1 perchè v è figlio di u
Se Dg(s,u) = h allora u è a livello h nell'albero di visita

**Utilizziamo il lemma e Supponiamo che Dg(s,v) <LIVa(v)**

Se il livello del nodo v fosse > della distanza del nodo v sul grafo, allora vuol dire che v è figlio di un altro nodo w che si trova a livello > h ( dove h era il livello di u dato dall'ipotesi induttiva)

Questa supposizione viene esclusa per il fatto che l'algoritmo fa uscire dalla coda prima u che w dato che si trova a un livello inferiore di w.
Quindi v uscirà dopo u, visto che è suo figlio.
Quindi abbiamo dimostrato che Dg(s,v) = LIVa(v)

**BFS:**

**Nella dimostrazione del fatto che “un nodo è al livello k dell’albero di visita BFS se e solo se ha distanza k dalla sorgente” che ruolo ha il fatto che la visita adopera una coda?**

Nella visita BFS, si visitano ciclicamente i nodi ad un certo livello dalla sorgente(per questo il concetto di livello è molto importante) ,perché prima vengono visitati i nodi di livello 1, poi di livello 2 e così via. Ora qui introduco il concetto di frontiera(che possiamo vederla come una frontiera immaginaria che viene posta tra un nodo visitato e i suoi vicini),e da un nodo si passerà a visitare quello successivo solo quando saranno stati "scoperti" tutti i nodi che superano la frontiera, per questo il ruolo della CODA è molto importante, perché verranno messi ad ogni visita di un nodo tutti i nodi vicini che superano la frontiera in coda, in questo modo si avrà la certezza di visitare appunto i nodi corretti che superano la frontiera e solo quelli!

**BFS e DFS:**

**Per determinare se un grafo NON orientato ha un ciclo si deve usare una visita DFS o BFS? Si motivi la risposta**

La visita DFS risulta essere più adatta per determinare se un grafo non orientato ha un ciclo perché questo algoritmo di visita raggiunge rapidamente i vertici lontani dal vertice di partenza.

Anche la BFS può determinare se un grafo non orientato ha un ciclo, ma visto che esamina in ordine crescente di distanza dal nodo di partenza S, è più adatto a risolvere i problemi per decidere il cammino minimo da un nodo a un’altro.

Sono entrambi accettabili perché entrambi gli algoritmi di visita, ogni volta che devono prendere in considerazione un nuovo arco, lo prendono solo nel caso in cui uno dei due nodi non è stato scoperto, così che non possono formare cicli.
Visto che si parla di grafi non orientati, ogni volta che si prende in considerazione un arco si avrà un ciclo, ovvero se prendo l’arco (u,v) si avrà il ciclo u,v,u quindi si deve tenere in memoria chi è il nodo padre di ciascun nodo.

==========

Per determinare se un grafo NON orientato ha un ciclo o meno si possono utilizzare entrambe le visite, dato che basta tenere traccia dei padri dei nodi. In un grafo non orientato esiste un ciclo se durante la visita di un nodo v se ne scopre un altro già visitato che non sia suo padre.

**DFS:**

**Si consideri l'albero di visita ottenuto con una visita in profondità di un grafo NON orientato. Si dica quali tipi di archi aggiuntivi ( non presenti nell'albero di visita ) possono essere presenti nel grafo e quali non possono esserci ( ad esempio: archi trasversali, archi all'indietro, ecc..) Si motivi la risposta.**

Una visita DFS su un grafo NON orientato genera un albero DFS che NON può avere archi trasversali , mentre gli archi in avanti o all'indietro sono possibili. Questo è dovuto alla strategia della visita in profondità,che operando in profondità non sono possibili archi traversali perché si andrebbe a visitare dei nodi che sono già stati scoperti.

**Ordine topologico:**
**Si spieghi dove e perché fallisce la dimostrazione di correttezza dell'algoritmo per l'ordine topologico se il grafo ha cicli**

Con ordine topologico si sta parlando di DAG ( Directed Acyclic Graph) ovvero di grafi orientato aciclico.La dimostrazione di correttezza di questo algoritmo ha lo scopo di dimostrare che per ogni arco u,v del grafo ot[u]<ot[v]

Se è presente un ciclo la dimostrazione di correttezza fallisce in due punti:

- La visita di un nodo non termina dopo la visita dei suoi discendenti, nel caso in cui è presente un arco all'indietro che collega un nodo ad un suo antenato, la visita dell'antenato finisce prima di quella del suo successore, di conseguenza ot[u]<ot[v] non viene rispettata.
- di conseguenza il primo nodo a terminare avrà degli archi uscenti

---

La dimostrazione di correttezza di questo algoritmo ha lo scopo di dimostrare che per ogni arco u,v del grafo eta[u]<eta[v]. La dimostrazione fallisce perchè nei cicli la visita di un nodo termina prima di quella di un suo discendente, perche' il discendente è anche antenato per colpa del ciclo e di conseguenza eta[u]<eta[v] non viene rispettata.
Commento: questa spiegazione è corretta

**Ordine Topologico:**

**Si introduca la nozione di ordine topologico, e si spiega nel modo più preciso possibile, quali grafi hanno un ordine topologico, quali non lo hanno e perché.**

Quando parliamo di ordine topologico ci riferiamo ai DAG, grafi orientati aciclici . Se un grafo ha un ciclo non può avere un ordine topologico, inoltre un ordine topologico gli archi vanno tutti da sinistra verso destra, per questo il grafo dev'essere orientato. Dato un DAG G(u,v) possiamo dire che il suo ordine topologico è una numerazione dei suoi vertici num: V-> {0,1,...,n-1} tale che per ogni arco (u,v) appartenente a E vale num(u)<num(v). Graficamente se disponiamo i nodi lungo una riga orizzontale in base alla loro numerazione num e tracciamo gli archi tra di loro, risultano tutti orientati da sinistra verso destra. Utilizzando la DFS nella visita, il primo nodo che termina ha ordine topologico n-1 ed è corretto perchè non ha discendenti, se fosse presente un ciclo non sarebbe verificata questa cosa. inoltre i grafi devo essere orientati perchè altrimenti non si riuscirebbe a darne un ordine.

**SCC:**
**Si discuta se, nell'algoritmo di Kosaraju per determinare le componenti fortemente connesse, le due visite possono essere visite qualunque e perchè. (Chi ha studiato l'algoritmo basato sulla contrazione di cicli risponda invece a questa domanda: si illustri ad alto livello la dimostrazione di correttezza e se ne sottolineino gli aspetti fondamentali)**

Per determinare le SCC, bisogna applicare due visite per svolgere l'algoritmo, la prima volta (al primo passaggio), deve essere per forza una DFS, perché devo trovare l'ordine di fine visita, e devo essere sicuro che quando un padre termina tutti i figli sono stati visitati, e per questo uso la DFS. Mentre la seconda volta il tipo di visita non ha molta importanza, è una visita qualunque (potrebbe anche usare un ordine ne' BFS ne' DFS, basta che sia una visita e raggiunga tutti i nodi raggiungibili e non ancora visitati.

**SCC:**
**Si enunci e si dimostri il lemma sull'ordine di fine visita di nodi in due componenti connesse C1 e C2 tali che esiste un arco del grafo da C1 a C2, e si dica che funzione ha questo lemma nella dimostrazione di correttezza dell'algoritmo di Kosaraju.**
**( Chi ha studiato l'algoritmo basato sulla contrazione di cicli risponda invece a questa domanda: si dimostri quando termina la visita del primo nodo visitato di una componente fortemente connessa con un arco uscente rispetto ai nodi della componente fortemente connessa in cui entra l'arco)**

**Lemma:**
Se consideriamo 2 componenti fortemente connesse e avviene una visita DFS completa su queste 2 componenti possiamo dire con certezza che il nodo che termina per ultimo appartiene alla componente connessa che ha un arco uscente verso la seconda

Questo lemma in kosaraju viene applicato durante la prima visita che dev'essere per forza una DFS perché è necessario determinare l'ordine di fine visita per poi calcolare il grafo

trasposto e applicare la seconda visita che invece non ha importanza se è una BFS o DFS perché si basa sull'ordine di fine visita ottenuto nella prima visita.

**SCC:**
**Si dica cos'è il grafo delle componenti fortemente connesse di un grafo orientato e che ruolo gioca nell'algoritmo di Kosaraju ( Chi ha studiato l'algoritmo basato sulla contrazione di cicli risponda invece a questa domanda: si dimostri perché il fatto che la visita è una DFS implica che quando termina la visita del rappresentante di una classe esso è sulla cime della pila dei rappresentati e invece tutti i nodi della componente sono sulla pila dei parziali sopra al rappresentante)**

Il grafo delle componenti fortemente connesse di un grafo orientato è un DAG ovvero un grafo orientato aciclico, perchè nel caso in cui contenesse un ciclo non si potrebbe determinare l'ordine di fine visita.
La prima visita che dev'essere applicata in kosaraju è una DFS per determinare appunto l'ordine di fine visita  quindi non può essere presente un ciclo e deve essere direzionato altrimenti non potrebbero essere considerate componenti connesse distinte

**SCC:**
**Si spieghi se, nell'algoritmo di Kosaraju per determinare le componenti fortementi connesse, ciascuna delle due visite può essere indifferentemente in ampiezza o in profondità. ( Chi ha studiato l'algoritmo basato sulla contrazione di cicli risponda invece a questa domanda: si dica se si possono utilizzare indifferentemente una visita in ampiezza o una visita in profondità per realizzare l'algoritmo e si spieghi il perchè potete ragionare sul caso base in cui le componenti fortemente connesse sono due e una delle due non ha archi uscenti.)**

Kosaraju per trovare SCC deve inizialmente trovare l'ordine di fine visita (l'ordine in cui terminano i nodi) utilizzando la visita DFS ( non funziona con la BFS), dopo calcola il grafo trasposto e infine esegue una seconda visita sul grafo trasposto utilizzando l'ordine di fine visita (la seconda visita è indifferente perché segue l'ordine di fine visita)

**Bene**

Un grafo delle componenti fortemente connesse indica che raggruppo i nodi tra loro in modo da avere un DAG(ovvero),in questo modo ottengono nuovo grafo(appunto un dag).
Mi serve questo nell'algoritmo di kosaraju perché poi nello svolgimento invertirò tutti gli archi per procedere diciamo che è un passo dell'algoritmo, e poi partirò dal primo nodo che è diventato l'ultimo a ritroso.
La dfs viene usata perché un nodo termina solo quando tutti i suoi figli sono terminati quindi il padre del dfs (sorgente)terminerà per ultimo una volta appunto che tutti i suoi figli saranno terminati, questo ci da la certezza che una volta che il nodo padre termina tutti i figli sono stati visitati ed è quello che ci serve.

**Commento: va bene**

**SCC**

**Si dica cos'è una componente fortemente connessa di un grafo orientato, cos'è il grafo delle componenti fortemente connesse, e si spieghi com'è fatto ( motivando la risposta). (Chi ha studiato l'algoritmo basato sulla contrazione di cicli risponda invece a questa domanda: si dimostra perché il fatto che la visita è una DFS implica che quando termina la visita del rappresentante di una classe esso è sulla cima della pila dei rappresentanti e invece tutti i nodi della componente sono sulla pila dei parziali sopra al rappresentante)**

Una componente fortemente connessa è un sottoinsieme massimale di vertici V' (dato un grafo G =(V,E) ) tale che presi due nodi "u" e "v" all'interno di questo sottoinsieme esiste un cammino da "u" a "v" e viceversa. Il grafo delle SCC è un grafo i cui nodi sono le SCC stesse, e gli archi, gli archi che nel grafo originale collegavano le SCC. Questo SCC graph non deve contenere cicli, in quanto la presenza di un ciclo comporterebbe la creazione di una SCC contenente più nodi, e quindi le due SCC precedenti non sarebbero massimali.

# *Secondo Esonero*

**Algoritmo di Dijkstra:**
**Si enunci il passo induttivo della dimostrazione di correttezza dell'algoritmo di Dijkstra e illustri in che modo si utilizza nella dimostrazione la scelta greedy,**

**Ipotesi Induttiva:**
S=K
Sono stati visitati k nodi e vale che per ogni v appartenente a S dist[v] = gamma(sorg,v)

**Tesi Induttiva:**
S=K+1
Sono stati visitati k+1 nodi e vale che per ogni v appartenente a S dist[v]= gamma(sorg,v)

E' stato scelto l'arco(u,v) da Dijkstra perchè viene attraversata la frontiera  e prende l'arco di peso minimo dalla sorgente.

Qualsiasi altro cammino che parte da sorg e arriva a v è sicuramente peggiore di quello preso da Dijkstra

Sappiamo che
dist[x]+peso(x,y) >= dist[u]+peso(u,v)

Supponiamo di avere 2 percorsi:

C=lungh(sorg,u)+peso(u,v)
C'=lungh(sorg,x)+peso(x,y)+lungh(y,v)

Possiamo dire che:
C' >= lungh(sorg,x)+peso(x,y) (perché i pesi sono tutti >= 0)  >= gamma(sorg,x)+peso(x,y) utilizziamo l'ipotesi induttiva e diciamo che è uguale a dist[x]+peso(x,y) il quale è >= dist[u]+peso(u,v) = C questo perchè l'algoritmo è DIjkstra e sapendo che la distanza fino ad u è calcolata correttamente e dato che è un algoritmo Greedy C è migliore di C'

**Caso base**: S={sorg} |S|=1 (cardinalità nodi visitati)
dist{sorg}=$\wp$(sorg,sorg)
La distanza della sorgente nell'albero di visita è uguale alla distanza nel grafo.
Infatti l'algoritmo la pone=0;
**Ip Induttiva:**|S|=k
Sono stati visitati k nodi e vale che:  per ogni v $\in$ S dist[v]=$\wp$(sorg,v).
**Tesi:**|S|=k+1: per ogni v $\in$ S  dist[v]=$\wp$(sorg,v).
Viene visitato il nodo v perchè viene attraversato un arco che attraversa la frontiera.

E’ stato scelto quell’arco perchè era quello con peso minimo tra gli altri archi connessi alla sorgente.
Consideriamo qualsiasi altro cammino qualunque che arriva a v.
Questo cammino passa dai nodi scoperti e oltrepassa la frontiera.
Qualsiasi cammino io prenda avrà distanza >= di quello preso dall’algoritmo:
Dist[x]+peso(x,y)>= dist[u]+peso(u,v)

Supponendo 2 percorsi:
c=lungh(sorg,u)+peso(u,v)
c'=lungh(sorg,x)+peso(x,y)+lungh(y,v)
Abbiamo che il percorso
lungh(c')=lungh(sorg,x)+peso(x,y)+lungh(y,v)>=lungh(sorg,x)+peso(x,y)
essendo una minoranza (grazie al fatto che i pesi sono tutti positivi) siccome la distanza dalla sorgente a x nel grafo è il cammino minimo sarà >=δ(sorg,x)+peso(x,y) il quale per ipotesi induttiva è =dist[x]+peso(x,y) il che è >=dist[u]+peso(u,v)=lungh(c) perchè l'algoritmo di dijkstra usa una tecnica greedy, sapendo infatti che la distanza[u] per ipotesi induttiva è calcolata correttamente allora il percorso c risulta essere migliore del percorso c'.

**Algoritmo di Dijkstra:**
**Si dica quale vincolo sul grafo è fondamentale per la correttezza dell’algoritmo di Dijkstra e si dica se l’algoritmo calcola sempre una soluzione sbagliata, nel caso in cui tale vincolo non è rispettato**

Il vincolo principale è che i pesi devono essere tutti >=0 perché nel caso in cui fossero negativi l’algoritmo fallisce

Supponendo di avere un qualunque cammino minimo che arriva a v dalla sorgente
C = sorg → (u,v) → v
lunghezza(C) = lunghezza(sorg→ u) + peso(u→ v)
e un secondo cammino che arriva sempre a v ma con un percorso diverso
C’ = sorg→ (x,y) → v
lunghezza(C’) = lunghezza(sorg→ x) + peso(x,y) + lunghezza(y,v)

Si verifica che il percorso di C >= C’ implica che tutti i pesi devono essere positivi.
Nel caso siano presenti dei pesi negativi la lunghezza di C’ potrebbe essere < della lunghezza di C.

**Algoritmo di Dijkstra:**
**Si dica come si confronta la soluzione ottenuta con Dijkstra per il problema dei cammini minimi da una sorgente con soluzioni ottenute in altro modo, nella dimostrazione di correttezza per l’algoritmo.**

Supponiamo di avere 2 percorsi, C è il percorso preso da Dijkstra per arrivare a v e C' e un qualsiasi altro percorso per arrivare a v che NON è stato scelto da Dijkstra:

C=lungh(sorg,u)+peso(u,v)
C'=lungh(sorg,x)+peso(x,y)+lungh(y,v)

Possiamo dire che:
C' >= lungh(sorg,x)+peso(x,y) (perché i pesi sono tutti >= 0)  >= gamma(sorg,x)+peso(x,y) utilizziamo l'ipotesi induttiva e diciamo che è uguale a dist[x]+peso(x,y) il quale è >= dist[u]+peso(u,v) = C questo perchè l'algoritmo è DIjkstra e sapendo che la distanza fino ad u è calcolata correttamente e dato che è un algoritmo Greedy C è migliore di C'

**Sequenziamento dei processi:**
**Si spieghi come si confronta la soluzione greedy per il problema del sequenziamento dei processi con un'altra qualsiasi soluzione ( nella dimostrazione di correttezza)**

Si hanno n processi, con lunghezza positive
I processi vengono eseguiti sequenzialmente uno dietro l'altro uno alla volta e l'obiettivo è minimizzare il tempo di attesa medio che è dato da Am = somma tempo attesa dei processi/ numero di processi.

Se utilizziamo un algoritmo greedy, sceglie come prossimo processo da eseguire quello con la lunghezza minima non ancora eseguita, quindi tutti i processi verranno disposti in ordine dal più piccolo al più grande.

Per vedere se questa soluzione è ottimale prendiamo una soluzione diversa da quella greedy e poi confrontiamo il tempo di attesa medio e si nota che il tempo di attesa medio della soluzione greedy è migliore.

Per migliorare la soluzione non greedy possiamo effettuare uno scambio ( algoritmo dello scambio) dei processi, prendendone uno di peso maggiore e  scambiandolo di posizione con uno di peso minore, in modo che quello di peso minore sia posizionato prima di quello di peso maggiore.

Confrontano il tempo di atteso medio con quello della soluzione non greedy precedente e con quella attuale, notiamo che quello attuale è migliore, quindi si può concludere che ogni soluzione non greedy è migliorabile.

**Teorema del taglio:**

**In base al teorema del taglio un arco che non è l'arco di peso minimo che attraversa un taglio del grafo può appartenere al minimo albero ricoprente del grafo ?**
**Giustificare la proprio risposta.**
**(Ovviamente, usando la nomenclatura del libro di Crescenzi: in base alla condizione di taglio, un arco che non è l'arco di peso minimo di un taglio può appartenere al minimo albero ricoprente del grafo ? Giustificare la propria risposta)**

L'arco di peso minimo appartiene sempre al MST

Per assurdo supponiamo che in un MST non sia presente l'arco di peso minimo, **e.**
Quindi aggiungiamo l'arco **e** al MST che abbiamo ora, ma facendo ciò si forma un ciclo poiché essendo un MST, tutti i nodi sono stati scoperti e visitati quindi aggiungendo un arco si forma un ciclo (**c)**.
Per eliminare **c** togliamo l'arco **e'** che attraversa lo stesso taglio di **e** ma ha un peso maggiore, **peso(e)<peso(e')**
Quindi ora che non c'è più il ciclo abbiamo un nuovo MST il cui peso totale degli archi sarà minore di quello iniziale **peso(T)>peso(T')** perché **P(T') = P(T) - P(e') + P(e)**
Quindi l'arco di peso minimo deve appartenere al MST.

Esiste un solo MST a meno che ci siano più archi abbiano pesi uguali.

**Prim:**
**Cosa bisogna dimostrare per dimostrare la correttezza dell'algoritmo di Prim, e su cosa si basano le varie parti della dimostrazione?**

**Enunciato:**
L'algoritmo di Prim calcola correttamente il minimo albero ricoprente di un grafo non orientato e connesso
**Ipotesi:**
Le ipotesi sono: il grafo è non orientato e connesso, l'algoritmo è una visita di un grafo in cui la scelta del nuovo nodo da visitare è greedy e minimizza il peso del nuovo arco scelto
**Tesi:**
La tesi è che questo algoritmo calcola il minimo albero ricoprente del grafo ( ovvero che costruisce un albero, che tale albero ricopre il grafo, e che l'albero è di peso minimo tra tutti gli albero ricoprenti)

**Prim:**
**Dire come si dimostra l'albero costruito dall'algoritmo di Prim è un albero ricoprente del grafo ( e in quali ipotesi questo è vero)**

Prim genera, un minimo albero ricoprente o MST (minimum spanning tree) e ciò è dimostrato dal teorema del taglio che divide il grafo in due sottoinsiemi e di questi va a prendere l'arco minimo che attraversa il taglio. Questo sicuramente apparterrà al MST

E’ ricoprente poiché essendo un algoritmo di visita di un grafo connesso, vengono visitati tutti i nodi del grafo

E’ un albero poichè è aciclico e connesso → E’ connesso poiché vi è un cammino da una radice a tutti gli altri nodi e anche perchè ad ogni iterazione viene aggiunto un arco tra un nodo e visitato e un nodo non ancora visitato. Non ha cicli poiché ogni nodo ha uno e un solo padre.

**Unicità del minimo albero ricoprente:**
**Se un grafo ha più archi aventi lo stesso peso, è unico il minimo albero ricoprente ? Si discuta brevemente la propria risposta**

Se un grafo ha più archi aventi lo stesso peso, può essere che l’MST sia unico ma è anche possibile che esistano più MST, dipende dal peso di questi archi.
Se essi sono tutti l’arco di peso minimo che attraversano più tagli, allora potranno essere più MST ma nel caso in cui non siano l’arco di peso minimo che attraversa il taglio, l’MST sarà unico perchè quegli archi non vengono presi in considerazione visto che non sono l’arco di peso minimo che attraversa il taglio, questo avviene secondo il teorema del taglio.

**Unicità del minimo albero ricoprente:**
**Si dica qual è l’idea alla base della dimostrazione di unicità del minimo albero ricoprente**

**Teorema:** Sia G un grafo pesato non orientato. Se non ci sono due archi di G che hanno lo stesso peso, G ha un unico albero ricoprente minimo

**Dimostrazione:** Per assurdo supponiamo di avere due MST dello stesso grafo, M1 ed M2 e sono distinti, quindi esiste almeno un arco che appartiene ad uno di essi e non all’altro.

Sia **e** l’arco di peso minimo che appartiene a M2 e non a M1.
Quindi aggiungiamo **e** ad M1 ma facendo ciò si viene a formare un ciclo **c,** perché M1 è un albero ricoprente quindi è connesso e contiene tutti i nodi del grafo.

Per togliere il ciclo **c** è necessario togliere l’arco **e’** che ha peso maggiore di **e.**
Quindi togliendo **e’** da M1 otterremo M1’, ovvero diverso da quello iniziale

Si tratta di un albero ricoprente di G perché il ciclo non esiste più, M1’ ricopre il grafo ed è connesso perché e’ faceva parte di un ciclo

**Kruskal:**
**Si discuta l’implementazione dell’algoritmo di Kruskal. Si dica quale scelta per la struttura dati è preferibile e perché.**

L’algoritmo di Kruskal è un algoritmo che non è di visita, che prende in considerazione sempre tutti gli archi di peso minimo controllando che non ci siano cicli ( in tal caso li scarta e

passa al successivo) e in questo modo crea il minimo albero ricoprente ( ovvero che l'arco preso in considerazione non abbia i due nodi che appartengono alla stessa componente connessa ).

Per fare questo si utilizza una struttura dati detta **Union Find** che " mappa "  ogni nodo che descrivendo la componente connessa di appartenenza, questa struttura può utilizzare la **Quick Find** per velocizzare la fase di ricerca detta componente connessa a discapito dell'operazione di unione che ha complessità O(n)
Si può utilizzare anche la **Quick Union** per velocizzare la fase di Union che diventa O(i) a discapito della ricerca che dovrà scorrere tutto l'albero e quindi diventa O(n).

Per incrementare l'efficienza può usare il lemma riguardante l'albero unione, **Union By Rank,** che diminuisce la complessità della Find nella Quick Union, appendendo sempre l'albero che ha profondità minore a quello che ha profondità maggiore.

**Kruskal:**
**Su cosa si basa la dimostrazione che l'albero ricoprente costruito dall'algoritmo di Kruskal é minimo ? Spiegare brevemente la propria risposta.**

La dimostrazione che l'albero ricoprente costruito dall'algoritmo di Kruskal si basa su

**Minimo**
E' minimo perchè ogni volta che considera degli archi, prende in considerazione un arco(u,v) dove u appartiene a una componente connessa e v a un'altra componente connessa ( che non forma cicli )
Riusciamo a definire quindi un taglio che ha "u" in una componente e "v" in un'altra, e l'algoritmo non va a prendere in considerazione altri archi che appartengono a componenti connesse già prese in considerazione.
Quindi gli archi sono tutti quelli che non avremo ancora preso in considerazioni e di questi prende il minimo.
Siccome prende il minimo grazie al teorema del taglio sicuramente apparterrà al MST.
Intanto va specificato che quando Kruskal sta per aggiungere (u,v) all'albero che sta costruendo, lo fa perchè è l'arco di peso minimo che non ha ancora considerato e non forma cicli.
Poi il punto è che si riesce a definire un taglio attraversato dall'arco (u,v) ma non da archi già presi in considerazione, perché questo taglio non spezza alcuna componente connessa.
Quindi gli archi già presi in considerazione hanno ambedue gli estremi da una parte o dall'altra del taglio. Quindi (u,v) è il  minimo arco che attraversa il taglio.

**Albero:**
Per essere un albero deve essere **connesso e senza cicli**
E' connesso e aciclico: è connesso perchè se così non fosse avremmo almeno 2 componenti connesse distinti, ma dato che il grafo di partenza è connesso, esiste sicuramente un arco che collega le due componenti.
Il primo arco che connette le due componenti sicuramente non forma un ciclo perché collega due componenti connesse distinte

**Ricoprente:**
Per costruzione l'algoritmo tocca tutti i nodi del grafo poichè MST=(V,A)

==========

Il fatto che l'albero ricoprente calcolato da Kruskal sia minimo è dovuto al teorema del taglio ( l'arco di peso minimo che attraversa un taglio appartenente all'MST), dato che kruskal considera un taglio che non spezza nessuna componente connessa già note prendendo l'arco di peso minimo che le connette ( ovvero che attraversa il taglio). L'albero risulta ricoprente per costruzione.

==========

**Minimo**
E' minimo perchè ogni volta che considera degli archi, prende in considerazione un arco(u,v) dove u appartiene a una componente connessa e v a un'altra componente connessa ( che non forma cicli )
Quindi gli archi sono tutti quelli che non avremo ancora preso in considerazioni e di questi prende il minimo, lo fa perchè è l'arco di peso minimo che non ha ancora considerato e non forma cicli.
Siccome prende il minimo grazie al teorema del taglio sicuramente apparterrà al MST.
Poi il punto è che si riesce a definire un taglio attraversato dall'arco (u,v) ma non da archi già presi in considerazione, perché questo taglio non spezza alcuna componente connessa.

**Albero:**
Per essere un albero deve essere **connesso e senza cicli**
E' connesso e aciclico: è connesso perchè se così non fosse avremmo almeno 2 componenti connesse distinte, ma dato che il grafo di partenza è connesso, esiste sicuramente un arco che collega le due componenti.
Il primo arco che connette le due componenti sicuramente non forma un ciclo perché collega due componenti connesse distinte

**Ricoprente:**
Per costruzione l'algoritmo tocca tutti i nodi del grafo poichè MST=(V,A)

**P,NP e problemi difficili:**
**Si definiscono le classi P ed NP e si parli del loro rapporto.**

**P:** di problemi di decisione perché ammettono come risposte SI o NO che sono risolvibili attraverso algoritmi polinomiali, questi sono problemi facili

Polinomiali perchè i polinomi permettono ancora un costo " ragionevole " rispetto a costi computazionali quadratici o cubici.

**NP:** di problemi di decisione che ammettono una dimostrazione verificabile in tempo polinomiale, cioè: data una soluzione, si verifica, in tempo polinomiale, se quella soluzione e valida o no.

Relazione tra P e NP

P è tutta contenuta in NP, cioè i problemi risolvibili attraverso algoritmi polinomiali hanno anche dimostrazioni verificabili in tempo polinomiale.
Ma non sappiamo se NP=P, cioè il costo computazionale è uguale per entrambi, ma possiamo trovare alcuni problemi più difficili, quindi vederne il costo e relazionali.

**P:** sono problemi di decisione, hanno degli algoritmi che li risolvono in tempo polinomiale e accettano come risposta si o no.

**NP:** sono i problemi in cui la dimostrazione è facile, nel quale data un'istanza del problema , ovvero una variabile input in più, riusciamo a verificare in tempo polinomiale se la soluzione soddisfa il requisito della variabile in input in più

Quindi se abbiamo un problema in cui non conosciamo l'algoritmo di ottimizzazione ma ci viene data la soluzione, riusciamo a verificare questa soluzione in tempo polinomiale.
Il fatto è che se anche un problema abbiamo algoritmi che lo risolvono in tempo polinomiale può essere considerato dalla classe NP, se ci viene dato un problema di cui conosciamo l'algo poli e ci viene dato la soluzione ovvero il certificato di appartenenza, e la variabile in più possiamo ignorarlo CA e usare l'algo polinomiale per risolverlo così per verificare che sia un'istanza si o no del problema.
I problemi in P sono contenuti in NP ma non sappiamo se NP=P per il fatto che non ci sono algoritmi risolvibili in tempo polinomiale però non li conosciamo adesso ma potremmo conoscerlo in futuro quindi dato che ci sono prob np-completi per cui se trovi una soluzione in tempo poli di 1, attraverso la riduzione puoi trovarla di tutti, nel caso in cui ne troviamo uno possiamo risolvere tutti i problemi in NP in tempo polinomiale quindi NP=P
Però visto che ora non li conosciamo un algo che li risolve in tempo polinomiale, NP contiene P visto che ci sono problemi che non conosciamo un algo che li risolve in tempo polinomiale

**NP:** sono problemi di decisione che ammettono una dimostrazione verificabile in tempo polinomiale, cioè data una S, si verifica in tempo polinomiale se quella soluzione è valida o no.
Se esiste un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP-completi potrebbero essere risolti in tempo polinomiale, P=NP
(Quindi o tutti i problemi in NP-completi sono risolvibili deterministicamente o nessuno di essi lo è)

**P e NP:**

**dimostrare che dato un grafo non orientato pesato G e un intero positivo k, il problema di determinare se il grafo ha un MST di peso al massimo k è un problema in NP**

Posso  esprimere questo problema in NP, se ho un'istanza x che è un certo grafo e mi viene fornito un albero A, posso verificare se A ( A in questo caso Sx) verifica la condizione e posso farlo in tempo polinomiale ( sommo tutti i pesi e verifico che non ci siano cicli). Il problema di decisione del MST è un problema in NP ( di conseguenza anche in P ) ma in realtà sappiamo che avendo un grafo conosciamo anche degli algoritmi per determinare l'MST e verificare A.
Di conseguenza sto trasformando un problema di ottimizzazione in un problema di decisione. Possiamo ignorare il certificato di appartenenza e risolvere il problema

**TSP metrico:**
**Dimostrate che il problema del commesso viaggiatore è un problema in NP**

**TSP metrico:**
**Da cosa deriva il fattore di approssimazione 2 per l'algoritmo che approssima la soluzione del problema del commesso viaggiatore nel caso di un grafo in cui i pesi degli archi soddisfano la disuguaglianza triangolare ?**
Supponiamo che un grafo sia derivato da una soluzione S*

Se tolgo un arco diventa un albero ricoprente quindi non ha cicli.

c(S*) è il costo ottimale che vorrettmo raggiungere, ma togliendo l'arco otteniamo una soluzione
**T=S*\{arco}**

Quindi il costo della soluzione S* è maggiore o uguale al costo della soluzione T poichè gli archi hanno tutti pesi positivi e togliendone uno il peso totale è minore o uguale ( perchè potrei togliere un arco di peso 0= quindi:
**c(S*)>=c(T)**

Se calcoliamo l'MST A con prim possiamo dedurre che :
**c(S*) >= c(T) >= c(A)** → questo perchè A è il mST quindi c(T) >= c(A) per ipotesi.

**1.Prim → MST A**

**c(W) = 2c(A)**
Questo perchè il viaggiatore percorre due volte gli archi dell'albero per fare il tour.
A questo punto sorge un dubbio:
Al commesso viaggiatore conviene tornare indietro o prendere un arco in modo da permettergli di tagliare il percorso e ottenere una scorciatoia?

**Conviene una scorciatoia!**

**c(W') <= c(W)** → secondo la disuguaglianza triangolare

Come facciamo a fargli prendere sempre delle scorciatoie?

**2. Visita DFS → per ottenere l'ordine dei nodi scoperti**

Quindi si ottiene che
**c(S) <= c(W) = 2c(A)**

Se combiniamo con questa relazione → **c(S*) >= c(T) >= c(A)**
Si ottiene una nuova relazione:
**c(S) <= c(W) =2c(A) <= 2c(S*)**

c(S) con le scorciatoie, sicuramente non è peggiore di due volte c(S*)

In definitiva si ottiene che:
**c(S) <= 2c(S*)**
Sappiamo quindi che la soluzione S non è peggiore di due volte la soluzione (ottimale)S*
In questo caso il fattore di approssimazione è uguale a 2 → gamma = 2

Potrebbe comunque esserci un percorso migliore di quello calcolato in questo modo, perchè l'MST non ha nulla a che fare con i cammini minimi.
Non riesco a dimostrare che questa soluzione sia ottimale.

**TSP metrico:**
**Perché l'algoritmo che approssima la soluzione del problema del commesso viaggiatore nel caso di un grafo in cui i pesi degli archi soddisfano la disuguaglianza triangolare calcola un minimo albero ricoprente?**

**Algoritmi di approssimazione:**
**Si definisca e si spieghi il significato di fattore di approssimazione per un algoritmo di approssimazione per un problema difficile.**

Quando non si riesce a raggiungere la soluzione ottimale per un determinato problema, si potrebbe cercare di trovarne una abbastanza buona che possa andare bene.
In caso facessimo un'approssimazione, è necessario comprendere quanto è buona questa approssimazione per capire se potrebbe andare bene o meno.

Prendiamo ad esempio un problema di minimizzazione e una soluzione S calcolata dall'algoritmo approssimato, mentre Sm corrisponde alla soluzione ottimale per il problema di minimizzazione preso in considerazione.
E' evidente che val(S) >= val(Sm), però noi vogliamo sapere quanto è 'cattiva' questa soluzione e in un certo senso limitarla superiormente.
Supponiamo che val(Sm) <=val(S)<=p*val(Sm) per p>=1 (se = 1 allora S=Sm) e p potrebbe dipendere anche dall'istanza del problema. Val(S)/val(Sm)<=p misura il rapporto tra le due. soluzioni, dicendoci quanto è 'cattiva' la nostra soluzione.

# *Terzo esonero*

**Zaino:**
**Data una soluzione ottimale del problema dello zaino con n oggetti e capacità c che contiene l'oggetto n, come si estrae da questa soluzione una soluzione ottimale per un sottoproblema?**
**Dire anche di quale sottoproblema si tratta, spiegare perché la nuova soluzione è una soluzione per il sottoproblema e il motivo per cui è ottimale.**

Data una soluzione ottimale del problema dello zaino con n oggetti e capacità c che contiene l'oggetto n, da questa soluzione estraggo una soluzione ottimale tramite gli algoritmi di programmazione dinamica si ricavano due sottoproblemi.
1)Data la soluzione ottimale S per un P(c,n) in cui l' oggetto n è incluso nella soluzione, andrò a considerare una Soluzione S' per un sottoproblema P(c-voln,n-1), supponiamo che S' non sia ottimale. Se prendiamo una soluzione S'' ottimale val(S'')>val(S') . Se

aggiungiamo n alla soluzione S" otteniamo val(S")U n > val(S) ma questo è assurdo in quanto S era ottimale, quindi anche S' era ottimale per P(n-voln,n-1)
2)

**Zaino:**
**Si dica in quale caso e in che modo nel problema dello zaino si riduce un sottoproblema ad un altro in cui lo zaino ha capacità inferiore**

Partendo dal problema dello zaino con capacità c e i oggetti avente una soluzione per esso ottimale e ammissibile S, questo si può ridurre a uno di questi due sottoproblemi:
-se l'oggetto i-esimo è stato preso, allora la soluzione S meno l'oggetto i-esimo è ammissibile e ottimale per il problema dello zaino con C-volume(i) e i-1 oggetti. Quindi si è ridotto il problema dello zaino con capacità c e i oggetti a un sottoproblema con capacità minore (infatti togliamo il volume di i) e con un oggetto in meno.
-se non è stato preso l'oggetto i-esimo, allora la soluzione S è ammissibile e ottimale anche per il sottoproblema con capacità c e i-1 oggetti.
Si riesce a parlare di soluzioni ammissibili e ottimali perché stiamo usando un algoritmo di programmazione dinamica, che prende il massimo tra i due sottoproblemi sopracitati e che permette di ricostruire la soluzione per tutto il problema dello zaino.

**Zaino:**
**Si scriva l'enunciato del teorema di sottostruttura ottima per il problema dello zaino, e si illustri la dimostrazione per un solo caso a scelta.**

**Ipotesi:**
S è una soluzione ottimale e ammissibile per il problema Pc,i dove c è la capacità e i il numero di oggetti.

**i appartiene a S**
**TESI: Se i appartiene a S, quindi S'=S-{i} è una soluzione ammissibile e ottimale per Pc-voli,i-1**
S' è una soluzione derivata da S, quindi è ammissibile in Pc-voli,i-1, questo perché ha in totale i-1 oggetti e la somma dei volumi degli oggetti presi e minore o uguale a C-voli.
Per dimostrare che S' è ottimale ipotizzo per assurdo che esista una soluzione S'' migliore di S' per il problema di Pvoli,c-1 val(S'')>val(S')
Allora, se aggiungo l'i-esimo elemento ad S'', per il teorema della sottostruttura ottima, ottengo una soluzione migliore rispetto ad S per il problema in Pc,i val(S'' U {i} ) > val(S) ma questo è assurdo perchè S è la nostra soluzione ottimale per il problema in Pc,i

**i non appartiene a S**
**TEST: i non appartiene a S, quindi S è una soluzione ottimale e ammissibile per Pc,i-1**
Per dimostrare che S è una soluzione ottimale per Pc,i-1 per assurdo ipotizziamo che esiste una soluzione S* migliore di S per il problema in Pc,i-1 val(S*)>val(S)
Ma questo è assurdo perché se fosse così, avremmo preso S* anche quando l'i-esimo oggetto era presente nel problema, ovvero in Pc,i

**Memoization: Si spieghi che cosa si intende per memoization, a che cosa serve e come la si utilizza nel caso di implementazione iterativa e nel caso di un' implementazione iterativa**

La memoization serve per memorizzare il risultato dei sottoproblemi già calcolati, e quando li ri incontro, invece di ricalcolare, mi limito ad andare a prendere il risultato già calcolato. Quindi risolvo un problema una sola volta, e il costo senza di esso era O(2^n) invece ora diventa lineare.

Nell'implementazione ricorsiva, risolvo prima il problema di dimensione n, e ricorsivamente vado a risolvere i sottoproblemi via a via più piccoli; aumento però l'insieme dei parametri passati con una tabella nella quale memorizzare i risultati già calcolati.

Nell'iterazione parto dai problemi più piccoli, e li risolvo andando in ordine crescente di dimensione; quando arrivo a risolvere un problema di dimensione i, ho già risolto tutti i problemi di dimensione <i

**Massimo sottoinsieme indipendente:**
**Si dica quali sono i sottoproblema che si considerano per l'algoritmo per il problema del massimo sottoinsieme indipendente su un grafo lineare, e si illustri il loro mutuo rapporto.**

**Se i appartiene a S**
**Tesi:** S' = S-{i} ammissibile e ottimale per Pi-2

**Se i non appartiene a S**
**Tesi:** S è ammissibile e ottimale per Pi-1

Questi vengono usati per progettare l'algoritmo, se vogliamo sapere il valore ottimale per il problema i
val(i) = max(val(i-2)+p(i),val(i-1))

**Dimostrazione:**
Se noi sappiamo che val(i-2)+p(i) è ottimale oppure val(i-1) è ottmale allora val(i) è ottimale per Pi perchè se non lo fosse, siccome vale il teorema di sottostruttura ottima si potrebbe scomporre e ottenere 2 soluzioni migliori per i sottoproblemi che è assurdo.

**Massimo sottoinsieme indipendente:**
**Si scriva l'enunciato del teorema di sottostruttura ottima per il problema del massimo sottoinsieme indipendente su un grafo lineare, e si illustri la dimostrazione per un solo sottocaso a scelta**

**Enunciato:**
**Ipotesi:** Supposto S ottimale per Pi ( i nodi )

**Se i appartiene a S**

**Tesi:** S' = S-{i} ammissibile e ottimale per Pi-2

S' è indipendente ed è derivata da S, ed è massimale per Pi-2, lo dimostro per assurdo. Ipottiamo che S' non sia ottimale per Pi-2 allora esiste una soluzione S'' che è massimale per questo problema val(S'')>val(S'), quindi se aggiungo i ad S'' sarà una soluzione ammissibile per Pi. val(S'' U {i}) > val(S) ma questo è assurdo perché abbiamo supposto che S outtimale.

**Se i non appartiene a S**
**Tesi:** S è ammissibile e ottimale per Pi-1

**Cammini minimi tra ogni coppia di nodi:**
**Si discuta la complessità in termini di tempo e di spazio dell'algoritmo di Floyd-Warshall**

Questo algoritmo è stato ideato per calcolare il cammino minimo tra ogni coppia di nodi anche se i pesi sono negativi con costo O(n^3)
Se avessimo un grafo sparso con pesi positivi ci converrebbe iterare n volte Dijkstra
Invece può valere la pena utilizzare Floyd Warshall nel caso in cui il grafo sia denso al posto di utilizzare Dijkstra o bellman Ford.Il costo di FW è O(n^3) poiché sono necessari 3 cicli

Servono almeno 2 tabelle una vecchia e una nuova. Quindi avremo un costo in spazio dato da O(2*n^2).

**Cammini minimi tra ogni coppia di nodi:**
**Si spieghi quali sono i sottoproblemi utilizzati dall'algoritmo di Floyd Warshall; si faccia anche un esempio**

**Ipotesi: S soluzione ottimale per Pi,j,k**
Vogliamo ridurre in sottoproblemi in cui k diminusce

Quindi abbiamo 2 possibilità:
**1| K non appartiene ai nodi intermedi**
**2| K appartiene ai nodi intermedi**

**Tesi 1| Se k non appartiene ai nodi intermedi S è ammissibile per il problema Pi,j,k-1**
**2|**

In questo caso avremo un percorso i→ j→ k e possiamo dividerlo in due soluzioni
**Tesi 2.1| S1 ( i→ k) è ammissibile e ottimale per Pi,k,k-1**
**Tesi 2.2| S2 (k→ j) è ammissibile e ottimale per Pk,j,k-1**

**Cammini minimi tra ogni coppia di nodi:**
**Si scriva l'enunciato del teorema di sottostruttura ottima per il problema del calcolo dei cammini minimi tra ogni coppia di nodi in un grafo orientato pesato che può avere archi con peso negativo, e si illustri la dimostrazione per un solo sottocaso a scelta.**

L’algoritmo in questione è Floyd Warshall.

**Enunciato:**

**Ipotesi:** Supponiamo S soluzione ammissibile e ottimale per il problema Pi,j,k

Cioè S è ammissibile e ottimale per il problema di trovare il cammino minimo da i a j passando al massimo per i primi K nodi ( cioè i nodi da 0 fino a K)

**Tesi 1:** Se k appartiene ai nodi intermedi possiamo dividere la soluzione in 2 soluzioni:

**S1:** cammino che va da i a k è soluzione ammissibile e ottimale per il problema Pj,k,k-1
**S2:** cammino che va da k a j è soluzione ammissibile e ottimale per il problema Pk,j,k-1

**Tesi2:**
Se k non appartiene ai nodi intermedi S è soluzione ammissibile e ottimale per il problema Pi,j,k-1

**Dimostrazione caso k non appartiene ai nodi intermedi:**
Se k non appartiene ai nodi intermedi S è soluzione ammissibile e ottimale per il problema Pi,j,k-1.
Se per assurdo S non fosse ottimale al problema Pi,j,k-1 allora deve esistere un’altra soluzione S* ammissibile e migliore di S per Pi,j,k-1 questa soluzione sarà ammissibile anche per il problema Pi,j,k in quanto scegliere anche quando il k-esimo nodo era presente quindi risulterebbe migliore di S per il problema Pi,j,k il che è assurdo.

## Bellman-Ford:
## Dimostrazione

**Enunciato**
**Ipotesi: S è una soluzione ottimale e ammissibile per Pv,i**

**i non appartiene a S**
**Tesi: S è una soluzione ottimale e ammissibile per Pv,i-1**
Per dimostrare che S è una soluzione ottimale per Pv,i-1, per assurdo ipotizzo che esiste una soluzione migliore per Pv,i-1 ovvero S* L(S*)<L(S) perciò S* sara ammissibile e ottimale anche per il problema Pv,i. Ma questo è assurdo perchè avevamo definito S ottimale per Pv,i

**i appartiene a S**
**Tesi: S’ è una soluzione ottimale e ammissibile per Pu,i**

## Cammini minimi da una sorgente con pesi arbitrari:
## Si dica se i cammini minimi calcolati da Bellman-Ford in caso di cicli negativi sono corretti e si motivi la risposta.

In caso di cicli negativi, i cammini calcolati da bellman-ford non sono corretti, dato che dopo n(numero nodi) iterazioni, l'algoritmo ha trovato tutti i possibili cammini minimi, e nelle ulteriori iterazioni i valori nell'array dist, array che tiene conto delle distanze dei nodi da una sorgente, non dovrebbero ottenere nessuna variazione, mentre in caso di ciclo di lunghezza negativa ci sarebbero valori in dist[v] che continuerebbero a decrementarsi indefinitamente

**Cammini minimi da una sorgente con pesi arbitrari:**
**su quale proprietà si basa l'algoritmo di Bellman-Ford per determinare la presenza di cicli negativi? Si dimostri la proprietà.**

Dimostriamo che al momento della terminazione dell'algoritmo abbiamo dist[v]=δ(s,v) per ogni nodo v.

Ragionando per induzione si verifica che all'inizio dell'iterazione i abbiamo che dist[v]=δ(s,v) per ogni nodo v per il quale il cammino minimo da s a v è composto da al più i archi e questa proprietà è vera per i=0 perché s è il solo nodo tale che il cammino da s a s è composto da 0 archi e dist[s]=0=δ(s,s).

Per induzione, ipotizziamo che la proprietà sia vera all'inizio dell'iterazione i:
se consideriamo un qualunque nodo v tale che il cammino s,..u,v comprende i+1 archi, quindi il cammino minimo s,..,u comprende i primi i archi del cammino precedente e abbiamo dist[u]=δ(s,u), ma allora nel corso dell'interazione i, abbiamo che dist[v] è aggiornato in modo che dist[v]=dist[u]+W(u,v)=δ(s,u)+W(u,v) che è uguale a δ(s,v) per l'ipotesi che il cammino s,..u,v sia minimo.

Quindi la proprietà risulta soddisfatta quando inizia l'iterazione i+1.
Dato che non esistono per ipotesi cicli di lunghezza negativa nel grafo, un cammino minimo comprende al più n-1 archi e quando i=n tutti i cammini minimi sono stati individuati. Se venissero effettuate più di n iterazioni non ci sarebbe nessun aggiornamento dei valori, se però il grafo avesse cicli di lunghezza negativa i valori continuerebbero a decrementarsi, da questo deriva che questo algoritmo può essere usato per verificare se un grafo presenta cicli di lunghezza negativa.

**Cammini minimi tra ogni coppia di nodi:**
**Si dica se i cammini minimi calcolati dall'algoritmo di Floyd-Warshall in caso di cicli negativi sono corretti ( attenzione la risposta non è nè sì nè no). Si motivi la risposta.**

Nel caso in cui si è in presenza di cicli negativi, l'algoritmo di Floyd-Warshall non calcolerà i cammini minimi in modo corretto, infatti se in un grafo è presente un ciclo negativo, noi potremmo percorrerlo all'infinito ottenendo sempre cammini minimi minori quindi il problema non è ben posto. Durante l'esecuzione dell'algoritmo di Floyd-Warshall ci possiamo accorgere della presenza di un ciclo negativo poichè, per definizione di ciclo si chiuderà su

almeno un nodo e quindi sulla diagonale della matrice ( che dovrebbe contenere sempre tutti 0) che viene creata dall'algoritmo troveremo un valore negativo nella posizione v(i,j) dove i=j ( con 'i' si intende il nodo dove si chiude il ciclo).

**LCS:**
**Si scriva l'enunciato del teorema di sottostruttura ottima per il problema del calcolo della più lunga sottosequenza comune a due stringhe, e si illustri la dimostrazione per un caso a scelta.**

Nel problema del LCS si vuole confrontare due stringhe di caratteri per stabilire quanto differiscono una dall'altra, LCS non tiene conto di quali caratteri compaiono nelle due stringhe ma anche dell'ordine in cui essi compaiono

Infatti una sottosequenza è un insieme di elementi posti in un certo ordine e si cerca appunto di trovare questi elementi che sono comuni tra le due stringhe e con l'ordine adeguato

Il teorema di sottostruttura ottime per LCS, ovvero massima sottosequenza comune, ha il seguente enunciato:

**Ipotesi:**
Si ha Zk = Z1,Z2,...Zk che è LCS(Xm,Yn) per il problema Pm,n dove m è la lunghezza della sequenza di X ed n la lunghezza della sequenza di Y
**Tesi:**

1. **Zk=Xm=Yn** allora Zk-1 è una soluzione ottimale e ammissibile per il problema Pm-1,n-1
2. **Zk!=Xm** allora Zk è una soluzione ottimale e ammissibile per il problema Pm-1,n Se per assurdo Zk non è ottimale per Pm.n-1 esiste una soluzione Z* migliore di Zk LCS(Z*)>LCS(Z). Quindi questa scelta la potevo fare anche quando l'm-esimo carattere era presente c'era dato che Z* è ammissibile per il problema Pm,n sarebbe anche migliore di Zk, per il medesimo problema il che è assurdo
3. **Zk!=Yn** allora Zk è una soluzione ottimale e ammissibile per il problema Pm,n-1

**LCS:**
**Si scriva l'enunciato del teorema di sottostruttura ottima per il problema del calcolo della più lunga sottosequenza comune a due stringhe, e si mostri come su questo teorema si possa basare un algoritmo per risolvere il problema.**

In questo problema si vogliono confrontare i caratteri comuni tra due stringhe e si tiene conto anche dell'ordine in cui sono posizionate.

Infatti, una sottosequenza è un insieme di elementi posti in un certo ordine e si cerca appunto di trovare questi elementi che sono comuni tra le due stringhe e con l'ordine adeguato.

Il teorema di sottostruttura ottime per LCS, ovvero massima sottosequenza comune, ha il seguente enunciato:

**Ipotesi:**
Si ha una soluzione ottimale Zk = Z1 Z2 Z3 … Zk che è LCS(Xm,Yn) per il problema Pm,n dove m è la lunghezza della sequenza X e n è la lunghezza della sequenza Y.

**Tesi:**
Se Zk = Xm = Yn allora Zk-1 è una soluzione ottimale per il problema Pm-1,n.1
Se Zk != Xm allora Zk è una soluzione ottimale per il problema Pm-1,n
Se Zk != Yn allora Zk è soluzione ottimale per il problema Pm,n-1

Si dovrebbero inizializzare i valori quando stringa vuoto quindi i valori

Su questo si potrebbe appunto basare l'algoritmo che va a prendere come soluzioni e le va a inserire in un array bidimensionale:

for i to n:// dove n-1 e m-1 la lunghezza delle due sottosequenze

for i to m

se Xi=Yj
    val[i][j] = val(Xi-1,Yj-1) +1
se Xi!=Yj
    val[i][j] = max(val(Xi-1,Yj), val(Xi,Yj-1))

Dove se i=j rappresenta il caso Zk = Xm e Yn e i!=j rappresenta i due sottoproblemi in cui Zk!=Xm oppure Zk!=Yn infatti si andrà a prendere il valore massimo risultante da questi due sottoproblemi