

ALGORITMI E STRUTTURE DATI

By Edoardo

Qualità algoritmi = efficiente.

Lo risulta più "pregiata" che deve ottimizzare un algoritmo è **IL TEMPO**

Col tempo lo spazio di memoria è passato in secondo piano rispetto al tempo di elaborazione.

- ANALISI ALGORITMO (Tempo e Spazio)

N.B. In casi così lo spazio passa in primo piano (limiti fisici di spazio es. IoT)

Abbassa i costi di elaborazione

- ALGORITMI FONDAMENTALI

- OTTIMALITÀ

- STRUTTURE DATI

- RICORSIONE

NO SIDEFFECT! (tranne se richiesto)

ED Funzione Ricorsiva Fibonacci

```
int fibonacci (int n)
    if (n==0) return 0;
    if (n==1) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
```

N.B. Non LINEARE perché ha più chiamate ricorsive

Ricorsione di Coda: quando la chiamata della ricorsione è in fondo al codice dc può essere sostituita con un ciclo

ES.

```
reverse - array (A, i, j)
    if (i < j){
        "SCAMBIO i con j";
        reverse - array (A, i+1, j-1);
    }
```



```
while (i < j){
    "SCAMBIO i con j";
    i++;
    j--;
}
```

N.B Se vuoi togliere la ricorsione in alcuni casi va aggiunta una nuova struttura dati (Pila)

ES.

```
pow (a, n)
    if (n==0) return 1
    return a * pow (a, n-1)
```

RICORSIONE BINARIA

Ho due ricorsioni per ciascun caso non base
chiamate esponenti.

Tecnico Riconoscimento (esso è ricorsione binaria nida)

totali chiamate ricorsive

$$T(n) = \begin{cases} n-1 & 1 \\ n > 1 & 2T\left(\frac{n}{2}\right) = 2 \left(2T\left(\frac{n}{2^2}\right)\right) = 2^2 T\left(\frac{n}{2^2}\right) \dots \end{cases}$$

$$T(n) = 2^k + \left(\frac{n}{2^k}\right)$$

$$\frac{n}{2^k} = 1 \quad k = \log_2 n$$

$$= 2^{\log_2 n} + \boxed{1} = \boxed{n}$$

RICORSIONE MULTICO

Più di due chiamate per ogni passo non base

Per i casi più complicati si usano cicli per gestire le chiamate.

ALGORITMI

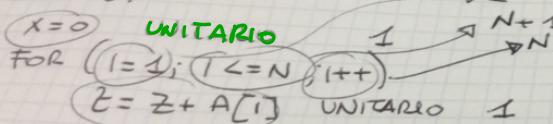
Un algoritmo è una procedura passo passo per risolvere un problema in tempo finito.

Il tempo dipende dalla dimensione dell'input (**ANALISI IN FUNZIONE DELLA DIM. DELL'INPUT**)

Nel caso in cui l'argomento "tende a ∞ " si parla di **ANALISI ASINTOTICA**. Non ha senso studiare il comportamento dell'algoritmo con input "piccoli". La funzione che descrive l'algoritmo in questo caso è asintotica (funzione di costo). L'**ANALISI DEL CASO PEGGIORE** studia il numero di passi nel caso peggiore per capire il costo massimo dell'algoritmo (**ANALISI DEL CASO PEGGIORE \Rightarrow WORST CASE \Rightarrow W.C.**)

L'**ANALISI DEI COSTI UNIFORMI** ipotizza che ogni passo elementare abbia lo stesso costo (assegnazione variabile = 1, addo $k = k + 1$ ecc.)

ESEMPIO:



$$\begin{array}{c}
 1 \\
 N \\
 1 \\
 N \\
 N+1
 \end{array} = 3N + 3 \sim N
 \stackrel{\approx}{\downarrow} \text{ASINTOTICO}$$

Considerando i costi uniformi si introducono errori e quindi eliminiamo le costanti moltiplicative.

Se $Z = \dim \text{input} = \log_2 n$ e il costo = $\frac{\sqrt{n}}{2}$

$$n = 2^Z$$

$$\sqrt{n} \Rightarrow \sqrt{2^Z} = \boxed{2^{Z/2}} \Rightarrow \text{Costo in funzione della dimensione dell'input}$$

O-grande

$f(n) \in O(g(n))$ se esistono due costanti positive c e n_0 tali che $f(n) \leq c g(n)$ per $n \geq n_0$

$$2 \cdot 2^{n+10} \in O(n)$$

$f(n) \in O(g(n))$ significa che il tasso di crescita di $f(n)$ non supera quello di $g(n)$ **UPPER BOUND**

Ω -grande

$f(n) \in \Omega(g(n))$ se esiste una costante $c > 0$ e una costante intera $n_0 \geq 1$ tali che $f(n) \geq c \cdot g(n)$ per $n \geq n_0$

$$\Rightarrow f(n) \in O(g(n)) \text{ allora } g(n) \in \Omega(f(n))$$

$f(n) \in \Omega(g(n))$ significa che il tasso di crescita di $f(n)$ è \geq di quello di $g(n)$ **LOWER BOUND**

Θ -grande

$f(n) \in \Theta(g(n))$ se esistono due costanti $c' > 0$ e $c'' > 0$ e una costante intera $n_0 \geq 1$ tali che $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ per $n \geq n_0$

$$\Rightarrow f(n) \in \Theta(f(n)) \text{ allora } g(n) \in \Theta(f(n))$$

Θ -grande oltre a dare un upper bound specifica anche che lo raggiunge **SICURAMENTE**

— ISTRUZIONE DOMINANTE

Determina il costo asintotico dell'algoritmo.

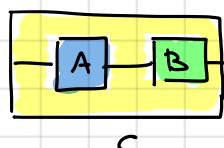
Esiste sempre! e basta trovarne una di tutte quelle che potrebbero esserci: è sempre la funzione seguita più volte

$p(n)$ è il costo dell'istruzione dominante ed è $\Theta(f(n))$

— OUTPUT SIZE

Problema: dato $\mathcal{D}[x]$ ordinato, stampare i numeri nell'intervallo $[x, y]$ RANGE QUERY (non lineare)

Soluzione: ricerca binaria con costo $\log(n)$ per trovare il primo elemento + k costo per mandare in output gli elementi da stampare $\Rightarrow \mathcal{O}(\log(n) + k)$



Quanto costa c se l'input di B è l'output di B?

C

$$f_c = f_a + f_b \Rightarrow f_c(n) = f_a(n) + f_b(n)$$

$$f_c(n) = f_a(n) \dots$$

perché stesso input

Algoritmo come funzione matematica: $A(x) = Y$ dove x è l'input e Y è l'output

FUNZIONE DI DILATAZIONE: $\frac{|y|}{|x|}$

$$\Rightarrow \delta_A(n) = \max \left\{ \frac{|y|}{|x|} \right\} \text{ con } |x|=n$$

$$f_c(n) = f_A(n) + f_B(n \delta_A(n))$$

Costo chiamata ricorsiva LINEARE

$$\begin{cases} T(n) = C' + T(n-1) \\ T(0) = C'' \end{cases} \text{ ricorsione}$$

$$T(n) = C + T(n-1) = C + C + T(n-2) = 2C + T(n-2) = 2C + C + T(n-3) = \dots = kC + T(n-k)$$

$$(quando k=n) C'n + T(0) = C'n + C'' \in \Theta(n)$$

$$z = |n| = \log_2(n) \Rightarrow n = 2^z \rightarrow \Theta(2^z)$$

Nel caso l'input sia un numero, la sua dimensione sarà la sua rappresentazione binaria $\Rightarrow z = \log_2 n$

Nel caso di un array la sua lunghezza.

COSTO ricorsione KUOTPLA

```

if((i < 0) || (j < 0)) return false;
if((i > mat.length - 2) || (j > mat[0].length - 2)) return false;
int p = mat[i][j], nord = mat[i-1][j], est = mat[i][j+1],
    sud = mat[i+1][j], ovest = mat[i][j-1];
if((p > nord) && (p > sud) && (p < est) && (p < ovest)) return true;
if((p < nord) && (p < sud) && (p > est) && (p > ovest)) return true;
return false;
}

static int contaPDS(int[][] mat, int up, int right, int down, int left) {
    if((up > down) || (left > right)) return 0;
    if((up == down) && (left == right))
        if(isPDS(mat, up, left)) return 1; else return 0;
    int o = (up+down)/2, v = (left+right)/2;
    return contaPDS(mat, up, v, o, left) + contaPDS(mat, up, right, o, v+1) +
        contaPDS(mat, o+1, v, down, left) + contaPDS(mat, o+1, right, down, v+1);
}

static int contaPDS(int[][] mat) {
    return contaPDS(mat, 0, mat[0].length, mat.length, 0);
}

```

$$T(m, n) = \begin{cases} C & m \leq 1, n \leq 1 \\ T\left(\frac{m}{2}, \frac{n}{2}\right) + T\left(\frac{m}{2}, \frac{n}{2}\right) + T\left(\frac{m}{2}, \frac{n}{2}\right) + T\left(\frac{m}{2}, \frac{n}{2}\right) + C'' & \text{1}^{\text{a}} \text{diomata} \quad \text{2}^{\text{a}} \text{diomata} \quad \text{3}^{\text{a}} \text{diomata} \quad \text{4}^{\text{a}} \text{diomata} \end{cases} = 4T\left(\frac{m}{2}, \frac{n}{2}\right) + C'' \Rightarrow \text{costi uguali asintoticamente } \left(\frac{m}{2} = \frac{n}{2} + 1\right)$$

$$T'(z) = \begin{cases} C''' & \\ 4T'\left(\frac{z}{2}\right) + C''' & \end{cases}$$

$$T(m, n) = 4 \underbrace{T\left(\frac{m}{2}, \frac{n}{2}\right)}_{4T\left(\frac{m}{2}, \frac{n}{2}\right) + C} + C = 4^2 \underbrace{T\left(\frac{m}{2^2}, \frac{n}{2^2}\right)}_{9T\left(\frac{m}{2^2}, \frac{n}{2^2}\right) + C} + 4C + C = 4^3 T\left(\frac{m}{2^3}, \frac{n}{2^3}\right) + 4^2 C + 4C + C = \dots = c \sum_0^{\infty} 4^i$$

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

Ipotizzando $m \leq n$ prendiamo che quello che ormai prima a 1 (caso base) è m

$$\frac{m}{2^k} = 1 \Rightarrow k = \log_2 m \quad (2)^{\log_2 m} - (2^{\log_2 m})^2 \Rightarrow 2^{\log_2 m} = m \Rightarrow m^2$$

$$T(m, n) = (2)^{\log_2 m} C' + C'' \cdot \frac{4^{\log_2 m} - 1}{3} = C' m^2 + C'' \frac{m^2 - 1}{3} = \Theta(m^2)$$

$$T'(z) = 4^k T\left(\frac{z}{4^k}\right) + C \sum_{i=0}^{k-1} 4^i = 4^{\log_4 z} C' + C'' \sum_{i=0}^{\log_4 z - 1} 4^i = \frac{4^{\log_4 z} - 1}{3} = \frac{z - 1}{3}$$

$$\sum_0^p q^i = \frac{q^{p+1} - 1}{q - 1}$$

Serie geometrica

UPPERBOUND DI UN ALGORITMO A

$$O(f(n)) \rightarrow C_A(n) \in O(f(n))$$

Entrambi per il caso peggiore

LOWERBOUND DI UN ALGORITMO A

$$\Omega(f(n)) \rightarrow C_A(f(n)) \in \Omega(f(n))$$

Un problema P ha U.B. $O(f(n))$ se $\exists A$ che risolve P e A ha U.B. $O(f(n))$.

P ha L.B. $\Omega(g(n))$ se $\forall A$ che risolve P, A ha L.B. $\Omega(g(n))$

ALBERI

Sono una particolare categoria di grafo

È un modello astratto di struttura gerarchica

Il modo principale è detto **radice**.

Nodi senza figli sono detti **follia** (atomi)

N nodi $\rightarrow n-1$ rami

Un albero **K-ario** è un albero con massimo K figli (arci a K)

Nod con lo stesso padre sono detti **fratelli**

Nodi con figli sono detti nodi interni

Il concetto di **visita** indica in che modo è possibile esplorare un albero (ordine)

ALTEZZA = lunghezza ramo più lungo (radice - foglia)

PROFOUNDITÀ = distanza nodo - radice

LEVEL = insieme di nodi allo stesso profondità

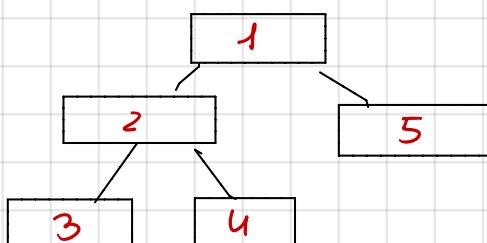
Le visite si dividono in visite di profondità (scenduglio) che privilegiano la scissione verticale e visite di ampiezza (ventaglio) che privilegiano lo scorrimento orizzontale [DEPTH FIRST SEARCH (DFS) e BREADTH FIRST SEARCH (BFS)]

Esempi visita

- VISITA IN PREORDINE (DFS)

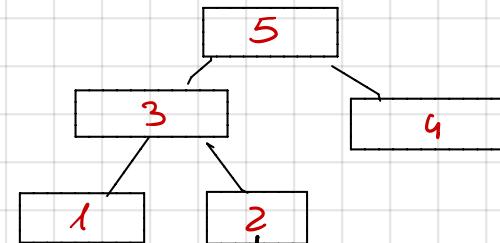
Ad ogni nodo (partendo dalla radice)

richiama la funzione ricorsiva per ogni figlio (una sola volta)



- VISITA IN POSTORDINE (DFS)

Un nodo viene visitato dopo tutti i suoi figli (ma solo volta)



Algoritmo di visita costa $\Theta(n)$!

$$T(n) = \begin{cases} c & n=1 \\ ? T(?) + c & n>1 \end{cases} \Rightarrow \text{Non è possibile scrivere la relazione di ricchezza} \\ \# \text{ ricorsioni} = \# \text{ figli} \quad \text{input}$$

Se non ho vincoli si utilizzano altri strumenti di analisi!

COSTO IN SPAZIO

Sicuramente lo spazio dell'albero

+ variabili [adi ex] (costante in spazio e non dipendente dall'input): qualsiasi sia il numero delle variabili a noi non importa perché non dipende dall'input.

+ spazio di "storage" che dipende dello spazio di input (array temporaneo da tenere tutti i valori di input, copie di input ecc)

Algoritmi che usano questo tipo di spazio extra sono algoritmi **NON IN PLACE** (in place altrimenti)

Soltanmente algoritmi ricorsivi non sono in place perché salvano in stack gli elementi che esaminano per arrivare al return finale.

Max stack = altezza albero

Algoritmi BFS solitamente non sono ricorsivi e non hanno spazio extra in uso (riguardo all'input)

Lo spazio può dipendere anche dal linguaggio (per albergare un array in C devo fare realloc e "usare 2 volte lo spazio")

ALBERI BINARI

Un albero binario è nullo o un albero lo cui lo radice ha un figlio destro e uno sinistro (ognuno di quelli li può avere a sua volta)



Esempi di alberi binari sono gli **expression trees**. Per calcolarli usiamo DFS in postordine. (alberi "modelli")

Gli alberi binari sono alberi 2-ari particolari: quelli binari fanno una distinzione netta tra figlio destro e sinistro

Un altro esempio sono gli **alberi di decisione** in cui al nodo interno è associata una domanda che accetta come risposta sì o no e nei nodi esterni le risposte (decisioni)

ALBERO COMPLETO: k-ario, tutti i livelli hanno il massimo numero di nodi

- Albero binario

$$\# \text{ nodi} = \sum_0^h 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1 = n \quad \text{dove } h \text{ è l'altezza dell'albero}$$

$$h = \log_2(n+1) - 1$$

- Albero k-ario

$$\# \text{ nodi} = \sum_0^h k^i = \frac{k^{h+1}-1}{k-1} = n$$

$$h = \log_k(n+1) - 1$$

ALBERO PIENO: ogni nodo ha 0 o k figli (condizione necessaria, ma non sufficiente per essere completo)

$$\# \text{ foglie} = (k-1) n_{\text{interni}} + 1$$

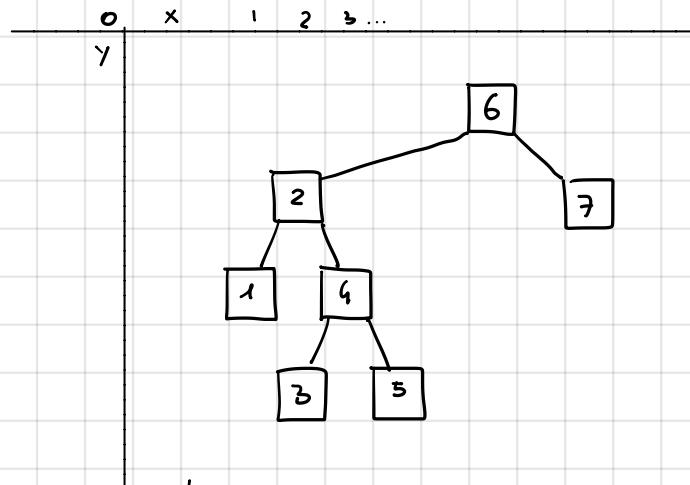
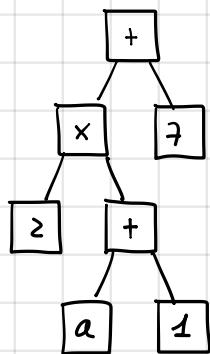
$$n = n_f + n_i$$

Un albero è QUASI completo se al massimo l'ultimo livello non lo è.

VISITA IN ORDINE (solo binari)

Un nodo è visitato dopo il suo sottoalbero sinistro e prima del suo sottoalbero destro.

Applicazione: stampa di un expression tree.



L'ordine è dato dalla x

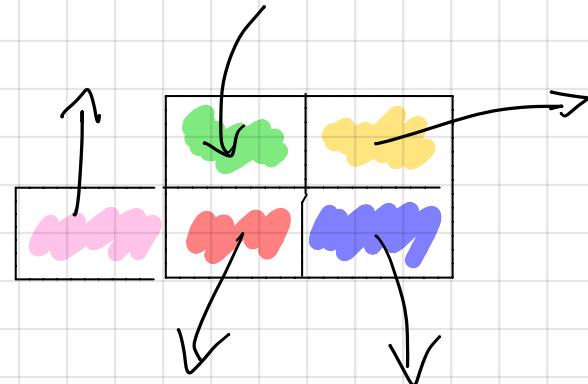
STRUCT / CLASSE NODO (binario)

key (int...)

Puntatore (inutile per quanto riguarda l'algoritmo)

LeftChild

RightChild



Puntatore Parent (qualsiasi inferiore, usare solo se necessario)

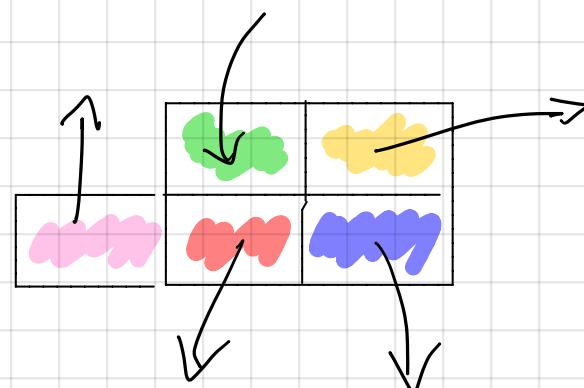
STRUCT / CLASSE NODO (k-ario)

key (int...)

Puntatore (inutile per quanto riguarda l'algoritmo)

FirstChild

NextSibling



Puntatore Parent (qualsiasi inferiore, usare solo se necessario)

Nel secondo caso dato un nodo non mi interessa andare direttamente a un figlio specifico, ma avrò bisogno partendo dal primo figlio: posso al primo costo $\Theta(1)$, arrivare al k-esimo $\Theta(k)$

Se necessito invece posso direttamente al figlio k si utilizza un solo puntatore che punta ad un array di puntatori che puntano ai figli, o un singolo array come la radice al primo posto e i figli del nodo i-esimo nella posizione $2i + 1$ e $2i + 2$

VISITA CON CAMMINO EULERIANO

Vista ogni nodo 3 volte: preordine, inordine, postordine

CODE DI PRIORITÀ

Memorizziamo una collezione di elementi.

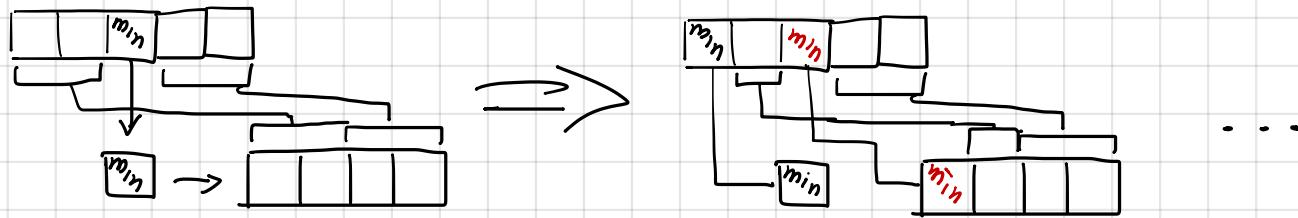
La priorità è espressa da un elemento di un insieme che ha relazione d'ordine totale (es. numeri). Gli elementi in entrata nello code (entry) sono coppie chiave - valore.

Formulo GAUSS:

$$\sum_{i=1}^k i = \frac{k(k+1)}{2}$$

Possiamo usare lo code di priorità per ordinare una lista di elementi confrontabili:

- Selection - Sort: code di priorità non ordinata. $\Theta(n^2)$



Tale algoritmo deve operare in place "suddividere" la sequenza non ordinata in due parti: la prima parte sarà composta dagli elementi che non sono ancora ordinati e la seconda da quelli già ordinati. L'algoritmo seleziona di volta in volta l'elemento più piccolo della sequenza non ordinata e lo sposta in quella ordinata.

- Insertion - Sort

Simile al precedente, lavora anch'esso in place. Il concetto di fondo è lo stesso su cui si ordina un motivo di codice: la prima parte dell'array sarà la parte ordinata e la seconda non ordinata. Si prende (solitamente a caso) un elemento dalla sequenza non ordinata e lo si inserisce nella giusta posizione in quella ordinata.

Il primo scambia il min/max della sequenza non ordinata con l'elemento subito dopo la sequenza ordinata.

Il secondo invece per ogni elemento si accorta che sia nella giusta posizione, in caso contrario "lo scolpisce" la sequenza e inserisce l'elemento al giusto posto.

Heap

Lo heap è rappresentato come un albero completo (o quasi) ed è composto da sinistra verso destra, inoltre è una memoria concreta. Lo chiave rappresenta la priorità e deve essere minore scendendo verso il basso. Il padre ha priorità più bassa rispetto ai figli e la priorità più alta è rappresentata dal valore più basso [MIN HEAP]. MAX HEAP invece è il caso in cui i valori più grandi hanno maggiore priorità. Quando inserisco un nuovo nodo nell'heap, il nodo va inserito nella prima posizione libera del livello. Se il livello è pieno va inserito nel primo nodo del livello successivo.

- Ogni sotto albero è un heap
- Se devo scambiare un nodo in una MIN HEAP, faccio lo scambio sempre con il fratello con valore minore, per cercare di lavorare sempre sullo stesso

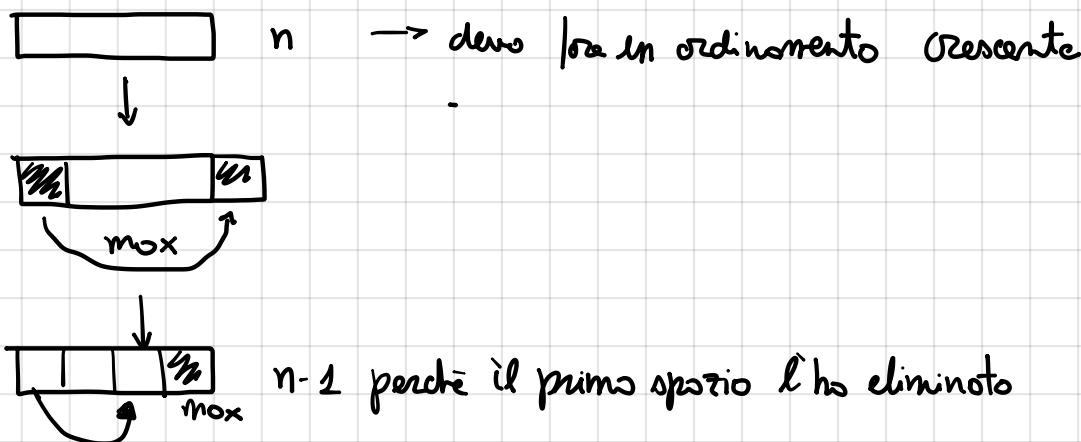
IMPLEMENTAZIONE HEAP IN ARRAY

Un heap di n livelli si rappresenta su un array di $n+1$ elementi.

Per il nodo n , il figlio sx è $2n$ mentre quello dx è $2n+1$.

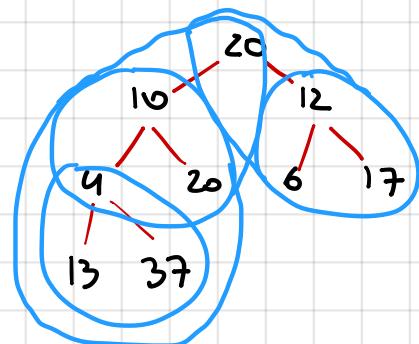
L'indice 0 dell'array non è usato per i nodi, ma per inserirci le dimensioni dell'heap.

HEAP-SORT



copio quindi il max in una variabile d'appoggio e lo copio nel primo spazio vuoto.

ORDINARE HEAP



Ordino tutti i min heap a partire dal basso
Prendo il primo nodo non foglia e lo ordino come min heap. Continuo così lavorando su tutti i nodi fino all'ultimo. Al termine avrò l'heap completo.

N.B. Ogni volta che riordino un min heap devo sempre controllare che quello del livello più basso sia ordinato e in caso non lo fosse faccio gli opportuni scambi.

-Heapify (min)

Heapify(A, i)

$l = \text{left}(i)$

$r = \text{right}(i)$

IF $l \leq \text{heapsize}(A) \ \& \ A[l] < A[i]$

$\min = l$

ELSE $\min = i$

IF $r \leq \text{heapsize}(A) \ \& \ A[r] < A[\min]$

$\min = r$

IF $\min \neq i$

scambia $A[i]$ con $A[\min]$

Heapify(A, \min)

(Questa funzione ausiliaria serve per mantenere le proprietà della heap partendo dall'array dell'array (può essere implementata direttamente sulla heap) controllando (nel caso min heap) che la radice sia minore dei suoi figli, in caso contrario lo scambio con il minore dei suoi figli. Dopo il primo passaggio continua tenendo in "memoria" l'indice con valore minimo (\min).

È usata per mantenere in ordine la heap.

HEAPSORT

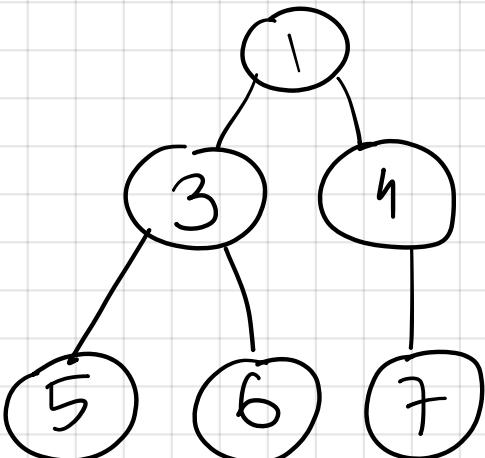
- Costruisce heap:

Dato un array, heapify MAX sullo prima metà degli elementi, heapsize = # elementi

- Per ogni elemento, dalla fine, lo scambio con il primo, decremento heapsize e eseguo heapify MAX sul modo che ho scambiato

BUILD (HPY)	5 7 3 6 4 1 5
SWP	7 5 6 4 1 3
HPY	3 5 6 4 1 7
SWP	6 5 3 4 1 7
HPY	1 5 3 4 6 7
SWP	5 1 3 4 6 7
HPY	4 1 3 5 6 7
SWP	6 1 3 5 6 7
	3 1 4 5 6 7

heapsize = 2
 $k > 2$ non lo conto! (4)



HPY	3 1 4 5 6 7
SWP	1 3 4 5 6 7
	INVARIATO

$n=1$ FINITO

MAPPA = dizionario python senza duplicazioni di chiavi

TABELLA HASH

In principio si voleva trovare una Regola che inserisse ogni identificatore in una casella di una tabella. Tale Regola (funzione) doveva avere costo $\Theta(1)$.

$$R: X \rightarrow T \text{ (tabella: array)}$$

Guesto creava problemi poiché i due insiemi potevano avere cardinalità diverse e non esiste una funzione che possa creare una "connessione" biunivoca.

La soluzione è stata usare una Regola biettiva chiamata **FUNZIONE DI HASH**

Potranno però avvenire delle **COLLISIONI** ovvero due identificatori (chiavi) nello stesso casellino: adesso bisogna trovare una soluzione alle collisioni, ma in caso non le avessi ho eseguito le operazioni di hashing in tempo costante

Cosa mi serve sapere?

- 1) Dimensione tabella
- 2) Algoritmo risolutivo collisioni
- 3) Funzione di hashing che genera **PSEUDOCHIAVI** ovvero lo casellino della tabella.

Una volta inserite in tabella devo poter eseguire **get**, **put** e **remove**.

ARRA Y ORDINATO: preserverà l'adiacente chiavi (**LOCALITA'**)

TABELLA HASH: se l'otto per bene non preserverà l'adiacente delle chiavi

Soltanto una funzione hash è composta da due funzioni:

h_1 = genera un intero

h_2 = fa sì che l'intero entri nell'intervallo $[0, N-1]$ delle caselle.

$$h(x) = h_2(h_1(x))$$

I codici hash polinomiali sono molto frequenti: si calcola un polinomio per ogni chiave. Tali polinomi possono essere "risolti" in tempo $\Theta(1)$ grazie all'algoritmo di Horner.

-GESTIONE COLLISIONI

- **CONCATENAZIONE SEPARATA**: a ogni celletta (con collisione) viene associata una lista collegata di chiavi che vengono mappate a quella celletta.

Contro: richiede spazio aggiuntivo

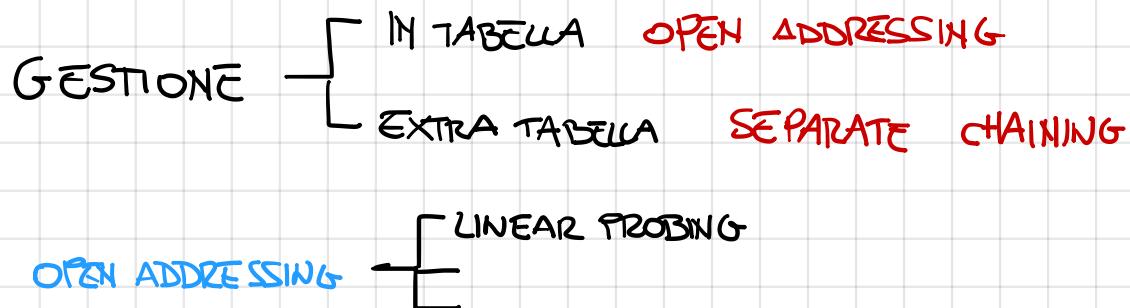
Pro: facile da implementare all'esterno della tabella

La tabella si trasforma in una pseudo-matrice di puntatori

Costo lineare per scandire la lista su **get**, **set**, **remove**!

Costo medio costante poiché la lunghezza è fissata da una costante.

Il costo medio è riducibile se la tavola non è TROPPO PIENA,
 $\text{LOADFACTOR} = \frac{n}{N} = \frac{\# \text{ KEYS}}{\text{TAVOLA}}$ < 70 %



LINEAR PROBING: Scansione tabella finché non trova uno casello libero
 Costo costante se ho poche caselle piene prima di quello vuoto.

Con questo procedimento di gestione possono avere **CLUSTERING PRIMARIO**
 ovvero, più lo tabella ovù zone occupate, maggiore sarà la
 probabilità di collisione. Il clustering non dipende dal fatto se le zone
 occupate sono contigue, ma dall'algoritmo di scansione.

Pur gestire lo concellazione di una chiave invece, non basta trovare lo
 casello giusto e cancellare il contenuto perché intal caso spetterei la
 sequenza di scansione. Per fare ciò si usa un array di flags che indici
 per ogni celo se lo parte o meno della sequenza: così facendo posso
 cancellare il contenuto, ma mantenere la sequenza.

ESEMPIO RICHIESTA

Se elimino una chiave all'interno di una sequenza senza usare
 i flag posso avere il seguente problema. Se in precedenza avevo
 inserito due chiavi nello stessa celia, lo secondo scelava fino allo
 primo libero A REMOVED B. In caso di ricerca di B, l'algoritmo
 arriverebbe prima in A, ma non trovando B, scelto a destra: se a
 destra è vuoto (come mai utilizzato = senta flag) la ricerca fallisce.
 Con i flag invece l'algoritmo capisce di dover proseguire.

(RE-HASHING)

HASHING DOPPIO : in caso di collisione uso un'altro funzione di
 hashing. Lo secondo funzione viene usata per generare il posso di scansione!
HASHING QUADRATICO : si utilizza una funzione polinomiale
 $K(i) = \text{polinomio che dipende da } h_i^2$. Il posso vorreto cost di un fattore
 quadratico.

CLUSTERING SECONDARIO: due chiavi con lo stessa chiave collidono
 e hanno lo stessa sequenza di scansione. Piuttosto al paronio è
 trascurabile.

DICTIONARIO

Al contrario della tavola hash, il dizionario accetta chiavi duplicates. Le operazioni base sono comunque ricerca, inserimento e cancellazione. Avendo la possibilità di più chiavi uguali oltre al metodo find base ho anche il metodo findAll() che restituisce un iteratore per tutte le chiavi uguali concrete.

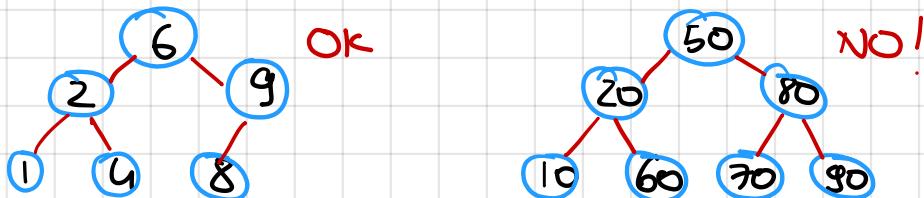
Dizionari e Mappe possono essere ordinate.

ALBERO BINARIO DI RICERCA (BST)

Un albero binario che memorizza le chiavi nei nodi interni.

Deve soddisfare queste proprietà:

se dato un nodo la sua chiave sarà sempre maggiore delle chiavi del suo sottoalbero sinistro e minore di quelli nel sottoalbero destro.



In vista in ordine di questo albero incontrerà le chiavi in maniera ordinata.

Data la sua proprietà, per trovare una data chiave BASTA LAVORARE SU UN SOLO RAMO

TreeSearch (int k, node v) {

```
if (v == null) return null;  
if (v.key == k) return v;  
if (v.key > k) return TreeSearch(k, v.left);  
else return TreeSearch(k, v.right);}
```

$\Theta(h)$

$h = \text{altezza albero}$

N.B. allo stesso insieme di chiavi possono corrispondere diversi alberi.



L'algoritmo di inserimento è uguale a quello di ricerca aggiungendo l'inserimento come foglia.

Connessione nodo

- ① v - figlia
② v ho 1 figlio
③ v ho 2 figli

- ① Metto a null il puntatore del padre di v a v. $\Theta(h)$

② Cambio il puntatore del padre di v al figlio di v. $\Theta(h)$

③ ④ Cerco predecessore di v ovvero il max del suo sottoalbero di sinistra.

⑤ Scriv il predecessore al posto di v

⑥ Cancello predecessore

- come trovo il predecessore?

Socò il nodo più a destra del sottocollo sinistro di ventra figlio
destra.

L'algoritmo di ricerca semplicemente partendo dallo root "scende" l'albero: se lo chiude del modo \rightarrow allo chiude di v scendo a sinistra altrimenti a destra e mi segno il predecessore corrente

RANGE - QUERY - COUNT

```

if(v == NULL) return 0
C = 0
if(v.key >= a) [ ] [ ]
    C = IN-ORDER(v.left, --)
if(v.key ∈ [a, b]) C += 1, [ ] [ ]
    if(v.key <= b)
        C += IN-ORDER(v.right, --)
return C

```

Costo $\Theta(n+k)$ $k = \#$ chiavi

IPL: internal path length = somma lunghezza percorsi nodo - radice per tutti i nodi.

$$\text{LUNGHEZZA MEDIA} = \text{IPL} / \# \text{nodi}$$

key	4	8	10	→ RANK	1	2	3
key	8	4	10	→ RANK	2	1	3

Sostituire le chiavi con il loro rank non cambia nulla.

Se ho n chiavi in n nodi con $P_n(i) =$ lungh. perc. medio ch. n con i come radice ($\text{IPL}/\# \text{nodi}$)

$$P_n = \text{IPL} / \# \text{nodi}$$

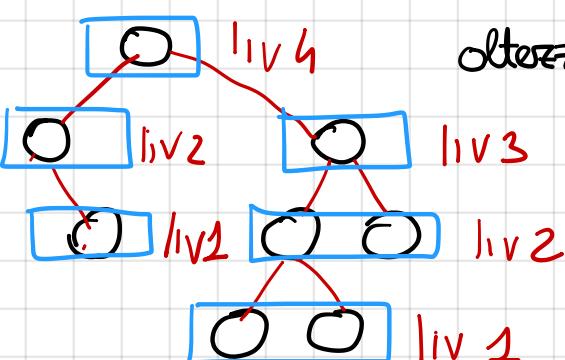
$$P_n(i) = \frac{(P_{i-1} + 1)(i-1) + (P_{n-i} + 1)(n-i)}{n}$$

[Se $k(\text{radice}) = i$ allora:
sottoalbero sx ha $i-1$ nodi,
sottoalbero dx ha $n-i$ nodi.]

$$P_n = \frac{\sum_i^n P_n(i)}{n} = O(\log n) \rightarrow \Theta(\log n)$$

ALBERI AVL

Sono alberi **BIANCIATI**. Sono alberi binari di ricerca tali che per ogni nodo interno, le altezze dei suoi sottosalberi figli possono differire al massimo di 1 (o i livelli)



Fattore di Bilanciamento =

altezza sottosalbero destro - altezza sottosalbero sinistro

Tale fattore deve essere al più 1

I nodi intorno ± 1

ALBERI DI FIBONACCI: alberi AVL con il minimo numero di nodi (finita l'altezza)

(In un albero AVL di n nodi ho altezza $\Theta(\log n)$)

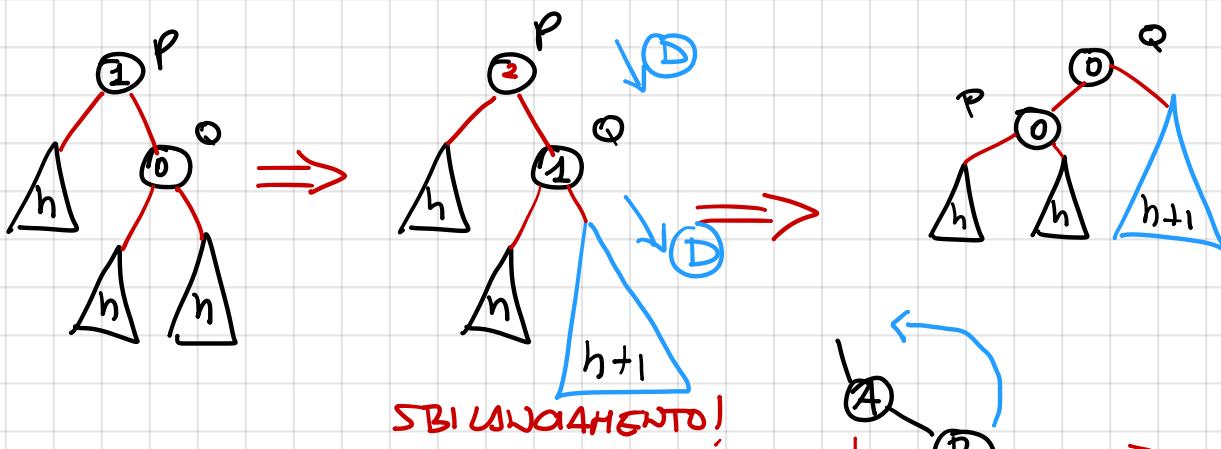
h	F_h	AVL_h
0	0	0
1	1	1
2	1	2
3	4	4
4	7	7
5	12	12
6	20	20
7	33	33

Inserimento in un AVL

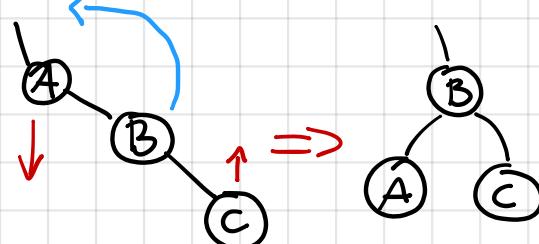
L'inserimento in caso di sbilanciamento lo crea solo nel suo ramo. Per eliminare tutti gli sbilanciamenti provocati da un inserimento basta risolvere il primo che si manifesta.

Per fare ciò si usa la **rotazione**:

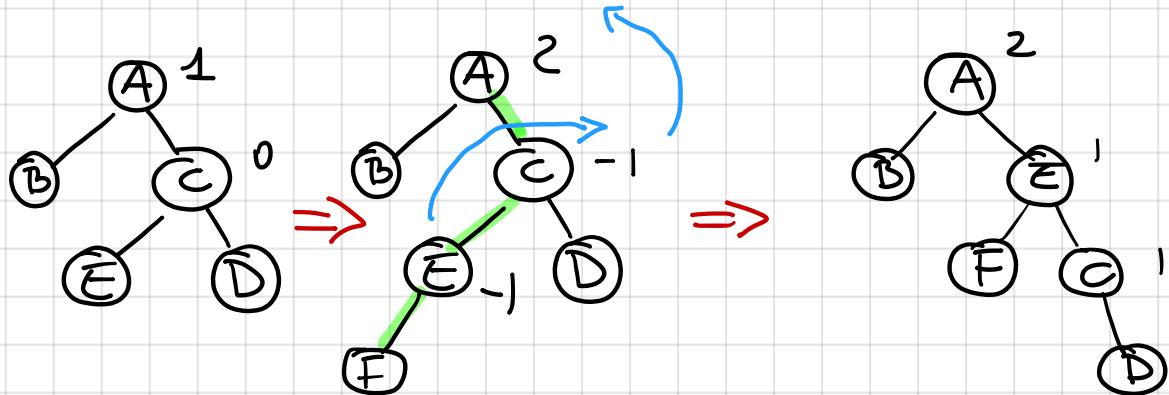
Esempio DD (destra - destra) (ss è simmetrico)



Custo rotazione: $\Theta(\log n)$ solo per trovare il nodo sbilanciato, costante per i combi.

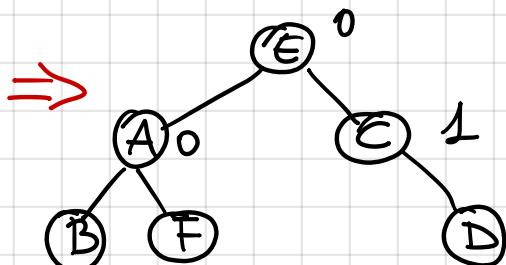


ROTAZIONE DOPPIA (SD o DS)

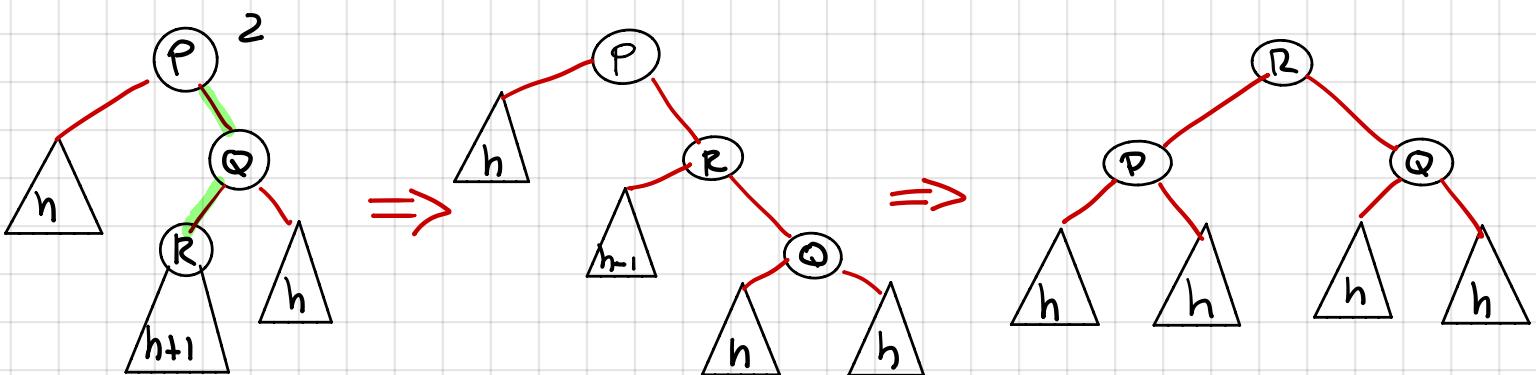


SBALANCIA MENTO! Applico
due rotazioni.

Rotazione su E



Rotazione su E



Le rotazioni risolvono problemi di sbilanciamento massimo a 2!

Gli alberi AVL rappresentano un algoritmo di ordinamento: lo costruisco e poi lo visito in ordine.

```

IF albero right-heavy {
    IF sottoalbero destro Left Heavy {
        DOPPIA ROTAZIONE SINISTRA
    }
    ELSE
        ROTAZIONE SINGOLA SINISTRA
}
ELSE IF albero LEFT HEAVY {
    IF sottoalbero sinistro RIGHT HEAVY {
        DOPPIA ROTAZIONE DESTRA
    }
    ELSE
        ROTAZIONE SINGOLA DESTRA
}

```

N.B. Doppia rotazione sinistra
= prima rotazione destra
su sottoalbero destro e poi
rotazione sinistro.

• Cancellazione in AVL

Essendo un BST elimino come tale
Se ho sbilanciamento PER OGNI NODO SBIANCATO applico la rotazione.
In questo caso la rotazione avverrà non sul ramo del nodo eliminato
ma sull'altro poiché si applica sul ramo più lungo.

DIVIDE - ET - IMPERA

È il paradigma gerarchico con cui si elaborano gli algoritmi: dividere un problema in due sottoproblemi di uguale dimensione (anche più di due, ma con la stessa dimensione e stessa tipologia) **DIVIDE**

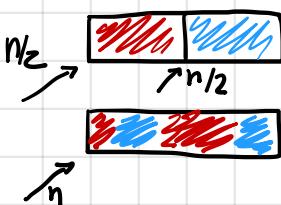
In seguito unisco i due risultati **IMPERA**

Lo stesso discorso vale anche per i sotto problemi

Un esempio di questo paradigma è l'algoritmo **Merge-Sort**:

dota una sequenza S di n elementi lo divide in due sequenze di $n/2$ elementi. Ricorsivamente le ordina entrambo e poi le fonde (merge) insieme.

Si prenda un'area di memoria di dimensione n (la somma delle due sotto parti).



Ad ogni iterazione confronto il valore dei puntatori delle due sotto parti e aggiungo lo più piccolo all'area condivisa. Il puntatore "minore" viene incrementato di 1 come il puntatore dell'area condivisa.

Essendo le due parti ordinate, finito una sotto parte copio lo restante parte dall'altra nell'area condivisa.

L'algoritmo funziona anche con array di diverse dimensioni.

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \quad \begin{matrix} n > 1 \\ \text{ordinare 2} \\ \text{sottoinsiemi} \end{matrix} = 2 \left[2T\left(\frac{n}{2^2}\right) + cn \frac{n}{2} \right] + cn =$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2cn = 2^2 \left[2T\left(\frac{n}{2^3}\right) + cn \frac{n}{2^2} \right] + 2cn = 2^3 T\left(\frac{n}{2^3}\right) + 3cn$$

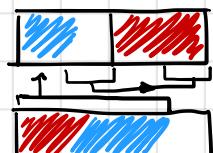
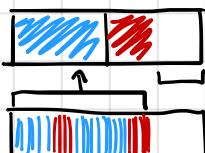
$$\Rightarrow T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$\text{Quando } \frac{n}{2^k} = 1 \Rightarrow k = \log_2 n \quad T(1) = c$$

$$\Rightarrow 2^{\log_2 n} T(1) + \log_2 n \cdot cn = cn + cn \log_2 n$$

Non IN PLACE!

Al contrario di insertion sort è poco sensibile a ordinamenti pre-esistenti.



Migliorie (massimo $n/2$)

In questi casi non serve copiare lo j-th nel array auxiliario, ma direttamente allo fine dell'array final

Altro esempio è **QUICK-SORT**
nel caso peggiore ha costo $\Theta(n^2)$

Si sceglie un pivot random e si creano due sotto insiemi: elementi minori e maggiori del pivot **Passo di partizionamento**

L P G

Si esegue ricorsivamente la funzione sui due gruppi creati: il pivot che rimane in attesa è già nella sua posizione finale

N.B. L e G sono minori e maggiori di P, ma non sono ordinati

Il caso base sarà con la dimensione del subarray 0 o 1.

Il processo di partizionamento su array di piccole dimensioni porta al loro ordinamento

ES.

$$2 \boxed{1} 3 \rightarrow 1 \underbrace{3}_{} 2 \rightarrow 3 \boxed{2} \rightarrow 23$$

$$= 1 2 3$$

Quando L e G sono molto diversi avranno ad avere costo $\Theta(n^2)$
 $\Theta(n \log n)$ altrimenti

CASO PEGGIORE:

L P (G vuoto)
L \downarrow P (G vuoto)

Brendo il Pivot massimo
(o minimo)

A ogni livello ho $n, n-1, n-2, \dots$ iterazioni $\Rightarrow \Theta(n^2)$

$$T(n) = T(n-1) + T(1) + cn$$

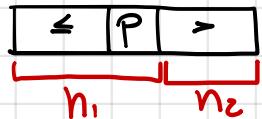
COUNT SORT

- ① Scansione per il max $\Theta(n)$
- ② Alloco array temp. dim max+1 $\Theta(\max)$
- ③ Nuova scansione input $\Theta(n)$
elem input $\rightarrow \text{Temp}[\text{elem}]++$
- ④ Scansione temp $\Theta(\max)$ Popolo l'array con i valori di temp > 0
 $\Rightarrow \Theta(n + \max)$

Se so che max è limitato $\Rightarrow \Theta(n)$

SELEZIONE

- **Quick Select**: algoritmo randomizzato basato su quick sort



$\Theta(n)$ per il partizionamento

Trovare il k-esimo elemento:

$k < n_1 \rightarrow k$ sx

$k = n_1 \rightarrow$ OK

$k > n_1 \rightarrow k - n_1$ dx

INSIEMI

Dati due insiem (array ordinati) trovare le intersezioni

Usa un approccio simile al merge-sort

1	5	9	13
↑	↑	↑	↑
2	3	5	13

Scorso entrambi gli array se l'elemento è uguale lo salvo in un array, altrimenti incremento l'indice che punta sull'elemento minore. Aggiunto un elemento entrambi gli indici.

Lo stesso sistema può essere usata per trovare l'unione (senza ripetizioni)

Se sono uguali aggiro una volta e incremento entrambi, altrimenti aggiro il minore e incremento l'indice che punta ad esso.

Vettore Correttore

$$U = \{u_0, u_1, u_2, \dots, u_n\}$$

$|0|1| \dots |i| |N| N+1$

$$\begin{array}{ll} S[i] = 0 & u_i \notin U \\ S[i] = 1 & u_i \in U \end{array} \quad \text{es. } U = \{A, B, C, D, E\}$$

$S[00110] \rightarrow [CD]$

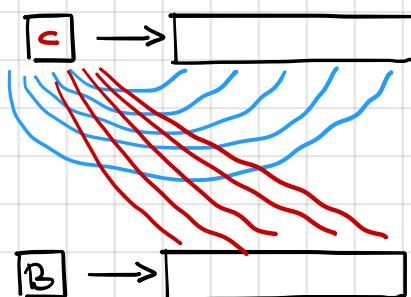
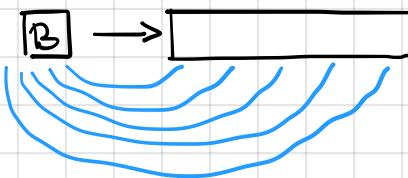
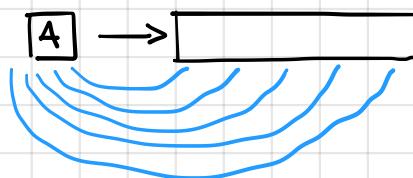
Per insiemi di dimensione limitata è una buona soluzione.

UNION, FIND, MAKESET

Makeset crea un insieme di un singolo elemento: lista con un solo elemento che avrà un valore di ritorno al nome dell'insieme

Find ritorna il nome dato l'elemento

Union è DISTRUTTIVA ovvero ritorna l'unione di due insiemi distruggendo i due precedenti.



Combio nome all'insieme maggiore così da dover "spostare" meno elementi possibili

La dimensione finale sarà almeno doppia di quella dell'insieme minore. Le unioni quindi aumentano esponenzialmente

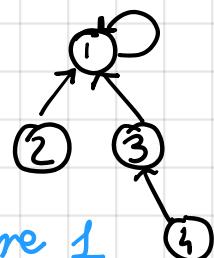
$$1 - 1 \Rightarrow 2 - 2 \Rightarrow 4 - 4 \dots$$

Ogni elemento può essere spostato massimo $O(\log n)$ volte essendo n il numero totale di elementi.

Tempo totale $\log n \cdot n \Rightarrow \Theta(\log n)$ AMMORTIZZATO per operazione.

Gli insiemi vengono rappresentati con alberi particolori:

- La radice ha un riferimento a se stessa
- Ogni nodo ha un riferimento al genitore e non al figlio
- L'elemento radice rappresenta il nome dell'insieme.



Lo schema dell'albero è dato dalla UNION

Lo find ha costo che aumenta con l'aumentare della lunghezza dei cammini \Rightarrow bisogna fare lo Union in modo da avere un'altezza più bassa possibile.

Lo Union combina il puntatore della radice da se stessa ad un'altra radice \Rightarrow bisogna decidere chi verrà collegato a chi!

EURISTICA 1:

UNION BY RANK

Ogni nodo ha un campo in più con l'altezza del suo sottoalbero (a volte con un upperbound)

La radice con RANK minore viene collegata a quella con RANK maggiore.

EURISTICA 2:

PATH - COMPRESSION

Durante lo find faccio puntare tutti i nodi che attraverso alla radice con costo $\Theta(K)$ dove K sono i nodi attraversati.

MAKE-SET (x)

1 $p[x] \leftarrow x$ parent di $x \rightarrow x$

2 $rank[x] \leftarrow 0$

UNION (x, y)

1 LINK(FIND-SET(x), FIND-SET(y))

LINK (x, y)

1 IF $rank[x] > rank[y]$

2 $p[y] \leftarrow x$

3 ELSE $p[x] \leftarrow y$

4 IF $rank[x] = rank[y]$

5 $rank[y] \leftarrow rank[y] + 1$

FIND-SET (x)

1 IF $x \neq p[x]$ se non è radice

2 $p[x] \leftarrow FIND-SET(p[x])$

3 return $p[x]$

L'unione delle due euristiche impiegherà tempo $O(m \alpha(n))$ per eseguire m operazioni su n elementi con $\alpha(n)$ l'inversa di della funzione di Ackermann.

Custo ammortizzato $\Theta(\alpha(n))$

GRAFI

Un grafo è una coppia di insiemi V e E

V = insieme nodi detti VERTICI

E = insieme coppie vertici (collegamenti) detti SPIGGI (o ARCHI)

$e = \{u, v\} = \{v, u\}$ non orientato (insieme)

$e = (u, v)$ orientato (coppia ordinata)

Se gli spigoli sono orientati lo è anche il grafo, semplifica altrimenti

Il grado di un vertice è il numero degli spigoli che interseca (detti incidenti perché incidono sullo stesso nodo).

Proprietà 1

$$\sum \text{gradi} = 2m \quad (m = \# \text{spigoli})$$

Proprietà 2

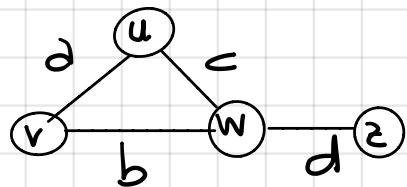
In un grafo non diretto e senza loop o spigoli multipli:

$$m \leq n \frac{(n-1)}{2}$$

Definizione matematica di grafo:

Dato V insieme finito non vuoto, un grafo è una relazione binaria R su V

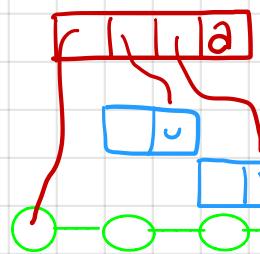
$$R \subseteq V \times V$$

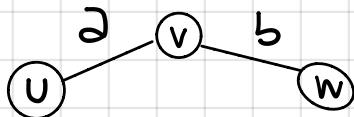


OGGETTO VERTEX

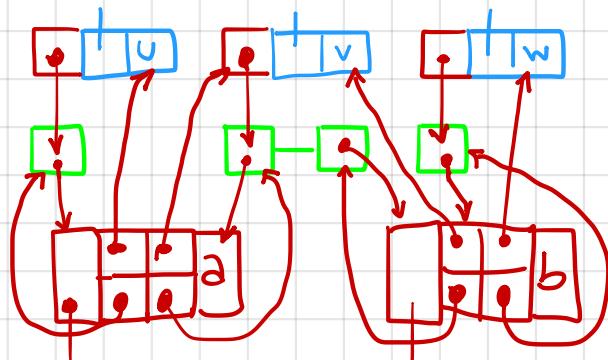
 Contiene l'elemento vertice e un riferimento alla posizione nella lista dei vertici

OGGETTO EDGE

 Contiene un riferimento alla posizione nella sequenza degli spigoli, uno ol vertice d origine , uno a quello di destinazione , e l'elemento spigolo.



Ogni vertice può "collegarsi" allo sua sequenza di incidente

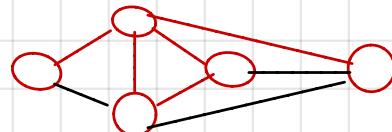


- └ Vertici
- └ Sequenze di incidente (3)
- └ Questi due sono oggetti spigolo incrementati

SOTTOGRAFI

Un sottografo è un grafo i cui vertici e spigoli appartengono tutti ad un altro grafo.

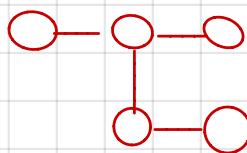
Un sottografo ricoprente è un grafo che contiene tutti i vertici del grafo "padre", ma non tutti gli spigoli.



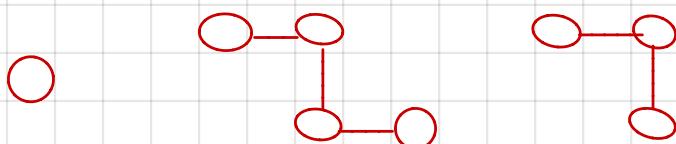
Un grafo si dice CONNESSO se esiste uno spigolo tra ogni coppia di vertici.

ALBERI E FORESTE

Un albero è un grafo non orientato, connesso e senza cicli.
Se non è connesso è detto foresta.

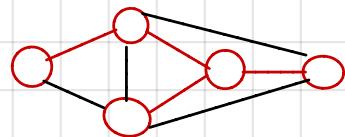


ALBERO



FORESTA

Un albero si dice ricoprente se oltre ad essere un sottografo ricoprente è anche un albero.



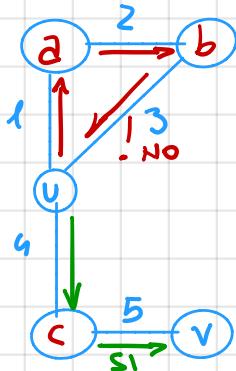
Se in particolare non è connesso, è una foresta ricoprente.

• VISITA IN PROFONDITÀ (DFS)

Dati n vertici e m spigoli impiega $O(n+m)$ per visitare tutti i vertici e spigoli.

Lo DFS può essere modificato per trovare il percorso tra due verti: basta settare un "contatore" quando trovo il vertice di arrivo stabilire salvando tutti vertici passati.

Lo strutturo dati accessoria che mi salva gli spigoli è una pila (stack) che verrà aumentata o decrementata con push e pop nel corso della ricerca verso o meno.



Da u devo scendere verso v: l'algoritmo può andare verso a e salire 1 in stack, poi passa a b e salire 2.
Da qui però non può andare avanti poiché u è già DISCOVER. In questo caso inizio una serie di pop finché non trovo alternative valide e inizio di nuovo a fare push. ovvero ritorno a u e vo verso c.

Stessa cosa può essere usata per trovare cicli semplici ($u, 1, 2$): con lo stesso pilo non appena trovo un back-edge (3) restituiamo il ciclo ovvero il percorso dall'inizio a se stesso. (La pila può essere racciatata per dare il giusto ordine)

Se applicata su grafi orientati si parla di D-DFS che usa un'etichettatura più raffinata della DFS: gli archi possono quindi essere oltre a tree (DISCOVER) e back anche cross e forward. In particolare back e forward dipendono dall'ordine con cui vengono incontrati, informazione che adesso diventa importantissima.

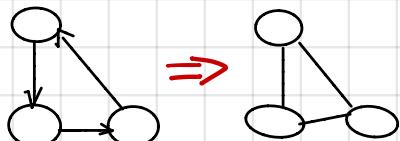
BACK: arco (u, v) con u discendente di v discendente vuol dire che è stato incontrato tramite la ricorsione partita da v . Chiudono un ciclo.

FORWARD: arco (u, v) con v discendente di u

CROSS: gli altri archi non tree.

u e v sono nello stesso albero DF e uno non è discendente dell'altro.

Un grafo orientato ha **connessione debole** se lo stesso grafo non orientato è连通的.



Ha **connessione forte** se per ogni coppia di vertici c'è un percorso orientato che li collega

Raggiungibilità: albero DFS radicato in v che comprende i vertici raggiungibili da v tramite percorsi orientati.

Per verificare se un grafo è fortemente连通:

- DFS su un qualsiasi vertice \Rightarrow se esiste un vertice non visitato allora non è fortemente連通.
- Se G' è il trasposto del grafo G (vertici invertiti) $\Rightarrow G$ è fortemente連通 se e solo se G' lo è.
 - DFS con verifica su un qualsiasi vertice

Tempo $O(n+m)$ (2 DFS)

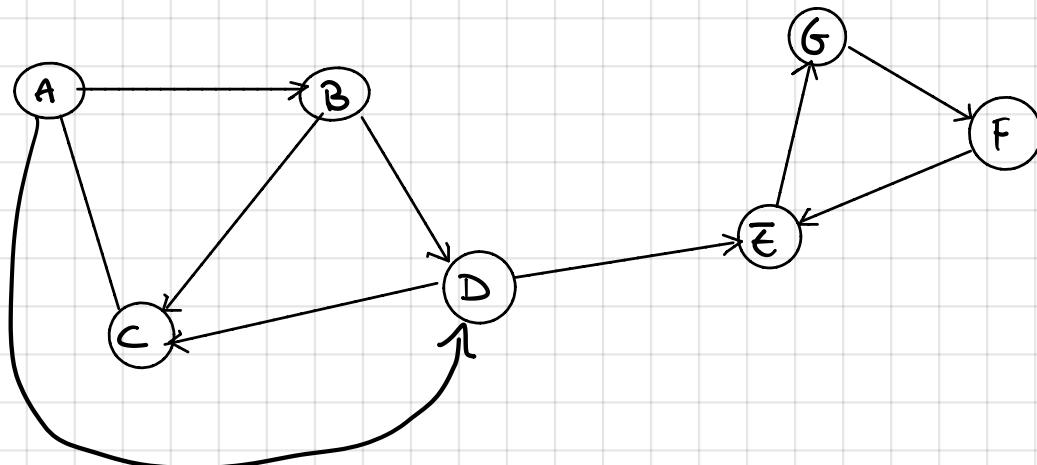
Lo prima DFS verifica che da v arrivo a qualsiasi vertice.
Lo secondo DFS verifica che da qualsiasi vertice posso arrivare a v .

$$\begin{aligned} \textcircled{1} \quad & \exists \pi(v,w) \Rightarrow \pi''(u,w) = \pi'(u,v) \circ \pi(v,w) \\ \textcircled{2} \quad & \exists \pi'(u,v) \end{aligned}$$

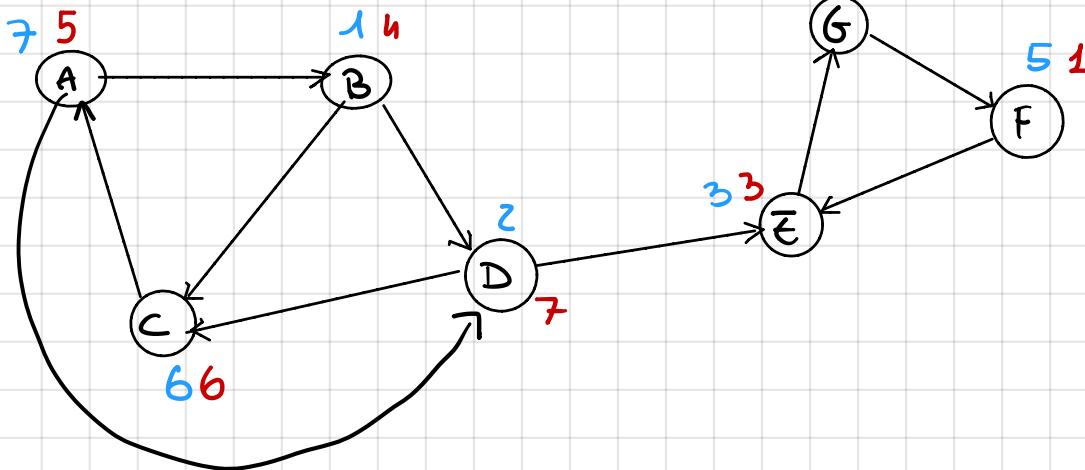
L'insieme dei vertici di un grafo può essere partitionato per formare sottografi massimali fortemente連通.

STRONGLY CONNECTED COMPONENTS (SCC)

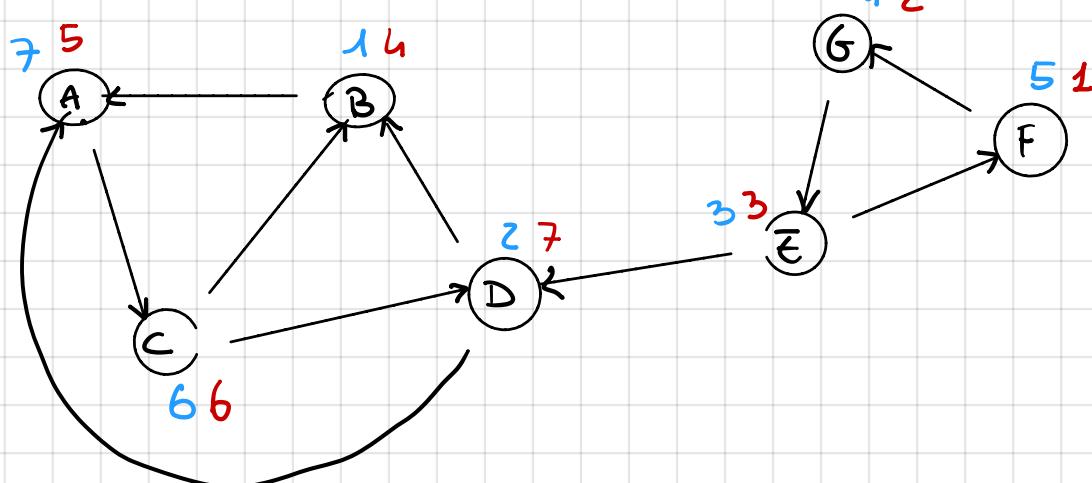
- ① D-DFS (doppia etichetta)
- ② Trasposizione
- ③ D-DFS Vertici esaminati secondo l'ordine della seconda etichetta (o contrario)



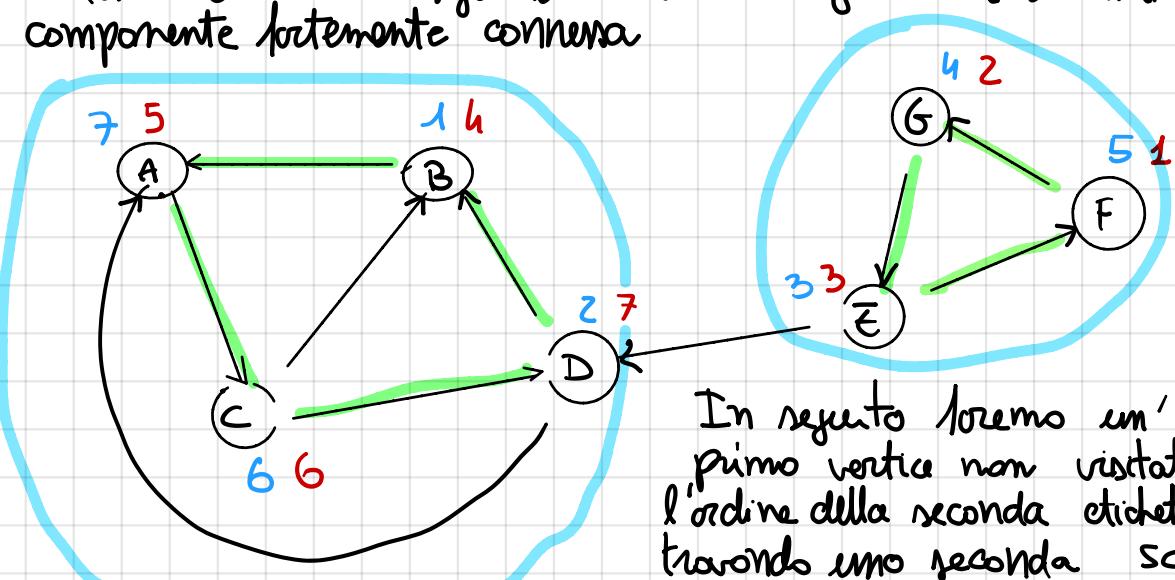
Supponiamo di partire da B in DFS. Seguendo il verso degli spigoli vado avanti inserendo la prima etichetta: al binio in D procediamo a destra. Arrivati a F non abbiamo completato i vertici, ma non possiamo più andare avanti: a ritorno fino all'ultimo binio (D) metto la seconda etichetta e procedo con la prima.



Faccio il trasposto ed eseguo una DFS dall'ultimo della seconda etichetta:



Partendo da D e seguendo i versi degli occhi delimitiamo la prima componente fortemente连通 (SCC).



In seguito faremo un'altra DFS sul primo vertice non visitato seguendo sempre l'ordine della seconda etichetta → E trovando una seconda SCC.

È possibile usare una stack per la seconda etichetta e fare push alla prima DFS e pop alla seconda.

CHIUSURA TRANSITIVA

Dato un digrafo G la sua chiusura transitiva G^* è un digrafo che ha gli stessi vertici di G ma se G ha un percorso da u a v , G^* ha anche uno spigolo da u a v .



Per calcolarlo posso eseguire una DFS su ogni vertice $\Rightarrow \Theta(n(n+m))$ oppure ...

Algoritmo di Floyd-Warshall



- ◆ denoma vertici di G come v_1, \dots, v_n e calcola una serie di digrafi G_0, \dots, G_n
 - $G_0 = G$
 - G_k ha uno spigolo orientato (v_i, v_j) se G ha un percorso orientato da v_i a v_j con vertici intermedi nell'insieme $\{v_1, \dots, v_k\}$
- ◆ Per definizione $G_n = G^*$
- ◆ Nella fase k , viene calcolato il digrafo G_k a partire da G_{k-1}
- ◆ Tempo di esecuzione $O(n^3)$ se `areAdjacent` viene eseguito in $O(1)$ (ad esempio, usando una matrice di adiacenza)

Algorithm *FloydWarshall(G)*

Input digrafo G

Output la chiusura transitiva G^* di G

```

 $i \leftarrow 1$ 
for all  $v \in G.vertices()$ 
    denota  $v$  come  $v_i$ 
     $i \leftarrow i + 1$ 
 $G_0 \leftarrow G$ 
for  $k \leftarrow 1$  to  $n$  do
     $G_k \leftarrow G_{k-1}$ 
    for  $i \leftarrow 1$  to  $n$  ( $i \neq k$ ) do
        for  $j \leftarrow 1$  to  $n$  ( $j \neq i, k$ ) do
            if  $G_{k-1}.areAdjacent(v_i, v_k) \wedge$ 
                 $G_{k-1}.areAdjacent(v_k, v_j)$ 
            if  $\neg G_k.areAdjacent(v_i, v_j)$ 
                 $G_k.insertDirectedEdge(v_i, v_j, k)$ 
return  $G_n$ 

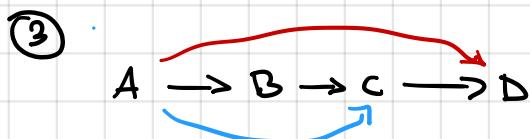
```

Il primo for è utilizzato per numerare tutti i vertici.

Il primo dei 3 cicli omnidati prende il grafo precedente ($G_0 = G$ per def.) e oppure modifica se necessario.

Gli altri due cicli cercano per ogni coppia di vertici se esiste un percorso $i \rightarrow k \rightarrow j$, in caso esistesse aggiungono uno spigolo da i a j

ES. ① $A \rightarrow B \rightarrow C \rightarrow D \Rightarrow$ ② $A \rightarrow B \rightarrow C \rightarrow D \Rightarrow$



Al secondo passaggio aggiungo l'arco $A-C$ che mi permette al terzo passaggio di aggiungere $A-D$ ($A \rightarrow C \rightarrow D$)

DAG (direct acyclic graph) è un grafo orientato che non ha cicli orientati

Solo un DAG permette un ordinamento topologico, ovvero la possibilità di enumerare tutti i vertici tali che per ogni spigolo (u, v) $u < v$

L'ordinamento topologico utilizza una chiomata ricorsiva su ogni vertice per trovare un **pozzo** ovvero un vertice che non ha uscite. Dato un grafo imposta a "non visitato" tutti i nodi e spigoli. e $N = \# \text{ nodi}$

In seguito, per ogni nodo v chiama una funzione ricorsiva.

Tale funzione:

- 1) imposta il nodo v come "visitato" (se già visitato esce).
- 2) per ogni spigolo, se "non visitato", se il nodo dall'altro parte è "non visitato" impasto lo spigolo come "ricerca" e chiama ricorsivamente la funzione sull'altro nodo.
- 3) Terminate le chiamate ricorsive interne, avendo sui nodi "topologicamente" più in fondo, etichetta v con h (che sarà diminuito tante volte quanti sono i suoi nodi più profondi) e riduco N di 1.

BREATH-FIRST SEARCH (BFS)

- L'algoritmo usa un meccanismo per assegnare e accedere "etichette" di vertici e archi

Algorithm **BFS(G)**

Input graph G

Output labeling of the edges and partition of the vertices of G

```

for all  $u \in G.vertices()$ 
    setLabel( $u$ , UNEXPLORED)
for all  $e \in G.edges()$ 
    setLabel( $e$ , UNEXPLORED)
for all  $v \in G.vertices()$ 
    if getLabel( $v$ ) = UNEXPLORED
        BFS( $G$ ,  $v$ )
    
```

Algorithm **BFS(G, s)**

```

 $L_0 \leftarrow$  new empty sequence
 $L_0.insertLast(s)$ 

```

```

setLabel( $s$ , VISITED)

```

```

 $i \leftarrow 0$ 

```

```

while  $\neg L_i.isEmpty()$ 

```

```

 $L_{i+1} \leftarrow$  new empty sequence

```

```

for all  $v \in L_i.elements()$ 

```

```

for all  $e \in G.incidentEdges(v)$ 

```

```

if getLabel( $e$ ) = UNEXPLORED

```

```

     $w \leftarrow$  opposite( $v, e$ )

```

```

if getLabel( $w$ ) = UNEXPLORED

```

```

        setLabel( $e$ , DISCOVERY)

```

```

        setLabel( $w$ , VISITED)

```

```

         $L_{i+1}.insertLast(w)$ 

```

```

else

```

```

        setLabel( $e$ , CROSS)

```

```

 $i \leftarrow i + 1$ 

```

L'algoritmo BFS è una tecnica generale per visitare i grafici e permette di sapere se un grafo è连通的, re calcola le componenti connesse e una sua foresta ricoprente.

Come di solito imposta tutti i vertici e tutti gli spigoli come "inesplorati" e poi per ogni vertice non esplorato lo chiama con un'altra funzione non ricorsiva.

Tale funzione imposta $i=0$, crea una lista L_i (L_0) e ci inserisce il primo nodo. Nel while crea una nuova lista L_{i+1} : per ogni elemento della lista precedente fa un controllo sui suoi spigoli adiacenti, se sono stati visitati sono cross, altrimenti imposta lo spigolo come "ricerca" e il vertice opposto come "visitato" e lo inserisce in L_{i+1} . Incrementa in fine i .

Ad ogni iterazione si effettuano le operazioni su vertici già visitati: quando tutti i vertici connessi saranno stati visitati L_i sarà vuota poiché nessun "opposite(v, e)" sarà "non visitato" ($L_{i+1} \Rightarrow i+1 \Rightarrow L_i$)

Usciti dalla prima esecuzione della seconda funzione avremo la prima componente连通的 del grafo (1 per ogni iterazione del 3° ciclo della prima funzione).

Tempo $O(n+m)$

CAMMINI MINIMI (DISTANZA)

Un grafo può essere PESATO, ovvero ad ogni suo arco può essere associato un valore (costo)

Il cammino minimo tra due vertici è il percorso tra i due con costo totale minore.

Proprietà:

- 1) Il sottocammino di un cammino minimo è un cammino minimo
- 2) L'insieme di tutti i cammini minimi da un vertice agli altri forma un albero

ALGORITMO DIJKSTRA

Si crea una struttura dati per contenere tutti i nodi: ognuno di essi nella sua struttura avrà il campo per la distanza dal nodo stabilita

Con un for si inizializza questa distanza a infinito, mentre a zero sul nodo di partenza.

Finché questa struttura dati non è vuota:

- 1) Trovo nella struttura il nodo con distanza minima, lo salvo in una variabile momento v e lo elimino dalla struttura (alla prima iterazione verrà preso il nodo di partenza)
- 2) Per ogni suo vertice adiacente z controllo che la distanza di z sia minore della distanza di v + il peso dell'arco che li unisce, in caso contrario aggiorno l'etichetta del nodo z con $\text{dist}(v) + \text{peso arco}$.
- 3) Terminato il ciclo ritorno al punto 1

N.B. Quando aggiorno l'etichetta devo aggiornarlo sia nella lista dei vertici del grafo sia nello struttura dati creata a inizio algoritmo.

Tempo $O(m \log n)$ quando connesso, $O((n+m) \log n)$ oltrimenti

Per trovare il percorso più corto da una sorgente agli altri vertici basterà salvare il nodo precedente ad ogni nodo che si visita. Per farlo basterà nel passo 2 oltre ad aggiornare la distanza di z ne aggiorno anche il predecessore con v .

Per avere il cammino minimo basterà procedere seguendo lo linea dei predecessori partendo dal nodo di destinazione fino alla sorgente.

RIEPILOGO

VISITA PREORDINE: ogni nodo è visitato prima dei suoi figli

VISITA POSTORDINE: ogni nodo è visitato dopo i suoi figli

VISITA IN ORDINE: ogni nodo è visitato dopo il suo sottosalbero sinistro e prima di quello destro.

SELECTION-SORT: Dato un array, per ogni posizione cerco il min nelle posizioni successive e lo scambio con quello corrente.

INSERTION-SORT: Dato un array, per ogni posizione verifico che il precedente sia minore della posizione corrente, in caso contrario scorro l'array al contrario fino alla posizione corretta.

CHIUSURA TRANSITIVA: dato il grafo G , la chiusura transitiva G^* è un grafo come G , ma per ogni percorso in G che collega due vertici, G^* ha anche un arco che li collega.

