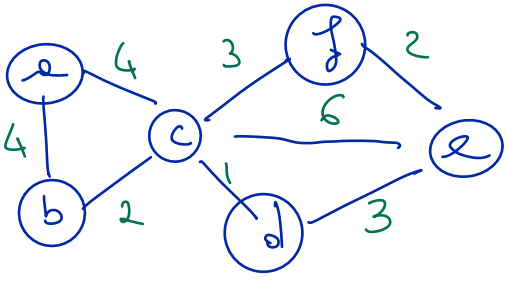
```
Djikstra(G,s)
  for each (u nodo in G)  dist[u] = ∞ // tutti i nodi sono bianchi
  parent[s] = null; dist[s] = 0 // s diventa grigio
  Q = heap vuoto
  for each (u nodo in G)  Q.add(u,dist[u])
  while (Q non vuota)
    u = Q.getMin() //estraggo nodo a distanza provvisoria minima, u diventa nero
      for each ((u,v) arco in G) //v diventa o resta grigio
          if (dist[u] + c_{u,v} < dist[v]) // se v nero falso
            parent[v] = u; dist[v] = dist[u] + c_{u,v}
            Q.changePriority(v, dist[v]) //moveUp
```

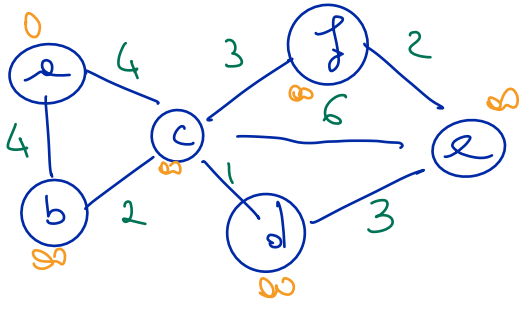
$n \log n$

$n \log n$

$m \log n$

$\Theta((n+m)\log n)$



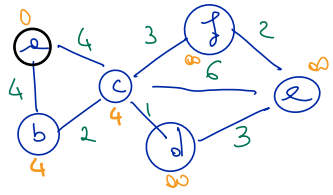


$Q = [(a,0), (b,\infty), (c,\infty), (d,\infty), (e,\infty), (f,\infty)]$

a

While (Q non vuota)

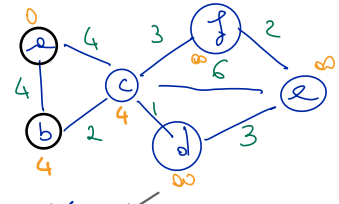
u = Q.getMin() = (a,0) viene eliminato in Q



a — c : 0+4 < ∞ → dist[c] = 0+4 ; parent[c] = a
a — b : 0+4 < ∞ → dist[b] = 0+4 ; parent[b] = a

Q = [(a,0), (b,~~∞~~ 4), (c,~~∞~~ 4), (d,∞), (e,∞), (f,∞)]

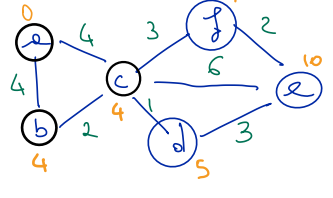
u = Q.getMin() = (b,4)



b — c : 4+2 > 4 → skip

Q = [~~(b,4)~~, (c,4), (d,∞), (e,∞), (f,∞)]

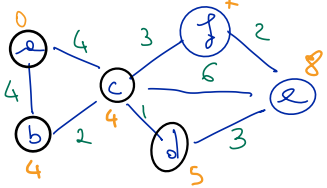
u = Q.getMin() = (c,4)



c — f : 4+3 < ∞ → dist[f] = 4+3 ; parent[f] = c
c — e : 4+6 < ∞ → dist[e] = 4+6 ; parent[e] = c
c — d : 4+1 < ∞ → dist[d] = 4+1 ; parent[d] = c

Q = [~~(c,4)~~, (d,~~∞~~ 5), (e,~~∞~~ 10), (f,~~∞~~ 7)]

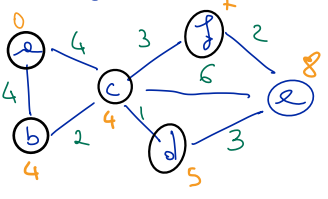
$u = Q.getMin() = (d, 5)$



$d - e$: $5+3 < 10 \rightarrow dist[e] = 5+3$, $parent[e] = d$

$Q = [(d,5), (e,10), (f,7)]$ → 8

$u = Q.getMin() = (f, 7)$



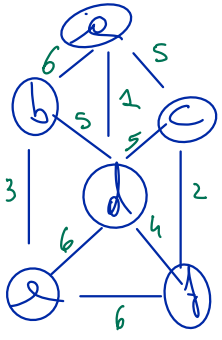
$f - e$: $7+2 > 8 \rightarrow$ skip

$Q = [(e,8), (f,7)]$

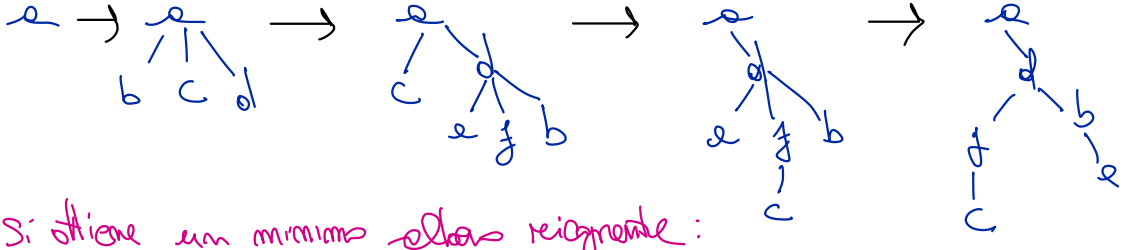
FINE: e raggiunge tutti i nodi visitati

Si ottengono le distanze ma anche quindi $parent[s] = u$ nell'algo ottengono anche l'albero

```
Prim(G,s)
  for each (u nodo in G) marca u come non visitato //necessario, avvio
  for each (u nodo in G) dist[u] = ∞
  parent[s] = null; dist[s] = 0
  Q = heap vuoto
  for each (u nodo in G) Q.add(u, dist[u])
  while(Q non vuota)
    u = Q.getMin() //estraggo nodo a minima distanza dai neri
    marca u come visitato (nero)
    for each ((u,v) arco in G)
      if (v non visitato  &&  c_{u,v} < dist[v] )
        parent[v] = u; dist[v] = c_{u,v}
        Q.changePriority(v,dist[v]) //moveUp
```



estratto	a	b	c	d	e	f
	0	∞	∞	∞	∞	∞
a	2	6	5	1	∞	∞
d		5		2	6	4
f			1			2
c			2			
b		2			3	
e					2	



Si ottiene un minimo albero ricoprente:

- albero libero: grafo connesso aciclico
- Ricoprente : contiene tutti i nod.
- la somma dei pesi degli archi è minima tra tutti

```
Kruskal(G)
  s = sequenza archi di G in ordine di costo crescente
  T = foresta formata dai nodi di G e nessun arco
  while (s non vuota)
    estrai da s il primo elemento (u,v)
    if (u,v non connessi in T) T = T + (u,v)
  return T
```

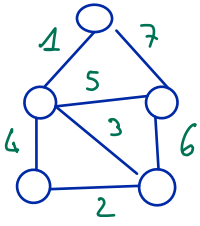
$m \log m$

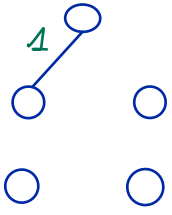
A ogni giro prendo l'arco di peso minimo

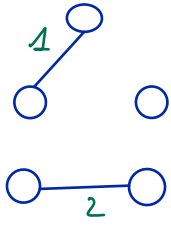
m

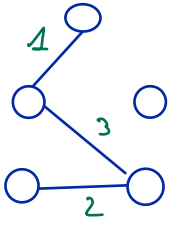
m

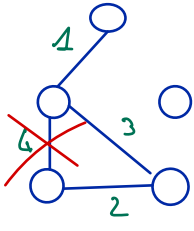
$O(m \cdot m)$



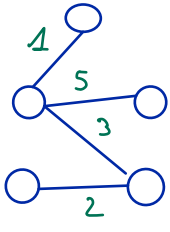




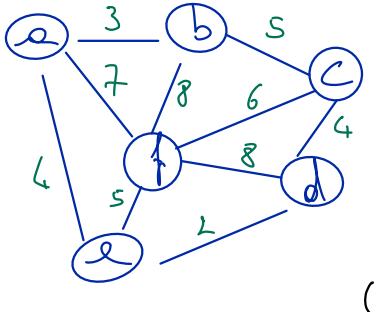




altrimenti ciclo



```
Kruskal(G)
  s = sequenza archi di G in ordine di costo crescente
  T = foresta formata dai nodi di G e nessun arco
  UF = struttura union-find vuota
  for each (u nodo in G) UF.makeSet(u)
  while (s non vuota)
    estrai da s il primo elemento (u,v)
      if (UF.union_by_need(u,v))
      // restituisce vero ed esegue union delle radici se UF.find(u) ≠ UF.find(v)
      // falso altrimenti
        T = T + (u,v)
  return T
```



ARCO ESTRATTO | FORESTA COERENTE | UF by RANK + path compression

(a,d) — a — d

(a,b) — a — d, a — b

(a,e) — a — b, a — d, d — e

(c,d) — a — b, a — d — c, d — e

(b,c) — skip xke ciclo

(e,f) — a — b, a — d — c, e — f

(a,f), (b,f), (f,d) — skip xke ciclo

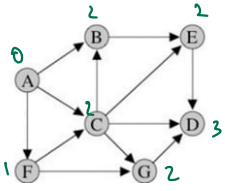
UF by RANK := ogni volta faccio puntare la radice dell'albero con il rango minore

Path compression := la FIND rende i nodi che incontra figli della radice.

```
topologicalsort(G) ordina tabella stretta TC ∀u,v∈V → nodi
                                              (u,v)∈E → archi
  S = insieme vuoto
  Ord = sequenza vuota
  for each (u nodo in G) indegree[u] = indegree di u //m passi
  for each (u nodo in G) if (indegree[u] = 0) S.add(u) //n passi
  while (S non vuoto) // in tutto m passi
    u = S.remove()
    Ord.add(u) //aggiunge in fondo
    for each ((u,v) arco in G)
      indegree[v]--
      if (indegree[v]=0) S.add(v)
  return Ord
```

Quando decremento elimino gli archi entranti...

$O(n+m)$

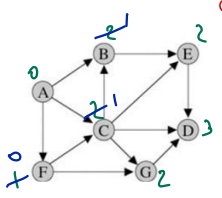


$S = \{a\}$

while (S non vuoto)

S.remove

Ord = [a]

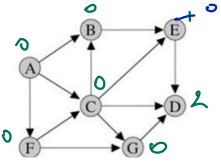


$S = \{b, g\}$

while (S non vuoto)

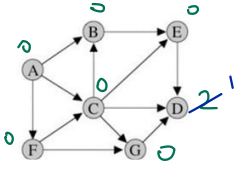
S.remove

Ord = [a, f, c, b]



S.remove

Ord = [a, f, c, b, g]

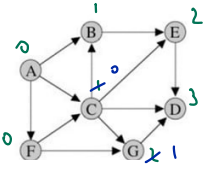


$S = \{f\}$

while (S non vuoto)

S.remove

Ord = [a, f]

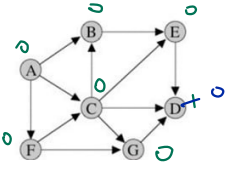


$S = \{e\}$

while (S non vuoto)

S.remove

Ord = [a, f, c, b, g, e]



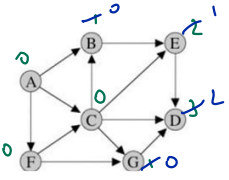
Ord = [a, f, c, b, g, e, d]

$S = \{c\}$

while (S non vuoto)

S.remove

Ord = [a, f, c]



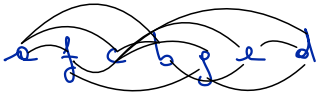
□ L'ordine di rimozione è indifferente che intanto ∃ + ordinamenti topologici

○ decremento nodi adiacenti

Prendo Ord

a f c b g e d

Collego i nodi basandomi sul grafo

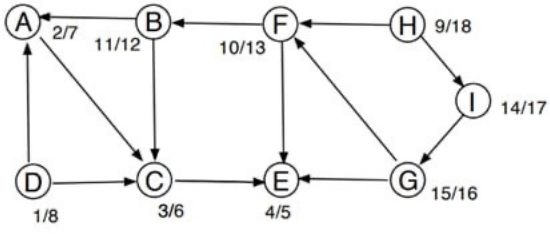


Algoritmo alternativo con DFS timestamp

```
DFS(G)
  for each (u nodo in G) marca u come bianco; parent[u]=null
  time = 0
  for each (u nodo in G) if (u bianco) DFS(G,u)

DFS(G,u,T)
  time++; start[u] = time //inizio visita
  visita u; marca u come grigio
  for each ((u,v) arco in G)
    if (v bianco)
      parent[v]=u
      DFS(G,v)
  time++; end[u] = time //marca u come nero
  //fine visita
```

iniziamo per esempio la visita da D

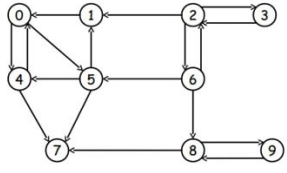


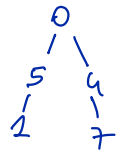
H, I, G, F, B, D, A, C,E

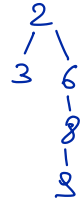
prendendo i tempi di fine visita in ordine inverso otteniamo un ordinamento topologico

COMPONENTI FORTEMENTE CONNESSE

```
SCC(G)
  DFS(G,Ord) //aggiunge i nodo visitati a Ord in ordine di fine visita   O(m+n)
  //si noti che non occorre calcolare i tempi di fine visita
  G^T = grafo trasposto di G    O(m+n)
  Ord↔ = sequenza vuota //ordine topologico delle c.f.c.
  while (Ord non vuota)                                                  O(m+n)
    u = ultimo nodo non visitato in Ord //si trova in una c.f.c. sorgente
    C = insieme di nodi vuoto
    DFS(G^T,u,C) //aggiunge nodi visitati in C
    Ord↔.add(C) //aggiunge in fondo
  return Ord↔
```





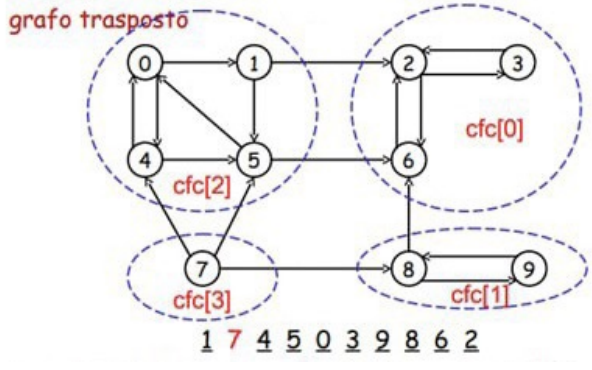


Ord = [1, 7, 4, 5, 0 3, 9, 8, 6, 2]

G è trasposto, visito tutti i nodi raggiungibili da esso e li inserisco nella cfc

VISITATO

1	cfc[0] = {2,3,6}
6	raggiunge solo nodi non ancora visitati → già parte di cfc[0]
8	cfc[1] = {8,9}
9, 3	raggiunge solo nodi non ancora visitati → già parte di cfc
0	cfc[2] = {0,1,4,5}
5, 4	raggiunge solo nodi non ancora visitati → già parte di cfc[2]
7	cfc[3] = {7} raggiunge solo nodi non visitati (oppure 1) ma non fa parte di nessuna cfc → che è già in cfc[2]
1	—



LCS con PROGRAMMAZIONE DINAMICA

Si ricava la soluzione di un problema delle soluzioni di sottoproblemi più piccoli

		A	T	C	B	A	B
	0	0	0	0	0	0	0
B	0	0↑	0↑	0↑	↖ 1	← 1	↖ 1
A	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
A	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
T	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
B	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
A	0	↑ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

ATCBAB
BACATBA
=
BACA

sottosequenza comune di lunghezza massima

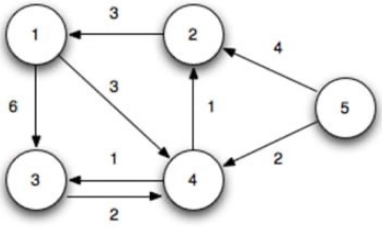
- Se stessi caratteri riga - colonna ↖ aumentando di 1 la lunghezza rispetto alla cella precedente
- Se caratteri diversi ← o ↑ e record di quella cella ha lunghezza maggiore (se uguale lunghezza allora non riceplo sopra)

LCS = si parte dall'ultimo elemento di prima seguendo le frecce e memorizzando solo quelle con ↖

$\Theta(m \cdot m)$ = memorizzo lunghezze e sequenza = $\Theta(m^2)$

Floyd e Warshall

Cammini minimi in un grafo orientato
V coppia di nod.
Sono ammessi archi negativi



Passo 0 no intermed.

D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π = nodo padre

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Passo 1 passo per il nodo 1

D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Passo 2 passo anche per 2

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**2**	5	/

Passo 3 passo anche per 3

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**2**	5	/

Passo 3 passo anche per 4 (se ci sono più strade prendo la più corta)

D

	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	**6**	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Π

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**4**	**4**	/	3	/
4	2	4	4	/	/
5	2	**4**	**4**	5	/

SOTTOGRAFO DEI CAMMINI MINIMI