

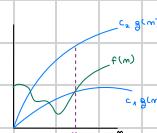
RIASSUNTO



NOTAZIONE ASINTOTICA

• $\Theta(g(m)) = \left\{ f(m) : \exists c_1, c_2, m_0 > 0 \text{ t.c.} \right.$

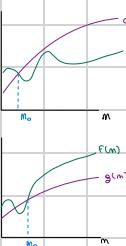
$$\left. 0 \leq c_1 \cdot g(m) \leq f(m) \leq c_2 \cdot g(m) \quad \forall m \geq m_0 \right\}$$



$$f(m) = \Theta(g(m)) \rightarrow a = b$$

• $O(g(m)) = \left\{ f(m) : \exists c, m_0 > 0 \text{ t.c.} \right.$

$$\left. 0 \leq f(m) \leq c \cdot g(m) \quad \forall m \geq m_0 \right\}$$



$$f(m) = O(g(m)) \rightarrow a \leq b$$

• $\Omega(g(m)) = \left\{ f(m) : \exists c, m_0 > 0 \text{ t.c.} \right.$

$$\left. 0 \leq c \cdot g(m) \leq f(m) \quad \forall m \geq m_0 \right\}$$



$$f(m) = \Omega(g(m)) \rightarrow a \geq b$$

RULE OF THUMB

Funzioni logaritmiche crescono più lentamente di polinomiali che a loro volta crescono più lentamente di esponenziali.

SCALA INFINITESIMI: $K, \log(\log m), \log m, m, m \log m, K^{\log m}, K^{\sqrt{m}}, m^K, m^{\log m}, m^{\sqrt{m}}, K^m, m!, m^m$

MASTER THEOREM

ricorrenza $T(m) = aT(m/b) + f(m)$ • $a \geq 1, b \geq 1$ costanti • $f(m)$ funzione asintoticamente positiva

3 CASI: 1) Se $f(m) = O(m^{\log_b a - \epsilon})$ (con $\epsilon > 0$ cost.), allora $T(m) = O(m^{\log_b a})$

2) Se $f(m) = \Theta(m^{\log_b a})$, allora $T(m) = \Theta(m^{\log_b a} \log m)$

3) Se $f(m) = \omega(m^{\log_b a + \epsilon})$ (con $\epsilon > 0$ cost.) e se $a f(\frac{m}{b}) \leq c \cdot f(m)$ per $c > 1$ cost. e m sufficientemente grande,
allora $T(m) = \Theta(f(m))$

ALGORITMI DI ORDINAMENTO

• MERGE SORT • utilizza D&I • $T(m) = O(m \log m)$ • L'algoritmo opera nel seguente modo:
 1- DIVIDE, divide la sequenza degli elementi da ordinare in due sotto sequenze di $n/2$ elementi ciascuna.
 2- IMPERA: ordina le sotto-sequenze in modo ricorsivo.
 3- COMBINA, fonde le due sotto-sequenze ordinate per generare un'unica sequenza ordinata del problema originale.

• BINARY SEARCH • algoritmo di ricerca • utilizza D&I • $T(m) = O(\log m)$ (è ottimale) • affinché funzioni la sequenza fornita deve essere ordinata.

↳ Trova un elemento "TARGET":
 1) DIVIDE: controlliamo l'elemento di mezzo (mid);
 2) IMPERA: se l'elemento target è minore del mid allora andremo a cercare il target nel sotto array di sinistra, mentre se il target è maggiore del mid, lo andremo a cercare nel sotto array di destra.
 3) COMBINA: triviale.

• MAX SOTTOARRAY • utilizza D&I • $T(m) = O(m \log m)$ • Dato un array di dimensione m , trovare un sottoarray i cui valori hanno somma max

• COUNTING INVERSIONS • utilizza D&I • merge and count • $T(m) = O(m \log m)$ • Data una permutazione determinare il numero di inversioni.

- 1) Divide: separa la lista in 2
- 2) impera: conta ricorsivamente il numero di inversioni in ciascuna delle metà
- 3) combina: conta il numero di inversioni quando unisco le due metà ritornando la somma di questa e di quelle ottenute nell'impresa.

• QUICKSORT: • utilizza D&I • $T(m) = \Theta(m \log m)$ • divide in loco sull'array stesso.

- 1) divide: partiziono l'array $A[p..r]$ in due parti $A[p..q-1]$ e $A[q..r]$, tali che ogni elemento della prima metà sia minore o uguale dell'elemento di mezzo, che a sua volta è minore o uguale a ogni elemento della seconda metà
- 2) impera: ordino i 2 sotto array chiamando ricorsivamente quicksort
- 3) combina: gli array sono già ordinati

-COUNTING SORT

- basato sul conteggio e non sui confronti

$$\bullet T(m) = \Theta(m+k) \quad \bullet E' stabile!$$

-RADIX SORT

- non si basa sui confronti

- E' stabile!

- utilizzato quando l'array ha dimensione elevata

- utilizza altri algo stabili (e.g. counting sort)

- trova la somma di un sottodarray contiguo

- algoritmo di Kadane

- suppone che ciascun elemento input sia compreso fra 1 e k.

$$\bullet \# passaggi = \# di cifre dell'elemento maggiore (da 0 a 9)$$

$$\bullet T(m) = \Theta(d(m+k))$$

costo di ogni chiave

\# di numeri a counting sort

STRUTTURE DATI

OP. TIPICHE: **INSERT**, **REMOVE**, **SEARCH**

- DIZIONARIO: insieme di coppie (elem, key) dove ogni key appartiene ad un insieme totalmente ordinato.

REMOVE

op. possibili

SEARCH

- ARRAY: collezione di celle numerate (m celle)

PROP.: 1) indici delle celle sono numeri consecutivi

accesso in tempo costante

2) non è possibile aggiungere nuove celle

+ è possibile realizzare un

dizionario con gli array

- STACK e QUEUE: insiemi di dati che consentono INSERT e DELETE

LIFO
FIFO

l'elemento rimosso è predefinito

- LISTE CONCATENATE: INSERT & DELETE in $T(m) = O(1)$

SEARCH $O(1)$

ALBERI

struttura dati composta da: NODI, RADICI, FIGLI.

PROFONDITÀ: distanza tra un nodo x e la radice

IN-ORDER: LEFT[x] | x | RIGHT[x] (radice in mezzo)

LIVELLO: insieme dei nodi ad una determinata profondità

POST-ORDER: LEFT[x] | RIGHT[x] | x (radice alla fine)

ALTEZZA: profondità massima di un nodo in un albero

PRE-ORDER: x | LEFT[x] | RIGHT[x] (radice all'inizio)

BINARY SEARCH TREE

Ogni nodo ha al più due figli. Le operazioni di base richiedono un tempo proporzionale all'h dell'albero

(albero completo $\rightarrow \Theta(\log m)$)
(albero lineare $\rightarrow \Theta(m)$)

PROP: Sia x un nodo e key il suo valore: - y è un nodo nel sottoalbero sinistro di x, allora $y.key < x.key \quad value[y] > value[LEFT(x)]$

$value[RIGHT(x)] > value[x] > value[LEFT(x)] \quad$ - z è un nodo nel sottoalbero destro di x, allora $z.key > x.key \quad value[x] < value[RIGHT(x)]$

(è possibile ordinare didatticamente tutte le chiavi con IN-ORDER)

HEAP

Albero binario quasi completo, tutti i livelli sono completamente riempiti tranne eventualmente l'ultimo (riempito da sx)

Due tipi di heap: - MAX-HEAP: $A[\text{PARENT}(i)] > A[i]$ (elem più grande nella radice)

- MIN-HEAP: $A[\text{PARENT}(i)] \leq A[i]$ (elem più piccolo nella radice) + utilizzato per le code di priorità

Altezza albero $\Theta(\log n)$, le op. richiedono un tempo proporzionale all'altezza

CODA DI PRIORITÀ \rightarrow applicazione più diffusa dell'heap. Due tipi di cod: - MAX-PRIORITÀ, basato sul MAX-HEAP

Mantiene un insieme S di elementi con un valore associato (chiave)

- MIN-PRIORITÀ, basato sul MIN-HEAP

ALBERI ROSSO-NERI

Utili per bilanciare gli alberi di ricerca in modo da garantire che le op. richiedano $O(\log m)$.

bst con un bit di memoria in più per il colore.

- PROP:**
- la radice è meno
 - se un nodo è rosso entrambi i figli sono neri
 - ogni foglia NIL è nera
 - per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri

Altezza massima è $2 \log(m+1)$

SELF-BALANCING TREE (B-ALBERI)

Usato per la gestione dei dati in memoria secondaria. Possiamo avere più di due figli.

- PROP:**
- ogni nodo x ha i seguenti campi:
 - $x.m$ è il numero di chiavi correttamente memorizzate in x .
 - le $x.m$ chiavi sono memorizzate in ordine crescente t.c. $x.key_1 \leq x.key_2 \leq \dots \leq x.key_m$
 - $x.leaf$ è un campo booleano con valore true se x è foglia
 - ogni nodo interno x contiene anche $x.m+1$ puntatori ai suoi figli. I nodi foglia non hanno figli perciò i loro attributi ci non sono definiti
 - le chiavi $x.key_i$ separano gli intervalli delle chiavi memorizzate in ciascun sottoalbero
 - tutte le foglie hanno la stessa profondità, ovvero l'altezza dell'albero
 - ci sono limiti superiori e inferiori per il numero di chiavi che un nodo può contenere. Questi limiti possono essere espressi in termini di un intero $t \geq 2$ chiamato GRADO MINIMO del B-Albero:
 - ogni nodo, tranne la radice, deve avere almeno $t-1$ chiavi. Ogni nodo interno, tranne la radice, quindi ha almeno t figli. Se l'albero non è vuoto, la radice deve avere almeno una chiave
 - Ogni nodo può contenere al massimo $2t-1$ chiavi. Quindi, un nodo interno può avere al massimo $2t$ figli.
- Diciamo che un nodo è pieno se contiene esattamente $2t-1$ chiavi.

Il B-Albero più semplice si ha quando $t=2$. Ogni nodo interno ha 2, 3 o 4 figli. (Solitamente però t è molto più grande)

ALBERI DI INTERVALLI

Un intervallo chiuso è una coppia ordinata di numeri reali $[t_1, t_2]$, con $t_1 \leq t_2$. Un albero di intervalli è un albero rosso-nero che gestisce un insieme dinamico di elementi in cui ogni elemento x contiene un intervallo $\text{int}[x]$. Oltre all'intervallo $\text{int}[x]$, ogni nodo x contiene un valore $\text{max}(x)$ che è il massimo tra tutti gli estremi degli intervalli memorizzati nel sottoalbero con radice x .

ALBERI PER STATISTICHE D'ORDINE

È un albero rosso-nero con una informazione aggiuntiva in ogni nodo, $x.size$, ovvero il numero di nodi interni nel sottoalbero con radice in x , incluso x stesso. $x.size = x.left.size + x.right.size + 1$

Qualsiasi statistica d'ordine può essere determinata in $O(\log m)$, anche il lungo di un elemento è determinato in $O(\log m)$

TAVOLE HASH

TAVOLE A INDIZZAMENTO DI RETTO

Tecnica che funziona quando l'universo delle chiavi è ragionevolmente piccolo. Si rappresentano con un array dove ogni cella corrisponde a una chiave dell'universo. Supponiamo che due elementi mai possono avere la stessa chiave.

PROBLEMI:

- l'universo delle chiavi potrebbe essere troppo grande \Rightarrow inefficiente in termini di memoria

- l'insieme delle chiavi effettivamente memorizzate può essere estremamente più piccolo dello spazio allocato sprecando memoria

• TAVOLE HASH

Richiedono meno spazio di una tonda ad indirizzamento diretto, ma non utilizza l'indice dell'array come key ma una funzione hash per calcolare la cella della chiave K . Quindi diremo che $h(K)$ è il valore hash della chiave K .

SPAZIO RICHIESTO $\approx O(1)$, TEMPO PER UNA OP.: $\approx O(1)$

PROBLEMA: due chiavi possono essere mappate nella stessa cella \Rightarrow COLLISIONE

→ una soluzione potrebbe essere il CONCATENAMENTO, ovvero poniamo tutti gli elementi associati alla stessa cella in una lista concatenata.

FATORE DI CARICO: data una tonda hash T con m celle dove sono memorizzati n elementi, definiamo FATORE DI CARICO d di T il rapporto n/m , ovvero il numero medio di elementi memorizzati in una lista.

TIPI DI FUNZIONI HASH:

• HASHING UNIFORME SEMPLICE: qualsiasi chiave ha la stessa probabilità di essere mappata in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono mappate le altre.

→ la probabilità che scegliendo chiave a caso questa vada in cella i è $\frac{1}{m}$

⇒ se le collisioni sono ridotte con il concatenamento, una qualciasi ricerca richiede $O(1+d)$ nel caso medio nell'hashing uniforme semplice.

$$\text{se } m = O(n), \text{ allora } d = \frac{n}{m} = \frac{O(n)}{O(m)} = O(1)$$

• METODO DELLA DIVISIONE: una chiave K viene associata a una delle m celle prendendo il resto della divisione fra K e m : $h(K) = K \bmod m$
 $m \rightarrow$ numero PRIMO non troppo vicino a una potenza di 2

• METODO DELLA MOLTIPLICAZIONE: moltiplichiamo la chiave K per una costante A nell'intervallo $0 < A < 1$ ed estraiamo la parte frazionaria di KA , poi moltiplichiamo per m e prendiamo la parte intera inferiore del risultato. $h(K) = L.m - (KA \bmod 1)$ (parte da sinistra)

$m \rightarrow$ potenza di 2

• HASHING UNIVERSALE: invece di usare una singola funzione di hash, utilizza un insieme di funzioni di hash scelte casualmente da \mathcal{H} (collezione di funzioni)

→ scegliendo a caso una funzione da \mathcal{H} , la probabilità di collisione tra due chiavi K e L è $\leq \frac{|\mathcal{H}|}{m} \cdot \frac{1}{|\mathcal{H}|} = \frac{1}{m}$

• INDIRIZZAMENTO APERTO: ogni cella della tonda contiene un elemento dell'insieme o NIL.

→ il fattore di carico d non supera mai 1.

→ esclude i puntatori calcolando la sequenza delle celle da esaminare.

Vengono esaminate in successione le posizioni della tonda finché non trova una cella vuota in cui inserire la chiave.

• HASHING UNIFORME: ogni posizione nella tabella hash deve avere all'inizio la stessa probabilità di essere scelta per qualsiasi chiave.

Per l'implementazione vengono utilizzate delle approssimazioni accettabili come il doppio hashing.

• DOPPIO HASHING: usa una funzione hash della forma $h(K, i) = (h_1(K) + i h_2(K)) \bmod m$

→ h_1, h_2 funzioni ausiliarie

GRAFI

• TERMINOLOGIA

- **GRAFO** $G=(V,E)$ consiste di:
 - un insieme $V=\{v_1, \dots, v_n\}$ di vertici o nodi
 - un insieme $E=\{(v_i, v_j) : v_i, v_j \in V\}$ di coppie (non ordinate) di vertici (distinti) dette archi.
- **GRAFO DIRETTO**: - insieme $V=\{v_1, \dots, v_m\}$ di vertici o nodi
 - insieme $E=\{(v_i, v_j) : v_i, v_j \in V\}$ di coppie ordinate di vertici, dette archi diretti.
- m numero di vertici, m numero di archi
- **ADJACENTI**: due nodi collegati da un arco
- un arco (x,y) si dice incidente ad x e ad y (estremi)
- **GRADO** di un nodo: numero di archi ad esso incidenti.
- il GRADO DEL GRAFO $\Delta(G)$ è pari al massimo tra tutti i gradi dei suoi vertici
- un CAMMINO in $G=(V,E)$ tra una coppia di nodi (x,y) è una sequenza alternata di nodi e di archi in G che parte da x e arriva in y dove ogni arco è incidente ai due nodi tra cui è compreso nel cammino.
- LUNGHEZZA del cammino: numero di archi che lo compongono
 - | in caso di GRAFO DIRETTO il cammino deve rispettare il verso di orientamento degli archi e si chiama CAMMINO ORIENTATO
- **DISTANZA**: la lunghezza del più corto cammino (semplice) tra due nodi.
- **DIAMETRO**: massima distanza tra due nodi del grafo ha DIAMETRO INFINITO
- **GRAFO CONNESSO**: esiste un cammino tra ogni coppia di nodi (altrimenti si dice disconnesso)
- **CAMMINO CHIUSO**: cammino da un nodo a se stesso, se non contiene altre ripetizioni di nodi si dice CICLO.
- un grafo $H=(V',E')$ è un SOTTOGRAFO di $G=(V,E) \iff V' \subseteq V$ e $E' \subseteq E$
- SOTTOGRAFO DEGENERI di G : $H = \emptyset$ e $H = G$
- dato un grafo $G=(V,E)$ il SOTTOGRAFO INDOTTO da un insieme di nodi $V' \subseteq V$ è il grafo $H[V'] = (V', E')$ con $E' = \{(x,y) \in E : x, y \in V'\}$
- CRITERIO DI CONNESSIONE: Sia $G=(V,E)$ un grafo. Per ogni $X \subseteq V$, denotiammo con $f(X)$ il sottoinsieme di E contenente gli archi con un nodo in X e l'altro in V/X
 - Un grafo $G=(V,E)$ è CONNESSO se e solo se $f(X) \neq \emptyset$ per ogni $\emptyset \neq X \subseteq V$
 - **GRAFO TOTALMENTE DISCONNESSO** è un grafo $G=(V,E)$ tale che $V \neq \emptyset$ e $E = \emptyset$
 - **GRAFO COMPLETO**: è un grafo tale che per ogni coppia di nodi esiste un arco che li congiunge
 - un grafo senza coppi (archi da un nodo a se stesso né ARCHI PARALLELI (due archi che collegano la stessa coppia di nodi) può avere un numero di archi m compreso tra 0 e $m(m-1)/2 = O(m^2)$. Se il grafo è CONNESSO necessariamente $m \geq m-1 \Rightarrow m-1 \leq m \leq m(m-1)/2$
 - | combinazione necessaria
 - $m \geq m-1 \Rightarrow m = O(m^2)$
 - **GRAFO DENSO**: $m = \Theta(m^2)$
 - **GRAFO SPARSO**: $m = O(m)$
 - **GRAFO EULERIANO**: se e solo se contiene un cammino (non semplice) che passa una e una sola volta su ciascun arco in E .
 - **GRAFO CONNESSO EULERIANO**: se e solo se ha tutti i nodi di grado pari, oppure se ha esattamente due nodi di grado dispari
 - **ALBERO**: grafo connesso aciclico
 - **FORESTA**: grafo aciclico

- Sia $G = (V, E)$ un grafo. Le seguenti affermazioni sono equivalenti:

- 1) G è un ALBERO
- 2) G contiene un unico cammino tra ogni coppia di vertici
- 3) G è connesso, ma il grafo ottenuto rimuovendo un qualsiasi arco da E è disconnesso
- 4) G è aciclico, ma il grafo ottenuto aggiungendo un qualsiasi arco ad E contiene un ciclo
- 5) G è aciclico e $|E| = |V| - 1$
- 6) G è CONNESSO e $|E| = |V| - 1$

- ALBERO RADICATO, è un albero in cui uno dei vertici si distingue da tutti gli altri, tale vertice è detto RADICE.

- Siamo T un albero radicato con radice $r \in V$ e $x \in V$ qualsiasi vertice di T . Un ANTENATO di x è un qualsiasi vertice nell'unico cammino tra r e x . Se $y \in V$ è un antenato di x , allora x è un DISCENDENTE di y . Il SOTTOALBERO con radice in x (T_x) è l'albero indotto dai discendenti di x con radice in x . Se l'ultimo arco nell'unico cammino tra r e x è (y, x) , allora y è il PADRE di x e x è un FIGLIO di y .

- GRAFO BIPARTITO: grafo $G = (V = (A, B), E)$ t.c. ogni arco ha come estremi un modo in A ed un modo in B . Un grafo bipartito è COMPLETO se $\forall x \in A, y \in B, (x, y) \in E$

RAPPRESENTAZIONE DEI GRAFI

• LISTE DI ADIACENZA: permettono di rappresentare in modo compatto i grafi SPARSI ($|E| \ll |V|^2$)

- ovvero Adj di $|V|$ liste (una per ogni vertice)

- $\forall u \in V$, la lista $Adj[u]$ contiene tutti i vertici v t.c. $\exists (u, v) \in E$

↳ contiene tutti i vertici adiacenti a u in G → le liste rappresentano gli archi in G

- a: a) ORIENTATO: somma lunghezza liste di adiacenza $\Rightarrow |E|$

b) NON ORIENTATO: " " " " " $\Rightarrow 2 \cdot |E|$

- MEMORIA RICHIESTA $\Rightarrow \Theta(|V| + |E|)$

- possono rappresentare i GRAFI PESATI

- non c'è un modo veloce per determinare se un arco (u, v) è presente, si può solo controllare la lista di adiacenza di u .

• MATRICE DI ADIACENZA: si suppone che i vertici siano numerati $(1, 2, \dots, |V|)$

- consiste in una matrice $A = (a_{i,j})$ di dimensione $|V| \times |V|$ t.c.: $a_{i,j} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{negli altri casi} \end{cases}$

- MEMORIA RICHIESTA: $\Theta(|V|^2)$

- la matrice di un grafo non orientato è uguale alla sua trasposta. (infatti si può risparmiare memoria memorizzando solo gli elementi sopra e lungo la diagonale)

- possono rappresentare i GRAFI PESATI

VISITA IN AMPIEZZA

- per tenere traccia dei vertici già visitati colora i vertici di: BIANCO, vertici mai visitati
GRIGIO, potrebbero avere qualche modo bianco adiacente
NERO, vertici già scoperti e visitati completamente
- un vertice viene scoperto quando viene visitato la prima volta, a quel punto non è più Bianco
- il colore di ogni $u \in V$ è memorizzato nell'attributo $u.\text{color}$ e il predecessore di u in $u.\pi$. da distanza dalla sorgente s al vertice u calcolata è memorizzata in $u.d$. L'algoritmo utilizza una coda Q con schema FIFO per gestire i vertici grigi.

BFS(G, s)

```

1. for ogni vertice  $u \in G.V - \{s\}$           10. while  $Q \neq \emptyset$ 
2.      $u.\text{color} = \text{WHITE}$                   11.      $u = \text{DEQUEUE}(Q)$ 
3.      $u.d = \infty$                          12.     for ogni  $v \in \text{Adj}[u]$ 
4.      $u.\pi = \text{NIL}$                       13.         if  $v.\text{color} = \text{WHITE}$ 
5.      $s.\text{color} = \text{GRAY}$                  14.              $v.\text{color} = \text{GRAY}$ 
6.      $s.d = 0$                            15.              $v.d = u.d + 1$ 
7.      $s.\pi = \text{NIL}$                      16.              $v.\pi = u$ 
8.      $Q = \emptyset$                        17.              $\text{ENQUEUE}(Q, v)$ 
9.      $\text{ENQUEUE}(Q, s)$                 18.              $u.\text{color} = \text{BLACK}$ 
```

$$T(m) = O(|V| + |E|)$$

VISITA IN PROFONDITÀ

Dato un grafo G , DFS visita G sempre più in profondità fino a quando esistono vertici non scoperti. Se restiamo vertici da scoprire, allora uno di essi viene selezionato come nuova sorgente e la visita riparte da esso. Infatti il sottografo dei predecessori è formato da più alberi ed è definito foresta DF. È utile per:

- 1- ordinamento topologico
- 2- componenti fortemente connesse
- 3- ricerca di cicli

- i vertici vengono colorati durante la visita per fare in modo che gli alberi siano disgiunti:

- BIANCO, vertice non scoperto
- GRIGIO, vertice scoperto in fase d'ispezione
- NERO, la lista di adiacenza del vertice è stata completamente ispezionata

- ogni vertice v ha due informazioni temporali:

\leftarrow numero compreso

• $v.f$ registra il momento in cui la visita di v è completa (e colorato nero)

Fra 1 e $2|V|$ (quando viene scoperto il vertice + quando viene completata la visita)

DFS(G)

```

1. for ogni vertice  $u \in G.V$ 
2.      $u.\text{color} = \text{WHITE}$ 
3.      $u.\pi = \text{NIL}$ 
4.      $\text{time} = 0$  // var globale
```

$$T(m) = O(|V| + |E|)$$

DFS-VISIT(G, u)

```

5.     for ogni vertice  $u \in G.V$ 
6.         if  $u.\text{color} = \text{WHITE}$ 
7.              $u.d = \text{time}$ 
8.              $u.\pi = \text{NIL}$ 
9.              $u.\text{color} = \text{BLACK}$ 
10.             $u.f = \text{time}$ 
```

ORDINAMENTO TOPOLOGICO

E' un ordinamento lineare dei vertici tale che, se $(u,v) \in E$, allora u appare prima di v nell'ordinamento. Se il grafo contiene un ciclo \Rightarrow non ordinamento lineare.

TOPLOGICAL-SORT (G)

- call DFS(G) to compute finishing times $v.f$ for each vertex v
- as each vertex is finished, insert it onto the front of a linked list
- return the linked list of vertices

CAMMINI MINIMI (grafo aciclico)

DAG - SHORTEST-PATH (G, w, s)

- topologically sort the vertices of G
- INITIALIZE-SINGLE-SOURCE(G, s)
- for each vertex v , take $v.d = \infty$
- for each vertex $v \in G.Adj[u]$
- $RELAX(u, v, w)$

BELLMAN-FORD

Restituisce un valore booleano che indica se esiste o meno un ciclo di peso negativo raggiungibile da s: 1) Esiste: il problema non ha

BELLMAN-FORD (G, w, s)

- INITIALIZE-SINGLE-SOURCE(G, s)
- for $i=1$ to $|G.V|-1$
 - for each edge $(u, v) \in G.E$
 - $if v.d > u.d + w(u, v)$
 - return false
- $RELAX(u, v, w)$
- return true

INITIALIZE-SINGLE-SOURCE(G, s)

- for each vertex $v \in G.V$
- $v.d = \infty$
- $v.r = NIL$
- $s.d = 0$

RELAX (u, v, w)

- $if v.d > u.d + w(u, v)$
- $v.d = u.d + w(u, v)$
- $v.r = u$

SOLUZIONE

- NON ESISTE: l'algoritmo commette errori con pesi negativi.

DIJKSTRA

Risolve il problema dei cammini minimi da sorgente unica in un grafo pesato nel caso in cui tutti i pesi degli archi sono non negativi. Mantiene un insieme S di vertici i cui pesi finali dei cammini minimi dalla sorgente s sono stati già determinati. L'algoritmo seleziona ripetutamente il vertice $u \in V-S$ con la stima minima del cammino minimo, aggiunge u ad S e rilassa tutti gli archi che escono da u .

DIJKSTRA (G, w, s)

- INITIALIZE-SINGLE-SOURCE(G, s)
- $S = \emptyset$
- $Q = G.V$
- while $Q \neq \emptyset$
 - $u = EXTRACT-MIN(Q)$
 - $S = S \cup \{u\}$
 - for ogni vertice $v \in G.Adj[u]$
 - $RELAX(u, v, w)$

riga 1: inizializza i valori d e r di tutti i vertici

riga 2: inizializza S come insieme vuoto

riga 3: inizializza la coda di min-priorità Q in modo che contenga tutti i vertici in V

righe 4-6: un vertice u viene estratto da $Q = V-S$ e aggiunto all'insieme S . Il vertice u quindi ha la stima minima del cammino minimo di tutti i vertici in $V-S$

Righe 7-8: rilassiamo ogni arco (u, v) che esce da u e aggiorniamo la stima $v.d$ e il predecessore $v.p$ se il cammino minimo che arriva a v può essere migliorato passando per u .

$$T(m) = O(|E| \log |V|)$$

COMPONENTI CONNESSE DI UN GRAFO

CONNECTED-COMPONENTS (G)

1. for ogni vertice $v \in G.V$
2. MAKE-SET(v)
3. for ogni arco $(u, v) \in G.E$
4. if FIND-SET(u) ≠ FIND-SET(v)
5. UNION(u, v)

SAME-COMPONENT (u, v)

1. if FIND-SET(u) == FIND-SET(v)
2. return true
3. else return false

KRUSKAL

Trova un arco sicuro da aggiungere alla foresta in costruzione scegliendo, fra tutti gli archi che collegano due alberi qualsiasi nella foresta, un arco (u, v) di peso minimo.

MST-KRUSKAL (G, w)

1. $A = \emptyset$
2. for ogni vertice $v \in G.V$
3. MAKE-SET(v)
4. ordina gli archi di $G.E$ in senso non decrescente rispetto al peso w
5. for ogni arco $(u, v) \in G.E$, preso in ordine di peso non decrescente
6. if FIND-SET(u) ≠ FIND-SET(v)
7. $A = A \cup \{(u, v)\}$
8. UNION(u, v)
9. return A

$$T(m) = O(|V| \cdot \log |E|)$$

Righe 1-3: inizializza l'insieme A come un insieme vuoto e crea $|V|$ alberi, uno per ogni vertice.

Righe 5-8: esamina gli archi nell'ordine dal più leggero al più pesante. Il ciclo verifica, per ogni arco (u, v) , se l'estremità u e v appartengono allo stesso albero; in caso affermativo, l'arco (u, v) non può essere aggiunto alla foresta senza generare un ciclo, quindi l'arco viene scartato. Altrimenti i due vertici appartengono ad alberi differenti. In questo caso, l'arco (u, v) viene aggiunto ad A nella riga 7 e i vertici dei due alberi vengono fusi nella riga 8.

Quale delle seguenti sequenze di funzioni è tale che ogni funzione è O della successiva?

A - $\log_2 m$, $\log_{10} m$, $m(\log_2 m)^2$, $m^{\log_2 m}$, $2^{\sqrt{m}}$

B - $(\log m)^{2024}$, \sqrt{m} , $m \log m!$, $m^{\frac{\log_2 3}{2}}$, m^2

C - ~~nessuna~~