CODE DI PRIORITà: FIFO in cui viene estratto per primo l'elemento (key, value) a maggiore priorità (key minore). Si possono implementare con liste non ordinate (insert ha costo O(1), removeMin e min hanno costo O(n)) o con liste ordinate (insert ha costo O(n), removeMin e min hanno costo O(1), implementazione preferibile se serve cercare il min molte volte).
Operazioni: insert(k,v), min(), isEmpty(), removeMin(), size()

le code di priorità supportano lo schema PQ-SORT:
PQ-SORT(s,c) costo quadratico O=(n^2)
input: lista s, comparatore c
output: lista s ordinata in ordine crescente
p <- coda di priorità con comparatore c
while not s.isEmpty() //fase 1 di caricamento della PQ

    e <- s.remove(s.first()) si rimuove ogni elemento di s

    p.insert(e,null) e si inserisca (ordinatamente) in p
while not p.isEmpty() //fase 2 di svuotamento della PQ

    e <- p.removeMin().getKey() si rimuovono uno ad uno gli elementi (ordinati) di p

    s.addLast(e) e si aggiungono ad s
PQ-SORT ha due varianti: selection sort (usa PQ implementata con lista non ordinata) e insertion sort (usa PQ implementata con lista ordinata).

HEAPS: albero binario quasi completo (completo per ogni livello tranne al più l'ultimo, riempito da sx verso dx), in cui vale key(v)>=key(parent(v)) cioè le key dei genitori sono <= di quelle dei figli. la radice ha la key minore. l'altezza di un heap con n nodi è O(log(n)). l'inserimento di un nodo necessita di algoritmo di UPHEAP e ha costo O(log(n)). la rimozione di un nodo (rimozione del minimo = rimozione della radice) necessita di algoritmo di DOWNHEAP (si rimuove la radice, si sostituisce con nodo foglia, si esegue downheap) e ha costo O(log(n)). si implementa con array in base alla regola: radice in indice=0, figlio sx in indice=2i+1, figlio dx in indice=2i+2 con i=indice del genitore.
inserimento: si inserisce il valore nella 1ª posizione di lastnode disponibile e poi si swappa verso l'alto finché le condizioni non sono ristabilite.
eliminazione: si scambia la radice w con il lastnode. si elimina w che ora è in posizione di lastnode. si swappa la nuova radice verso il basso finché le condizioni non sono ristabilite (e' un removeMin()).

HEAP-SORT: ha costo O(n log(n)). è un PQ-SORT che carica tutti gli elementi in una coda realizzata con un heap e poi scarica dalla coda in ordine. è composto di due fasi: nella prima l'array viene trasformato in un heap, nella seconda l'heap viene svuotato in ordine per formare l'array che viene restituito in output.
for e in p s.addLast(p.removeMin(e))

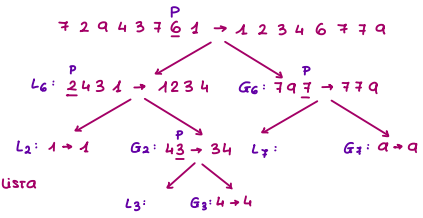
MERGE-SORT: algoritmo ricorsivo basato sul dividi et impera. divide l'array in 2 ricorsivamente fino ad arrivare a soluzioni unitarie. a quel punto fa merge risalendo nelle chiamate ricorsive e ordinando gli elementi.

QUICK-SORT: basato sul MERGE-SORT ma l'array non viene diviso a metà ma rispetto ad un pivot casuale. ha costo O(n^2) nel caso peggiore e O(n log(n)) nel caso medio.
dividi: si sceglie a caso il pivot x. si divide l'array in L:elem<x, E:elem=x, G:elem>x
ricorsione: si ordinano L e G

impera: si uniscono L, E, G

↗ questo va fatto ricorsivamente e i vari L,E,G vanno poi concatenati

```
QUICK-SORT_PARTITION(s, p)→pivot casuale
L,E,G <- sequenze vuote
x<- s.remove(p)
while not s.isEmpty()  finchè ci sono elementi in S
      y<-s.remove(s.first())  si rimuove l'elemento
      if y<x L.addLast(y)
      else if y=x E.addLast(y)   → si mette nella giusta sottolista
      else if y>x G.addLast(y)
return L,E,G
```



```
QUICK-SORT_INPLACE(s,l,r) //l e r sono rank non indici
input: sequenza s, posizioni l e r
output: sequenza s con gli elementi nelle posizioni da l a r ordinati
if l>r return
i<- intero random tra l e r
x<- s.elemAtRank(i)
(h,k)<- inPlacePartition(x)
QUICK-SORT_INPLACE(s,l,h-1)
QUICK-SORT_INPLACE(s,k+1,r)
```

implementazione:

```
static int partition (int*a, int left, int right){
int l=left+1 ;
int r=right ;
int p=left + nextInt(right-left); //posiz pivot random
int pivot= a[p];
swap(&a[left], &a[p]); //pivot al primo posto
while (l<r){
      while((l<r) && (a[l]<= pivot)) l++;
      while (a[r] > pivot) r--;
      if (l<r) swap(&a[l], &a[r]);
      }
if (a[left] > a[r]) swap (&a[left], &a[r]);
return r;
}

static void _quickSort(int*a, int first, int last){
if (first >= last) return;
int p=partion(a, first, last);
_quickSort(a, first, p-1);
_quickSort(a, p+1, last );
}

void quickSort(array*a){
_quickSort(a->arr, 0, a->size -1);
```

```
return;
}
```

BUCKET-SORT: algoritmo di tipo counting sort.(fase 1) data una sequenza di interi in un range [0, N-1] si usa una array di N puntatori a liste, inizialmente vuote. si scansiona la sequenza in input e ogni entry viene inserita alla fine della lista il cui puntatore ha l'indice corrispondente alla entry.(fase 2) dopo aver scandito la sequenza si scandisce l'array e le sue liste. ogni entry di ogni lista si inserisce in coda alla sequenza che poi viene restituita. ha costo O(n+N).

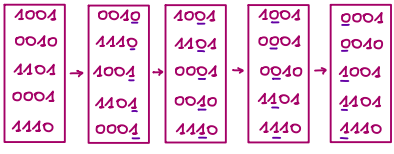
```
BUCKET-SORT(s)
B<-array di N liste
for each entry e in s do

      k= key of e   prende la key dell'elemento

      remove e from s   rimuove l'elemento

      insert e at the end of B[key]   mette l'elemento in coda al bucket [key]
for i=0 to N-1 do

      for each entry in B[i] do
            remove e from B[i]
            insert e at the end of s
```

→ rimuove tutti gli elementi dai bucket in s ordinatamente

RADIX-SORT: data una sequenza s di tuple di d elementi, applica d volte il BUCKET-SORT. ha costo O(d(n+N)).

```
RADIX-SORT(s,N)
input: sequenza s di d-tuple tali che (0,..,0)<=(x1,..,xd) && (x1,..,xd)<=(N-
1,..,N-1) per ogni tupla (x1,..,x2)
output:sequenza s ordinata in ordine lessicografico
for i<-d down to 1

      BUCKET-SORT(s,N)
```

RADIX-SORT BINARIO: applica il concetto di ordinamento lessicografico agli interi, immaginando ogni intero come una tupla di b bit/elementi che viene quindi ordinata rispetto alle altre seguendo gli stessi criteri.
ha costo O(bn)

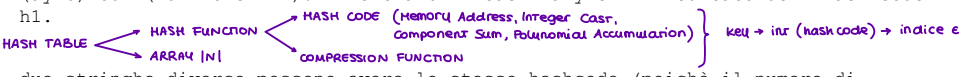



```
BINARYRADIX-SORT(s)
input: sequenza s di interi a b bit
output: sequenza s ordinata
replace each element x in s with item (0,x)
for i=0 to b-1

      replace key of each item (k,x) of s with bit xi of x

      BUCKET-SORT(s,2)
```

MAPS: collezione di entry (key, value) con key uniche.
Operazioni: get(k), put(k,v), remove(k), size(), isEmpty(), entrySet(), keySet(), values()
DIZIONARI: collezione di entry (key, value) in cui le keys possono non essere

uniche. ha come operazione ulteriore il findAll(k).

HASH TABLES: le hash table servono ad ottenere un intero a partire da una chiave. si basano su una hash function (a sua volta formata da hash code e compression function) e su un array di size N.
l'hash function (definita come l'applicazione di due funzioni h(x)=h2(h1(x)) in cui h2:int->[0,N-1] è una funzione di compressione e h1:keys->int è un hash code) prende una key=stringa, la converte in un intero e poi riconduce quell'intero ad un indice dell'array.
le funzioni di compressioni possono essere di due tipi h2(y)=y modN o h2(y)=(ay+b)modN (funzione MAD). in entrambi i casi la y è il risultato dell'hash code h1.



due stringhe diverse possono avere lo stesso hashcode (poichè il numero di hashcode è finito), o due stringhe con hashcode diverso possono avere stesso indice (poichè gli indici sono meno degli hashcode) -> si generano collisioni

GESTIONE COLLISIONI:
1 separate chaining: si costruisce una lista per ogni casella della tavola hash. la tavola hash diventa un array di liste. se la tabella ha N caselle il numero di elementi per lista è n/N. a questo punto se si ha a che fare con una mappa si trova la lista in cui va inserito l'elemento, si scandisce tutta e solo se non è già presente si aggiunge. se invece si ha a che fare con dizionari si inserisce il nuovo elemento in testa alla lista.
2 open addressing: si mette l'elemento che fa collisione nella tavola ma in un altro punto tramite un algoritmo.
2a linear probing: si guarda alla prima posizione libera
3 double hashing

ALBERI:
ALBERO BINARIO: ogni nodo ha <=2 figli.
ALBERO BINARIO PROPRIO: ogni nodo ha esattamente 2 figli.

proprietà: nodi esterni=nodi interni +1

nodi = 2(nodi esterni) -1

altezza <= (nodi-1)/2

altezza >= log in base 2 del numero dei nodi esterni

altezza >= [log in base 2 di (n+1)] -1

VISITE DI ALBERI BINARI: le visite si differenziano in visite in profondità (cioè visite DFS, tra cui quelle in preordine, postordine, ordine) e in ampiezza (cioè visite BFS).
VISITE DFS:
PREORDINE: preOrder(radice)

{if radice == null return;

visit(radice)

preOrder(radice->sx)

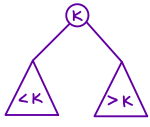
preOrder(radice-<dx)

return}

POSTORDINE:
```
postOrder(radice)
{if radice == null return;
postOrder(radice->sx)
postOrder(radice-<dx)
visit(radice)
return}
```
IN ORDINE:
```
inOrder(radice)
{if radice == null return;
inOrder(radice->sx)
visit(radice)
inOrder(radice-<dx)
return}
```
VISITA EULERO: esegue simultaneamente visita preorder, inorder e postorder. visita ogni nodo 3 volte, prima da sx poi da sotto poi da dx.

ALBERI BINARI DI RICERDA BST: sono alberi binari in cui esiste un ordinamento sulle chiavi (lo spazio delle chiavi è totalmente ordinato cioè per ogni coppia di chiavi k1, k2 vale (k1>=k2)V(k2>=k1)). ricorsivamente, data la radice di key k, tutti gli elementi nel sottoramo sx avranno key <k e tutti gli elementi nel sottoramo dx avranno key >k. dato un BST se si esegue una visita inOrder (in ordine simmetrico) si visiteranno le chiavi in ordine crescente.
grazie all'ordinamento sui BST esiste un algoritmo di ricerca di un valore v che non richiede la visita dell'intero albero:

```
TREESEARCH(k,v){
      if T.isExternal(v) return v
      if k<key(v) return TREESEARCH(k, left(v))
      else if  k=key(v) return v
      else if  k>key(v) return TREESEARCH(k, right(v))
}
```



ALBERI AVL: sono sempre alberi di ricerca tali che per ogni nodo |hsx - hdx|<=1 cioè per ogni nodo la differenza tra le altezze dei sottoalberi sx e dx è al più 1. |hsx - hdx| è detto fattore di (s)bilanciamento ed è calcolato su ogni nodo. un AVL è quindi un BST con fattore di bilanciamento compreso tra -1 e 1. un AVL ha un altezza O(log n). le operazioni su AVL hanno costo peggiore O(log n)=O(h)
INSERIMENTO IN AVL: a seguito di un inserimento l'albero potrebbe non rispettare più le condizioni di AVL. per ripristinarle si effettuano operazioni di
SEMPRE TRA 2 NODI ← rotazione (left-left, right-right, l-r, r-l) sul primo nodo sbilanciato (cioè sul primo nodo in cui il fattore di bilanciamento non è più compreso tra -1 e 1).
ELIMINAZIONE IN AVL: mentre dopo l'inserimento è sufficiente una rotazione per ristabilire le condizioni di AVL, dopo un'eliminazione un sola rotazione potrebbe non essere sufficiente.

GRAFI: un grafo è una coppia (V,E) con V:insieme di nodi ed E:insieme di archi orientati o meno.

proprietà: la sommatoria in v dei gradi dei v è uguale a 2(numero di archi)

in un grafo non orientato senza self loop e archi multipli vale che

numero archi <= [numero vertici(numero vertici - 1)]/2

operazioni: numVertices(), vertices(), numEdges(), edges(), getEdge(u,v), endVertices(e), opposite(v,e), outDegree(v), inDegree(v), outgoingEdges(v), incomingEdges(v), insertVertex(x), insertEdge(u,v,x), removeVertex(v), removeEdge(e)

stutture: con lista, con liste di adiacenza, con matrice di adiacenza

SOTTOGRAFO: dato G=(V,E), G'=(V',E') è sottografo di G se V' è incluso o uguale a V, E' è incluso o uguale a E, G' è un grafo. un sottografo che comprende tutti i vertici di V è detto ricoprente.

GRAFO CONNESSO: è un grafo in cui comunque si prende una coppia di vertici esiste un path che li collega

SOTTOGRAFO MASSIMALE CONNESSO: è un sottografo che non può essere reso più grande (includendo altri vertici) senza che perda la proprietà di connessione

ALBERO: è un grafo non orientato, connesso, senza cicli.

FORESTA: è un grafo non orientato e senza cicli (manca condizione di connessione rispetto all'albero).

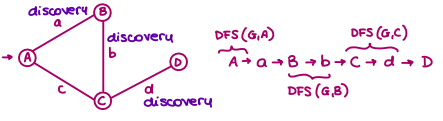
ALBERO RICOPRENTE DI UN GRAFO: è un sottografo ricoprente che è un albero.

FORESTA RICOPRENTE DI UN GRAFO: è un sottografo ricoprente che è una foresta.

VISITA DFS: visita tutti i vertici e gli archi di G, determina se è connesso, ne calcola le componenti connesse e le foreste ricoprenti con costo **O(n+m)** con n vertici e m nodi. con opportune modifiche può essere usato per trovare un ciclo o un percorso fra due vertici. la DFS si presta a risolvere problemi di topological sort.

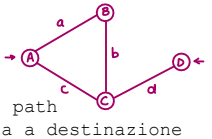
DFS DA UN VERTICE:

```
DFS(G,u)
input: grafo G e vertice u di G
output: insieme di vertici raggiungibili da u con i loro discovery edges
(l'insieme dei discovery edges definisce un sottografo connesso)
u<- visited
for each of u's outgoing edges, e=(u,v) do
    if v not visited then
        record e as discovery edge for v
        DFS(G,v)
```



DFS PATH TRA DUE VERTICI:

```
PATHDFS(G,v,z)
setLabel(v, visited) // v<- visited
s<- pila ausiliaria
s.push(v) //ho percorso v quindi aggiungo al path
if v=z return s.elements() //se sono arrivata a destinazione
for all e in G.incidentEdges(v) //per tutti gli archi di v
    if getLabel(e)= unexplored //se il lato non è visitato
        w<-opposite(v,e) //vado al vertice opposto w
```



PATHDFS(G,A,D)
A = visited
a = unexplored
B = unexplored
a = discovery
↳ PATHDFS(G,B,D)
B = visited
b = unexplored
C = unexplored
b = discovery
↳ PATHDFS(G,C,D)
C = visited
d = unexplored
D = unexplored
d = discovery
↳ PATHDFS(G,D,D)
D = visited
D = D
return S
→ S = A,a,B,b,C,d,D

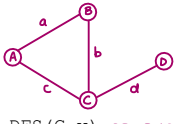
```
            if getLabel(w)= unexplored //e se w non è visitato
                  setLabel(e, discovery) //etichetto il lato come discovery
                  s.push(e) //e lo aggiungo al path
                  PATHDFS(G,w,z) //chiamata ricorsiva a partire da w
                  s.pop(e) //se la direzione è sbagliata rimuovo e
            else setLabel(e, back) //altrimenti lo etichetto come back
s.pop(v) //se ho sbagliato tolgo v
```

DFS CHE TROVA CICLI:

```
CYCLEDFS(G,v,z)
setLabel(v, visited)
s.push(v)
for all e in G.incidentEdges(v)
      if getLabel(e)= unexplored
            w<-opposite(v,e)
            s.push(e)
            if getLabel(w)= unexplored
            setLabel(e, discovery)
            PATHDFS(G,w,z) //se la chiamata non mi ha fatto trovare un ciclo
            s.pop(e) //allora tolgo e
      else t<- new empty stack
            repeat
                  o<-s.pop() //tolgo da s
                  t.push(o) //e metto in t
            until o=w //fin quando non trova w
            return t.elements()
s.pop(v) //se non sono mai entrata nell'else tolgo v poichè non è stato utile
per trovare il ciclo
```

DFS SU INTERO GRAFO:

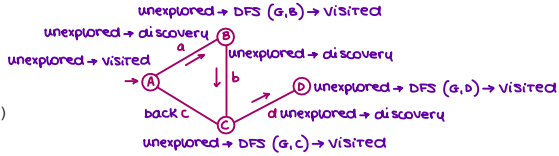
```
DFS(G)
input: grafo G
output: G con lati etichettati come discovery/back
for all u in G.vertices() setLabel(u, unexplored)
for all e in G.edges() setLabel(e, unexplored)
for all v in G.vertices() if getLabel(v)=unexplored DFS(G,v)
DFS(G,v)
input: grafo G e vertice di inizio v
output: componente connessa di v con lati (di G) etichettati come discovery/back
setLabel(v, visited)
for all e in G.incidentsEdges(v)
```



vertici unexplored
archi unexplored

esegue DFS (G,v) su ogni vertice marcato come unexplored

```
      if getLabel(e)= unexplored
      w<-opposite(v,e)
            if getLabel(w)= unexplored
                  setLabel(e, discovery)
                  DFS(G,w)
            else setLabel(e, back)
```



GRAFI ORIENTATI - DIGRAFI: sono rappresentabili con liste di adiacenza (senza ridondanze se si utilizza una sola lista di adiacenza per ogni nodo, poichè si segnano gli archi uscenti che quindi influiscono sulla lista di un solo nodo invece che due come nel caso di grafi non orientati. è frequente però rappresentare un digrafo con due liste di adiacenza per ogni vertice, una per gli archi entranti e una per quelli uscenti, quindi si reintroduce la ridondanza) o con matrice di adiacenza (che non sarà simmetrica come nel caso di grafi non orientati).

DFS SU DIGRAFI: produce una foresta ricoprente. ci sono 4 tipi di archi: discovery (definiscono la foresta ricoprente), back (identificano cicli poichè collegano due vertici che sapevamo già essere connessi ma in verso opposto), forward (identificano archi paralleli poichè collegano due vertici che sapevamo già essere connessi, andando nello stesso verso), cross (uniscono vertici già visitati attraverso altri collegamenti ma con cui non creano nè loop nè archi paralleli). c'è doppia marcatura dei nodi: exploring ed explored. un altro tipo di doppia marcatura è con le etichette DiscoveryLabel (intero che indica l'ordine di scoperta del nodo) e LeavingLabel (intero che indica l'ordine di abbandono, che avviene poichè tutti i discendenti sono stati visitati)

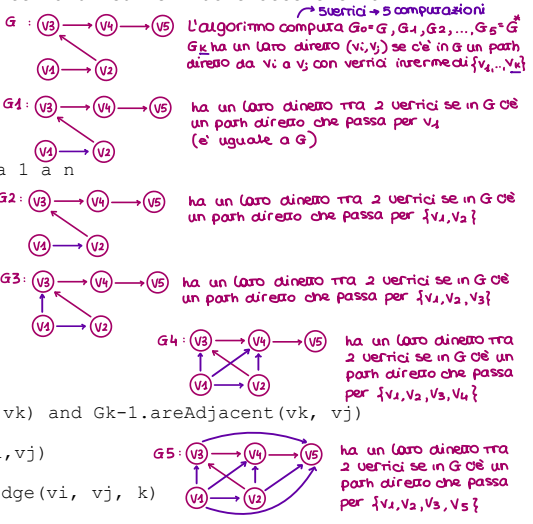
```
D-DFS(Digraph G, Vertex v){
setDiscoveryLabel(v, getNextDLabel());
for all e in outgoingEdges(v)
      if (getLabel(e) == UNEXPLORED)
            w=opposite(e,v)
            if (getDiscoveryLabel(w)==UNEXPLORED)
                  setLabel(e, DISCOVERY)
                  D-DFS(G,w)
            else if (getLeavingLabel(w)==0)
                  setLabel(e, BACK)
            else if (getDiscoveryLabel(v)<getDiscoveryLabel(w))
                  setLabel(e, FORWARD)
            else setLabel(e, CROSS)
      setLeavingLable(v, getNextLabel())
}
```

CONNETTIVITà:
DEBOLE: un grafo è debolmente connesso sse il grafo non orientato sottostante è connesso
FORTE: un grafo è fortemente connesso sse per ogni lato (u,v) esiste un percorso orientato da u verso v (anche da v verso u, cioè un ciclo; se il grafo è aciclico non è fortemente connesso). dato il grafo trasposto GT=(V,ET) di G=(V,E), cioè il grafo in cui ogni arco di E è stato invertito, vale che G è fortemente connesso sse GT è fortemente connesso.

prova: per testare la connettività forte di un grafo G si applica una visita DFS a partire da un vertice v in G. se vengono visitati tutti i nodi di V, allora si procede con la visita in DFS a partire dallo stesso vertice v su GT grafo trasposto. se vengono visitati tutti i nodi di V anche nella visita di GT allora G è fortemente connesso.

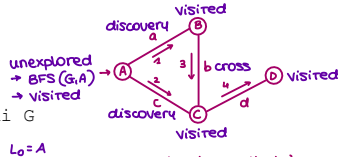
CHIUSURA TRANSITIVA: la chiusura transitiva di un grafo G può essere visualizzata con un grafo G* che su ogni vertice ha un arco diretto per ogni nodo che in G è raggiungibile tramite path.

FLOYD-WARSHALL PER CHIUSURA TRANSITIVA: l'algoritmo calcola una sequenza di grafi G0,..,Gn di cui G0 è il grafo di partenza di cui si vuole ottenere la chiusura transitiva.




```
FLOYD-WARSHALL(G)
input: digrafo G
output: digrafo G*
i<-1
for all v in G.vertices()

      denote v as vi //numera i vertici da 1 a n

      i<- i+1
G0<-G
for k=1 to n do //calcola Gk da Gk-1

      Gk<- Gk-1

      for i=1 to n, i!=k, do

            for j=1 to n, j!=i, j!=k do

                  if Gk-1.areAdjacent(vi, vk) and Gk-1.areAdjacent(vk, vj)

                  if not Gk.areAdjacent(vi,vj)

                        Gk.insertDirectedEdge(vi, vj, k)

      return Gn //Gn=G*
```

VISITA BFS: è un sistema di visita a livelli. partendo da un nodo visita prima i nodi a distanza 1, poi quelli a distanza 2 e così via. visita tutti i vertici e gli archi di G, determina se G è connesso, ne calcola le componenti connesse e le foreste ricoprenti. la BFS si presta a risolvere problemi di cammino minimo. ha costo **O(n+m)** come il DFS.




```
BFS(G)
input: grafo G
output: etichettamento dei lati e partizione dei vertici di G
for all u in G.vertices() setLabel(u, unexplored)
for all e in G.edges() setLabel(e, unexplored)
for all v in G.vertices()
```
→ setup

$L_0 = A$
$L_1 = B, C$ → quando finisce il 1° while i è incrementato di 1 e quindi il 2° while è applicato finché L_1 is not empty
$L_2 = D$

```
      if getLabel(v)=unexplored //se trova un vertice non esplorato

            BFS(G,v) //lancia BFS a partire da quel vertice
BFS(G,s) //fa uso di coda, non è ricorsivo
L0 <- new empty sequence //L0 conterrà i vertici a distanza 0 dal vertice
L0.addLast(s) //il vertice è a distanza 0 da sè stesso
setLabel(s, visited) //s è visitato
i <- 0
while not Li.isEmpty() //fin quando Li è diversa dalla coda vuota
                        nella 1° iterazione i=0, quindi L0 contiene il vertice da cui si inizia
      Li+1 <- new empty sequence //lista di vertici a distanza i+1 crea Li+1 = L1

      for all v in Li.elements() //per ogni v in Li nella 1° iterazione è solo quello di inizio

            for all e in G.incidentsEdges(v) //per ogni arco incidente in v

                  if getLabel(e)=unexplored //se e è unexplored

                        w<-opposite(v,e)                                  → se l'arco è unexplored
                                                                            e porta ad un vertice
                        if getLabel (w)=unexplored //se w è unexplored      unexplored

                              setLabel(e, discovery) //e è discovery

                              setLabel(w, visited) //e w è aggiornato a visited

                              Li+1.addLast(w) //w in coda a Li+1  il vertice opposto è aggiunto
                                                                  in Li+1 = L1 nella 1° iterazione
                        else setLabel(e, cross) //se w non è unexplored e è
cross

      i<-i+1 //incremento la distanza
```

VERSIONE SEMPLIFICATA DEL BFS IN CASO NON SIANO RILEVANTI LE LISTE DEI VERTICI ALLE VARIE DISTANZE: in questo caso i nodi sono in ordine di vicinanza su un'unica coda. si perde l'informazione sul numero di livelli del grafo.

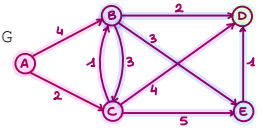
```
BFS(G,s)
L<-new empty sequence
L.endque()
setLabel(s, visited)
while not L.isEmpty()

      v=deque(L) //estraggo il vertice da L

      for all e in G.incidentEdges(v)

            if getLabel(e)=unexplored

                  w<-opposite(v,e)

                  if getLabel(w)=unexplored

                        setLabel(e, discovery)

                        setLabel(w, visited)

                        L.endque()

                  else setLabel(e, cross)
```

GRAFO PESATO: ha dei valori associati agli archi, ed è definito come G=(V,E,W) con W funzione peso. la distanza fra due nodi u,v sarà la lunghezza del percorso

più breve tra i due.
ci sono tre categorie di problemi di percorso minimo:
1 SP(G,u,v) percorso minimo in G da u a v (output: 1 percorso)
2 SSSP(G,s) percorso min tree radicato in s (output: n-1 percorsi)
3 ASPS(G) percorso minimo per ogni coppia (u,v) in V^2 (output: n^2 percorsi)

DIJKSTRA: introduce il concetto di livello pesato. si assume che il grafo sia connesso e che i pesi degli archi siano non negativi. parte dalla sorgente e trova uno ad uno tutti i percorsi minimi: parte da s, associa ad ogni vertice v una stima d(v) che rappresenta la distanza di v da s inizializzata a infinito e migliorata con l'avanzamento dell'algoritmo, ed ingloba ad ogni passo il vertice con la stima corretta (con la stima minore). inglobare un nodo fa migliorare anche altre stime (il miglioramento di una stima è detto rilassamento). ha costo O((m+n)log n)

DIJKSTRA(G,s)
input: grafo pesato G con pesi non negativi e vertice s di G
output: lunghezza di uno shortest path da s ad ogni vertice di G
initalize D[s]=0 and D[v]=infinito for each vertex v!=s
let a pq Q contain all vertices of G using D labels as keys
while Q not empty do //finchè la coda non è vuota



ordine di visita: 1 2 3 4 5

costi:

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	7
0	3	2	5	6
0	3	2	5	6
0	3	2	5	6

```
      //pull new vertex u into cloud

      u = value returned by Q.remove.min() //rimuovi il minimo e mettilo nella
nuvola

      for each vertex v adjacent to u such that v is in Q do

            if D[u]+w(u,v)<D[v] then

                  D[v]=D[u]+w(u,v)

                  change to D[v] the key of vertex v in Q
return label D[v] of each vertex v
```

BELLMAN-FORD: tenta tutti i possibili rilassamenti n-1 volte. all'i-esima iterazione del for trova shortest path di al più i archi. ha costo O(nm)

FLOYD-WARSHALL PER APSP: ha costo O(n^3)

```
FLOYD-WARSHALL(Graph G)
dist= new nxn matrix of distances
next= new nxn matrix of vertex indices
forall (u,v) in 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]
```

```
PATH(Vertex v, Vertex u)
if (next[u][v]==0) return []
path=[u]
while(u!=v)

      u=next[u][v]

      path.append(u)
return path
```

SHORTEST PATH PER DAG: trova shortest path in tempo lineare $O(n+m)$ anche se il grafo è pesato e ha pesi negativi

```
DAGDISTANCES(G,s)
for all v in G.vertices()

      if v=s setDistance(v,0)

      else setDistance(v,infinite) //fin qui inizializzazione
for u=1 to n do //dal primo nodo fino al nodo n

      for each e in G.outEdges(u) //per ciascun arco uscente

            z<-G.opposite(u,e)

            r<-getDistance(z)+weight(e)

            if r<getDistance(z) //se r migliora la stima di z

                  setDistance(z,r) //allora si aggiorna
```

ALBERI RICOPRENTI MINIMI - MST: un MST è un albero ricoprente (se esistono archi con pesi uguali non è unico) di un grafo pesato con minimo peso totale degli archi. se il grafo non è pesato l'albero ricoprente è sempre minimo. è diverso dall'albero dei cammini minimi SPT.

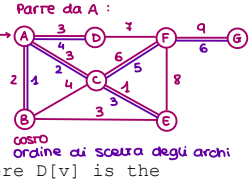
PROPRIETà DI CICLO: sia T un MST di un grafo pesato G e sia e un arco di G non in T. allora se si aggiunge e a T si forma un ciclo e vale che il peso di e è maggiore del peso di ogni altro arco in C.

PROPRIETà DI PARTIZIONE: siano U e V due sottoinsiemi dell'insieme dei vertici di G. allora preso un arco e di peso minimo tra le due partizioni, esisterà un MST di G che lo contiene.

$O(V^2)$ con matrice di adiacenza
$O(V \log V + E \log V)$ con lista di adiacenza

PRIM-JARNIK: parte arbitrariamente da un vertice s e fa crescere il MST come una nuvola. ad ogni vertice v associa una stima che rappresenta il più piccolo peso fra tutti gli archi che connettono v a qualunque vertice già incluso nella nuvola. ad ogni passo aggiungo alla nuvola il vertice esterno con l'etichetta più bassa possibile e aggiorno le distanze degli altri vertici coinvolti.

```
PRIM-JARNIK(G)
input: G non orientato, pesato, connesso con n vertici e m archi
output: MST T di G
pick any vertex s of G
D[s]=0
for each vertex v!=s do D[v]=infinite
initialize T=0
initialize pq Q with entry (D[v], (v,none)) for each vertex v where D[v] is the
```



key in the pq and (v,none) is the associated value
while Q is not empty do //finchè Q non è vuota

(u,e)=value returned by Q.remove_min() //estrae il minimo da Q

connect vertex u to T using edge e //connette u a T con e

for each edge e'=(u,v) such that v is in Q do //per ogni arco uscente e' con l'altro estremo ancora in Q

if w(u,v)<D[v] then //se l'arco (u,v) è < di D[v]

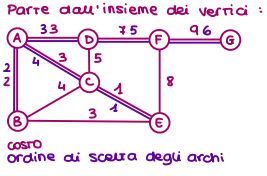
D[v]=w(u,v) //aggiorno D[v]

change the key of vertex v in Q to D[v] //cambio chiave di v in Q

change the value of vertex v in Q to (v,e') //cambio il valore di v in Q
return the tree T

$O(E \log E)$

KRUSKAL: è un algoritmo che inizia considerando tutti i nodi, senza archi, che quindi formano una foresta ricoprente. poi aggiunge gli archi di peso minimo per collegare coppie di nodi. andando avanti, aggiungendo altri archi, il numero di componenti connesse diminuisce. quando aggiungo l'n-1 esimo arco la foresta è diventata un albero ricoprente.



KRUSKAL(G)
input: grafo G pesato, connesso, con n vertici e m archi
output: MST T di G
for each vertex v in G do

define elementary cluster C(v)={v}
initialize priority queue Q to contain all edges in G, using weights as keys
T=0
while T has < n-1 edges do

(u,v)=value returned by Q.remove_min()

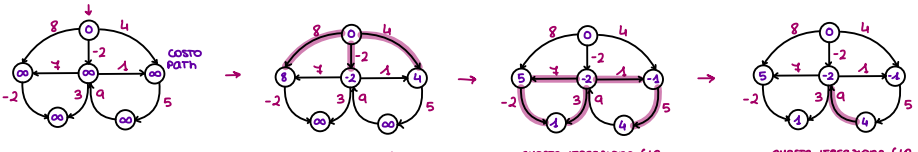
let C(u) be the cluster containing u and let C(v) be the cluster containg v

if C(u)!=C(v) then //se il cluster di u è diverso da quello di v

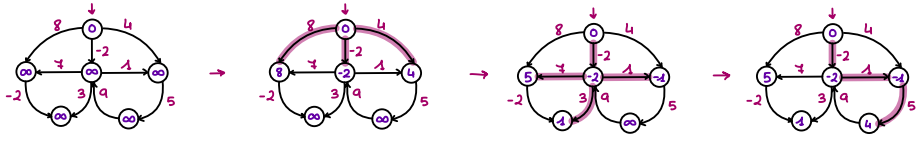
add edge (u,v) to T //aggiunge l'arco (u,v) a T

merge C(u) and C(v) into one cluster //fonde i due cluster
return tree T

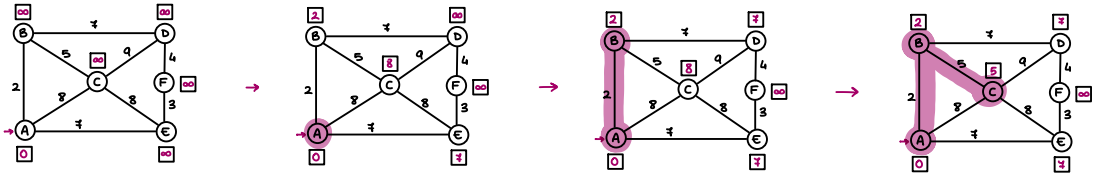
SHORTEST PATH:
Bellman-Ford

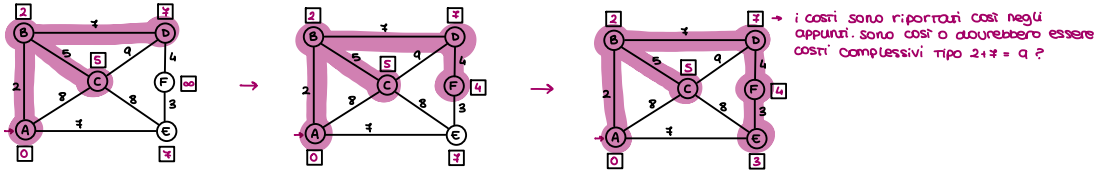


DAG-Based Algorithm



MINIMUM SPANNING TREE:
Prim-Jarnik's





Kruskal's