

## Lezione 1

Un algoritmo è un procedimento che descrive una sequenza di passi ben definiti finalizzati a risolvere un dato problema (computazionale)

Un algoritmo è diverso da programma:

- il programma è la codifica di un algoritmo
- un algoritmo può essere visto come un programma distillato da dettagli riguardanti il linguaggio di programmazione, ide, o SO
- Algoritmo è un concetto autonomo da quello di programma

Gli algoritmi devono essere:

corretti: producono correttamente il risultato desiderato.

efficienti: usano poche risorse di calcolo, come tempo e memoria.  
ma devono essere anche stabili, sicuri, modulari, semplici,...

Le idee algoritmiche non solo trovano soluzioni a problemi posti, quanto costituiscono il linguaggio che porta ad esprimere chiaramente il problema sottostante.

In ogni algoritmo è possibile individuare due componenti fondamentali:

- l'identificazione della appropriata tecnica di progetto algoritmico (struttura del problema)
- la chiara individuazione del nucleo matematico del problema stesso.

Es.

problema: individuare una moneta falsa fra  $n$  monete.

istanza:  $n$  specifiche monete: quella falsa è una di queste.

dimensione dell'istanza: il valore  $n$ .

modello di calcolo: bilancia a due piatti.

algoritmo: strategia di pesatura. La descrizione deve essere comprensibile e compatte. Deve descrivere la sequenza di operazioni sul

modello di calcolo eseguite per una generica istanza.

**correttezza dell'algoritmo:** la strategia di pesatura deve funzionare per una generica istanza.

**complessità temporale:** # di pesate che esegue prima di individuare la moneta falsa.

**complessità temporale nel caso peggiore:** # massimo di pesate che esegue su un'istanza di una certa dimensione

**efficienza:** l'algoritmo deve fare poche pesate, deve essere veloce.

Alg 1.

uso la prima moneta e la confronto con le altre  $x_1, x_2 \rightarrow x_1, x_3 \rightarrow x_1, x_n$

Alg 1. ( $X = \{x_1, \dots, x_n\}$ )

1. for  $i=2$  to  $n$  do

2. if  $\text{peso}(x_i) > \text{peso}(x_1)$  then return  $x_1$

3. if  $\text{peso}(x_i) < \text{peso}(x_1)$  then return  $x_i$

corretto

pesate: caso peggiore  $n-1$ , ma l'ultima pesata non serve quindi  $n-2$

Alg 2

peso le monete a coppie  $x_1, x_2 \rightarrow x_3, x_4 \rightarrow \dots$

Alg 2 ( $X = \{x_1, \dots, x_n\}$ )

1.  $K = \lfloor n/2 \rfloor$

2. for  $i=1$  to  $K$  do

3. if  $\text{peso}(x_{2i-1}) > \text{peso}(x_{2i})$  then return  $x_{2i-1}$

4. if  $\text{peso}(x_{2i-1}) < \text{peso}(x_{2i})$  then return  $x_{2i}$

5. return  $x_n$

corretto

pesate caso peggiore:  $n/2$

### Alg 3

peso le monete dividendole ogni volta in due gruppi  $x_1, x_2, x_3, x_4, x_5, x_6 \rightarrow x_1, x_2, x_3, x_4 \rightarrow \dots$

### Alg 3 (x)

1. if  $|x| = 1$  then return  $x$ .
2. dividi  $x$  in due gruppi  $x_1$  e  $x_2$  di uguale dimensione  $K = \lfloor |x| / 2 \rfloor$  se  $|x|$  è dispari una ulteriore moneta  $y$
3. if  $\text{peso}(x_1) = \text{peso}(x_2)$  then return  $y$
4. if  $\text{peso}(x_1) > \text{peso}(x_2)$  then return  $\text{Alg 3}(x_1)$   
else return  $\text{Alg 3}(x_2)$  corretto

# di pesate che Alg 3 esegue nel caso peggiore su un'istanza di dimensione  $n$ .  $P(x)$  è una funzione non decrescente in  $x$

$$\begin{aligned} P(n) &= P(\lfloor n/2 \rfloor) + \dots \quad P(1) = 0 \quad P(\lfloor n/2 \rfloor) + 1 = P(\lfloor (1/2)\lfloor n/2 \rfloor \rfloor) + 2 \\ &= P(\lfloor n/4 \rfloor) + 2 \\ &= P(\lfloor n/8 \rfloor) + 3 \\ &\leq P(\lfloor n/2^i \rfloor) + i \end{aligned}$$

$$\lfloor n/2^i \rfloor = 1 \text{ quando } i = \lceil \log_2 n \rceil$$

$$\leq P(1) + \lceil \log_2 n \rceil = \lceil \log_2 n \rceil \text{ pesate caso peggiore}$$

### Alg 4

posso dividere in tre gruppi invece di due

### Alg 4 (x)

1. if  $|x| = 1$  then return  $x$ .
2. dividi  $x$  in tre gruppi  $x_1, x_2$  e  $x_3$  di dimensione bilanciata siano  $x_1$  e  $x_2$  i gruppi con la stessa dimensione (ci sono sempre)
3. if  $\text{peso}(x_1) = \text{peso}(x_2)$  then return  $\text{Alg 4}(x_3)$
4. if  $\text{peso}(x_1) > \text{peso}(x_2)$  then return  $\text{Alg 4}(x_1)$   
else return  $\text{Alg 4}(x_2)$  corretto

# di pesate che Alg 4 esegue nel caso peggiore su un'istanza di dimensione  $n$ .

$$P(n) = P(\lceil n/3 \rceil) + 1, \quad P(1) = 0$$

$P(x)$  è una funzione non decrescente in  $x$

Sia  $K$  il più piccolo intero tale che  $3^K \geq n \quad n = 3^K$

$$\Rightarrow K = \log_3 n \xrightarrow{\text{interno}} K = \lceil \log_3 n \rceil$$

$$P(n) \leq P(n^i) = K = \lceil \log_3 n \rceil$$

$$P(n^i) = P(n^i/3) + 1$$

$$= P(n^i/3) + 2$$

$$= P(n^i/3^i) + i$$

$$\text{per } i = K$$

$$= P(1) + K = K$$

Un qualsiasi algoritmo che correttamente individua la moneta falsa fra  $n$  monete deve effettuare nel caso peggiore  $\lceil \log_3 n \rceil$  pesate.

La dimostrazione usa argomentazioni matematiche per mostrare che un generico algoritmo se è corretto deve avere almeno una certa complessità temporale nel caso peggiore.

Dimostrazione elegante e non banale che usa la tecnica dell'albero di decisione.

Alg 4 è un algoritmo **ottimo** per il problema.

## Lezione 2

Quanto velocemente si espanderebbe una popolazione di conigli sotto appropriate condizioni? (Leonardo da Pisa Fibonacci)

Partendo da una coppia di conigli quante coppie si avrebbero nell'anno n?

Regole:

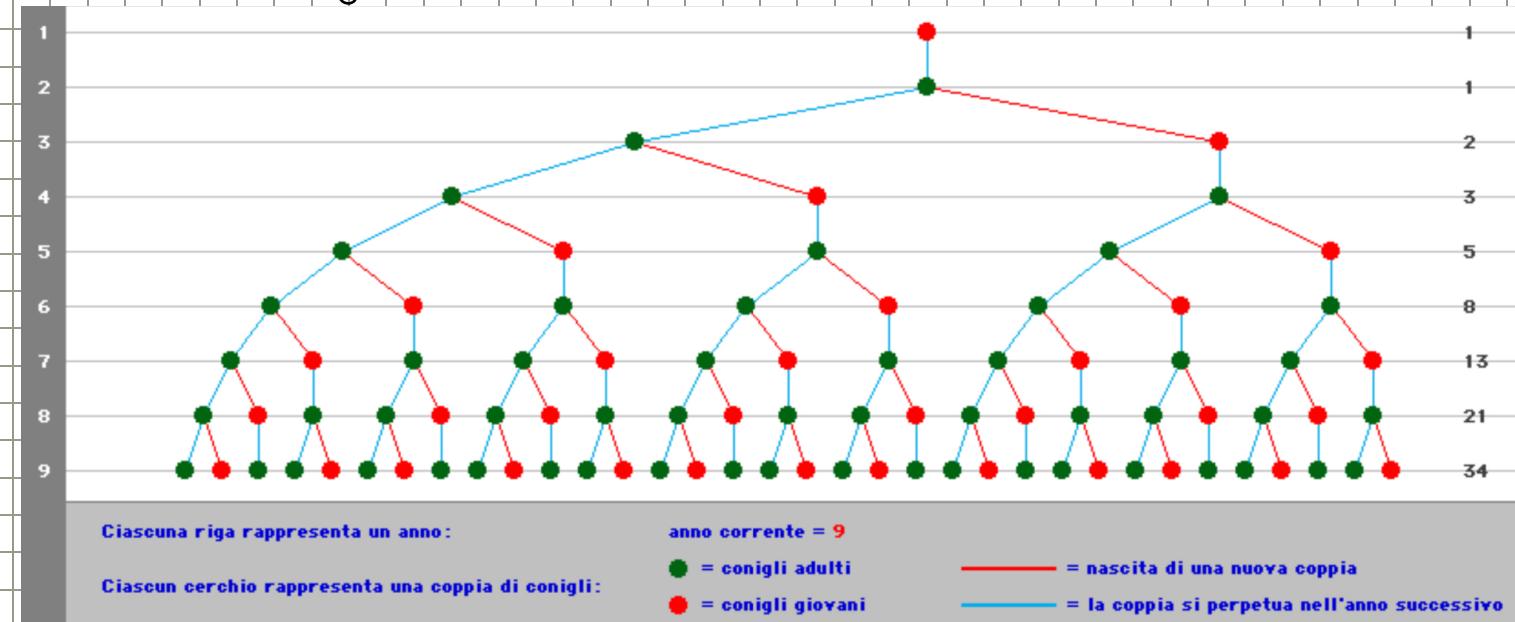
Una coppia di conigli concepisce due coniglietti di sesso diverso ogni anno, i quali formeranno una nuova coppia.

La gestazione dura un anno.

I conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita.

I conigli sono immortali.

### Albero dei conigli



Regole di espansione:

Nell'anno n ci sono tutte le coppie dell'anno precedente, e una nuova coppia di conigli per ogni coppia presente due anni prima.

Indicando con  $F_n$  il numero di coppie dell'anno n, abbiamo la seguente relazione di ricorrenza:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & n \geq 3 \\ 1 & n = 1, 2 \end{cases}$$

Come calcolare  $F_n$ ?

Alg fibo<sub>1</sub>

Possiamo usare una funzione matematica che calcoli direttamente i numeri di Fibonacci, si può dimostrare che:

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \bar{\phi}^n), \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1,618, \quad \bar{\phi} \approx \frac{1-\sqrt{5}}{2} = -0,618$$

- alg fibo<sub>1</sub>(int n) → intero

$$\text{return } \frac{1}{\sqrt{5}} (\phi^n - \bar{\phi}^n)$$

La correttezza di questo algoritmo varia in base alla precisione di  $\phi$  e  $\bar{\phi}$  in quanto:

n	fibo <sub>1</sub> (n)	arrotond.	$F_n$
3	1,9992	2	2
16	986,698	987	987
18	2583,1	2583	2584

diverso!

Alg fibo<sub>2</sub>

Si può utilizzare direttamente la funzione ricorsiva con la tecnica del divide et impera

- alg fibo<sub>2</sub>(int n) → intero  
if ( $n \leq 2$ ) then return 1  
else return fibo<sub>2</sub>(n-1) + fibo<sub>2</sub>(n-2)

E' sicuramente corretto ma non è efficiente:

$T(n) = \# \text{linee di codice eseguite (nel caso peggiore)} \text{ dell'algoritmo su input } n$ .

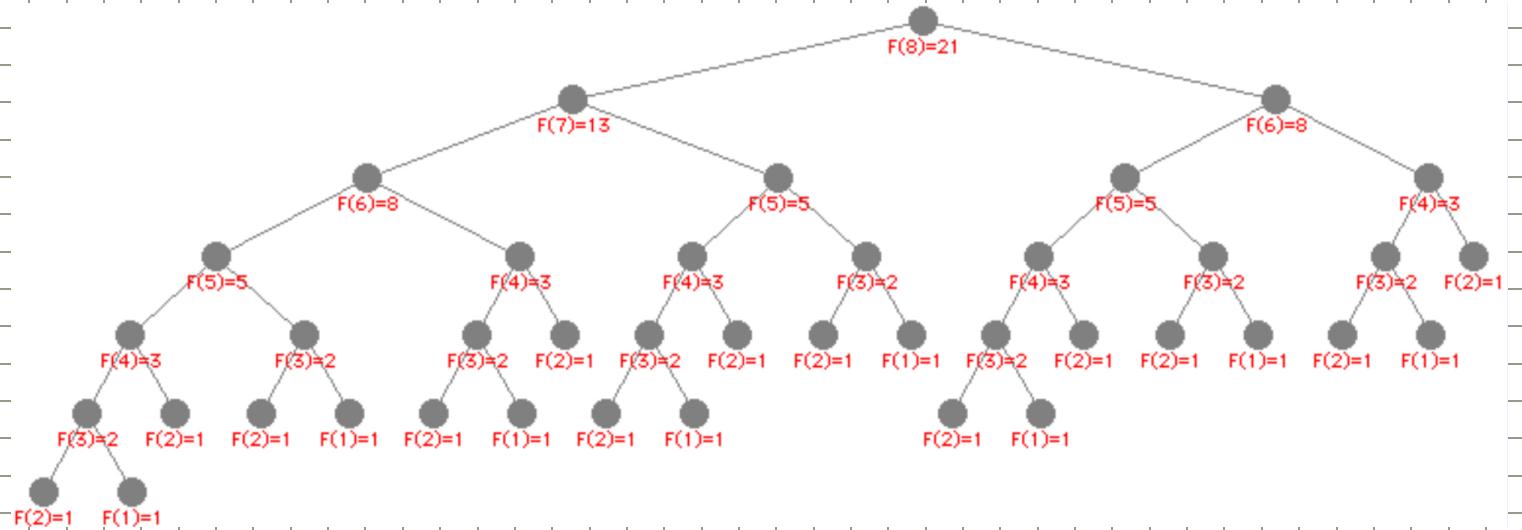
In ogni chiamata si eseguono due linee di codice, oltre a quelle eseguite

nelle chiamate ricorsive

$$T(n) = 2 + T(n-1) + T(n-2)$$

$$T(1) = T(2) = 1$$

E' utile rappresentare l'algoritmo tramite l'albero della ricorsione:



Etichettando i nodi dell'albero con il numero di linee di codice eseguite nella chiamata corrispondente:

- i nodi interni hanno etichetta 2
- le foglie hanno etichetta 1

Per calcolare  $T(n)$  ci basterà sommare il valore delle etichette

Lemma 1

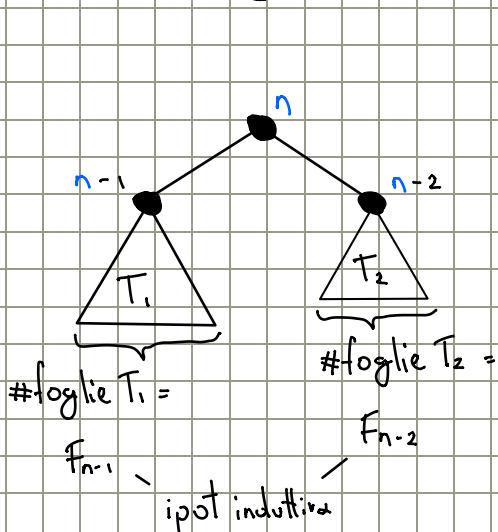
Il numero di foglie dell'albero della ricorsione  $\text{fiboz}(n)$  è pari a  $F_n$

D.m. induzione su  $n$

$$n=1, 2$$

$$\# \text{foglie}_1 = F_1 = F_2$$

$$n > 2$$



$$\# \text{foglie} = F_{n-1} + F_{n-2} = F_n$$

## Lemma 2

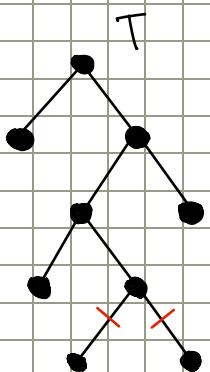
Il numero di nodi interni di un albero in cui ogni nodo interno ha due figli è pari al numero di foglie - 1

Dim. (induzione sul numero di nodi dell'albero n)

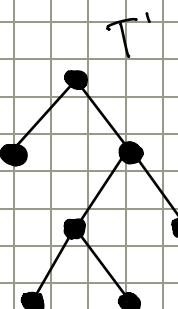
$$n \leq 2$$



$$n > 2$$



$$i = 0 \quad f = 1$$



per costruzione

$$i' = i - 1$$

$$f' = f - 1$$

$$i' = f - 1 \quad \text{per ipot. induttiva:}$$

$$i - 1 = f - 1 - 1 \quad \text{quindi} \quad i = f - 1$$

In totale le linee di codice eseguite sono:

$$F_{n+2}(T_{n-1}) = 3F_n - 2$$

fib02 è quindi un algoritmo lento,  $T(n) \approx F_n \approx \phi^n$ .

$n=8 = 61$ ,  $n=45 = 3404709508$ . non è efficiente.

## Algo fib03

Algo fib02 è lento perché continua a ricalcolare ripetutamente la soluzione dello stesso sottoproblema. Algo fib03 risolve questo problema memorizzando in un array le soluzioni dei sottoproblemi (tecnica **programmazione dinamica**)

- algo fib03 (int n) → intero

Fib ← array di n numeri

Fib[0] ← 1; Fib[1] ← 1

for i=3 to n do

Fib[i] ← Fib[i-1] + Fib[i-2]

return Fib[n]

n° di volte che la

riga viene eseguita

# 1

# 1

# n

# n

# 1

L'algoritmo è corretto e  $T(n) \leq n + n + 3 = 2n + 3$ .

Algo fibo3 impiega tempo proporzionale a  $n$  invece di esponenziale in  $n$  come Algo fibo2.

In questo caso però il tempo di esecuzione non è la sola risorsa di calcolo che ci interessa. Anche la quantità di memoria necessaria può essere cruciale. Con un algoritmo lento ci basterà attendere di più ma con uno che richiede più spazio di quello a disposizione non otterremo mai la soluzione.

Algo fibo4

Algo fibo3 usa un array quando in realtà non ci serve mantenere tutti i valori di  $F_n$  precedenti, ma solo gli ultimi due, riducendo lo spazio a poche variabili:

- algo fibo4 (int n)

a ← 1, b ← 1

for i = 3 to n do

c ← a + b

a ← b

b ← c

return c

L'algoritmo è corretto e  $T(n) \leq 4n + 2$ .

Si introduce quindi il concetto di notazione asintotica:

si esprime  $T(n)$  in modo qualitativo perdendo un po' in precisione ma guadagnando in semplicità.

Di  $T(n)$  vogliamo descrivere come cresce al crescere di  $n$  ignorando le costanti moltiplicative e termini di ordine inferiore

$$T(n) = sn + g = O(n), T(n) = 6n^2 + 8n + 13 = O(n^2) \quad T_{\text{fibo3}}(n) = T_{\text{fibo4}}(n)$$

Per un nuovo algoritmo possiamo utilizzare il Lemma 2:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} \quad F_0=0$$

Dim induzione su n:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Algo fibo5

- algo fibo5 (int n)

$$M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

for i=1 to n-1 do

$$M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

return M[0][0]

L'algoritmo è corretto ma il tempo di esecuzione è sempre  $O(n)$ .

Algo fibo6

Possiamo calcolare la n-esima potenza elevando al quadrato la  $\lfloor n/2 \rfloor$  esima potenza e se n è dispari facciamo un'ulteriore moltiplicazione

- algo fibo6 (int n) → int

$$A \leftarrow \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$M \leftarrow \text{potenz}(A, n-1)$$

return M[0][0]

func potenz (mat A, int K) → mat

$$\text{if } (K=0) \text{ return } \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\text{else } M \leftarrow \text{potenz}(A, \lfloor K/2 \rfloor)$$

$$M \leftarrow MM$$

$\text{if } (K \text{ è dispari}) \text{ then } M \leftarrow M \cdot A$

return M

L'algoritmo è corretto e tutto il tempo è speso nella procedura potenziale.

L'equazione di ricorrenza quindi è:

$$T(n) \leq T(\lfloor n/2 \rfloor) + c$$

$$T(n) \leq c + T(\lfloor n/2 \rfloor) \leq 2c + T(\lfloor n/2 \rfloor) \leq \dots \leq ic + T(\lfloor n/2^i \rfloor) \quad i = \lceil \log_2 n \rceil$$

$$T(n) \leq c \lceil \log_2 n \rceil + T(1) = O(\log_2 n).$$

fib6 è il più veloce di tutti. (esponenzialmente più di fibo 3)

L'algoritmo non ricorsivo ha una complessità spaziale che dipende dalla memoria (ausiliare) allocata mentre in un algoritmo ricorsivo dipende dalla memoria (ausiliaria) allocata da ogni chiamata e dal numero di chiamate che sono contemporaneamente attive.

Tramite l'albero della ricorsione è possibile capire quante chiamate sono attive nello stesso momento.

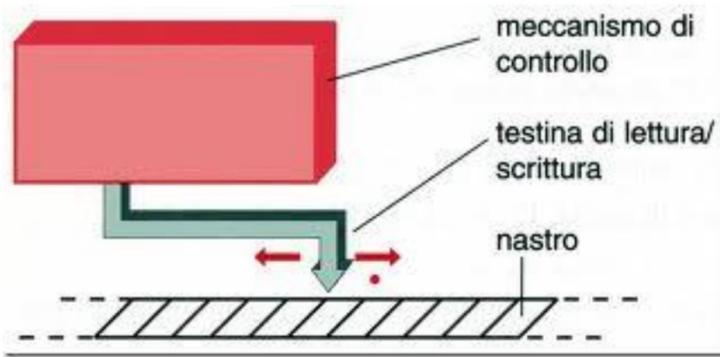
fib02 : spazio  $O(n)$  (Per ha al più  $n$  nodi e ognuno usa memoria costante)

fib06: spazio  $O(\log n)$  (altezza =  $O(\log n)$  e ogni nodo usa memoria costante)

## Lezione 3

Abbiamo un problema a cui sono associate diverse (infinte) istanze di diversa dimensione. Voglio risolvere (automaticamente) il problema progettando un algoritmo. L'algoritmo sarà eseguito su un modello di calcolo e deve descrivere in modo non ambiguo (utilizzando appositi costrutti) la sequenza di operazioni sul modello che risolvono una generica istanza. La velocità dell'algoritmo è misurata come numero di operazioni eseguite sul modello e dipende dalla dimensione e dall'istanza stessa. Analizzare la complessità computazionale di un algoritmo vuol dire stimare il tempo di esecuzione dell'algoritmo nel caso peggiore in funzione della dimensione dell'istanza. A volte si può dimostrare matematicamente che oltre una certa soglia di velocità non si può andare.

Un modello di calcolo storico è la macchina di Turing.



il programma sarebbe il nostro meccanismo di controllo, legge la cella sul nastro e in base al valore letto ed ad una tabella si muove e scrive.

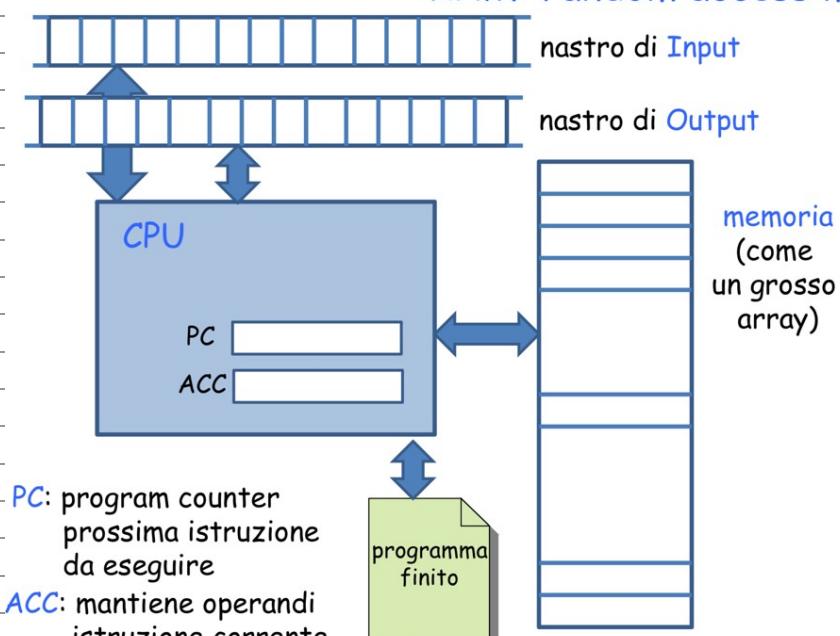
E' utile per parlare di calcolabilità ma meno utile per l'efficienza ed è troppo di basso livello differisce troppo dai calcolatori moderni.

Un modello più realistico è la macchina a registri (RAM: random access machine) è composta da:

- un programma finito.
- un nastro d'ingresso e uno d'uscita.
- una memoria strutturata come un array dove ogni cella può contenere un qualunque valore reale.
- due registri speciali: PC e ACC.

La RAM è un'astrazione dell'architettura di Von Neumann.

RAM: random access machine



passi elementari su una RAM:

istruzioni input/output, operazione aritmetico / logica e accesso e modifica del contenuto della memoria.

L'analisi della complessità di un algoritmo è basata sul concetto di passo elementare.

Sono due criteri di costo per i passi elementari:

- uniforme: tutte le operazioni hanno lo stesso costo e la complessità temporale è misurata come numero di passi elementari eseguiti (costante).
- logaritmico: il costo di una operazione dipende dalla dimensione degli operandi dell'istruzione. Un'operazione su un operando di valore  $x$  ha costo  $\log x$ . È un criterio di costo che modella meglio la complessità di algoritmi numerici.

Misureremo il tempo di esecuzione di un algoritmo in funzione della dimensione  $n$  delle istanze. Istanze diverse, a parità di dimensione, potrebbero però richiedere tempo diverso bisogna perciò distinguere la analisi nel caso peggiore e nel caso medio.

Sia  $\text{tempo}(I)$  il tempo di esecuzione di un algoritmo sull'istanza  $I$

$$T_{\text{worst}}(n) = \max_{\text{istanze } I \text{ di dimensione } n} |\text{tempo}(I)|$$

caso pegg

$T_{\text{worst}}(n)$  è il tempo di esecuzione sulle istanze in ingresso che comportano più lavoro, rappresenta una garanzia sul tempo di esecuzione di ogni istanza.

Sia  $P(I)$  la probabilità di occorrenza dell'istanza  $I$

$$T_{\text{avg}}(n) = \sum_{\text{istanze } I \text{ di dimensione } n} |P(I) \text{tempo}(I)|$$

caso medio

$T_{\text{avg}}(n)$  è il tempo di esecuzione sulle istanze in ingresso "tipiche" per il problema. Di solito però non si può conoscere la distribuzione di probabilità sulle istanze perciò la si assume.

Caso medio Alg1?

$$\sum_{i=1}^n \frac{1}{n} \left\{ \begin{array}{l} 1 \text{ se } A_{ij}=1 \\ 0 \text{ altrimenti} \end{array} \right\} = \frac{1}{n} \left( 1 + \sum_{j=1}^n (j-1) \right) = \frac{1}{n} \left( 1 + \frac{(n-1)n}{2} \right) = \frac{1}{n} + \frac{n-1}{2}$$

Per descrivere la complessità degli algoritmi si utilizza la notazione asintotica, espressa con una funzione  $T(n)$ .

$T(n)$ : passi elementari eseguiti su una RAM nel caso peggiore su un'istanza di dimensione  $n$ .

$$T(n) = \begin{cases} 71n^2 + 100\frac{n}{4} + 7 & \text{se } n \text{ è pari} \\ 70n^2 + 150\frac{n+1}{4} + 5 & \text{se } n \text{ è dispari} \end{cases}$$

descrivere  $T(n)$  in modo qualitativo:  
perdere in precisione ma guadagnare in semplicità

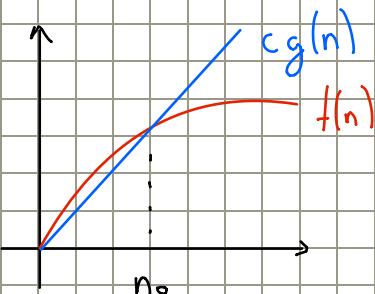
Scriveremo  $T(n) = \Theta(n^2)$ , intuitivamente vuol dire:  $T(n)$  è proporzionale a  $n^2$ , cioè ignoro costanti moltiplicative, termini di ordine inferiore.

$$f(n) = O(g(n)) \text{ se } \exists c > 0, n_0 \geq 0 : 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n))$$

$$f(n) = O(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} < \infty$$

es.  $f(n) = 2n^2 + 3n \quad f(n) = O(n^3), = O(n^2), \neq O(n)$

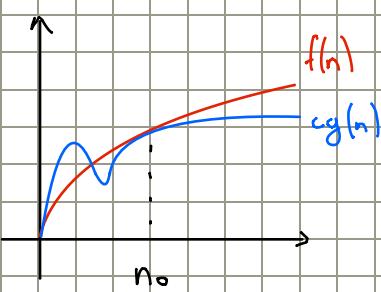


$$f(n) = \Omega(g(n)) \text{ se } \exists c > 0, n_0 \geq 0 : 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n))$$

$$f(n) = \Omega(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} > 0$$

es.  $f(n) = 2n^2 + 3n \quad f(n) = \Omega(n), = \Omega(n^2), \neq \Omega(n^3)$

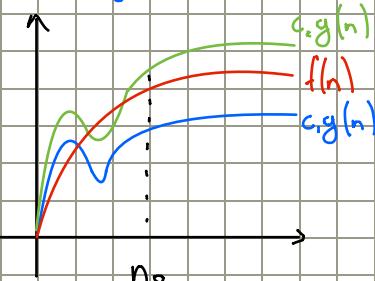


$$f(n) = \Theta(g(n)) \text{ se } \exists c_1, c_2 > 0 \text{ e } n_0 \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$$

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$$

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \text{ e } f(n) = O(g(n))$$



$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c > 0 \Rightarrow f(n) = \Theta(g(n)).$$

Scrivere che  $2n^2 + 4 = \Theta(n^2)$ ,  $= O(n^2)$ ,  $= \Omega(n)$  è un **abusò di notazione** in quanto si dovrebbe scrivere che  $2n^2 + 4 \in \Theta(n^2)$ ,  $\in O(n^2)$ ,  $\in \Omega(n)$

Dato una funzione  $g(n): \mathbb{N} \rightarrow \mathbb{R}$  si denota con  $o(g(n))$  l'insieme delle funzioni  $f(n): \mathbb{N} \rightarrow \mathbb{R}$ :

$$f(n) = o(g(n)) : \forall c > 0, \exists n_0 : 0 \leq f(n) < cg(n) \quad \forall n \geq n_0.$$

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$o(g(n)) \subset O(g(n))$$

Dato una funzione  $g(n): \mathbb{N} \rightarrow \mathbb{R}$  si denota con  $\omega(g(n))$  l'insieme delle funzioni  $f(n): \mathbb{N} \rightarrow \mathbb{R}$ :

$$f(n) = \omega(g(n)) : \forall c > 0, \exists n_0 : 0 \leq cg(n) < f(n) \quad \forall n \geq n_0.$$

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$\omega(g(n)) \subset \Omega(g(n))$$

Analogie

$$\underset{\leq}{\circ}, \underset{\geq}{\Omega}, \underset{=}{\oplus}, \underset{<}{\circ}, \underset{>}{\ominus}$$

Proprietà:

transitività:

$$f(n) = \Theta(g(n)) \text{ e } g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \text{ e } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \text{ e } g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

$$f(n) = o(g(n)) \text{ e } g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$$

$$f(n) = \omega(g(n)) \text{ e } g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

riflessiva:

$$f(n) = \Theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

simmetrica

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

simmetrica trasposta

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$

Velocità asintotica di funzioni composte

Date  $f(n)$  e  $g(n)$ :

La velocità ad andare a infinito della funzione  $f(n)+g(n)$  è la velocità della più veloce fra  $f(n)$  e  $g(n)$ .

La velocità ad andare a infinito della funzione  $f(n)g(n)$  è la velocità di  $f(n)$  "più" di  $g(n)$ .

La velocità ad andare a infinito della funzione  $f(n)/g(n)$  è la velocità di  $f(n)$  "meno" di  $g(n)$ .

Asintoti

polinomi:  $P(n) = a_dn^d + \dots + a_0$

$$P(n) = \Theta(n^d)$$

esponenziali:  $f(n) = a^n \quad a > 1$

$$\lim_{n \rightarrow \infty} \frac{a^n}{n^d} = \infty$$

$$f(n) = \omega(n^d)$$

logaritmici:  $f(n) = \log_b(n) \quad b > 1$

$$\lim_{n \rightarrow \infty} \frac{(f(n))^c}{n^d} = 0 \quad \forall c, d > 0$$

$$f(n) = o(n^d)$$

fattoriali:  $f(n) = n!$

$$f(n) = o(n^n), = \omega(n^n)$$

Analisi complessità

Consideriamo l'algoritmo di fibonacci 3 e facciamo l'analisi con la notazione asintotica.

Ogni singola istruzione sulla RAM compie un numero costante di operazioni.

fibonacci (int n)

1 Fib = []

2 Fib[0] ← Fib[1] ← 1

3 for i=3 to n do

4 Fib[i] ← Fib[i-1] + Fib[i-2]

5 return Fib[n]

Consideriamo  $c_1$  come # passi elementari eseguiti su una RAM e  $T(n)$  come complessità computazionale nel caso peggiore con input n allora:

$$T(n) \leq c_1 + c_2 + c_3 + n(c_3 + c_n) = \Theta(n) \quad T(n) = O(n) \quad \text{e}$$

$$T(n) \geq c_4(n-3) = \Theta(n) \quad T(n) = \Omega(n) \quad \text{quindi } T(n) = \Theta(n)$$

Il primo caso facciamo una maggiorazione **Upper Bound** mentre nel secondo caso si fa una minorazione **Lower Bound**.

Si usa quindi la notazione asintotica perché ci dà una misura indipendente dall'implementazione dell'algoritmo e dalla macchina reale su cui è eseguito. I dettagli nascosti sono poco rilevanti quando n è grande per funzioni asintoticamente diverse. Un analisi dettagliata del numero di passi realmente eseguiti sarebbe complessa e non direbbe molto di più. Descrivere bene in pratica la velocità degli algoritmi.

## Lezione 4

Dato un array, per capire se un certo elemento x appartiene a quella lista si utilizza l'elemento della **ricerca sequenziale**, l'algoritmo ritorna la posizione di x in L se x è presente, -1 altrimenti.

algoritmo RicercaSequentiale (array L, elem x)  $\rightarrow$  int

n = lunghezza di L

i = 1

for i = 1 to n do

if ( $L[i] = x$ ) then return i

return -1

$T_{\text{worst}}(n) = n = \Theta(n)$   $x \notin L$  oppure è in ultima posizione.

$T_{\text{avg}}(n) = (n+1)/2 = \Theta(n)$  assumendo che  $x \in L$  e si trovi in una qualsiasi posizione con la stessa probabilità.

Questo è il **miglior** algoritmo per cercare un elemento in un array non ordinato.  
Se invece l'array è ordinato si può utilizzare l'algoritmo di **ricerca binaria**.

algoritmo RicercaBinariaRic (array L, elem x, int i, int j)

if ( $i > j$ ) then return -1

$m = \lfloor (i+j)/2 \rfloor$

if ( $L[m] = x$ ) then return m

if ( $L[m] > x$ ) then return RicercaBinariaRic ( $L, x, i, m-1$ )

else return RicercaBinariaRic ( $L, x, m+1, j$ )

Gli indici i e j indicano la porzione di L in cui andare a cercare l'elemento x

$T(n) = T(n/2) + O(1) \rightarrow T(n) = O(\log n)$ .

Analizzare la complessità di un algoritmo **ricorsivo** significa risolvere equazioni di **ricorrenza**.  $T(n) = T(n')$ . Gli algoritmi ricorsivi vengono spesso utilizzati con la tecnica di divide et impera. Ci sono vari metodi per risolvere le equazioni di ricorrenza.

algoritmo fibo2 (int n)

if ( $n \leq 2$ ) then return 1

$T(n) = T(n-1) + T(n-2) + O(1)$

else return fibo2 ( $n-1$ ) + fibo2 ( $n-2$ )

### $\text{Alg}_n(x)$

1. if  $|x| = 1$  then return  $x$ .
2. dividere  $x$  in tre gruppi  $x_1, x_2$  e  $x_3$  di dimensione bilanciata siano  $x_1$  e  $x_2$  i gruppi con la stessa dimensione
3. if  $\text{peso}(x_1) = \text{peso}(x_2)$  then return  $\text{Alg}_n(x_3)$
4. if  $\text{peso}(x_1) > \text{peso}(x_2)$  then return  $\text{Alg}_n(x_1)$   
else return  $\text{Alg}_n(x_2)$

$$T(n) = T(n/3) + O(1)$$

La complessità computazionale di un algoritmo ricorsivo può essere espressa in modo naturale attraverso una eq. di ricorrenza. In ogni eq di ricorrenza il caso base sarà sempre:  $T(\text{costante}) = \text{cost.} \circ T(1) = 1$

### Metodo dell'iterazione

L'idea è di srotolare la ricorsione, ottenendo una sommatoria dipendente solo della dimensione  $n$  del problema iniziale.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 & T(n) &= 2T(n-1) + 1 = \\ &&&= 4T(n-2) + 2 + 1 \\ &&&= 8T(n-3) + 4 + 2 + 1 \\ &&&= 16T(n-4) + 8 + 4 + 2 + 1 & n-i = 1 \\ &&&= 2^i T(n-i) + \sum_{j=0}^{i-1} 2^j \end{aligned}$$

$$T(n) = 2^{n-1} T(1) + \sum_{j=0}^{n-2} 2^j = \Theta(2^n)$$

Non è sempre possibile utilizzare il metodo dell'iterazione:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 & T(n) &= T(n-1) + T(n-2) + 1 = \\ &&&= T(n-2) + T(n-3) + 1 + T(n-3) + T(n-4) + 1 = \\ &&&= T(n-3) + T(n-4) + 1 + 2T(n-4) + 2T(n-5) + 2 + \dots \\ &&&= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) + \dots \end{aligned}$$

$$\begin{aligned}
 \text{ese. } T(n) &= T(n-1) + n = \\
 &= T(n-2) + 2n = \\
 &= T(n-3) + 3n = \\
 &= T(n-i) + in = \quad n-i=1 \quad i=n-1 \\
 &= T(1) + (n-1)n = \Theta(n^2)
 \end{aligned}$$

$$\begin{aligned}
 \text{ese. } T(n) &= gT(n/3) + n = \\
 &= g(gT(n/9) + n/3) + n = g^2T(n/9) + 3n + n \\
 &= g^2(gT(n/27) + n/9) + 3n + n = g^3T(n/27) + gn + 3n + n \\
 &= g^i T(n/3^i) + \sum_{j=0}^{i-1} 3^j n \quad i = \log_3 n \\
 &= g^{\log_3 n} T(1) + \sum_{j=0}^{\log_3 n - 1} 3^j n = \Theta(g^{\log_3 n})
 \end{aligned}$$

### Metodo dell'albero della ricorsione

L'idea è di disegnare l'albero delle chiamate ricorsive indicando la dimensione di ogni nodo, stimare il tempo speso da ogni nodo dell'albero e stimare il tempo complessivo "sommando" il tempo speso da ogni nodo.

Se il tempo speso da ogni nodo è costante,  $T(n)$  è proporzionale al numero di nodi.

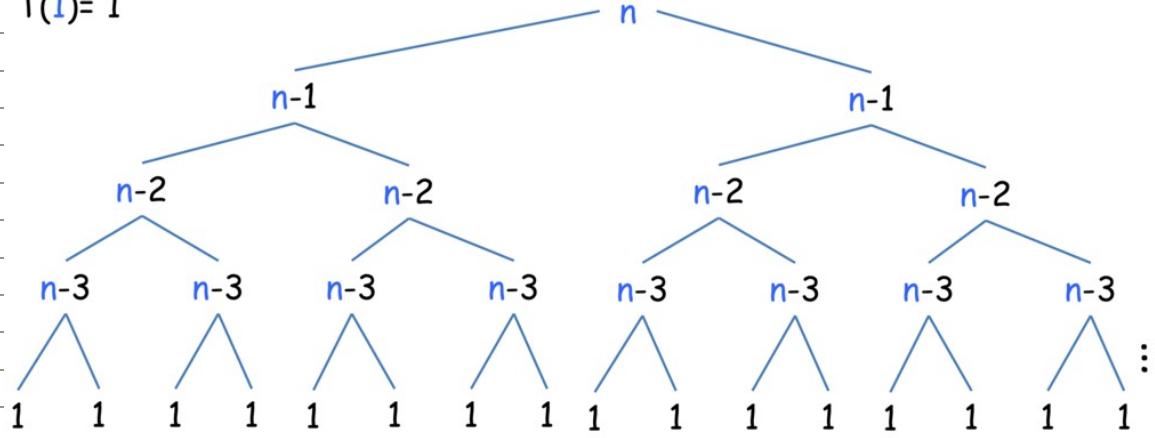
A volte conviene analizzare l'albero per livelli: analizzare il tempo speso su ogni livello e stimare il numero di livelli.

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

$\begin{array}{c} \uparrow \\ 1 \\ \uparrow \\ n-1 \\ \uparrow \\ n-2 \\ \vdots \\ n-i \\ \vdots \\ 1 \end{array}$ 
 ho  $n$  nodi e ogni nodo costa al più  $n$ , l'albero ha  
 $n$  nodi, quindi  $T(n) = \Theta(n^2)$   
 i primi  $n/2$  nodi costano  $n/2$  quindi:  $T(n) \geq (n/2)(n/2) = n^2/4$   
 quindi  $T(n) = \Omega(n^2)$   
 allora  $T(n) = \Theta(n^2)$

$$T(n) = 2T(n-1) + n$$

$$T(1) = 1$$



Questo albero è chiamato **albero binario completo**.

Ogni nodo costa al più  $n$ , l'albero è alto  $n-1$  e ci sono  $\sum_{i=0}^{n-1} 2^i = 2^{n+1} - 1$  nodi.

$$T(n) \leq n2^n = \Theta(n2^n)$$

Prima di usare l'idea dell'albero si potrebbe maggiorare così da ottenerne un Upper Bound:

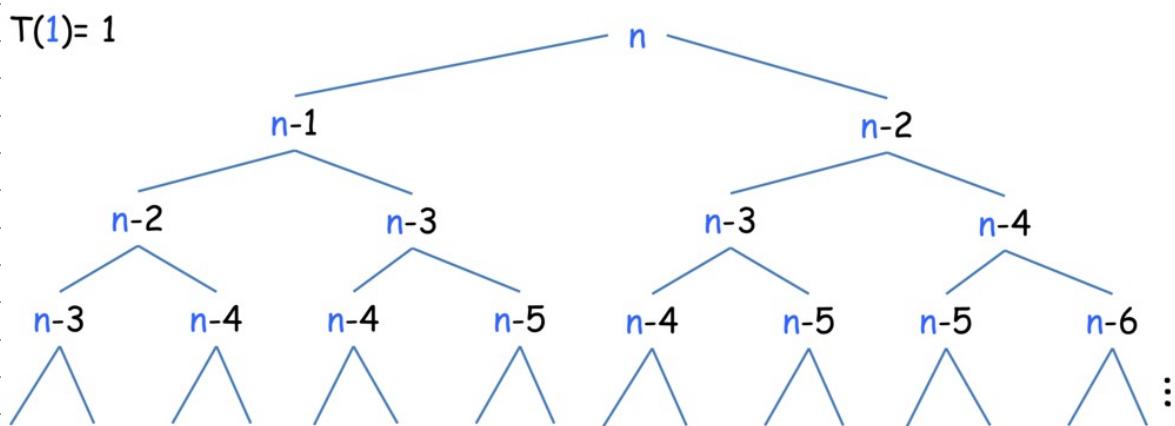
Prendiamo in considerazione l'algoritmo **fibonacci 2**

$$T(n) = T(n-1) + T(n-2) + 1 \quad T(n) \leq R(n) \text{ dove } R(n) = 2R(n-1) + 1$$

$$\text{quindi } T(n) = O(2^n) \text{ perché } R(n) = \Theta(2^n)$$

usando l'albero della ricorsione vediamo che:

$$T(1) = 1$$



Sappiamo dai lemmi che l'albero di fibonacci ha  $\phi^n$  nodi

quindi:

$$T(n) = \Theta(\phi^n)$$

$$T(n) = O(2^n)$$

## Lezione 5

### Metodo della sostituzione

L'idea è: indovinare la forma della soluzione, poi usare l'induzione matematica per provare che la soluzione è quella intuita e risolvere rispetto alle costanti.

$$T(n) = n + T(n/2), T(1) = 1$$

Vediamo se  $T(n) \leq cn$

Passo base:  $T(1) = 1 \leq c \quad \forall c \geq 1$

Passo induttivo: assumiamo  $T(K) \leq ck \quad \forall K < n$

$$T(n) = n + T(n/2) \leq n + c(n/2) = n(c/2 + 1) \quad \text{quando } T(n) \leq cn?$$

dopo avere  $c/2 + 1 \leq c$  quindi  $c \geq 2$

$$T(n) \leq 2n \quad \text{quindi } O(n)$$

es.  $T(n) = 4T(n/2) + n, T(1) = 1 \quad T(n) = O(n^3)$

Pb:  $T(1) = 1 \leq c \quad \forall c \geq 1$

PI: Assumiamo  $T(K) \leq ck^3 \quad \forall K < n$

$$T(n) = 4T(n/2) + n \leq n + 4c(n/2)^3 = n + \frac{1}{2}cn^3 = cn^3 - (\frac{1}{2}cn^3 - n) \leq cn^3$$

Se  $\frac{1}{2}cn^3 - n \geq 0 \quad c \geq 2$  allora  $T(n) \leq 2n^3 \quad T(n) = O(n^3)$

$T(n)$  è sicuramente  $O(n^3)$  ma è  $O(n^2)$ ?

Pb:  $T(1) = 1 \leq c \quad \forall c \geq 1$

PI: Assumiamo  $T(K) \leq ck^2 \quad \forall K < n$

$$T(n) = 4T(n/2) + n \leq n + 4c(n/2)^2 = n + cn^2 \neq cn^2 \quad \text{non lo è?}$$

Proviamo invece a dimostrare che  $T(n) \leq cn^2 - c_2n$

assumiamo:  $T(K) \leq c_1K^2 - c_2K \quad \forall K < n$

$$\begin{aligned} T(n) &= 4T(n/2) + n \leq 4c_1(n/2)^2 - 4c_2(n/2) + n = cn^2 - 2c_2n + n \\ &= cn^2 - c_2n - (c_2n - n) \end{aligned}$$

$$T(n) \leq c_1 n^2 - c_2 n \cdot (c_2 n - n) \leq c_1 n^2 - c_2 n \text{ se } c_2 n - n \geq 0 \quad c_2 \geq 1$$

caso base:

$$T(1) = 1 \leq c_1 - c_2 \quad c_2 \geq 1 \quad c_1 = 2 \quad \text{allora } T(n) \leq 2n^2 - n = O(n^2)$$

Per verificare se è un  $\Theta(n^2)$  bisogna dimostrare per induzione se è anche un  $\Omega(n^2)$  mettendo  $\geq$  e non  $\leq$  vedendo se è sia un upper che lower bound

## Metodo del teorema Master

Questo metodo si utilizza con algoritmi basati sulla tecnica del divide et impera:

- si divide il problema (di dimensione  $n$ ) in  $a$  sottoproblemi di dimensione  $n/b$
- risolvi i sottoproblemi ricorsivamente.
- ricombina le soluzioni.

Sia  $f(n)$  il tempo per dividere e ricombinare le istanze di dimensione  $n$ . La relazione di ricorrenza è data da:

$$T(n) = \begin{cases} aT(n/b) + f(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

e ha soluzione:

$$1. T(n) = \Theta(n^{\log_b a}) \text{ se } f(n) = O(n^{\log_b a - \epsilon}) \quad \epsilon > 0$$

$$2. T(n) = \Theta(n^{\log_b a} \log n) \text{ se } f(n) = \Theta(n^{\log_b a})$$

$$3. T(n) = \Theta(f(n)) \text{ se } f(n) = \Omega(n^{\log_b a + \epsilon}) \quad \epsilon > 0 \text{ e } af(n/b) \leq cf(n) \text{ per } c < 1.$$

Basta quindi vedere quali tra  $n^{\log_b a}$  e  $f(n)$  va più veloce a infinito, se hanno stesso ordine asintotico ci troviamo nel caso 2. altrimenti:  $T(n)$  ha ordine asintotico della più veloce.

Un esempio è fibo6 con:  $a=1, b=2, f(n)=O(1)$

L'algoritmo ottimo di pesatura Alg 4:  $a=1, b=3, f(n)=O(1)$

$$\text{es. } T(n) = n + 2T(n/2) \quad a=2 \quad b=2 \quad f(n)=n = \Theta(n^{\log_2 2})$$

$$T(n) = \Theta(n \log n)$$

$$T(n) = n + 3T(n/3) \quad a=3 \quad b=3 \quad f(n)=n = \Theta(n^{\log_3 3 - \epsilon})$$

$$T(n) = \Theta(n)$$

$$T(n) = c + 3T(n/3) \quad a=3 \quad b=3 \quad f(n)=c = \Omega(n^{\log_3 3 + \epsilon})$$

$$T(n) = \Theta(\sqrt{n})$$

In alcuni casi non è possibile utilizzare il teorema Master

$$T(n) = n \log n + 2T(n/2)$$

$$a=2, b=2, f(n) \neq \Omega(n \log^2 n + \epsilon) \text{ e } f(n) \neq O(n \log^{1-\epsilon}) \quad \forall \epsilon > 0$$

in questo caso non è possibile utilizzare il teorema master.

### Metodo del cambiamento di variabile

Quando ho dei casi che utilizzano esponenti, posso utilizzare il metodo del cambio di variabile

es.

$$T(n) = T(n^{\frac{1}{2}}) + O(1), T(1) =$$

$$n = 2^x \rightarrow x = \log_2 n \quad T(2^x) = T(2^{x/2}) + O(1) \quad R(x) = T(2^x)$$

$$R(x) = R(x/2) + O(1) \text{ quindi } R(x) = O(\log x) \text{ e } T(n) = O(\log \log n)$$

## Lezione 6

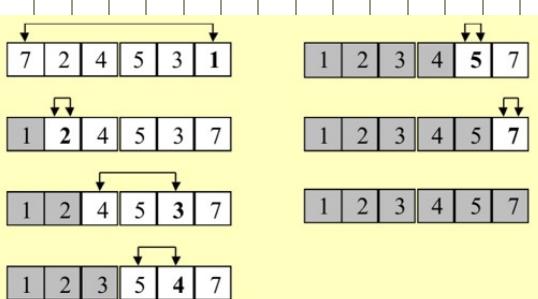
Dato un insieme  $S$  di  $n$  oggetti da un dominio totalmente ordinato, ordinare  $S$ .

Questo algoritmo è una subroutine in molti problemi, è possibile effettuare ricerche in array ordinati in tempo  $O(\log n)$  tramite la ricerca binaria.

In un problema di ordinamento in input è presente una sequenza di  $n$  numeri e in output una permutazione  $\langle a'_1, a'_2, \dots, a'_n \rangle$  della sequenza di input tale che  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Esistono vari algoritmi.

### Selection Sort



Approccio incrementale: estende l'ordinamento da  $K$  a  $K+1$  elementi, scegliendo il minimo degli  $n-K$  elementi non ancora ordinati e mettendolo in posizione  $K+1$ .

## SelectionSort (A)

```

for K=0 to n-2 do      K come valore di riferimento (si parte da m=K+1 cioè il primo elemento)
    m=K+1                il primo elemento come il minimo
    for j=K+2 to n do    dall'elemento successivo
        if (A[j] < A[m])  verifica se j è un minore
            then m=j        se sì, metti il suo indice come minore
    scambia A[m] con A[K+1]  cambia con il min appena trovato

```

Correttezza:

E' facile convincersi che l'algoritmo mantiene le seguenti invarianti: dopo il generico passo K ( $K=0, \dots, n-2$ ) abbiamo che:

- i primi  $K+1$  elementi sono ordinati e
- sono i  $K+1$  elementi più piccoli dell'array

Ragionare per invarianti è uno strumento utile per dimostrare la correttezza di un algoritmo, perché permette di isolare proprietà dell'algoritmo, spiegarne il funzionamento e capire a fondo l'idea su cui si basa.

Complessità:

$T(n) = \#$  operazioni elementari sul modello RAM a costi uniformi eseguite dall'algoritmo nel caso peggiore su istanze di dimensione  $n$ .

Il ciclo esterno viene eseguito al più  $n$  volte (1-5), il ciclo interno (3-5) è eseguito al più  $n$  volte per ogni ciclo esterno quindi sicuramente

$$T(n) \leq 5n^2 O(1) = \Theta(n^2) \rightarrow T(n) = O(n^2) \quad \text{upper bound}$$

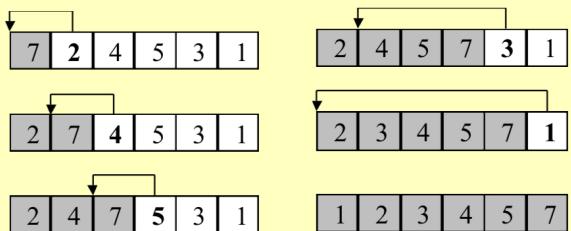
Se conto solo i confronti fra elementi, fai  $n-K-1$  confronti per ogni ciclo

$$T(n) \geq \sum_{K=0}^{n-2} (n-K-1) = \sum_{K=1}^{n-1} K = n(n-1)/2 = \Theta(n^2) \rightarrow T(n) = \Omega(n^2) \quad \text{lower bound}$$

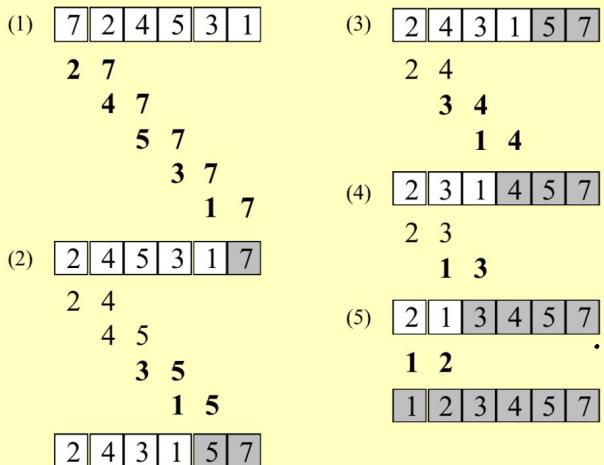
quindi  $T(n) = \Theta(n^2)$

$$\sum_{K=0}^{n-2} K = n-1 + n-2 + \dots + n-(n-2)-1 = 1 + 2 + \dots + n-1 = \sum_{K=1}^{n-1} K$$

## Insertion Sort



## Bubble Sort



## InsertionSort(A)

```

for k=1 to n-1 do
  x = A[k+1]
  for j=1 to k+1 do
    if (A[j] > x) then
      break
    if (j < k+1) then
      for t=k down to j do
        A[t+1] = A[t]
    A[j] = x
  
```

Approccio incrementale: estende l'ordinamento da  $k$  a  $k+1$  elementi, posizionando l'elemento  $(k+1)$ -esimo nella posizione corretta rispetto ai primi  $k$  elementi.

Approccio incrementale: esegue  $n$ -scansioni. Ad ogni scansione guarda coppie di elementi adiacenti e li scambia se non sono nell'ordine corretto.

## BubbleSort(A)

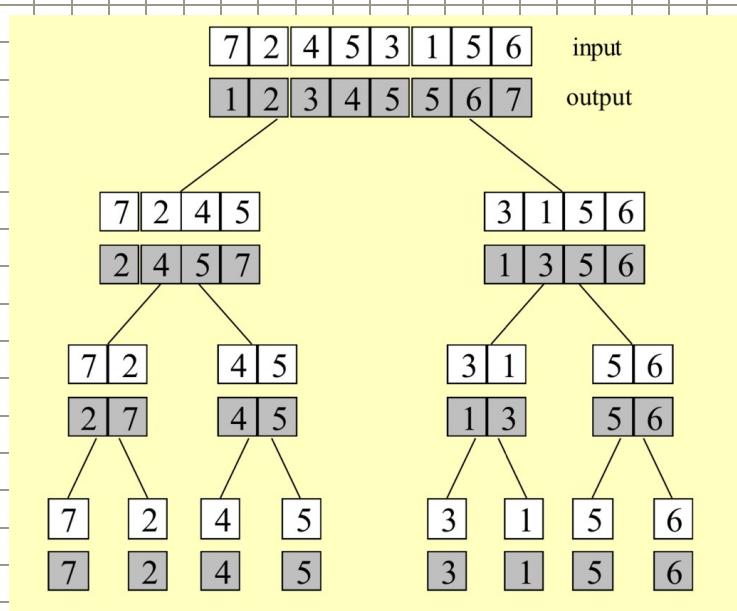
```

for i=1 to n-1 do
  for j=2 to n-i+1
    if A[j-1] > A[j] then
      scambia A[j-1] con A[j]
    if (non ci sono stati scambi) then
      break
  
```

## MergeSort

Usa la tecnica del divide et impera:

1. Dividi: divide l'array a metà
2. Risolvi i due sottoproblemi ricorsivamente
3. Impera: fonda le due sottosequenze ordinate



MergeSort ( $A, i, f$ )

:if ( $i < f$ ) then

$m = \lfloor (i+f)/2 \rfloor$

    MergeSort ( $A, i, m$ )

    MergeSort ( $A, m+1, f$ )

    Merge ( $A, i, m, f$ )

Due array ordinati  $A$  e  $B$  possono essere fusi rapidamente:

- estrai ripetutamente il minimo di  $A$  e  $B$  e copiarlo nell'array di output, finché  $A$  oppure  $B$  non diventa vuoto
- copia gli elementi dell'array non vuoto alla fine dell'array non vuoto alla fine dell'array di output

Questo algoritmo viene chiamato Merge:

## Merge(A, i, f<sub>1</sub>, f<sub>2</sub>)

Sia X un array ausiliario di lunghezza  $f_2 - i_1 + 1$  in caso di solo la parte  
della array è non è per esempio

$j = 1; k_1 = i_1; k_2 = f_1 + 1$   $k_1$  inizio prima array,  $k_2$  inizio secondo

while ( $k_1 \leq f_1$  e  $k_2 \leq f_2$ ) do

if ( $A[k_1] \leq A[k_2]$ )

then  $X[j] = A[k_1]$

$j++$ ;  $k_1++$

else  $X[j] = A[k_2]$

$j++$ ;  $k_2++$

if ( $k_1 \leq f_1$ )

then copia  $A[k_1; f_1]$  alla fine di X

else copia  $A[k_2; f_2]$  alla fine di X

copia X in  $A[i_1; f_2]$

utilizza un array ausiliario  
non si può senza

## Lemme

La procedura Merge fonde due sequenze ordinate di lunghezza  $n_1$  e  $n_2$  in tempo  $\Theta(n_1 + n_2)$

## Dim

Ogni confronto "consuma" un elemento di una delle due sequenze. Ogni posizione di X è riempita in tempo costante. Il numero totale di elementi è  $n_1 + n_2$ . Anche l'ultima linea (copia) costa  $\Theta(n_1 + n_2)$ .

Correttezza: corretto, chiamate ricorsive ordinano le due metà e il Merge le fonde

## Tempo di esecuzione:

La complessità temporale del MergeSort è descritta dalla seguente relazione di ricorrenza:  $T(n) = 2T(n/2) + O(n)$

Usando il Teorema Master si ottiene  $T(n) = O(n \log n)$   $a=b=2$ ,  $f(n)=O(n)$

Usiamo però un array ausiliario, la complessità spaziale del MergeSort è  $\Theta(n)$ .

La procedura merge utilizza memoria ausiliaria pari alla dimensione di porzione da fondere;

Non sono mai attive oltre procedure di Merge contemporaneamente;

Ogni chiamata di MergeSort usa memoria costante (esclusa quella usata dalla procedura Merge);

Numero di chiamate di MergeSort attive contemporaneamente sono  $O(\log n)$ ;

Il MergeSort non ordina in loco, occupa memoria ausiliaria  $\Theta(n)$ .

Un altro algoritmo per ordinare un array è il QuickSort

Usa la tecnica del divide et impera:

1. Divide: scegli un elemento  $x$  della sequenza (perno) e partiziona la sequenza in elementi  $\leq x$  ed elementi  $> x$ .

2 Risolvi i due sotto problemi ricorsivamente.

3 Impera: restituisce la concatenazione delle due sottosequenze ordinate.

Rispetto al MergeSort, divide complesso ed impera semplice.

Partizione (in loco)

Scegli il perno,

Scorri l'array "in parallelo" da sinistra verso destra e da destra verso sinistra: da sinistra verso destra, ci si ferma su un elemento maggiore del perno, da destra verso sinistra, ci si ferma su un elemento minore del perno.

Scambia gli elementi e riprendi la scansione

45	12	93	3	67	43	85	29	24	92	63	3	21
45	12	21	3	67	43	85	29	24	92	63	3	93
45	12	21	3	3	43	85	29	24	92	63	67	93
45	12	21	3	3	43	24	29	85	92	63	67	93
45	12	21	3	3	43	24	29	85	92	63	67	93

## Partition(A, i, f)

$x = A[i]$

$inf = i$

$sup = f + i$

while (true) do

do ( $inf = inf + 1$ ) while ( $inf \leq f$  e  $A[inf] \leq x$ )

do ( $sup = sup - 1$ ) while ( $A[sup] > x$ )

if ( $inf < sup$ ) then scambia  $A[inf]$  e  $A[sup]$

else break

scambia  $A[i]$  e  $A[sup]$  mette perno al centro

return sup restituisce l'indice del perno

Proprietà invariante: In ogni istante, gli elementi  $A[i], \dots, A[inf-1]$  sono  $\leq$  del perno, mentre gli elementi  $A[sup+1], \dots, A[f]$  sono  $>$  del perno.

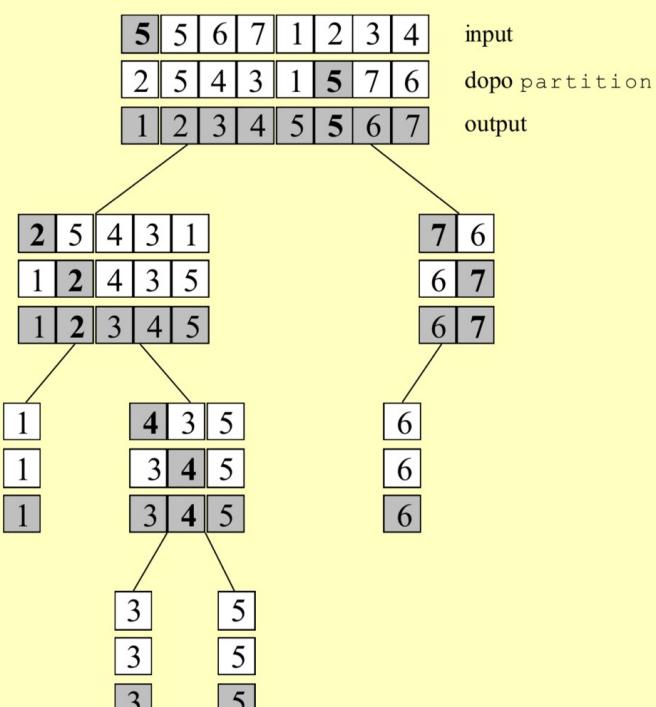
## QuickSort(A, i, f)

if ( $i < f$ ) then

$m = Partition(A, i, f)$

QuickSort(A, i, m-1)

QuickSort(A, m+1, f)



## Esempio di esecuzione

L'albero delle chiamate ricorsive può essere sbilanciato.

Correttezza: corretto, dopo Partition  $A[i:m-1]$  contiene elem  $\leq$  del perno,  $A[m]$  il perno e  $A[m+1:f]$  elem  $\geq$  del perno.

Le chiamate ricorsive ordinano  $A[i:f]$

Complessità caso peggiore:

Ogni invocazione di Partition posiziona almeno un elemento in modo corretto (il perno).

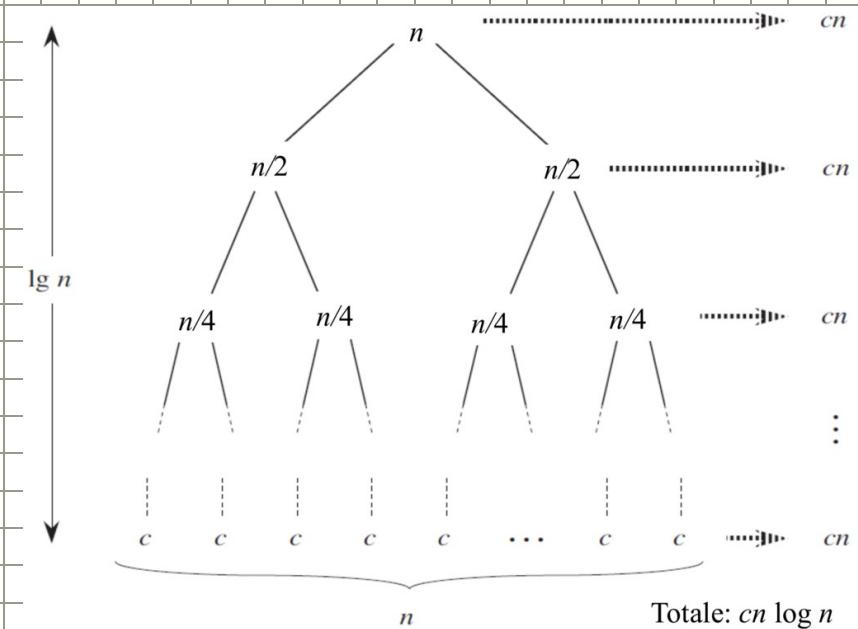
Quindi dopo  $n$  invocazioni di Partition, ognuna di costo  $O(n)$  ho il vettore ordinato. Il costo complessivo è quindi  $O(n^2)$ .

Il caso peggiore si verifica quando il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array.

La complessità in questo caso è:

$$\begin{aligned} T(n) &= T(n-1) + T(1) + O(n) \\ &= T(n-1) + O(1) + O(n) = \\ &= T(n-1) + O(n) = T(n) = O(n^2) \end{aligned}$$

Nel caso migliore  $O(n \log n)$  partizionamento sempre bilanciato, il perno è sempre il mediano

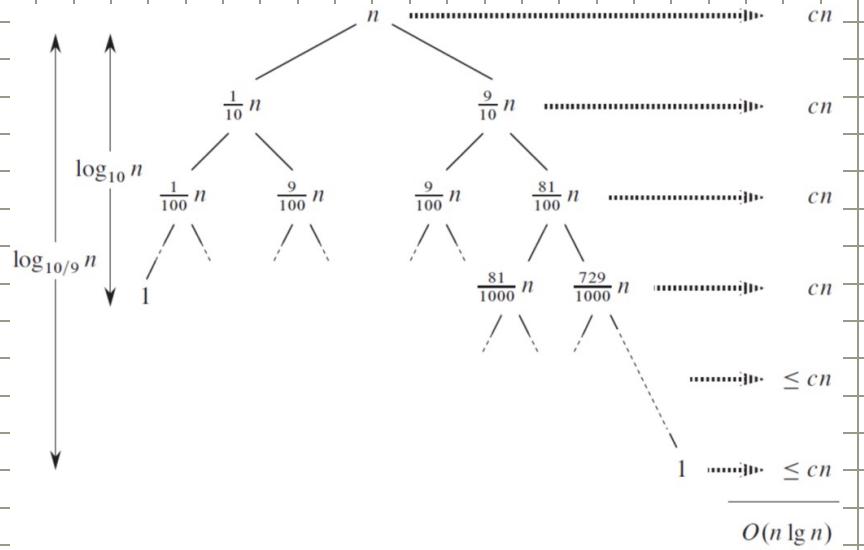


Nel caso medio, penso al caso di istanze equilibrate:

La partizione può essere sbilanciata, la probabilità che ad ogni passo si presenti la partizione peggiore è molto bassa, per partizioni che non sono "troppo sbilanciate".  
L'algoritmo è veloce.

Qual'è la complessità dell'algoritmo supponendo che l'algoritmo di partizionamento produca sempre una partizione 9-2-1? 99-2-1?

L'albero sarà così rappresentato:



Per migliorare il Quicksort si può randomizzare, scegliendo il perno  $x$  a caso fra gli elementi da ordinare.

L'algoritmo quickSort randomizzato ordina in loco un array di lunghezza  $n$  in tempo  $O(n^2)$  nel caso peggiore e  $O(n \lg n)$  con alta probabilità, ovvero con probabilità almeno  $1 - 1/n$ .

Il Quicksort randomizzato è diverso dal caso medio:

nessuna assunzione sulla distribuzione di probabilità delle istanze, nessun input specifico per il quale si verifica il caso peggiore infatti il caso peggiore determinato solo dal generatore di numeri casuali.

## Lezione 7

Progettare algoritmi veloci usando strutture dati efficienti come l'HeapSort

Usa lo stesso approccio incrementale del SelectionSort, seleziona gli elementi dal più grande al più piccolo e usa una struttura dati efficiente che permette di estrarre in tempo  $O(\lg n)$  il massimo.

Esiste una differenza tra il tipo di dato e una struttura dati:

**Tipo di dato:** Specifica una collezione di oggetti e delle operazioni di interesse su tale collezione (Dizionario)

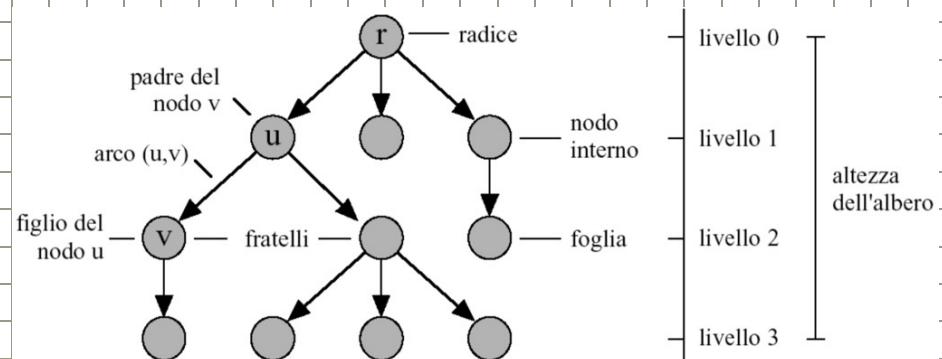
**Struttura dati:** Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile.

**HeapSort:** Progettare una struttura dati  $H$  su cui eseguire efficientemente le operazioni:

- dato un array, generare velocemente  $H$
- trovare il più grande oggetto in  $H$
- cancellare il più grande oggetto in  $H$

Il tipo di dato associato: coda con priorità

### Definizioni



lunghezza cammino = numero archi passati

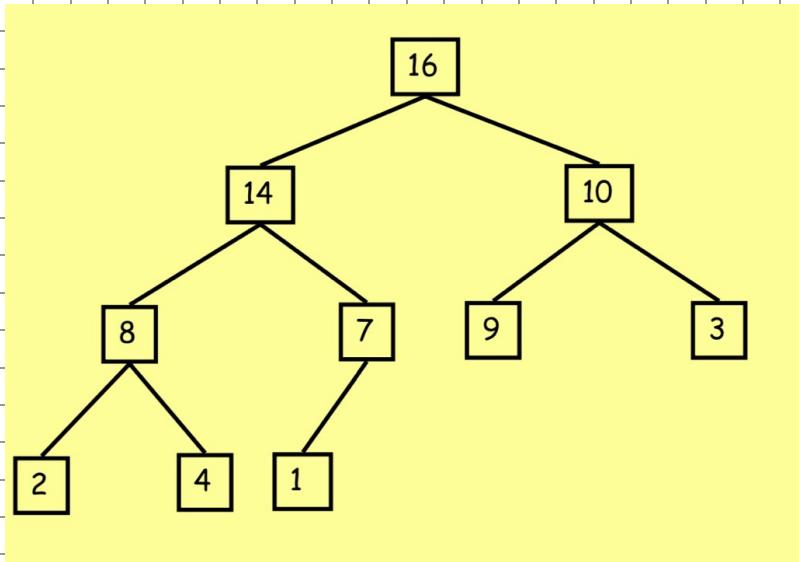
profondità: distanza rispetto alla radice (livello)

albero d-ario: albero in cui tutti i nodi interni hanno (al più)  $d$  figli.

un albero d-ario è **completo** se tutti i nodi interni hanno esattamente  $d$  figli e le foglie sono tutte allo stesso livello.

**HeapSort** è una struttura dati heap associata ad un insieme  $S$  = albero binario radicato con le seguenti proprietà:

- 1) completo fino al penultimo livello (struttura rafforzata: foglie sull'ultimo livello tutte compatte a sinistra)
- 2) gli elementi di  $S$  sono memorizzati nei nodi dell'albero (ogni nodo  $v$  memorizza uno e un solo elemento, denotato con chiave( $v$ ))
- 3) chiave(padre( $v$ ))  $\geq$  chiave( $v$ )  $\forall v \neq \text{root}$ .



è presente  
un ordinamento

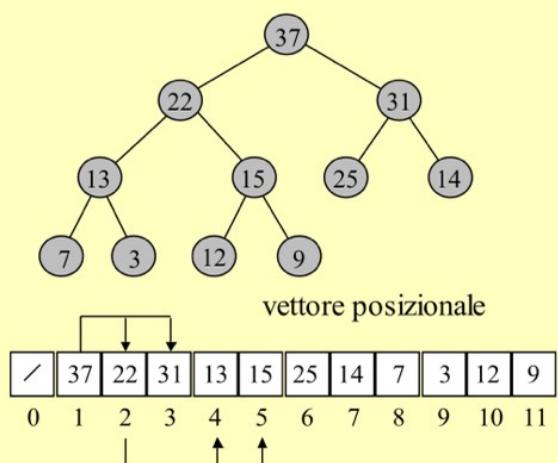
non è presente un ordinamento

### Proprietà degli heap

- 1) Il massimo è contenuto nella radice
- 2) L'albero ha altezza  $O(\log n)$
- 3) Gli heap con una struttura rafforzata possono essere rappresentati in un array di dimensione  $n$ .

La dimostrazione di (1) è banale.

Dim. rappresentazione tramite vettore posizionale (3)



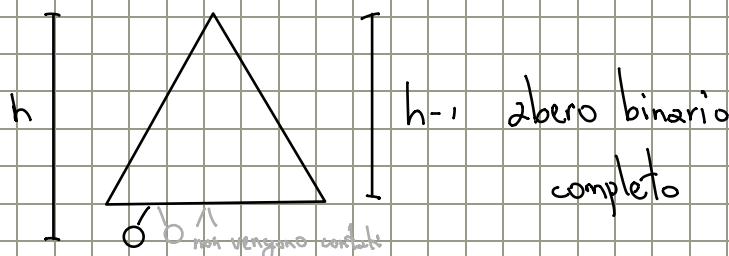
$$\begin{aligned} \text{sin}(i) &= 2i \\ \text{des}(i) &= 2i+1 \\ \text{padre}(i) &= \lfloor i/2 \rfloor \end{aligned}$$

la dimensione del vettore è diverso dal numero di elementi

Nello pseudocodice il numero di oggetti è indicato con `heapsize[A]` e volte memorizzato nella posizione 0, e diverso dalla lunghezza di `A`.

Dim. Altezza di un heap (2)

Sia  $h$  un heap di  $n$  nodi e altezza  $h$



$$n \geq 1 + \sum_{i=0}^{h-1} 2^i = 1 + 2^h - 1 = 2^h \quad h \leq \log_2 n$$

### fix Heap

Sia  $v$  la radice di  $H$ , si assume che i sottoalberi radicati nel figlio sinistro e destro di  $v$  sono heap, ma le proprietà di ordinamento delle chiavi non vale per  $v$ . Posso ripristinarla così:

fix Heap ( $i, A$ )

$s = \text{sin}(i)$

$d = \text{des}(i)$

if ( $s \leq \text{heapsize}[A]$  e  $A[s] > A[i]$ )

then massimo =  $s$

else massimo =  $i$

if ( $d \leq \text{heapsize}[A]$  e  $A[d] > A[\text{massimo}]$ )

then massimo =  $d$

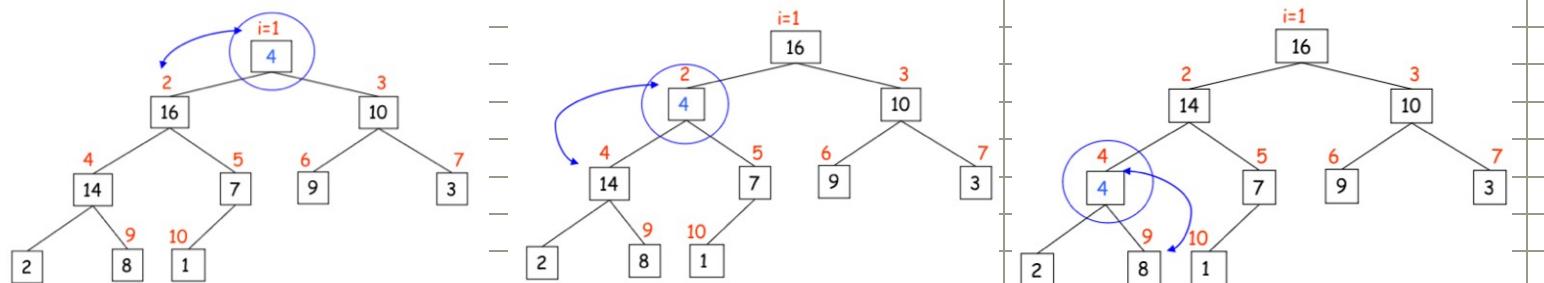
if (massimo  $\neq i$ )

then scambia  $A[i]$  e  $A[\text{massimo}]$

fix Heap (massimo,  $A$ )

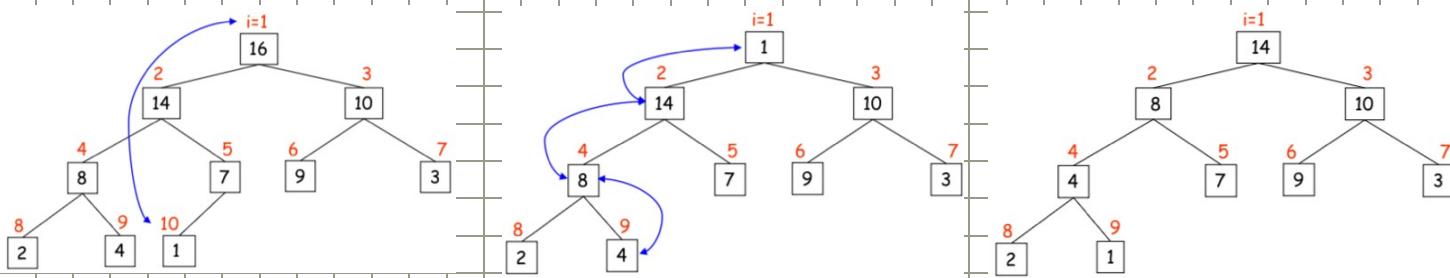
tempo di esecuzione:  $O(h \log n)$

altezza max =  $\log n$



## Estrazione del massimo:

- Copia nella radice la chiave contenuta nella foglia più a destra dell'ultimo livello (elemento in posizione heap-size)
- Rimuovi la foglia (decrementare heap-size)
- Ripristina la proprietà di ordinamento a heap richiamando fixHeap sulla radice



## Costruzione dell'heap:

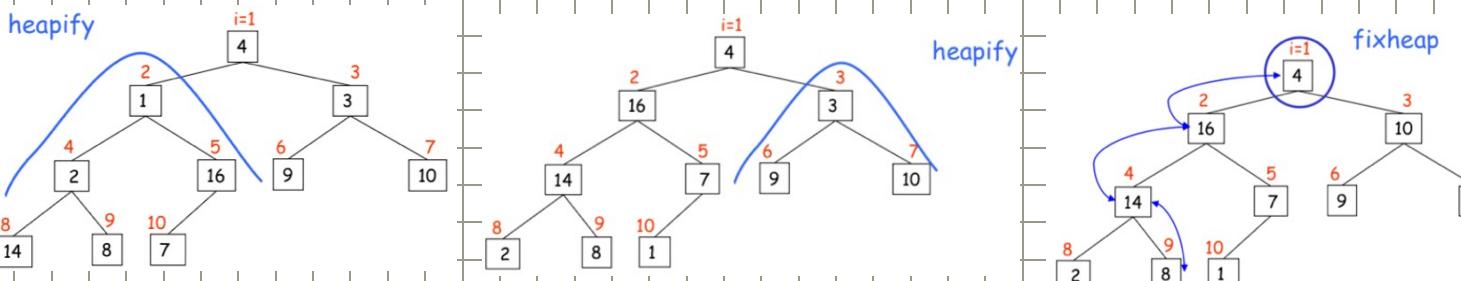
heapify (heap H)

if (H non vuoto) then

    heapify (sottoalbero sx di H)

    heapify (sottoalbero dx di H)

    fixHeap (radice di H, H)



## Complessità heapify

Sia  $h$  l'altezza di un heap con  $n$  elementi.

Sia  $n' \geq n$  l'intero tale che un heap con  $n'$  elementi ha altezza  $h$  ed è completo fino all'ultimo livello

$$T(n) \leq T(n') \text{ e } n' \leq 2n$$

$$\begin{aligned} T(n') &= 2T((n'-1)/2) + O(\log n) & n \leq n' \leq 2n \\ &\leq 2T(n'/2) + O(\log n) \end{aligned}$$

$$T(n') = O(n') \quad T(n) \leq T(n') = O(n)$$

## Max-Heap e Min-Heap

Se vogliamo una struttura dati che mi permette di estrarre il minimo, costruisco un min-heap invertendo le proprietà di ordinamento delle chiavi:  
 chiave(padre( $v$ ))  $\leq$  chiave( $v$ )  $\forall v \neq \text{root}$ .

L'uso del max-heap rispetto al min-heap ci permette di usare solo memoria costante

senza allo fine di HeapSort dovranno ribaltare l'array

## HeapSort

Costruisce un heap tramite heapify e estrae ripetutamente il massimo  $n-1$  volte (ad ogni estrazione memorizza il massimo nella posizione dell'array che si è appena liberata)

### heapSort(A)

Heapify(A) crea albero

$\} O(n)$

Heapsize[A] = n lunghezza

for  $i=n$  down to 2 do per tutti i nodi tranne rad.

scambia  $A[i]$  e  $A[1]$  Scambia root e min

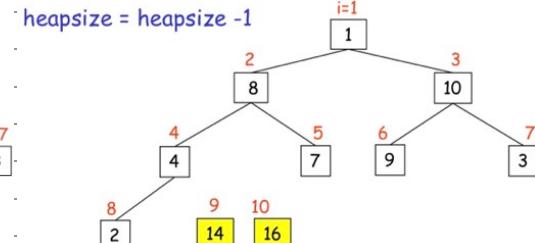
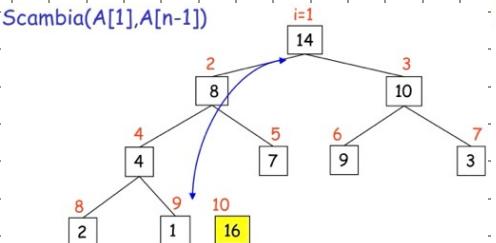
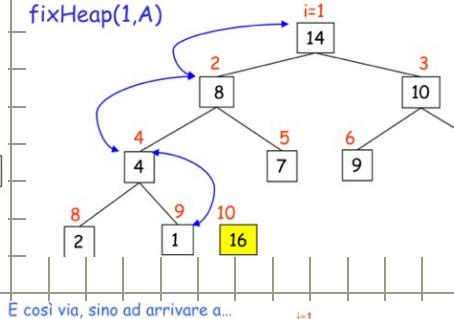
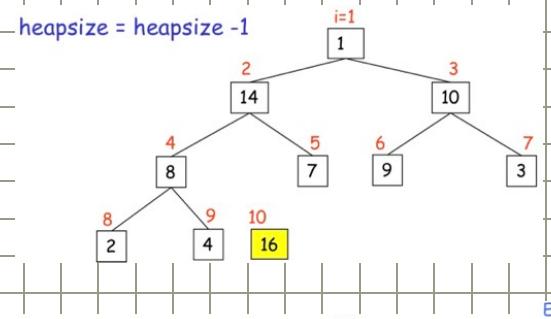
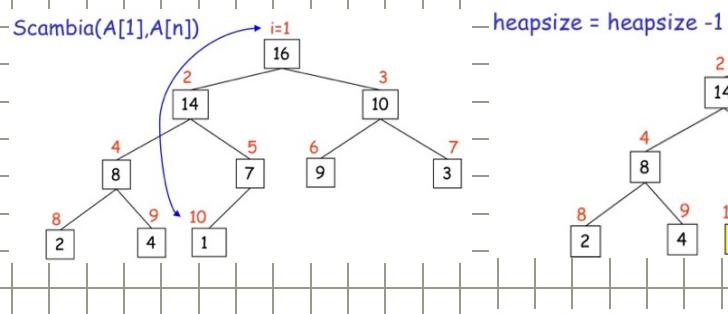
Heapsize[A] = Heapsize[A] - 1 estraggo il max

fixHeap(1, A) aggiusto l'heap

$n-1$  estrazioni di costo  $O(1 \log n)$

ordina in loco  $O(n \log n)$

L'algoritmo **HeapSort** ordina in loco un array di lunghezza  $n$  in tempo  $O(n \log n)$  nel caso peggiore.



## Lezione 8

C'è una differenza tra la complessità di un algoritmo e quello di un problema:

### Complessità algoritmo:

#### - upper bound

Un algoritmo A ha complessità (costo di esecuzione)  $O(f(n))$  rispetto ad una certa risorsa di calcolo, se la quantità  $r(n)$  di risorsa usata da A nel caso peggiore su istanze di dimensione n verifica la relazione  $r(n) = O(f(n))$ .

#### - lower bound

Un algoritmo A ha complessità (costo di esecuzione)  $\Omega(f(n))$  rispetto ad una certa risorsa di calcolo, se la quantità  $r(n)$  di risorsa usata da A nel caso peggiore su istanze di dimensione n verifica la relazione  $r(n) = \Omega(f(n))$ .

### Complessità problema:

#### - upper bound

Un problema P ha una complessità  $O(f(n))$  rispetto ad una risorsa di calcolo se esiste un algoritmo che risolve P il cui costo di esecuzione rispetto quella risorsa è  $O(f(n))$ .

#### - lower bound

Un problema P ha una complessità  $\Omega(f(n))$  rispetto ad una risorsa di calcolo se ogni algoritmo che risolve P ha costo di esecuzione nel caso peggiore  $\Omega(f(n))$  rispetto quella risorsa.

### Ottimalità di un algoritmo

Dato un problema P con complessità  $\Omega(f(n))$  rispetto ad una risorsa di calcolo, un algoritmo che risolve P è (asintoticamente) ottimo se ha costo di esecuzione  $O(f(n))$  rispetto a quella risorsa.

### Complessità temporale del problema dell'ordinamento

$O(n^2)$ : Insertion, Selection, Quick, Bubble

$O(n \log n)$ : Merge, Heap

$\Omega(n)$ : banale, ogni algoritmo che ordina n elementi li deve leggere tutti

Esistono due tipi di ordinamento: per confronto e non

Per confronto: Dati due elementi  $a_i$  e  $a_j$ , per determinare l'ordinamento relativo effettuiamo una delle seguenti operazioni di confronto:

$$a_i < a_j ; a_i \leq a_j ; a_i > a_j ; a_i \geq a_j ; a_i = a_j$$

Non si possono esaminare i valori degli elementi o ottenere informazioni sul loro ordine in altro modo.

**Teorema:**

Ogni algoritmo basato su confronti che ordina  $n$  elementi deve fare nel caso peggiore  $\Omega(n \log n)$  confronti.

si noti che il # di confronti che un algoritmo esegue è un lowerbound al # di passi elementari che esegue.

**Corollario**

Il MergeSort e l'HeapSort sono algoritmi ottimi (nella classe di Algo. basati su confronto)

Per dimostrare il teorema del problema degli algoritmi di ordinamento basato su confronti utilizziamo gli alberi di decisione e un generico algoritmo di ordinamento basato su confronti.

Un generico algoritmo di ordinamento per confronto lavora nel modo seguente:

- confronta due elementi  $a_i$  ed  $a_j$  (test:  $a_i \leq a_j$ )
- a seconda del risultato riordina e/o decide il confronto successivo da eseguire.

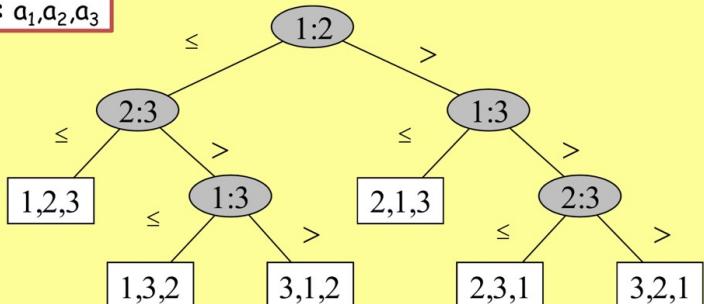
L'albero di decisione descrive i confronti che l'algoritmo esegue quando opera su un input di una determinata dimensione. I movimenti dei dati e tutti gli altri aspetti vengono ignorati.

L'albero di decisione descrive le diverse sequenze di confronti che A potrebbe fare su istanze di dimensione  $n$ .

Nodo interno:  $i:j$  confronto tra  $a_i : a_j$

Nodo foglia: permutazione degli elementi

Input:  $a_1, a_2, a_3$



se cambio algoritmo l'albero  
cambia

## Osservazioni

L'albero di decisione:

- non è associato ad un problema.
- non è associato solo ad un algoritmo.
- è associato ad un algoritmo e a una dimensione dell'istanza.
- descrive le diverse sequenze di confronti che un certo algoritmo può eseguire su istanze di una data dimensione.
- è una descrizione alternativa dell'algoritmo.

## Proprietà

Per una particolare istanza, i confronti eseguiti dall'algoritmo su quella istanza rappresentano un cammino radice-foglia.

L'algoritmo segue un cammino diverso a seconda delle caratteristiche dell'istanza: il caso peggiore è il cammino più lungo.

Il numero di confronti nel caso peggiore è pari all'altezza dell'albero di decisione.

Un albero di decisione di un algoritmo (corretto) che risolve il problema dell'ordinamento di  $n$  elementi deve avere necessariamente almeno  $n!$  foglie.

## Lemme

Un albero binario  $T$  con  $K$  foglie, ha altezza almeno  $\log_2 K$

Dim. (per induzione su  $K$ )

caso base:  $K=1$  altezza almeno  $\log_2 1 = 0$

caso induttivo:  $K > 1$

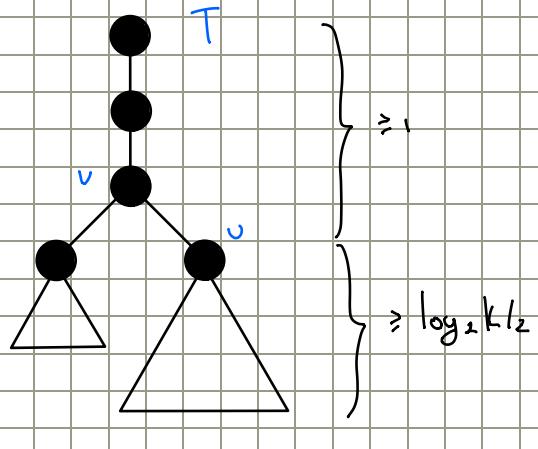
considera il nodo interno  $v$  più vicino alla radice che ha due figli ( $v$  potrebbe essere la radice).  $v$  deve esistere perché  $K > 1$ .

$v$  ha almeno un figlio  $u$  che è radice di un (sotto) albero che ha almeno  $K/2$  foglie e  $< K$  foglie.

$T$  ha altezza almeno  $1 + \log_2 K/2 =$

$$= 1 + \log_2 K - \log_2 2 =$$

$$= \log_2 K$$



Consideriamo l'albero di decisione di un qualsiasi algoritmo che risolve il problema dell'ordinamento di  $n$  elementi.

L'altezza  $h$  dell'albero di decisione è almeno  $\log_2(n!)$ .

Formula di Stirling:  $n! \approx (2\pi n)^{1/2} \cdot (n/e)^n$

$$\begin{aligned} h &\geq \log_2(n!) > \log_2(n/e)^n = \\ &= n \log_2(n/e) = \\ &= n \log_2 n - n \log_2 e = \\ &= \Omega(n \log n). \end{aligned}$$

## IntegerSort

Per ordinare  $n$  interi con valori in  $[1, K]$ , l'IntegerSort mantiene un array  $Y$  di  $K$  contatori tale che  $Y[x] = \text{numero di volte che il valore } x \text{ compare in } X$ .

Si scorre  $Y$  da sinistra verso destra e, se  $Y[x] = K$ , scrive in  $X$  il valore  $x$  per  $K$  volte

X	5   1   6   8   6	5   1   6   8   6	5   1   6   8   6
Y	0   0   0   0   1   0   0   0	1   0   0   0   1   0   0   0	1   0   0   0   1   1   0   0
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8
X	5   1   6   8   6	5   1   6   8   6	5   1   6   8   6
Y	1   0   0   0   1   1   0   1	1   0   0   0   1   2   0   1	
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	

(a) Calcolo di Y

fase 1

X	1	1	1   5
Y	1   0   0   0   1   2   0   1	0   0   0   0   1   2   0   1	0   0   0   0   1   2   0   1
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8
X	1   5   6   6	1   5   6   6	1   5   6   6   8
Y	0   0   0   0   0   2   0   1	0   0   0   0   0   0   0   1	0   0   0   0   0   0   0   1
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8

(b) Ricostruzione di X

fase 2

## IntegerSort ( $X, K$ )

Sia  $Y$  un array di dimensione  $K$

for  $i=1$  to  $K$  do  $Y[i] = 0$

for  $i=1$  to  $n$  do incrementa  $Y[X[i]]$

$j=1$

for  $i=1$  to  $K$  do

while ( $Y[i] > 0$ ) do

$X[j] = i$

$j++$

$Y[i]--$

$$\sum_{i=1}^K (i + Y[i]) = \sum_{i=1}^K i + \sum_{i=1}^K Y[i] = K + n$$

### Analisi

Tempo  $O(1) + O(K) = O(K)$  per inizializzare  $Y$  a 0.

Tempo  $O(1) + O(n) = O(n)$  per calcolare i valori dei contatori.

Tempo  $O(n+K)$  per ricostruire  $X$ .

Tempo lineare se  $K = O(n)$

## Bucket Sort

Per ordinare  $n$  record con campi (nome, cognome...), con chiavi intere in  $[1, K]$

Input del problema:

- $n$  record mantenuti in un array
- ogni elemento di un array è un record con campo chiave (rispetto al quale ordinare) e altri campi associati alla chiave (informazioni satellite).

Basta mantenere un array di liste, anziché di contatori, ed operare come per IntegerSort.

La lista  $Y[i]$  conterrà gli elementi con chiave uguale a  $i$ .

Concatenare poi le liste. Tempo  $O(n+K)$  come per IntegerSort.

$O(1)$  inizializza array di lunghezza  $K$

$O(K)$  lo imposti a 0

$O(n)$  fase 1

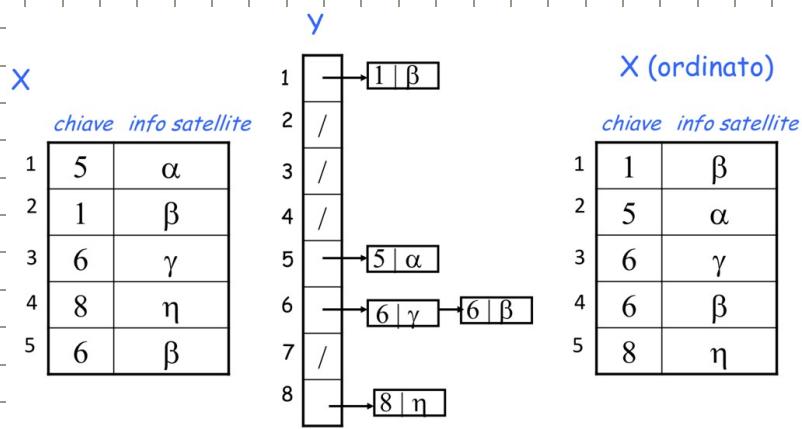
$O(1)$

$O(K)$  per ogni elemento dell'array  $Y$

} per  $i$  fissato

# volte eseguite  $\rightarrow O(K+n)$

}  $i$  al più  $i + Y[i]$



## BucketSort(X, K)

Sia Y un array di dimensione K

for  $i=1$  to  $K$  do  $Y[i]$  = lista vuota

for  $i=1$  to  $n$  do

appendi il record  $X[i]$  alla lista  $Y[\text{chiave}(X[i])]$

for  $i=1$  to  $K$  do

copia ordinatamente in X gli elementi della lista  $Y[i]$

## Stabilità:

Un algoritmo è **stabile** se preserva l'ordine iniziale tra elementi con la stessa chiave. Il BucketSort è stabile se si appendono gli elementi di X in coda alla opportuna lista  $Y[i]$

## Radix Sort

ricordarsi sempre quale è l'input del problema e quale problema l'algoritmo risolve

Ordina  $n$  interi con valori in  $[l, K]$

Rappresentiamo gli elementi in base  $b$ , ed eseguiamo una serie di

## BucketSort.

Partiamo dalla cifra meno significativa verso quella più significativa:

- Ordiniamo per l' $i$ -esima cifra con una passata di BucketSort. (stabile)
- $i$ -esima cifra è la chiave, il numero info satellite
- $i$ -esima cifra è un intero in  $[0, b-1]$

Per  $b=10$

2397	5924	5924	4368	4368	2397
4368	2397	4368	2397	2397	4368
5924	4368	2397	5924	5924	5924

## Correttezza

Se  $x$  e  $y$  hanno una diversa  $t$ -esima cifra, la  $t$ -esima passata di BucketSort li ordina. Se  $x$  e  $y$  hanno la stessa  $t$ -esima cifra, la proprietà di stabilità del BucketSort li mantiene ordinati correttamente.

Dopo la  $t$ -esima passata di BucketSort, i numeri sono correttamente ordinati rispetto alle  $t$  cifre meno significative.

## Complessità

$O(\log_b K)$  passate di BucketSort, # di cifre per rappresentare il valore massimo  $K$  in base  $b$ :  $O(\log_b K)$

Ciascuna passata richiede tempo  $O(n+b)$ , in ogni passata la chiave è un intero in  $[0, b-1]$

Costo:  $O((n+b) \log_b K)$

Se  $b = \Theta(n)$ , si ha  $O(n \log_n K) = O\left(n \frac{\log K}{\log n}\right)$

$$\log_2 K = \log_n K \log_2 n$$

Tempo lineare se  $K = O(n^c)$ ,  $c$  costante

Ordinare  $10^6$  numeri da 32 bit,  $10^6$  è compreso fra  $2^{19}$  e  $2^{20}$  scegliendo  $b = 2^{16}$  si ha: sono sufficienti 2 passate di bucketSort e ogni passata richiede tempo lineare.

Conviene scegliere come base una potenza di 2 in quanto passare da binario ad una potenza di due richiede meno operazioni per una macchina.

$$11001010_2 \rightarrow 11/00/10/10 \rightarrow 3022_4$$

## Lezione 9

Tipo di dato: Dizionario

dati: Un insieme  $S$  di coppie (elem, chiave).

operazioni: insert(elem e, chiave k)

aggiunge a  $S$  una nuova coppia (e, k).

delete(chiave k)

cancella da  $S$  la coppia con chiave k.

search (chiave K)  $\rightarrow$  elem

se la chiave K è presente in S restituisce l'elemento e ad essa associato, e null altrimenti.

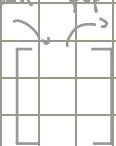
Tipo di dato: Pilza

dati: Una sequenza S di n elementi.

operazioni: isEmpty()  $\rightarrow$  result

restituisce true se S è vuota, e false altrimenti.

push (elem e)

aggiunge e come ultimo elemento di S.

pop ()  $\rightarrow$  elem

toglie da S l'ultimo elemento e lo restituisce.

top ()  $\rightarrow$  elem

restituisce l'ultimo elemento di S (senza toglierlo da S).

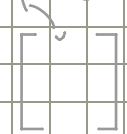
Tipo di dato: Coda

dati: Una sequenza S di n elementi.

operazioni: isEmpty()  $\rightarrow$  result

restituisce true se S è vuota, e false altrimenti.

enqueue (elem e)

aggiunge e come ultimo elemento di S.

dequeue ()  $\rightarrow$  elem

toglie da S il primo elemento e lo restituisce.

first ()  $\rightarrow$  elem

restituisce il primo elemento di S (senza toglierlo da S).

## Tecniche di rappresentazione dei dati:

Rappresentazioni **indizzate**: i dati sono contenuti (principalmente) in array  
Rappresentazioni **collegate**: i dati sono contenuti in record collegati fra loro mediante puntatori

### Proprietà:

#### Rappresentazione indizzata:

- Array: collezione di celle numerate che contengono elementi di un tipo prestabilito

Proprietà (forte): gli indici delle celle di un array sono numeri consecutivi

Proprietà (debole): non è possibile aggiungere nuove celle ad un array

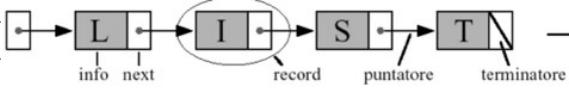
#### Rappresentazioni collegate:

- i costituenti di base sono i record
- i record sono numerati tipicamente con il loro indirizzo di memoria
- record creati e distrutti individualmente e dinamicamente
- il collegamento tra un record A e un record B è realizzato tramite un puntatore

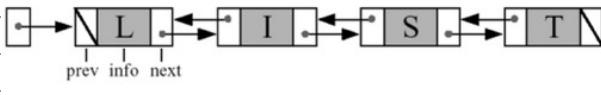
Proprietà (forte): è possibile aggiungere o togliere record a una struttura collegata

Proprietà (debole): gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi

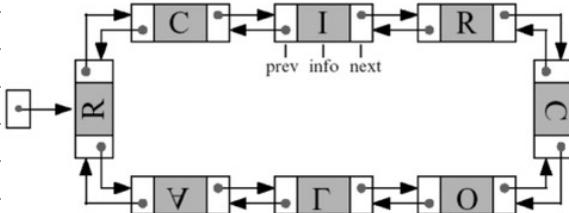
### Strutture collegate



Lista semplice



Lista doppialemente collegata



Lista circolare doppialemente collegata

## Pro e contro

### Rappresentazioni indizzate:

- Pro: accesso diretto ai dati mediante indici
- Contro: dimensione fissa (riallocazione array richiede tempo lineare)

### Rappresentazioni collegate:

- Pro: dimensione variabile (aggiunta e rimozione record in tempo costante)
- Contro: accesso sequenziale ai dati.

## Realizzazione di un dizionario

Metodo più semplice: array non ordinato (sorridimensionato)

Insert  $\rightarrow \mathcal{O}(1)$  - inserisco dopo l'ultimo elemento

Search  $\rightarrow \mathcal{O}(n)$  - devo scorrere l'array

Delete  $\rightarrow \mathcal{O}(n)$  - delete: search + cancellazione

2° Metodo: array ordinato

Insert  $\rightarrow \mathcal{O}(n)$  - ho bisogno di  $\mathcal{O}(\log n)$  confronti per trovare la giusta posizione in cui inserire l'elemento e  $\mathcal{O}(n)$  trasferimenti per mantenere l'array ordinato

Search  $\rightarrow \mathcal{O}(\log(n))$  - ricerca binaria

Delete  $\rightarrow \mathcal{O}(n)$  (come per insert)

3° Metodo: Lista non Ordinata

Insert  $\rightarrow \mathcal{O}(1)$

Search  $\rightarrow \mathcal{O}(n)$

Delete  $\rightarrow \mathcal{O}(n)$

4° Metodo: Lista Ordinata

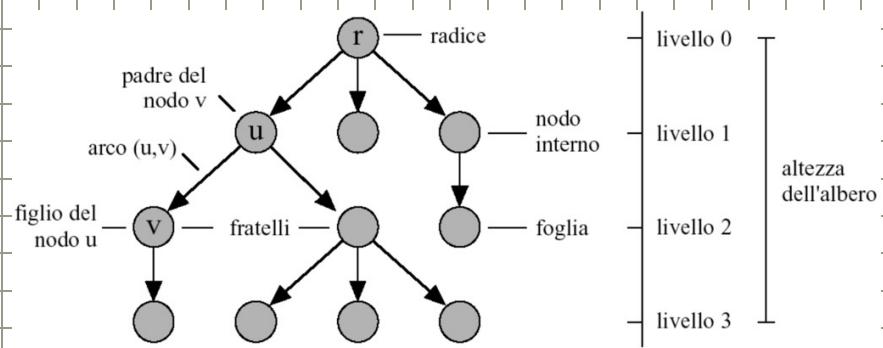
Insert  $\rightarrow \mathcal{O}(n)$  devo mantenere ordinata la lista

Search  $\rightarrow \mathcal{O}(n)$  non posso usare la ricerca binaria

Delete  $\rightarrow \mathcal{O}(n)$

## Alberi

### Organizzazione gerarchica dei dati



Dati contenuti nei **nodi**, relazioni gerarchiche definite dagli **archi** che li collegano. **Grado** di un nodo: numero dei suoi **figli**.

U **antenato** di v se v è raggiungibile da U risalendo di padre in padre. se stesso

V **discendente** di U se U è un antenato di V. se stesso

Rappresentazioni **indirizzate** di alberi

Ideaz: ogni cella dell'array contiene

- le informazioni di un nodo
- eventualmente altri indici per raggiungere altri nodi

Si può utilizzare il:

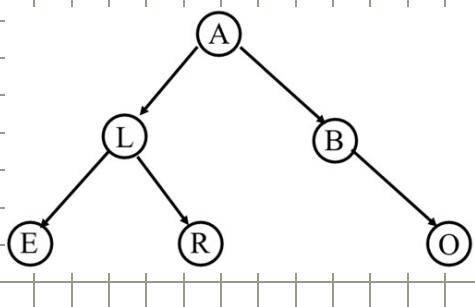
**Vettore dei padri**

Per un albero di n nodi uso un array P di dimensione (almeno) n.

Una generica cella i contiene una coppia (info, parent), dove:

info: contenuto informativo del nodo i.

parent: indice (nell'array) del nodo padre di i.



P[i].info: contenuto informativo nodo

P[i].parent: indice del nodo padre

(L,3)	(B,3)	(A,null)	(O,2)	(E,1)	(R,1)
1	2	3	4	5	6

(P[i].info, P[i].parent)

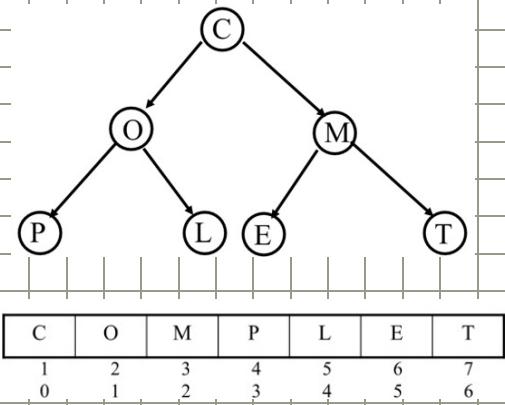
Oss. Numero arbitrario di figli, tempo  $O(1)$  per individuare il padre di un nodo, tempo  $O(d)$  per individuare uno o più figli di un nodo

Vettore posizionale: (per alberi d-ari (quasi) completi).

Nodi arrangiati nell'array "per livelli".

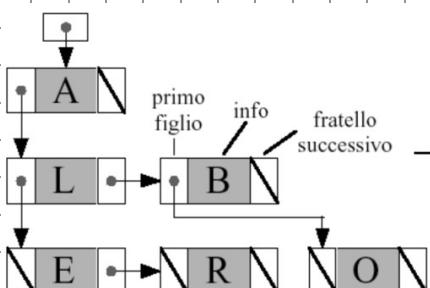
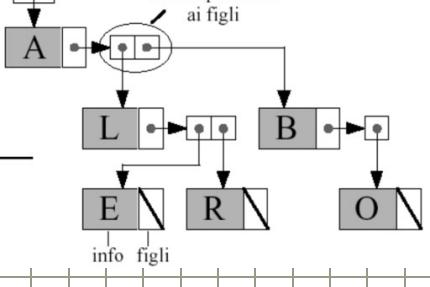
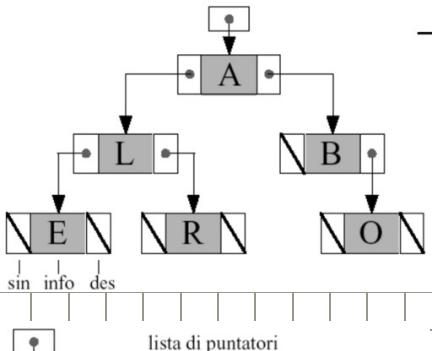
Indici a partire da 0: - j-esimo figlio ( $j \in \{1, \dots, d\}$ ) di i è in posizione  $d \cdot i + j$   
- il padre di i è in posizione  $\lfloor (i-1)/d \rfloor$

Indici a partire da 1: - j-esimo figlio ( $j \in \{1, \dots, d\}$ ) di i è in posizione  $d(i-1) + j - 1$   
- il padre di i è in posizione  $\lfloor (i-2)/d \rfloor + 1$



Oss. # di figli = d, solo per alberi completi o quasi completi, tempo  $O(1)$  per individuare il padre di un nodo, tempo  $O(1)$  per individuare uno specifico figlio di un nodo

Rappresentazioni: collegate di alberi



- Rappresentazione con puntatori ai figli (nodi con numero limitato di figli)

Rappresentazione con liste di puntatori ai figli (nodi con numero arbitrario di figli)

Rappresentazione di tipo primo figlio - fratello successivo (nodi con numero arbitrario di figli)

Tutte le rappresentazioni viste possono essere arricchite per avere in ogni nodo anche un puntatore al padre

## Visite di Alberi

Sono algoritmi che consentono l'accesso sistematico ai nodi e agli archi di un albero. Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai nodi.

### Algoritmo di visita in profondità

L'algoritmo di visita in profondità (DFS) parte da  $r$  e procede visitando nodi di figlio in figlio fino a raggiungere una foglia. Retrocede poi al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli.

#### algoritmo visita DFS (nodo $r$ )

pila  $s$

alberi binari

```
s.push(r)
while (not s.isEmpty()) do
    u ← s.pop()
    if (u ≠ null) then
        visita il nodo u
        s.push (figlio destro di u)
        s.push (figlio sinistro di u)
```

Ogni nodo inserito e estratto dalla Pila una sola volta.

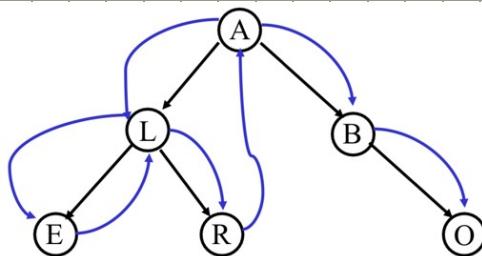
Tempo speso per ogni nodo:  $O(1)$  (se si individua i figli di un nodo in tempo costante)

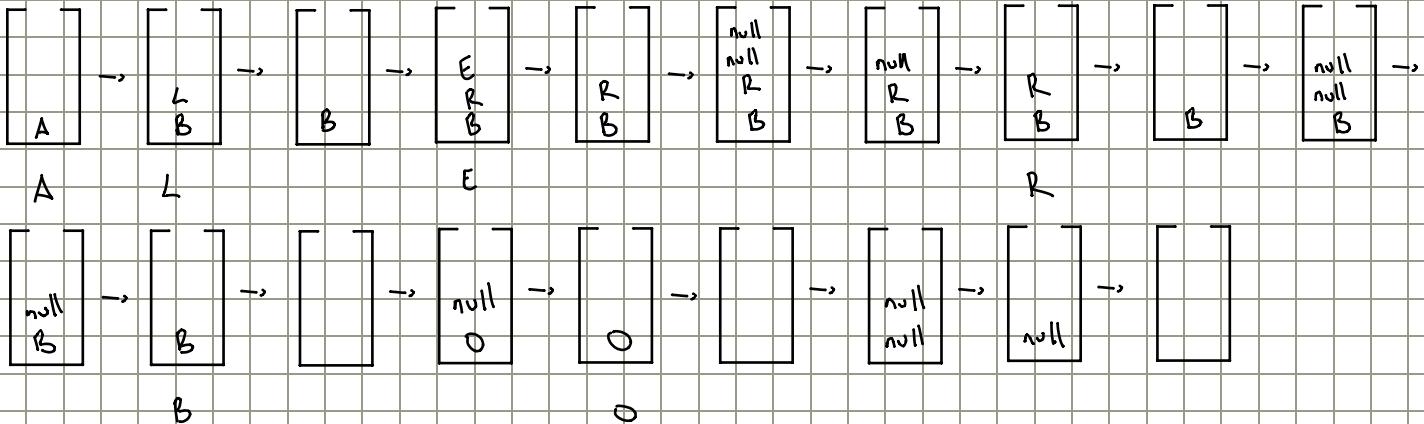
# nodi null inseriti/estratti:  $O(n)$

$$T(n) = O(n)$$

#### algoritmo visitaDFS ricorsiva (nodo $r$ )

```
if ( $r \neq$  null) then
    visita il nodo r
    visitaDFS ricorsiva (figlio sinistro di r)
    visitaDFS ricorsiva (figlio destro di r)
```





Diversi tipi di visita

Visita in **preordine**: radice, sottoalbero sin, sottoalbero des

Visita **simmetrica**: sottoalbero sin, radice, sottoalbero des (scambiare riga 2 con 3)

Visita in **postordine**: sottoalbero sin, sottoalbero des, radice (scambiare riga 2 dopo n)

Preordine: ALERBO , Simmetrica: ELRABO , Postordine: ERLOBA

Algoritmo di visita in ampiezza

L'algoritmo di visita in ampiezza (**BFS**) parte da r e procede visitando nodi per livelli successivi. Un nodo sul livello i può essere visitato solo se tutti i nodi sul livello i-1 sono stati visitati.

algoritmo **visit2BFS(nodo r)**

alberi binari

c. enqueue(r)

while (not c.isEmpty()) do

u ← c.dequeue()

if (u ≠ null) then

visita il nodo u

c.enqueue(figlio sinistro di u)

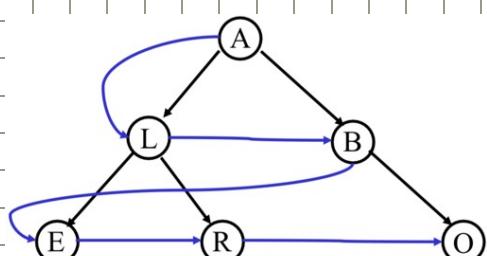
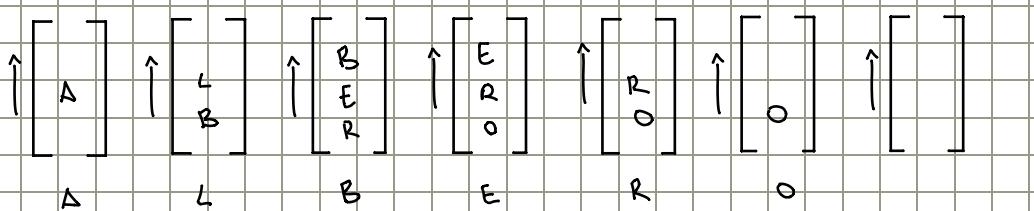
c.enqueue(figlio destro di u)

Ogni nodo inserito e estratto dalla Coda una sola volta.

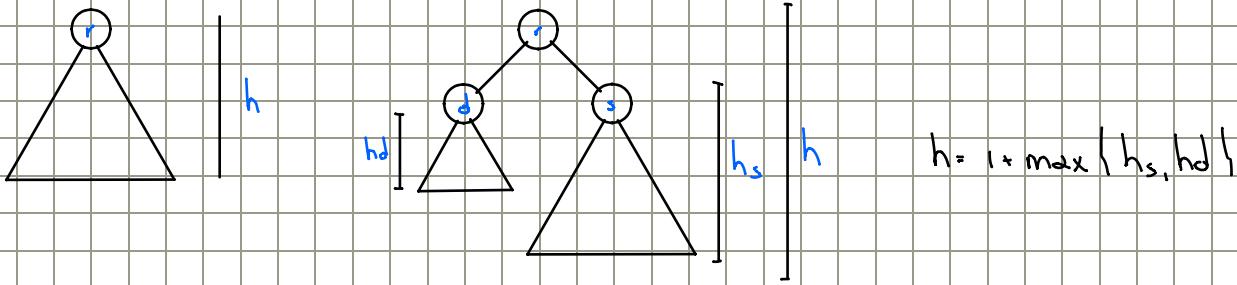
Tempo speso per ogni nodo:  $O(1)$  (se si individua i figli di un nodo in tempo costante)

# nodi null inseriti/estraitti:  $O(n)$

$$T(n) = O(n)$$



Utilizzo algoritmi di visita: calcolo dell'altezza

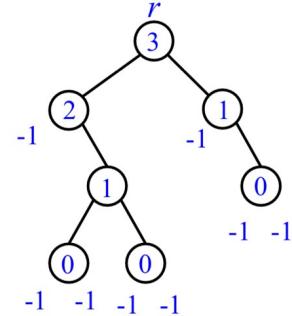


### Calcolo Altezza (nodo r)

```
if (r == null) then return -1
sin = CalcoloAltezza (figlio sinistro di r)
des = CalcoloAltezza (figlio destro di r)
return 1 + max {sin, des}
```

comp  $O(n)$

Calcola (e ritorna) l'altezza di un  
albero binario con radice r



### Problema 3.6

#### 1. Calcolo Foglie (nodo r)

```
if (r == NULL) then return 0
if (r.sin == NULL and r.des == NULL) then return 1
sin = CalcoloFoglie (figlio sinistro di r)
des = CalcoloFoglie (figlio destro di r)
return sin + des
```

#### 2. Calcolo Grado Med (nodo r)

n = numero nodi

nfoglie = CalcoloFoglie (nodo r)

if (r != null) then

return (SommaGradi (r) / (n - nfoglie))

#### SommaGradi (nodo r)

```
if (r == null) return 0
if (r è foglia) return 0
S = numero figli di r + SommaGradi (r.sin)
+ SommaGradi (r.des)
return S
```

### 3. CercaElemento (nodo r, chiave K)

if ( $r == \text{null}$ ) then return null

se chiave K non presente ritorna null

if ( $\text{chiave}(r) = K$ ) then return r

comp 3.6  $O(n)$

sin = CercaElemento (figlio sinistro r, K)

if ( $\text{sin} \neq \text{null}$ ) then return sin

return CercaElemento (figlio dest di r, K)

Per la rappresentazione di alberi tramite vettori posizionali, per memorizzare un albero non completo con n nodi (si assuma d=2) è necessario quanto spazio?

Si consideri un albero di n nodi che è una catena, ovvero un albero tale che ogni nodo ha al più un figlio

L'altezza di questo albero è  $n-1$ .

L'albero binario completo di altezza  $n-1$  ha  $2^{n-1}$  nodi.

Dimensione vettore posizionale  $2^n - 1$ .

Quantità di memoria necessaria per memorizzare albero è esponenzialmente più grande del numero di nodi.

Sia T un albero mantenuto attraverso un vettore dei padri. Progettare un algoritmo che, dato T e un nodo  $r'$  di T restituisce il vettore dei padri che rappresenta T radicato in  $r'$ .

### riRadice (T, j)

$x = j$

$px = T[j].\text{parent}$

$T[j].\text{parent} = \text{null}$

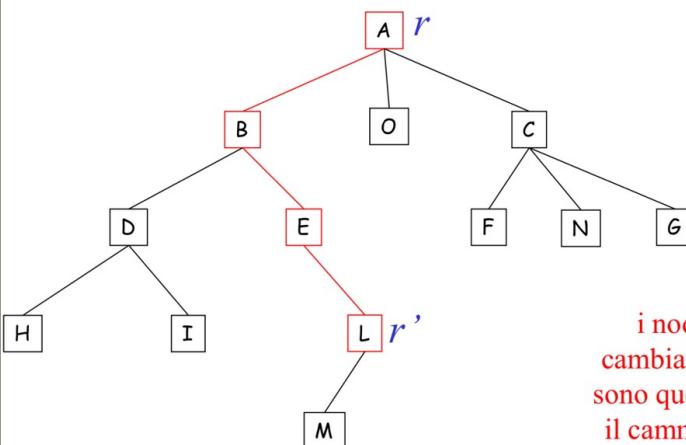
while ( $px \neq \text{null}$ ) do

$y = T[px].\text{parent}$

$T[px].\text{parent} = x$

$x = px$

$px = y$



i nodi che  
cambiano padre  
sono quelli lungo  
il cammino che  
unisce  $r'$  con  $r$

comp  $O(h)$

## Lezione 10

Il problema del dizionario

tipo Dizionario:

dati: un insieme  $S$  di coppie (elem, chiave)

operazioni:

$\text{insert}(\text{elem } e, \text{chiave } k)$

aggiunge a  $S$  una nuova coppia  $(e, k)$

$\text{delete}(e)$

cancella da  $S$  l'elemento  $e$

$\text{search}(\text{chiave } k) \rightarrow \text{elem}$

se la chiave  $k$  è presente in  $S$  restituisce un elemento  $e$  ad essa associato  
e null altrimenti.

E' possibile garantire che tutte le operazioni su un dizionario di  $n$  elementi  
abbiano tempo  $O(\log n)$  tramite due idee:

1. Definire un albero (binario) tale che ogni operazione richiede tempo  $O(h)$

(alberi binari di ricerca, BST = binary search tree)

2. Fare in modo che l'altezza dell'albero sia sempre  $O(\log n)$  (alberi AVL)

La prima idea:

Un BST è un albero binario che soddisfa le seguenti proprietà:

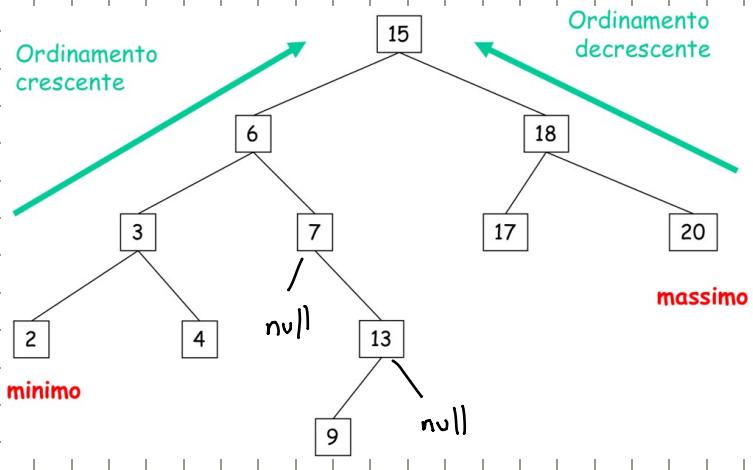
- ogni nodo  $v$  contiene un elem( $v$ ) cui è associata una chiave( $v$ ) presa da un dominio totalmente ordinato.

Per ogni nodo  $v$  tale che:

- le chiavi nel sottoalbero sinistro di  $v$  sono  $\leq$  chiave( $v$ )

- le chiavi nel sottoalbero destro di  $v$  sono  $\geq$  chiave( $v$ )

Il minimo non è detto sia una foglia, così come il massimo.

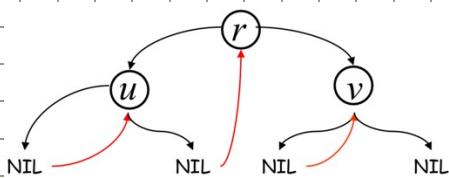


2 3 4 6 7 9 13 15 17 18 20

### Verifica correttezza

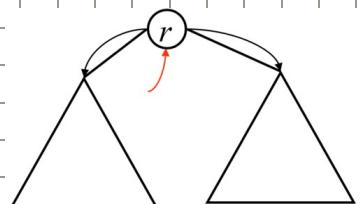
Indichiamo con  $h$  l'altezza dell'albero. Vogliamo dimostrare che la visita in ordine simmetrico restituisce la sequenza ordinata.

Per induzione:  $h=1$ .



$$\text{chiave}(u) \leq \text{chiave}(r) \leq \text{chiave}(v)$$

$h$  generico (ipot.  $\exists h$  che la procedura sia corretta per altezza  $< h$ )



1. Albero di  $h \leq h-1$ . Tutti i suoi elementi sono  $\leq$  della radice
2. Albero di  $h \leq h-1$ . Tutti i suoi elementi sono  $\geq$  della radice

### Operazioni sul BST:

Search:

algoritmo search(chiave K)  $\rightarrow$  elem

$v \leftarrow$  radice di T

while ( $v \neq \text{null}$ ) do

if ( $K = \text{chiave}(v)$ ) then return elem(v)

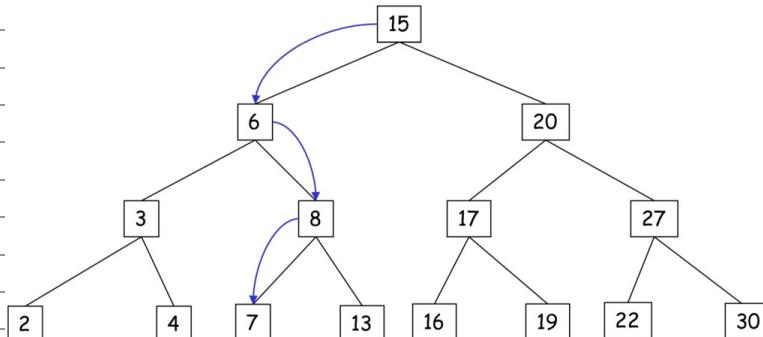
else if ( $K < \text{chiave}(v)$ ) then  $v \leftarrow v.\text{sin}$

else  $v \leftarrow v.\text{des}$

return null

Se visitiamo BST in ordine simmetrico  
i nodi saranno in ordine crescente  
di chiave (DFS)

comp  $O(h)$



Traccia un cammino nell'albero partendo dalla radice: su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro.

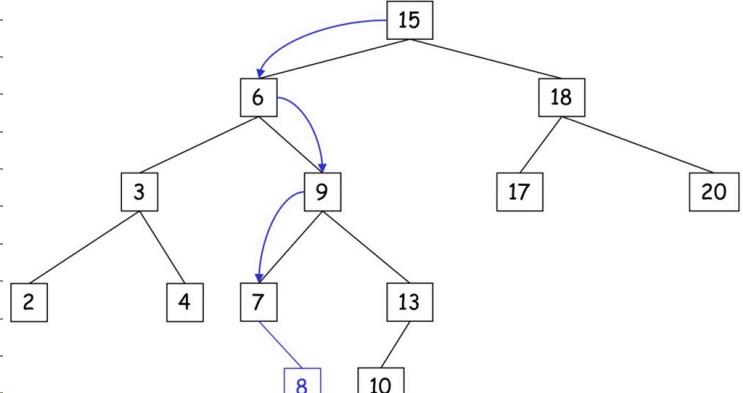
Insert:

Idea: aggiungere la nuova chiave come nodo foglia; per capire dove mettere la foglia simula una ricerca con la chiave da inserire

1. Crea un nodo  $v$  con  $\text{elem} = e$  e  $\text{chiave} = K$
2. Cerca la chiave  $K$  nell'albero, identificando così il nodo  $v$  che diventerà padre di  $v$
3. Appendi  $v$  come figlio sinistro/destro di  $v$  in modo che sia mantenuta la proprietà di ricerca

Verifica correttezza

Se seguo questo schema l'elemento è viene posizionato nella posizione giusta. Infatti, per costruzione, ogni antenato di  $e$  si ritrova  $e$  nel giusto sottoalbero.



Ricerca max e min

algoritmo  $\text{max}(\text{nodo } u) \rightarrow \text{nodo}$

$v \leftarrow u$

while ( $v.\text{des} \neq \text{null}$ ) do

$v \leftarrow v.\text{des}$

return  $v$

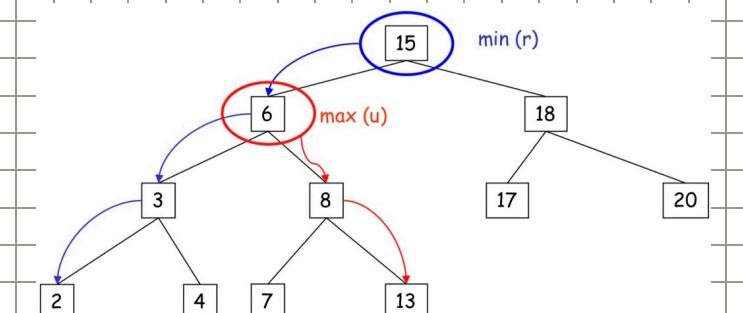
algoritmo  $\text{min}(\text{nodo } u) \rightarrow \text{nodo}$

$v \leftarrow u$

while ( $v.\text{sin} \neq \text{null}$ ) do

$v \leftarrow v.\text{sin}$

return  $v$



## Predecessore e Successore

Il predecessore di un nodo  $u$  in un BST è il nodo  $v$  nell'albero avente massima chiave  $\leq$  chiave( $u$ ).

Il successore di un nodo  $u$  in un BST è il nodo  $v$  nell'albero avente minima chiave  $\geq$  chiave( $u$ ).

## Ricerca del predecessore e successore

algoritmo  $\text{pred}(\text{nodo } u) \rightarrow \text{nodo}$

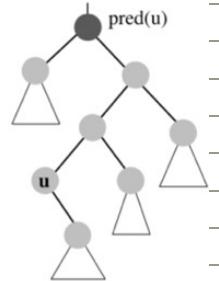
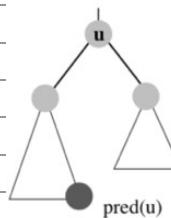
if ( $u$  ha figlio sinistro  $\text{sin}(u)$ ) then

return  $\max(\text{sin}(u))$

while ( $\text{parent}(u) \neq \text{null}$  e  $u$  è figlio sinistro di suo padre) do

$u \leftarrow \text{parent}(u)$

return  $\text{parent}(u)$



algoritmo  $\text{suc}(u) \rightarrow \text{nodo}$

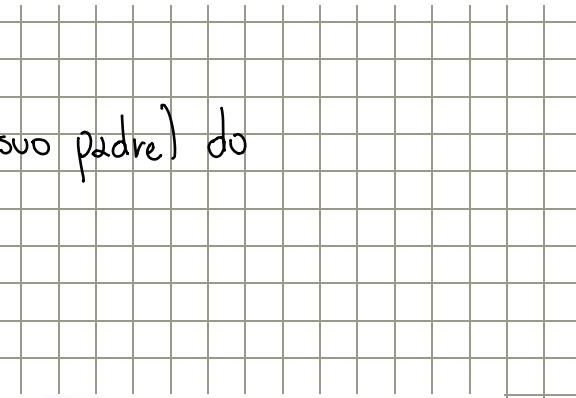
if ( $u$  ha figlio destro  $\text{des}(u)$ ) then

return  $\min(\text{des}(u))$

while ( $\text{parent}(u) \neq \text{null}$  e  $u$  è figlio destro di suo padre) do

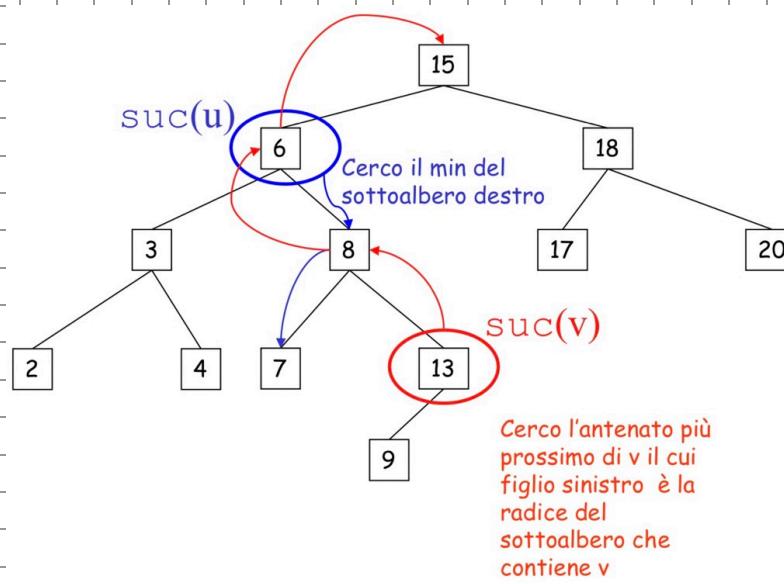
$u \leftarrow \text{parent}(u)$

return  $\text{parent}(u)$



In entrambi i casi

comp  $O(h)$

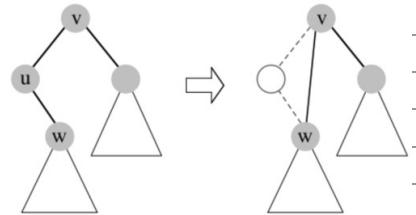


Delete:

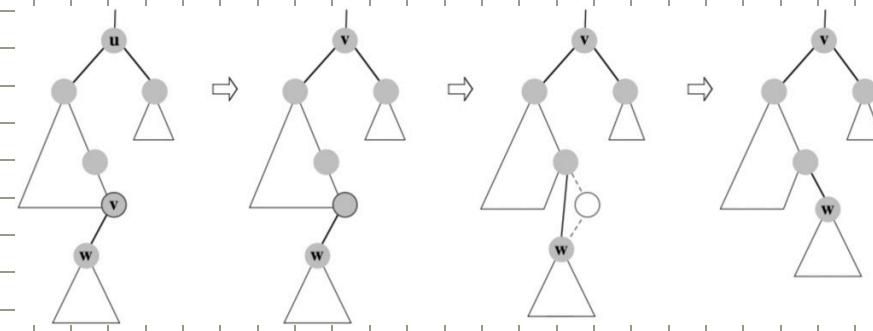
Sia  $v$  il nodo contenente l'elemento e da cancellare abbiamo tre casi:

1)  $v$  è una foglia: rimuovila

2)  $v$  ha un solo figlio:



3)  $v$  ha due figli: sostituiscelo con il predecessore (o successore) ( $v'$ ) e rimuovi fisicamente il predecessore (o successore) (che ha al più un figlio)



### Costo delle operazioni

Tutte le operazioni hanno costo  $O(h)$  dove  $h$  è l'altezza dell'albero, si avrà  $O(n)$  nel caso peggiore (alberi molto sbilanciati e profondi come catene), in un albero di ricerca bilanciato  $h = \Theta(\log n)$  (BTS completo).

### Lezione 11

La seconda idea:

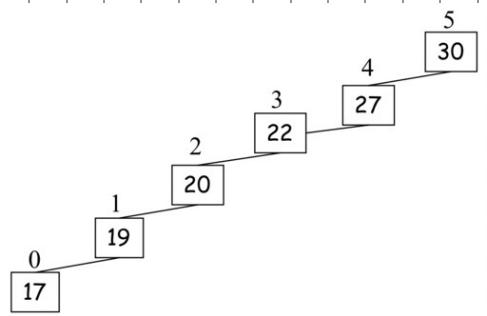
Per fare in modo che l'altezza dell'albero sia sempre  $O(\log n)$  si utilizzano alberi AVL.

Fattore di bilanciamento  $B(v)$  di un nodo  $v$ :  $h_{\text{sottoalbero sx}} - h_{\text{sottoalbero dx}}$

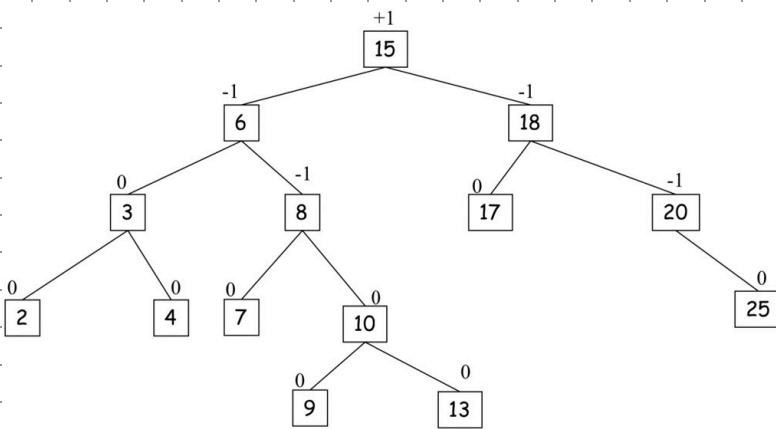
Un albero si dice bilanciato in altezza se ogni nodo  $v$  ha fattore di bilanciamento in valore assoluto  $\leq 1$ .

Alberi AVL sono alberi binari di ricerca bilanciati in altezza.

(Generalmente  $B(v)$  mantenuto come informazione aggiuntiva nel record relativo a  $v$ )



NO!



SI!

### Altezza di alberi AVL

Si può dimostrare che un albero AVL con  $n$  nodi ha altezza  $O(\log n)$

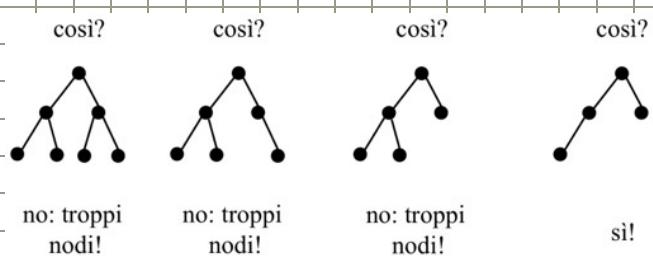
Idea della dimostrazione: considerare, tra tutti gli AVL, i più sbilanciati (albero Fibonacci)

L'albero di Fibonacci di altezza  $h$  sono alberi AVL con il minimo numero di nodi  $n_h$

$$\begin{array}{ccc} \text{minimizzare \# nodi} & = & \text{massimizzare altezza} \\ \text{fissata l'altezza} & & \text{fissato \# nodi} \end{array}$$

Se gli alberi di Fibonacci hanno altezza  $O(\log n)$  allora tutti gli alberi AVL hanno altezza  $O(\log n)$ .

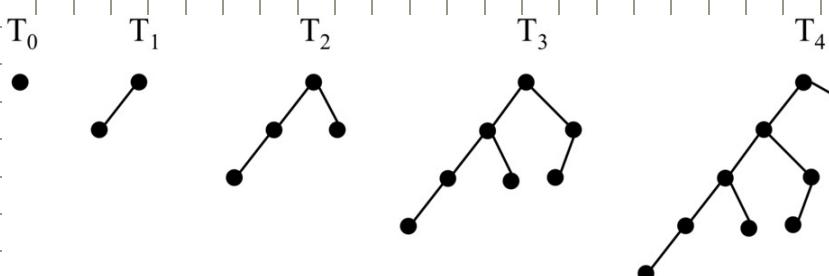
### Albero di Fibonacci $h=2$



se togliamo un nodo o diventa sbilanciato o cambia la sua altezza. Ogni nodo (non foglia) ha  $\beta(v)$  pari (in valore assoluto) a 1.

### Albero di Fibonacci per valori piccoli di altezza

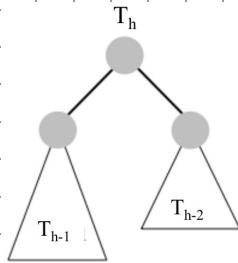
$T_i$ : albero di Fibonacci di altezza  $i$



se a  $T_i$  tolgo un nodo o diventa sbilanciato o cambia la sua altezza

Ogni nodo (non foglia) ha  $\beta(v)$  pari (in valore assoluto) a 1.

Vengono chiamati alberi di Fibonacci perché è possibile formare  $T_i$  utilizzando i due alberi  $T_{i-1}$  e  $T_{i-2}$  nel seguente schema:



**Lemma:** Sia  $n_h$  il numero di nodi di  $T_h$ , risulta  $n_h = F_{h+3} - 1$

dim: per induzione su  $h$  si usa  $\cap_h = \cap_{h-1} + \cap_{h-2}$

$$Pb: h=0 \quad n_0 = F_{3-1} = 1 \quad \checkmark$$

$$\text{PI: } n_h = 1 + n_{h-1} + n_{h-2} \Rightarrow F_{h+3} - 1 = 1 + (F_{h+2} - 1) + (F_{h+1} - 1) \Rightarrow F_{h+3} = F_{h+2} + F_{h+1} = F_h = F_{h-1} + F_{h-2} \quad \checkmark$$

## Corollario

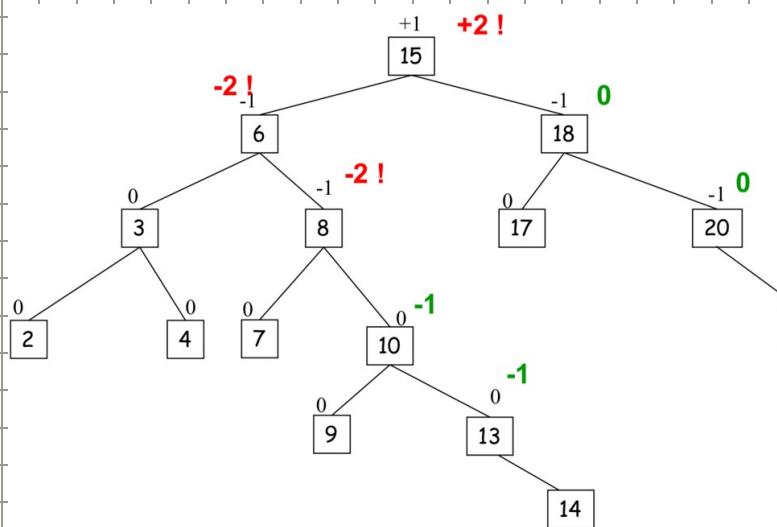
Un albero AVL con  $n$  nodi ha altezza  $h = O(\log n)$

$$\text{dim.: } n_h = F_{h+3} - 1 = \Theta(\phi^{h+3}) = \Theta(\phi^h \phi^3) = \Theta(\phi^h)$$

$$h = \Theta(\log n) = O(\log n)$$

albero AVL fibonaci di h fissato ha il minimo numero di nodi  
 $n_h \leq n \Rightarrow$  albero AVL normale con stesso h fissato

Di quanto e quali fattori di bilanciamento cambiano a fronte di un inserimento o cancellazione?



Se inserisco 14 l'albero si sbilancia.

Se cancellò 25 l'albero si  
sbilancia.

Se parto da un albero AVL e inserisco/cancello un nodo:

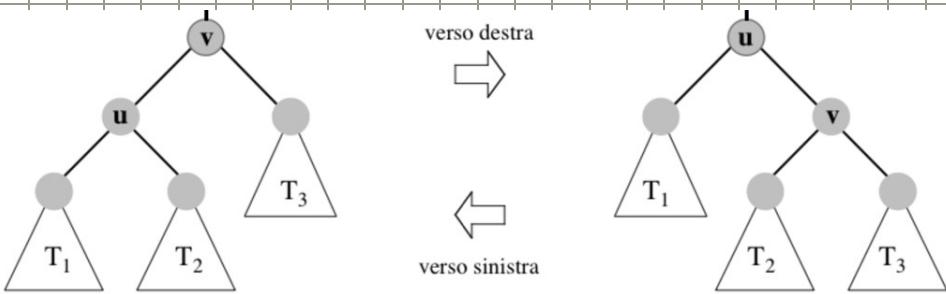
(quali) cambiano solo i fattori di bilanciamento dei nodi lungo il cammino radice-nodo inserito / cancellato

(quanto) i fattori di bilanciamento cambiano di  $\pm 1$

Quindi l'operazione search procede come in un BST, ma inserimenti e cancellazioni potrebbero sbilanciare l'albero.

Per mantenere il bilanciamento tramite opportune rotazioni.

Rotazione di base verso destra /sinistra rispettivamente sul nodo v/u



Mantiene le proprietà di ordinamento, e richiede tempo  $O(1)$  perché bisogna cambiare i puntatori ( $v, r, t_2, u, \text{padre di } v$ )

Ribilanciamento tramite rotazioni:

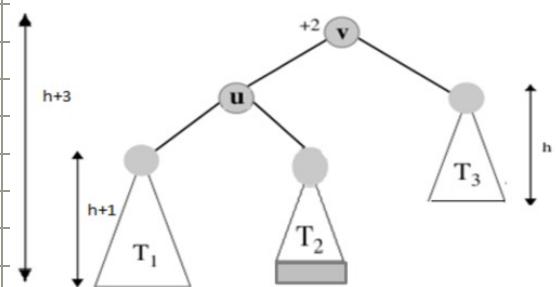
Le rotazioni sono effettuate su nodi sbilanciati.

Sia  $v$  un nodo di profondità massima (nodo critico) con fattore di bilanciamento  $B(v) \pm 2$ , esiste quindi un sottoalbero  $T$  di  $v$  che lo sbilancia, a seconda della posizione di  $T$  si hanno 4 casi:

- sinistra-sinistra **ss**  $B(v) = +2$   $T$  è il sottoalbero sinistro del figlio sinistro di  $v$
- destra-destra **dd**  $B(v) = -2$   $T$  è il sottoalbero destro del figlio destro di  $v$
- sinistra-destra **sd**  $B(v) = +2$   $T$  è il sottoalbero destro del figlio sinistro di  $v$
- destra-sinistra **ds**  $B(v) = -2$   $T$  è il sottoalbero sinistro del figlio destro di  $v$

Sia  $h$  l'altezza del sottoalbero destro di  $v$ .

(caso **ss** ( $B(v) = +2$ , altezza  $T_1 = h+1$ ))



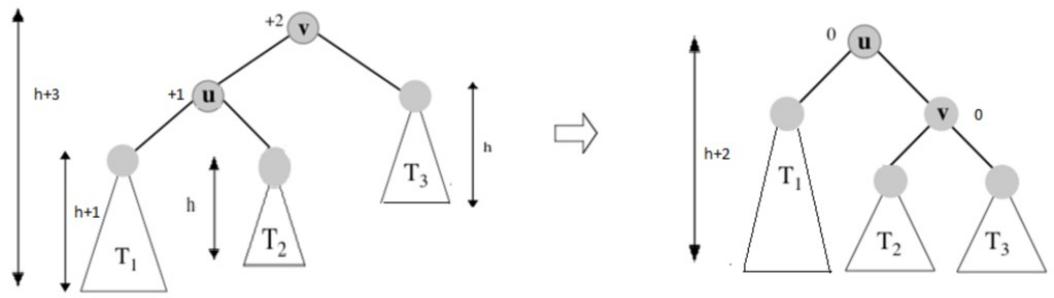
L'altezza di  $T(v)$  è  $h+3$ , l'altezza di  $T(u)$  è  $h+2$ , l'altezza di  $T_3$  è  $h$ , è l'altezza di  $T_1$  è  $h+1$ .  $B(v) = +2$  è lo sbilanciamento provocato da  $T_1$ .

Si applica una semplice rotazione verso destra su  $v$  e i fattori di bilanciamento di tutti i nodi torneranno ok.

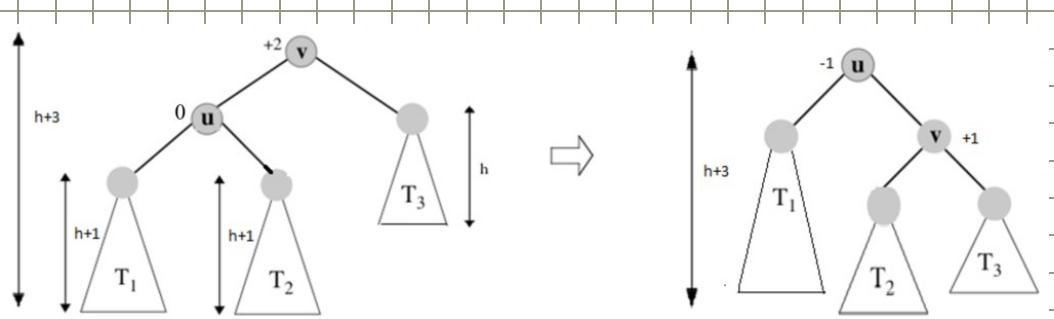
2 sottocasi possibili:

(i) l'altezza di  $T_2$  è  $h \Rightarrow$  l'altezza dell'albero coinvolto nella rotazione passa da  $h+3$  a  $h+2$ .

(ii) l'altezza di  $T_2$  è  $h+1 \Rightarrow$  l'altezza dell'albero nella rotazione rimane  $h+3$ .



primo caso.



secondo caso.

Dopo la rotazione l'albero è bilanciato (tutti i fattori di bilanciamento sono in modulo  $\leq 1$ ).

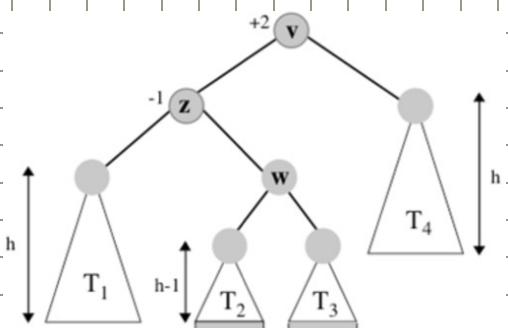
L'insertimento di un elemento nell'AVL (ovvero, l'aggiunta di una foglia a un albero bilanciato) può provocare solo il sottocaso (i) (perché altrimenti l'AVL era già sbilanciato!).

Invece, la  cancellazione di un elemento dall'AVL (che necessariamente fa diminuire l'altezza di qualche sottoalbero) può provocare entrambi i casi (ad esempio, se cancellando un elemento ho abbassato l'altezza di  $T_3$ ).

Nel caso (i), dopo la rotazione, l'albero diminuisce la sua altezza di uno.

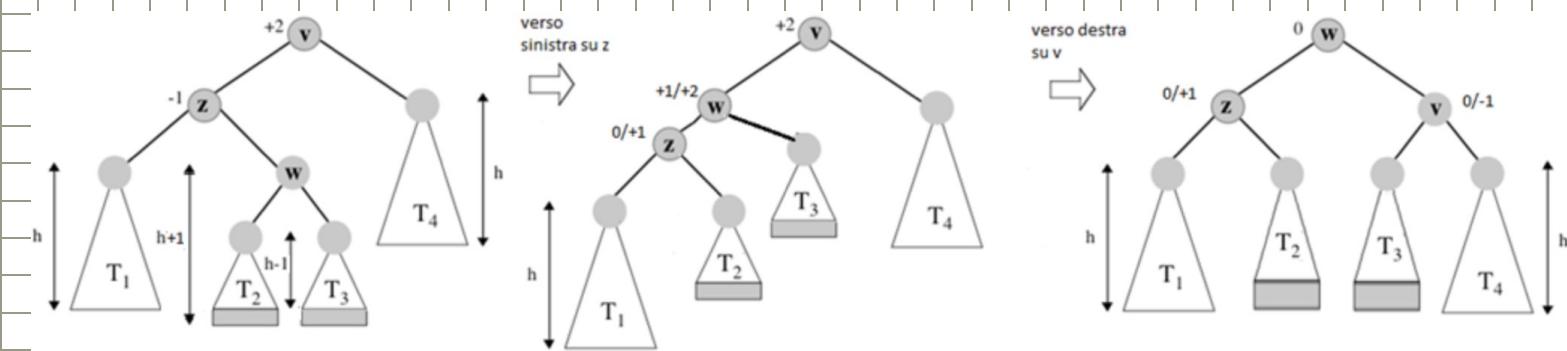
Simmetrico per DD.

Caso SD ( $\beta(v) = +2$ , altezza  $T_1 \neq h+1$ )



L'altezza di  $T(w)$  è  $h+1$ , l'altezza di  $T(z)$  è  $h+1$ ,  $\beta(v) = +2$  e sbilanciamento  $\beta(z) = -1$  è provocato dal sottoalbero destro di  $z$ .

S. applicano due rotazioni: una verso sinistra sul figlio sinistro del nodo critico (nodo  $z$ ), l'altra verso destra sul nodo critico (nodo  $v$ ).



I fattori di bilanciamento di tutti i nodi tornano ok, l'altezza dell'albero dopo la rotazione passa da  $h+3$  a  $h+2$ , poiché  $T_2$  e  $T_3$  sono alti al più  $h$ , e il fattore di bilanciamento di  $w$  diventa 0, mentre i fattori di bilanciamento di  $z$  e  $v$  sono 0 oppure  $\pm 1$ .

Il caso SD può essere provocato sia da inserimenti (in  $T_2$  o  $T_3$ ), sia da cancellazioni che abbassano di 1 l'altezza di  $T_n$ .

Simmetrico per DS.

### Inserimento

Crea un nodo  $u$  con elem-e e chiave =  $k$ .

Inserisci il nodo  $u$  come in un BTS.

Ricalcola i fattori di bilanciamento dei nodi nel cammino della radice a  $u$ : sia  $v$  il più profondo nodo critico  $\beta(v) = \pm 2$ .

Esegui una rotazione opportuna su  $v$ .

Oss: Un solo ribilanciamento è sufficiente, poiché l'altezza dell'albero coinvolto diminuisce di 1 (sottocaso (i) di SS o DD, o casi SD o DS), e quindi torna ad essere uguale all'altezza che aveva prima dell'inserimento.

### Cancellazione

Cancella il nodo  $u$  come in un BTS.

Ricalcola i fattori di bilanciamento del padre del nodo eliminato fisicamente (che potrebbe essere diverso dal nodo contenente  $e$ ), ed esegui l'opportuna rotazione.

Ripeti questo passo, sino ad arrivare eventualmente alla radice AVL:

Se l'altezza del sottoalbero appena ribilanciato è uguale a quella che aveva prima della cancellazione, termina.

Se l'altezza è diminuita, risali verso l'alto, calcola il fattore di bilanciamento e applica l'opportuno ribilanciamento.

Oss: potrebbero essere necessarie  $O(\log n)$  rotazioni: infatti eventuali diminuzioni di altezza indotte dalle rotazioni possono propagare lo sbilanciamento verso l'alto nell'albero (l'altezza del sottoalbero in cui è avvenuta la rotazione diminuisce di 1 rispetto a quella che aveva prima della cancellazione).

Tutte le operazioni hanno costo  $O(\log n)$  poiché l'altezza dell'albero è  $O(\log n)$  e ciascuna rotazione richiede solo tempo costante.

### Classe AVL

classe AlberoAVL estende AlberoBinarioDiRicerca:

dati: albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo.

operazioni: search(chiave K)  $\rightarrow$  elem

ereditata

insert(elem e, chiave k)

chiama insert() ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite  $O(1)$  rotazioni.

delete(elem e)

chiama delete() ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite  $O(\log n)$  rotazioni.

Nell'analisi delle complessità di insert/delete vengono implicitamente usate le seguenti tre proprietà.

(i) dato un nodo v, è possibile riconoscere  $\beta(v)$  in  $O(1)$ ;

(ii) dopo aver inserito/cancellato un nodo v nell'albero come se fosse un semplice BST, è possibile ric算olare i fattori di bilanciamento dei nodi lungo il cammino da v alla radice in tempo complessivo  $O(\log n)$ .

(iii) nell'eseguire le rotazioni necessarie per ribilanciare l'albero, è possibile aggiornare anche i fattori di bilanciamento dei nodi coinvolti in tempo complessivo  $O(\log n)$ .

## Lezione 12

Tipo di dato: Coda Priorità

Dati: Un insieme  $S$  di  $n$  elementi di tipo elem a cui sono associate chiavi di tipo chiave prese da un universo totalmente ordinato.

Operazioni:

findMin()  $\rightarrow$  elem

restituisce l'elemento in  $S$  con la chiave minima

insert(elem e, chiave K)

aggiunge a  $S$  un nuovo elemento e con chiave K

delete(elem e) {Suppongo mi venga dato un riferimento diretto all'elemento}

cancella da  $S$  l'elemento e.

deleteMin()

cancella da  $S$  l'elemento con chiave minima

increaseKey(elem e, chiave d)

incrementa della quantità d la chiave dell'elemento e in  $S$ .

decreaseKey(elem e, chiave d)

decrementa della quantità d la chiave dell'elemento e in  $S$ .

merge(CodaPriorita c<sub>1</sub>, CodaPriorita c<sub>2</sub>)  $\rightarrow$  CodaPriorita

restituisce una nuova coda con priorità  $c_3 = c_1 \cup c_2$

Applicazioni: gestione code in risorse condivise, gestione priorità in processi concorrenti, progettazione di algoritmi efficienti (calcolo cammini minimi di un grafo, ordinamento, ecc.)

Possiamo implementare una Coda con Priorità con:

1. Array non ordinato
2. Array ordinato
3. Lista non ordinata
4. Lista ordinata

Almeno una delle operazioni è lineare:  
(non vanno bene)

## 1. Array non ordinato

Lo dimensiono sufficientemente grande e tengo traccia del numero  $n$  di elementi nella coda in una variabile di appoggio.

FindMin:  $\Theta(n)$  devo guardare tutti gli elementi.

Insert:  $O(1)$  inserisco in coda.

Delete:  $O(1)$  ho riferimento diretto, lo posso cancellare sovraccoppiando l'ultimo elemento.

DeleteMin:  $\Theta(n)$  cerco il minimo e poi lo cancello.

## 2. Array ordinato

Lo dimensiono sufficientemente grande e tengo traccia del numero  $n$  di elementi nella coda in una variabile di appoggio, tenendolo ordinato in ordine decrescente.

FindMin:  $O(1)$  ultimo elemento.

Insert:  $O(n)$  trovo in  $\Theta(\log n)$  la giusta posizione ma poi devo fare  $O(n)$  spostamenti.

Delete:  $O(n)$  devo fare  $O(n)$  spostamenti.

DeleteMin:  $O(1)$  non devo fare spostamenti, è in fondo.

## 3. Lista non ordinata (bidirezionale)

FindMin:  $\Theta(n)$  devo guardare tutti gli elementi.

Insert:  $O(1)$  inserisco in coda o in testa.

Delete:  $O(1)$  ho riferimento diretto, lo posso cancellare e spostare puntatori.

DeleteMin:  $\Theta(n)$  cerco il minimo e poi lo cancello.

## 4. Lista ordinata

Lista bidirezionale ordinata in ordine crescente

FindMin:  $O(1)$  in testa alla lista

Insert:  $O(n)$  trovo in  $O(n)$  la giusta posizione e in  $O(1)$  l'inserimento.

Delete:  $O(1)$  cambio puntatori.

DeleteMin:  $O(1)$  cambio il primo puntatore con il secondo.

Tre implementazioni evolute

- d-heap: generalizzazione heap binari;

- Heap binomiali

- Heap di Fibonacci

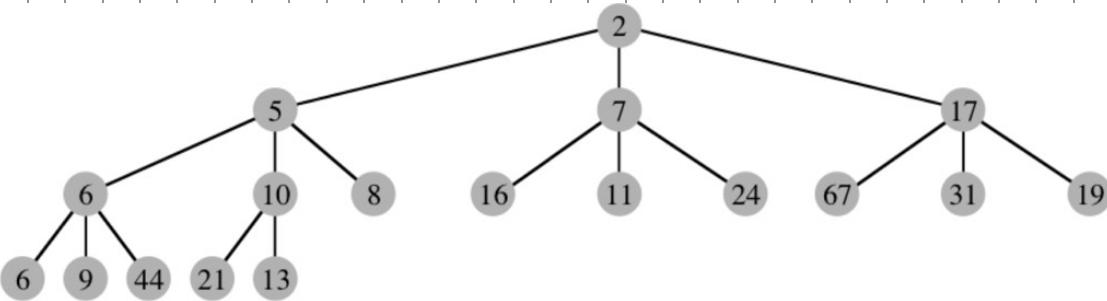
## D-Heap

E' un albero radicato d-ario con le seguenti proprietà:

**Struttura:** è completo fino al penultimo livello, e tutte le foglie sull'ultimo livello sono compatte a sinistra.

**Contenuto informativo:** ogni nodo  $v$  contiene un elemento  $\text{elem}(v)$  e una chiave  $\text{chiave}(v)$  presa da un dominio totalmente ordinato.

**Ordinamento parziale (inverso) dell'heap (min-heap):**  $\text{chiave}(v) \leq \text{chiave}(\text{parent}(v))$  per ogni nodo  $v$  diverso dalla radice.



### Proprietà:

1. Un d-heap con  $n$  nodi ha altezza  $h = \Theta(\log_d n)$ . più figli ho più l'albero è basso
2. La radice contiene elemento con chiave minima.
3. Può essere rappresentato implicitamente tramite vettore posizionale.

Procedure ausiliarie utili per ripristinare la proprietà di ordinamento a heap su un nodo  $v$  che non la soddisfi:

procedura muoriAlto( $v$ )  $T(n) = O(\log_d n)$

while ( $v \neq \text{radice}(T)$  and  $\text{chiave}(v) < \text{chiave}(\text{padre}(v))$ ) do  
scambia posto  $v$  e  $\text{padre}(v)$  in  $T$

procedura muoriBasso( $v$ )  $T(n) = O(d \log_d n)$  (FixHeap)

repeat

sia  $u$  il figlio di  $v$  con la minima chiave( $u$ ), se esiste  
if ( $v$  non ha figli o  $\text{chiave}(v) \leq \text{chiave}(u)$ ) break  
scambia posto  $v$  e  $u$  in  $T$

Operazioni:

$$T(n) = O(1)$$

findMin()  $\rightarrow$  elem

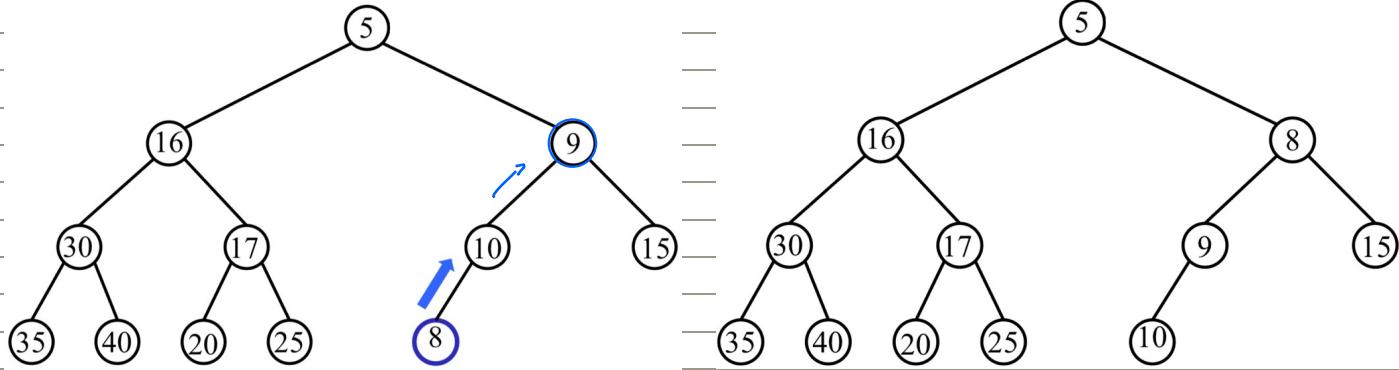
restituisce l'elemento nella radice di T

insert(elem e, chiave K)

$$T(n) = O(\log n)$$

per muoviAlto

crea un nuovo nodo v con elemento e e chiave K, in modo che diventi una foglia sull'ultimo livello di T. La proprietà dell'ordinamento a heap viene poi ripristinata spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi. (incrementare Heapsize)

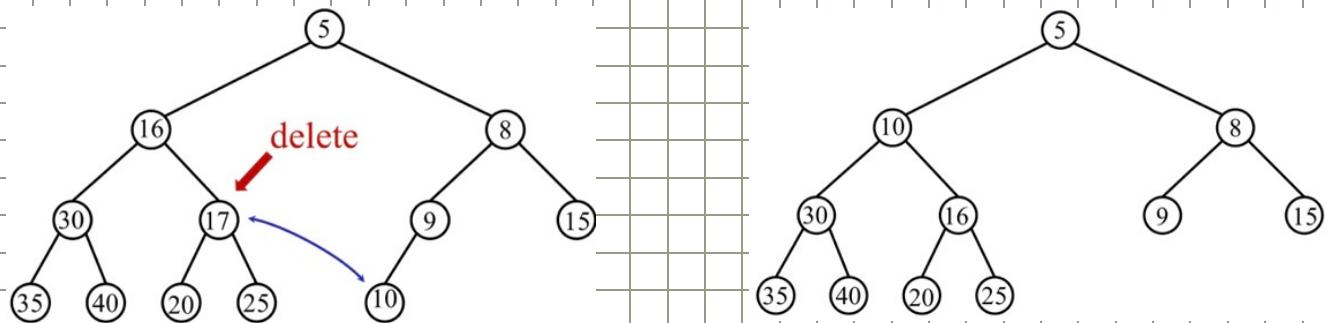


delete (elem e)  $T(n) = O(\log n) \circ O(d \log n)$

scambia il nodo v contenente l'elemento e con una qualunque foglia u sull'ultimo livello di T, e poi elimina v. Ripristina infine la proprietà dell'ordinamento a heap spingendo il nodo u verso la sua posizione corretta tramite muoviAlto o muoviBasso in base al valore della chiave

deleteMin()  $T(n) = O(d \log n)$

usa la stessa procedura di delete scambiando la root e applicando soltanto muoviBasso



decreaseKey (elem e, chiave d)

$$T(n) = O(\log n)$$

decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto (nuovo Alto)

increaseKey (elem e, chiave d)

$$T(n) = O(d \log n)$$

incrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso il basso (nuovo Bassso)

merge ((CodaPriorita c<sub>1</sub>, CodaPriorita c<sub>2</sub>)

Due modi:

**Costruire da zero:** si distruggono le due code e se ne crea una terza contenente l'unione degli elementi.

come: generalizzazione della procedura heapify, rendo i d sottoalberi della radice heap ricorsivamente e chiamo nuovoBasso sulla radice.

Complessità (d costante):  $T(n) = dT(n/d) + O(d \log n)$   $T(n) = \Theta(n)$   $n = |c_1| + |c_2|$

**Inserimenti ripetuti:** si inseriscono ripetutamente gli elementi della coda più piccola in quella più grande.

Sia  $K = \min(|c_1|, |c_2|)$  e  $n = |c_1| + |c_2|$  eseguiamo K inserimenti.

Costo:  $O(K \log n)$ .

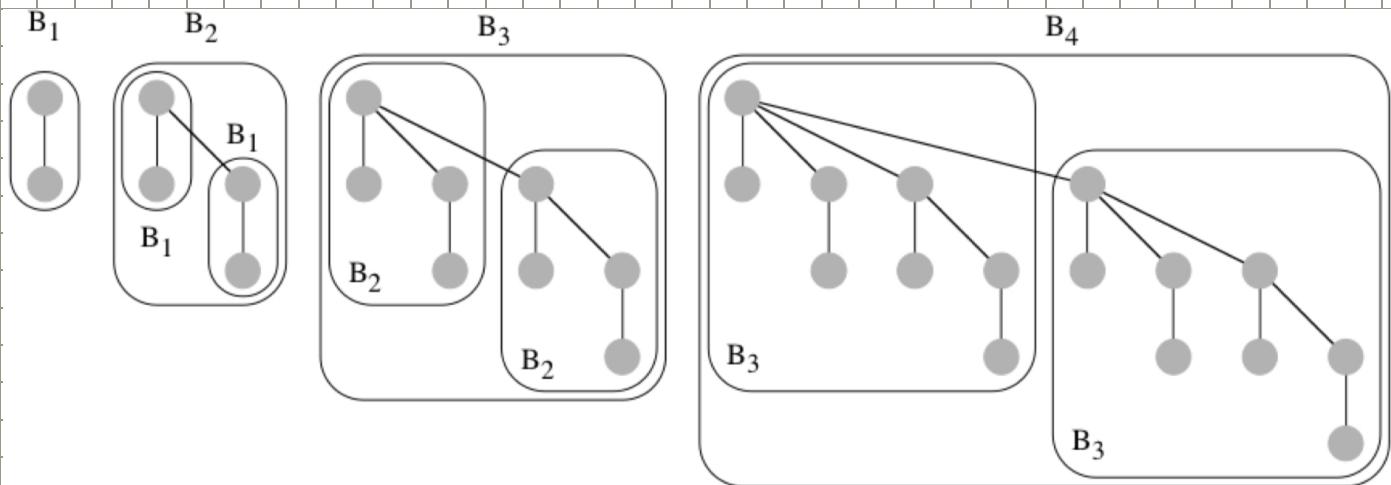
L'approccio conviene per  $K \log n = o(n)$  cioè per  $K = o(n/\log n)$ .

Nel caso peggiore entrambe le operazioni (costruzione da 0 e inserimenti ripetuti) hanno un costo di  $\Omega(n)$ . ( $|c_1| = |c_2| = \frac{n}{2}$ )

## Heap Binomiale

Un albero binomiale  $B_h$  è definito ricorsivamente come segue:

1.  $B_0$  consiste di un unico nodo
2. Per i/o  $B_{h+1}$  è ottenuto fondendo due alberi binomiali  $B_h$ , ponendo la radice dell'uno come figlia della radice dell'altro



Proprietà di un albero binomiale  $B_h$ :

Numero di nodi:  $n = 2^h$

Grado della radice:  $D(n) = \log_2 n$

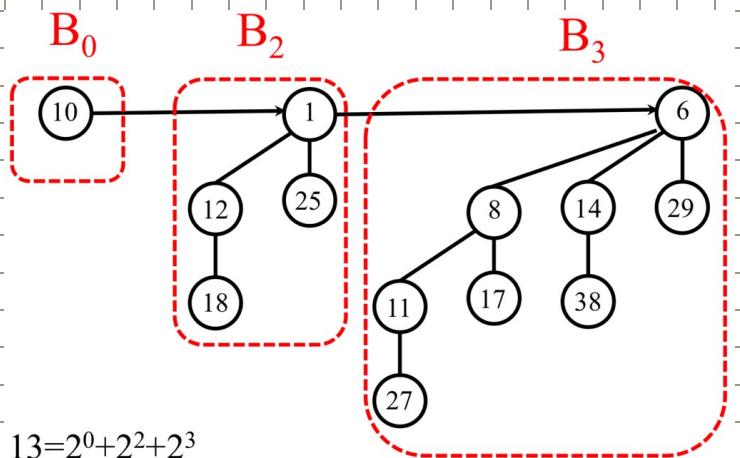
Altezza  $H(n) = h = \log_2 n$

Figli della radice: i sottoalberi radicati nei figli della radice di  $B_h$  sono  $B_0, \dots, B_{h-1}$ .

Un heap binomiale è una foresta di alberi binomiali che gode delle seguenti proprietà:

1. **unicità**:  $\forall i \geq 0$  esiste al più un  $B_i$  nella foresta.  
non posso avere due  $B_i$
2. **contenuto informativo**: ogni nodo  $v$  contiene un elemento  $\text{elem}(v)$  ed una chiave  $\text{chiave}(v)$  presa da un dominio totalmente ordinato.
3. **ordinamento a heap**:  $\text{chiave}(v) \geq \text{chiave}(\text{parent}(v))$  per ogni nodo  $v$  diverso da una delle radici.

Esempio di Heap Binomiale:



Il numero di nodi  $n=13$

$$\text{Heap Binomiale} = B_0 + B_2 + B_3$$

$$13 = 2^0 + 2^2 + 2^3 = 1101$$

Proprietà topologiche:

Dalla proprietà di unicità degli alberi binomiali che lo costituiscono, ne deriva che un heap binomiale di  $n$  elementi è formato dagli alberi binomiali  $B_{i_0}, \dots, B_{i_k}$ , dove  $i_0, \dots, i_k$  corrispondono alle posizioni degli 1 nella rappresentazione in base 2 di  $n$ . Ne consegue che in un heap binomiale con  $n$  nodi, vi sono al più  $\lceil \log n \rceil$  alberi binomiali, ciascuno con grado e altezza  $O(\log n)$ .

Per implementare l'heap binomiale si utilizza la struttura di tipo  
primo figlio - fratello successivo con puntatore a padre, fratello, figlio, etc.

Procedura ausiliaria per ripristinare le proprietà di unicità in un heap binomiale (ipotizziamo di scorrere la lista delle radici da sinistra verso destra, in ordine crescente rispetto all'indice degli alberi binomiali)

procedura `ristruttura()`

$i = 0$

while (esistono ancora due  $B_i$ ) do

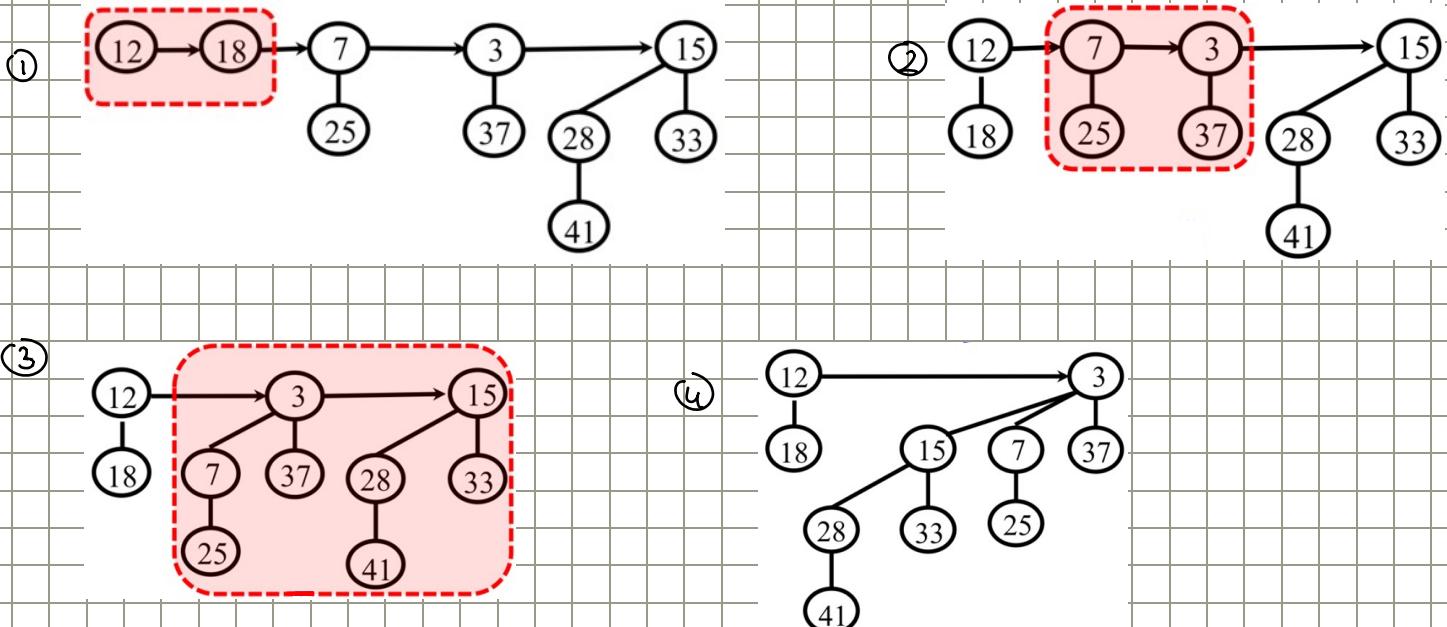
si fondono i due  $B_i$  per formare un albero  $B_{i+1}$  ponendo la radice con chiave più piccola come genitore della radice con chiave più grande.

$i = i + 1$

$T(n)$  = lineare nel numero di alberi binomiali in input

(ogni fusione diminuisce di uno gli alberi binomiali in input)

## Esempio Ristruttura



Realizzazione:

Classe `HeapBinomiale` implementa `CodaPriorita`:

Dati: una foresta  $H$  con  $n$  nodi, ciascuno contenente un elemento  $elem$  e una chiave di tipo `chiave` presa da un universo totalmente ordinato.

Operazioni:

`findMin() -> elem`

scorre le radici di  $H$  e restituisce l'elemento a chiave minima.

`insert(elem e, chiave k)`

aggiunge ad  $H$  un nuovo  $B_0$  con dati  $e$  e  $k$ . Ripristina poi la proprietà di unicità in  $H$  mediante fusioni successive dei doppioni  $B_i$ .

`deleteMin()`

trova l'albero  $T_h$  con radice a chiave minima. Togliendo la radice  $T_h$  esso si spezza in  $h$  alberi binomiali  $T_0, \dots, T_{h-1}$ , che vengono aggiunti ad  $H$ . Ripristina poi la proprietà di unicità in  $H$  mediante fusioni successive dei doppioni  $B_i$ .

`decreaseKey(elem e, chiave d)`

decrementa di  $d$  la chiave nel nodo  $v$  contenente l'elemento  $e$ . Ripristina poi la proprietà di ordinamento a heap spingendo il nodo  $v$  verso l'alto tramite ripetuti scambi di nodi.

`delete(elem e)`

richiama `decreaseKey(e, -∞)` e poi `deleteMin()`.

`increaseKey(elem e, chiave d)`

richiama `delete(e)` e poi `insert(elem e, K+d)` dove K era la chiave associata all'elemento e.

`merge(CodaPri c1, CodaPri c2) → CodaPri c3`

unisce c<sub>1</sub> e c<sub>2</sub> in un nuovo heap binomiale c<sub>3</sub>. Ripristina poi la proprietà di unicità in c<sub>3</sub> mediante fusioni successive dei doppioni B<sub>i</sub>.

Tutte le operazioni richiedono tempo  $T(n) = O(\log n)$ .

Durante ristruttura si avranno al più tre B<sub>i</sub>:  $H_i \geq 0$ .

## Heap di Fibonacci

Heap binomiale rilassato: si ottiene un heap binomiale rilassando la proprietà di unicità dei B<sub>i</sub> ed utilizzando un atteggiamento più "pigro" durante l'operazione insert (la foresta non verrà subito ristrutturata).

Heap di Fibonacci: si ottiene da un heap binomiale rilassato indebolendo la proprietà di struttura dei B<sub>i</sub> che non sono più necessariamente alberi binomiali.

Analisi sofisticata: i tempi di esecuzione sono ammortizzati su sequenze di operazioni, cioè dividendo il costo complessivo della sequenza di operazioni per il numero di operazioni della sequenza.

	Find Min	Insert	Delete	DelMin	Incr. Key	Decr. Key	merge
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$O(n)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$O(1)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
d-Heap	$O(1)$	$O(\log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$
d-Heap (d cost.)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Heap Binom.	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Fibon.	$O(1)$	$O(1)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)^*$	$O(1)^*$	$O(1)$

\* operazioni ammortizzate

Il costo ammortizzato di un'operazione è il costo "medio" rispetto a una sequenza qualsiasi di operazioni. Se un'operazione ha costo ammortizzato costante e eseguo una sequenza (qualsiasi) di  $K$  operazioni è possibile che il costo di una singola operazione può non essere costante, ma l'intera sequenza costerà  $\mathcal{O}(K)$ .

Diverso dal caso medio: non c'è nessuna distribuzione di probabilità (sulla sequenza da eseguire) e l'algoritmo è deterministico.

Molto utile quando si vogliono buone prestazioni sull'intera sequenza e non garanzie sulla singola operazione.

### Teorema

Usando un Heap di Fibonacci, una qualsiasi sequenza di  $n$  insert,  $d$  delete,  $f$  findMin,  $m$  deleteMin,  $\Delta$  increaseKey,  $\delta$  decreaseKey,  $\mu$  merge prende tempo (nel caso peggiore)

$$\mathcal{O}(n + f + \delta + \mu + (d + m + \Delta) \log n)$$

### Lezione 13

La teoria dei grafi fu affrontata da Eulero con il problema dei 7 ponti di Königsberg (Prussia): visitare tutta la città senza passare due volte sullo stesso ponte.

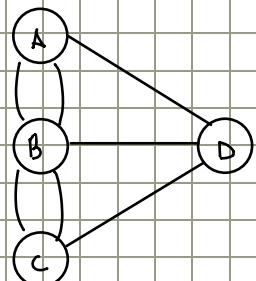
#### Definizione di grafo (non orientato)

Un grafo  $G = (V, E)$  consiste in:

- un insieme  $V$  di vertici (o nodi);
- un insieme  $E$  di coppie (non ordinate) di vertici, detti archi.

es:  $V = \{A, B, C, D\}$

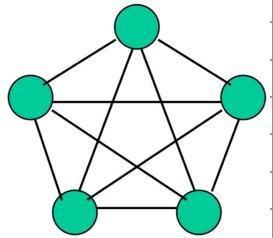
$E = \{(A, B), (A, B), (A, D), (B, C), (B, C), (B, D), (C, D)\}$



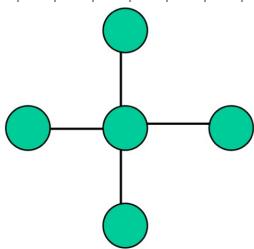
Nota: È più propriamente detto multigrafo, in quanto contiene archi paralleli.

contiene più occorrenze dello stesso elemento

es.



Ciclo, grafo completo  
ogni nodo ha un solo  
arco verso un altro  
nodo



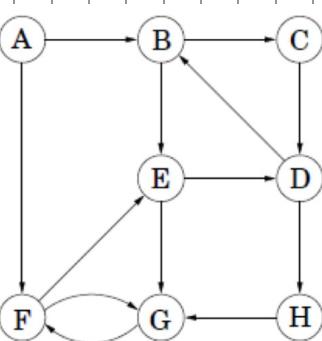
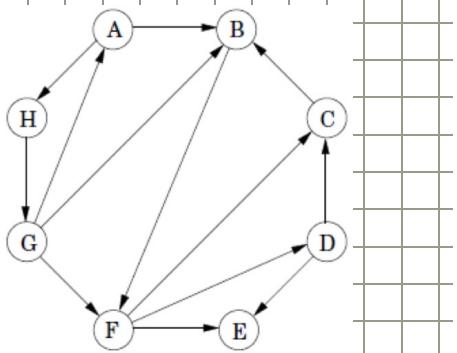
Stella, ogni nodo è  
collegato soltanto al  
nodo centrale

## Definizione di grafo diretto

Un grafo  $D = (V, A)$  consiste in:

- un insieme  $V$  di vertici (o nodi);
- un insieme  $A$  di coppie ordinate di vertici, detti archi diretti.

es.



## Terminologia

$G = (V, E)$  grafo non diretto

$n = |V|$  numero di vertici

$m = |E|$  numero di archi

$u$  ed  $v$  sono adiacenti (vicini)

$(u, v)$  è incidente a  $u$  e a  $v$  (detti estremi)

$\delta(v)$ : grado di  $v$ : # archi incidenti a  $v$

grado di  $G = \max_{v \in V} \{\delta(v)\}$

Esiste una relazione fra grado dei nodi e numero di archi: la somma dei gradi di ogni

nodo è:  $\sum_{v \in V} \delta(v) = 2m$ , - in ogni grafo il numero di nodi di grado dispari  
è pari.

nei grafici diretti:  $\sum_{v \in V} \delta(v) = \sum_{v \in V} \delta_{\text{out}}(v) = m$ .

cammino: sequenza di nodi connessi da archi.

lunghezza di un cammino: # archi del cammino.

**distanza:** la distanza del più corto cammino tra due vertici (in un grafo orientato deve rispettare il verso di ordinamento).

**è connesso** se esiste un cammino per ogni coppia di vertici. (esiste sempre un cammino da ogni nodo ad un altro)

**ciclo:** un cammino chiuso, ovvero un cammino da un vertice a se stesso.

**diametro:** la massima distanza fra due nodi,  $\max_{u,v \in V} \text{dist}(u,v)$ , il diametro di un grafo non connesso è  $\infty$ .

**Grafo pesato:** è un grafo  $G = (V, E, w)$  in cui ad ogni arco viene associato un valore definito dalla funzione peso  $w$  (definita su un opportuno insieme, di solito  $\mathbb{R}$ ).

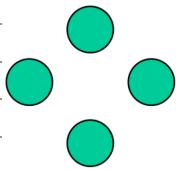
Due grafî particolari:

**Grafo totalmente sconnesso:** è un grafo  $G = (V, E)$  tale che  $V \neq \emptyset$  ed  $E = \emptyset$

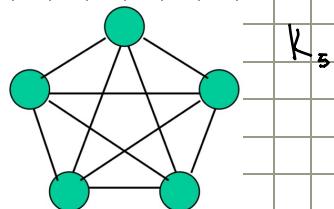
**Grafo completo:** per ogni coppia di nodi esiste un arco che li congiunge. Il grafo completo con  $n$  vertici verrà indicato con  $K_n$  e  $m = |E| = n(n-1)/2$

es:

grafo totalmente sconnesso



grafo completo



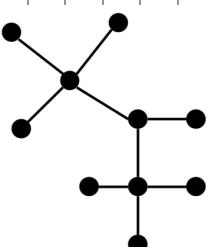
Un grafo (senza cappi o archi paralleli) può avere un numero di archi  $m$  compreso tra 0 e  $n(n-1)/2 = \Theta(n^2)$ .

**Definizione albero**

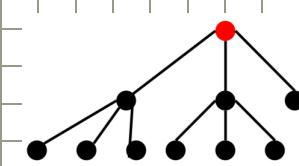
Un **albero** è un grafo connesso ed aciclico. (può essere libero o radicato)

es:

libero



radicato



## Teorema

Sia  $T = (V, E)$  un albero; allora  $|E| = |V| - 1$

dim. induzione

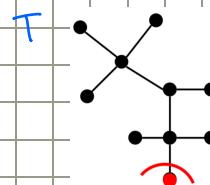
caso base:  $|V| = 1 \quad T \bullet \quad |E| = 0 = |V| - 1$

caso induttivo:  $|V| > 1$

Sia  $n$  il numero di nodi di  $T$ , poiché  $T$  è connesso e aciclico ha almeno una foglia (nodo con grado 1) (se tutti i nodi avessero grado almeno 2 ci sarebbe un ciclo.)

rimuovendo tale foglia si ottiene un grafo connesso e aciclico con  $n-1$  nodi che per ipotesi induttiva ha  $n-2$  archi

allora  $T$  ha  $n$  nodi e  $n-1$  archi.



Sia  $G = (V, E)$  un grafo non orientato, le seguenti affermazioni sono tutte equivalenti:

- $G$  è un albero;
  - due vertici qualsiasi di  $G$  sono collegati da un unico cammino semplice;
  - $G$  è connesso, ma se viene rimosso un arco qualsiasi da  $E$ , il grafo risultante non è connesso;
  - $G$  è connesso e  $|E| = |V| - 1$ ;
  - $G$  è aciclico e  $|E| = |V| - 1$ ;
  - $G$  è aciclico, ma se un arco qualsiasi viene aggiunto a  $E$ , il grafo risultante contiene un ciclo.
- (tocca i nodi una sola volta)

Per un grafo connesso con  $n$  nodi e  $m$  archi vale:  $n-1 \leq m \leq n(n-1)/2$ .

Se un grafo ha  $m \geq n-1$  archi, non è detto che sia connesso.

Se  $G$  è connesso  $m = \Omega(n)$  e  $m = O(n^2)$ .

## Definizione

Dato un grafo  $G$ , un **ciclo** (rispettivamente un **cammino**) **Euleriano** è un ciclo (rispettivamente un **cammino** non chiuso) di  $G$  che passa per tutti gli archi di  $G$  una e una sola volta.

## Teorema (di Eulero)

Un grafo  $G$  ammette un **ciclo Euleriano** sse tutti i nodi hanno grado pari.  
Inoltre, ammette un **cammino Euleriano** sse tutti i nodi hanno grado pari tranne due  
(gli estremi del cammino).

Il problema dei 7 punti non ammette quindi soluzioni. I grafi costituiscono un linguaggio  
potente per descrivere oggettivi problemi algoritmici.

reti **stradali** e di **trasporto** (strade, aeroporti, metro,...) dove un problema è trovare  
il cammino minimo tra due nodi dove ogni arco può avere un peso. (strada più breve,  
più veloce,...).

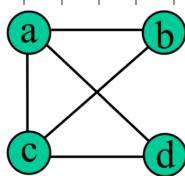
reti **sociali** (facebook, IMDb,...) dove i problemi sono: individuare i nodi centrali,  
individuare "comunità", calcolo distanza tra due nodi, trovare il diametro della rete.  
(di solito in ogni rete sociale il diametro è basso, Kevin Bacon Graph).

reti delle **dipendenze** (esami e propedeuticità, moduli software di un progetto e dipendenze,...)  
non ci sono cicli in questi grafi, un problema è: trovare un ordine in cui eseguire i compiti  
in modo da rispettare le dipendenze, trovare insieme indipendente (insieme  $X$  di nodi di cardinalità  
max t.c.  $\forall u, v \in X, u \neq v$  non sono adiacenti), trovare colorazione di un grafo (colorare i nodi  
del grafo risultante usando  $\min X$  di colori in modo che due nodi adiacenti non abbiano lo  
stesso colore).

## Lezione 14

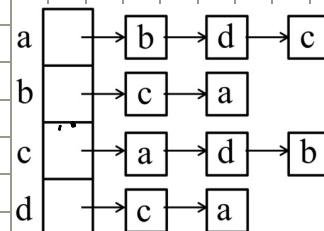
Due strutture dati che vengono utilizzate per rappresentare i grafi.

Grafi non diretti:



	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

matrice simmetrica



Matrice di adiacenza

Spazio:  $O(n^2)$

Operazione:

Elenco archi incidenti in  $v$ :  $O(n)$

C'è un arco  $(u,v)$ ? :  $O(1)$

scorre tutta la colonna riga

posizione  $(u,v)$

Liste di adiacenza

Spazio:  $O(m+n)$

Operazione:

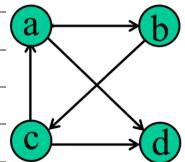
Elenco archi incidenti in  $v$ :  $O(\delta(v))$

C'è un arco  $(u,v)$ ? :  $O(\min\{\delta(u), \delta(v)\})$

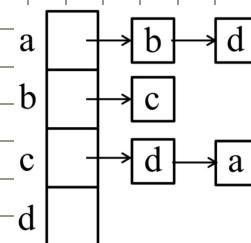
tutti gli archi collegati a  $v$

vedo un arco di  $v$  uno di  $v$  finché uno dei due non finisce e sopra se è presente un altro

Grafi diretti:



	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	1	0	0	1
d	0	0	0	0



Matrice di adiacenza

Spazio:  $O(n^2)$

Operazione:

Elenco archi incidenti in  $v$ :  $O(n)$

C'è un arco  $(u,v)$ ? :  $O(1)$

Liste di adiacenza

Spazio:  $O(m+n)$

Operazione:

Elenco archi incidenti in  $v$ :  $O(\delta(v))$

C'è un arco  $(u,v)$ ? :  $O(\delta(u))$

Per vedere quali parti del grafo sono raggiungibili da un certo nodo eseguo una **visita del grafo**. Una visita di un grafo  $G$  permette di esaminare i nodi e gli archi di  $G$  in modo **sistematico** (se  $G$  è connesso) generando un **albero di visita**.

È un problema di base in molte applicazioni ed esistono vari tipi di visite con diverse proprietà:

- visita in **ampiezza** (BFS = breadth first search)
- visita in **profondità** (DFS = depth first search)

### Visita in ampiezza

Dato un grafo  $G$  (non pesato) e un nodo  $s$ , trova tutte le distanze/cammini minimi da  $s$  verso ogni altro nodo  $v$ .

Applicazioni: web crawling, social networking, network broadcast, garbage collection, model checking, risolvere puzzle.

algoritmo **visitaBFS**(vertice  $s$ )  $\rightarrow$  albero

rendi tutti i vertici non marcati

$T \leftarrow$  albero formato da un solo nodo  $s$

Coda  $F$

marca il vertice  $s$

$F.\text{enqueue}(s)$

while ( $\text{not } F.\text{isempty}()$ ) do

$u \leftarrow F.\text{dequeue}()$

for each (arco  $(u, v)$ ) in  $G$  do

if ( $v$  non è ancora marcato) then

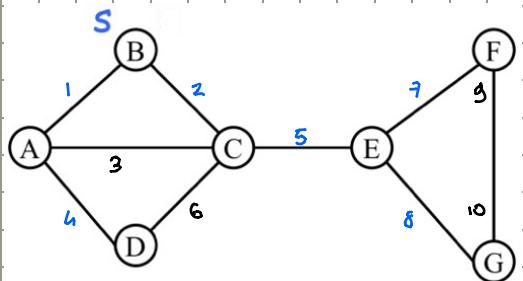
$F.\text{enqueue}(v)$

marca il vertice  $v$

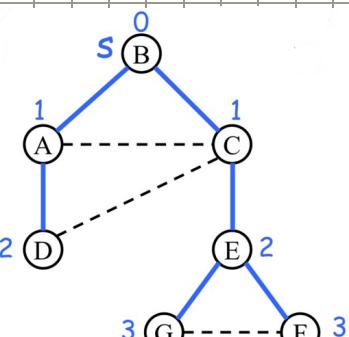
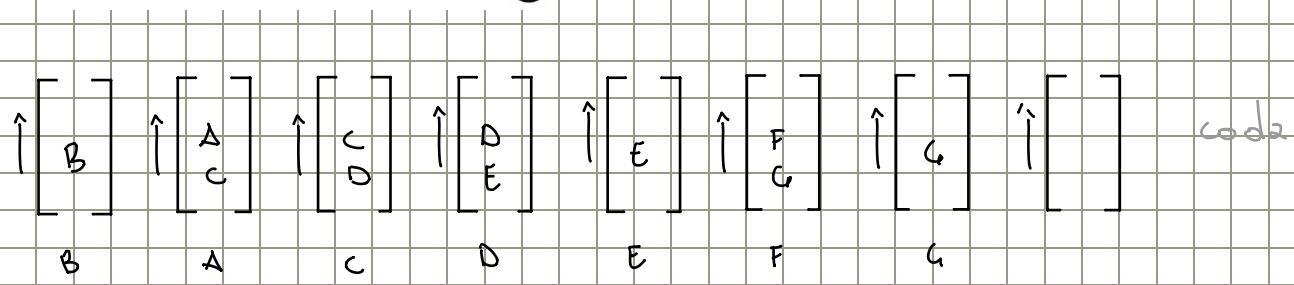
rendi  $u$  padre di  $v$  in  $T$

return  $T$

es:

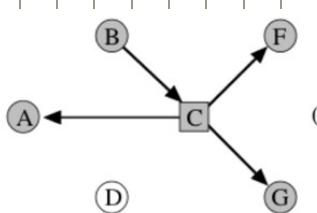
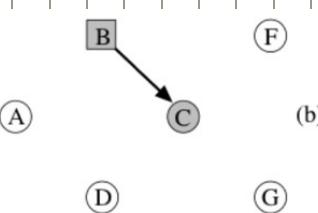
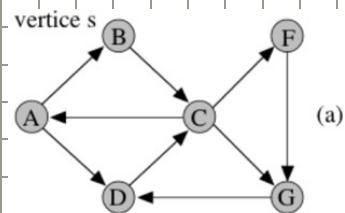


I numeri neri indicano che durante l'algoritmo quel nodo è già stato visitato perciò non verrà collegato.

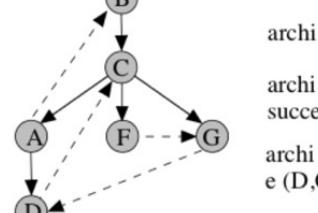
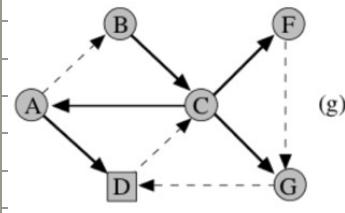
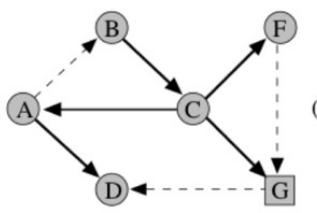
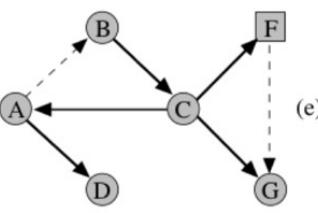
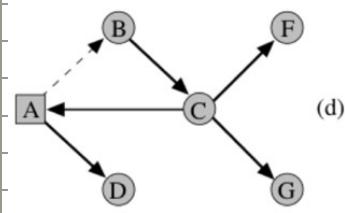


L'albero  $T$  ritornato dall'algoritmo

es:



grafi diretti



archi con estremi nello stesso livello: (F,G)

archi da un livello a quello immediatamente successivo: (G,D)

archi da un livello a uno precedente: (A,B) e (D,C)

Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo. (e dalla connettività o meno del grafo rispetto a s)

algoritmo `visitBFS`(vertice s) → albero (m matrici di ad., l liste di ad.)

vendi tutti i vertici non marcati

$T \leftarrow$  albero formato da un solo nodo s

Coda F

marca il vertice s

F.enqueue(s)

while (not F.isEmpty()) do per ogni nodo

$u \leftarrow F.dequeue()$

for each (arco  $(u, v)$ ) in  $\delta(u)$  do

if ( $v$  non è ancora marcato) then

F.enqueue(v)

marca il vertice v

vendi  $u$  padre di  $v$  in  $T$

return T

$m = O(n^2)$

$l = O(m+n)$

$\sum \delta(v) = O(m)$

$m = O(n)$

$l = O(\delta(v))$

Si noti che se il grafo è connesso allora  $m \geq n-1$  e quindi  $O(m+n) = O(m)$ , ricordando che  $m \leq n(n-1)/2$  si ha che  $O(m+n) = O(n^2) \Rightarrow m = O(n^2)$  la rappresentazione mediante liste di adiacenza è temporalmente più efficiente!

### Teorema

Per ogni nodo  $v$ , il livello di  $v$  nell'albero BFS è pari alla distanza di  $v$  dalla sorgente  $s$  (sia per grafi orientati che non):

- inserisco  $s$  in  $F$  (distanza 0)
- estraggo  $s$  e guardo tutti i suoi vicini; questi sono nodi di distanza 1; li inserisco in  $F$  e assegno loro livello 1; ora ho tutti i nodi a distanza 1.
- estraggo tutti i nodi di livello/distanza 1 e guardo per ognuno tutti i suoi vicini; i vicini non marcati sono a distanza 2 da  $s$ ; li inserisco in  $F$  e assegno loro livello 2; una volta finita l'estrazione avrò in  $F$  tutti i nodi di distanza 2 e così via.

## Visita in profondità

Applicazioni: trovare l'uscita di un labirinto: dove per marcire la strada presa utilizzo una variabile booleana e la pila con push e pop per arrotolare e srotolare la corda

procedura visita DFS ricorsiva (vertice v, albero T)

marca e visita il vertice v

for each (arco(v,w)) do

if (w non è marcato) then

aggiungi l'arco (v,w) all'albero T

visita DFS ricorsiva (w,T)

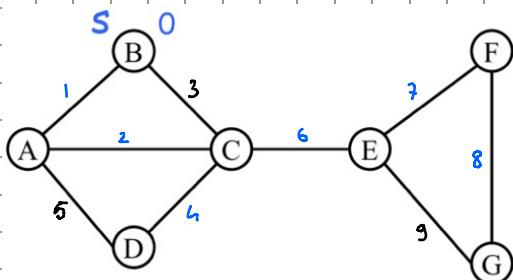
algoritmo visita DFS (vertice s) → albero

$T \leftarrow$  albero vuoto

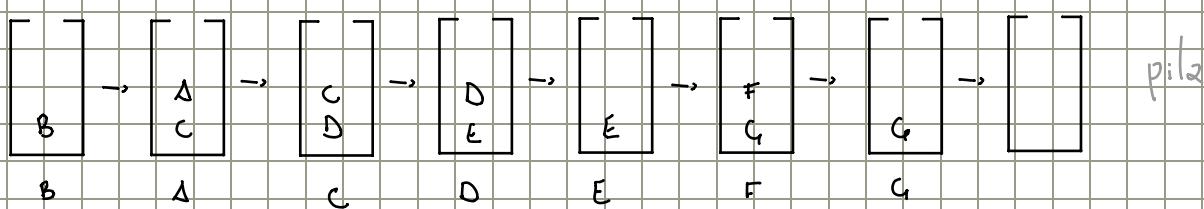
visita DFS ricorsiva (s,T)

return T

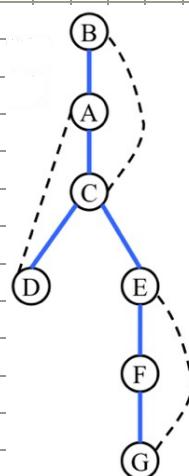
es:



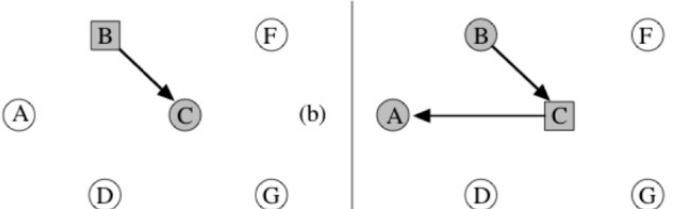
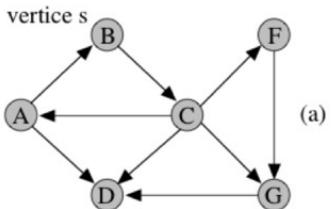
I numeri neri indicano che durante l'algoritmo quel nodo è già stato visitato perciò non verrà collegato.



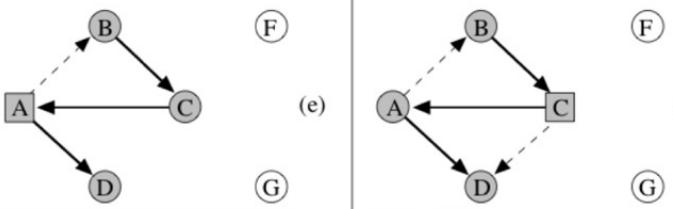
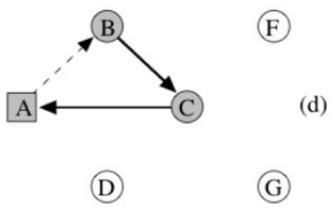
L'albero T ritornato dall'algoritmo



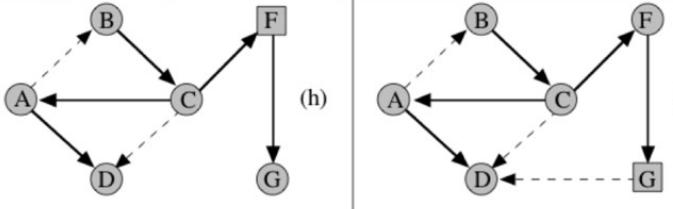
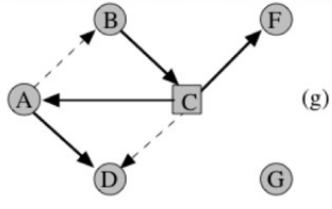
es:



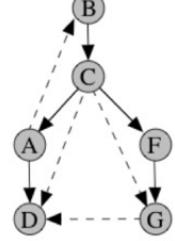
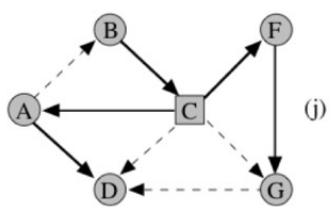
(c) grafi orientati



(f)



(i)



archi in avanti: (C,D) e (C,G)

archi all'indietro: (A,B)

archi trasversali a sinistra: (G,D)

Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo. (e della connettività o meno del grafo rispetto a s)

procedura visita DFS ricorsiva (vertice v, albero T)

marca e visita il vertice v

(in matrici di ad., | liste di ad.)

foreach (arco(v,w)) do

$m = O(n^2)$

if (w non è marcato) then

$| = O(m+n)$

aggiungi l'arco (v,w) all'albero T

visita DFS ricorsiva (w,T)       $| \leftarrow O(n)$

algoritmo visita DFS (vertice s)  $\rightarrow$  albero

$T \leftarrow$  albero vuoto

visita DFS ricorsiva (s,T)

return T

## Proprietà dell'albero DFS radicato in s

Se il grafo è **non orientato**, per ogni arco  $(u,v)$  si ha che  $(u,v)$  è un arco dell'albero DFS, oppure i nodi  $u$  e  $v$  sono l'uno discendente/antenato dell'altro.

Se il grafo è **orientato**, per ogni arco  $(u,v)$  si ha che  $(u,v)$  è un arco dell'albero DFS, oppure i nodi  $u$  e  $v$  sono l'uno discendente/antenato dell'altro

oppure  $(u,v)$  è un arco **trasversale a sinistra**, ovvero il vertice  $v$  è in un sottoalbero visitato precedentemente ad  $u$ .

## Lezione 15

### Usi della visita DFS

- Tenere il **tempo**

procedura **visitaDFS ricorsiva** (vertice  $v$ , albero  $T$ )

marca e visita il vertice  $v$

$pre(v) = \text{clock}$  prerisposta

tempo in cui viene scoperto il vertice

$\text{clock} = \text{clock} + 1$

for each (arco  $(v,w)$ ) do

if ( $w$  non è marcato) then

aggiungi l'arco  $(v,w)$  all'albero  $T$

visitaDFS ricorsiva ( $w, T$ )

$post(v) = \text{clock}$  postrisposta

tempo in cui viene abbandonato il vertice

$\text{clock} = \text{clock} + 1$

algoritmo **visitaDFS** (vertice  $s$ )  $\rightarrow$  albero

$\text{clock} = 1$

$T \leftarrow$  albero vuoto

visitaDFS ricorsiva ( $s, T$ )

return  $T$

Questo algoritmo è pensato per calcolare l'albero di visitaDFS a partire da una singola sorgente  $s$  a tutti i nodi da essa raggiungibili. Esistono però dei nodi non raggiungibili da  $s$ . E' utile far partire una visitaDFS dal nodo rimasto non marcato.

Quando non tutti i nodi sono raggiungibili dal punto di partenza si usa:

visit2DFS(graf G)

for each nodo  $v$  do

imposta  $v$  come non marcato

stessa complessità

clock = 1

$F \leftarrow$  foresta vuota

for each nodo  $v$  do

if ( $v$  è non marcato) then

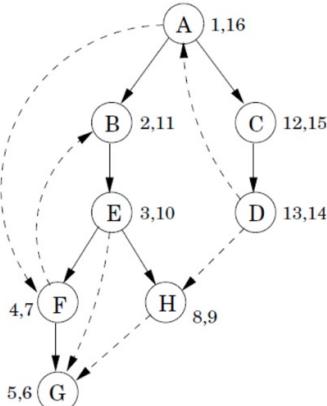
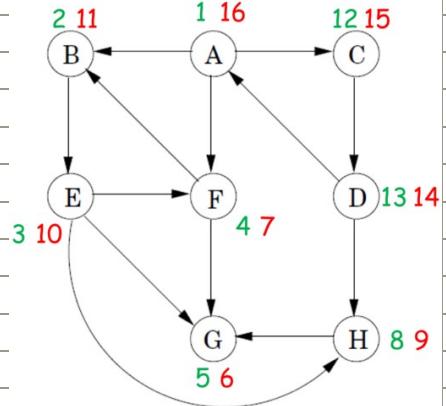
$T \leftarrow$  albero vuoto

visit2DFSricorsiva( $v, T$ )

aggiungi  $T$  ad  $F$

return  $F$

es:



Proprietà

Per ogni coppia di nodi  $u$  e  $v$  gli intervalli  $[pre(u), post(u)]$  e  $[pre(v), post(v)]$  o sono disgiunti o l'uno è contenuto nell'altro.

$v$  è antenato di  $v$  nell'albero DFS se  $pre(u) < pre(v) < post(v) < post(u)$ .

tipi di arco (nel grafo)

in avanti  $\begin{bmatrix} v \\ u \end{bmatrix}$

all'indietro  $\begin{bmatrix} u \\ v \end{bmatrix}$

trasversali  $\begin{bmatrix} v \\ u \end{bmatrix} \quad \begin{bmatrix} u \\ v \end{bmatrix}$

- Riconoscere la presenza di un ciclo in un grafo diretto.

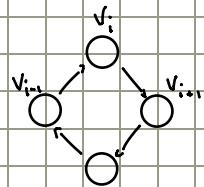
fa una visita DFS e controlla se c'è un arco all'indietro.

Un grafo diretto  $G$  ha un ciclo sse la visita DFS rivelà un arco all'indietro

$\Leftrightarrow$  se c'è un arco all'indietro chiaramente  $G$  ha un ciclo

$\Rightarrow$  se c'è ciclo  $\langle v_0, v_1, \dots, v_{k-1}, v_k = v_0 \rangle$

$v_i$  è il primo nodo scoperto nella visita, poiché  $v_{i-1}$  è raggiungibile da  $v_i$ , visito  $v_{i-1}$  prima di terminare la visita di  $v_i$ ; allora  $(v_{i-1}, v_i)$  è un arco all'indietro.

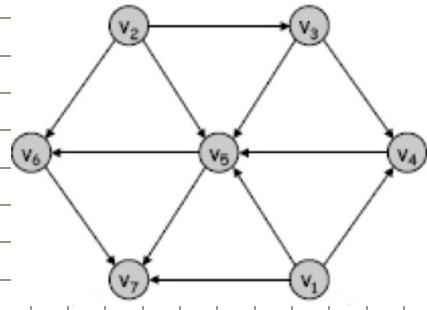


ha sicuramente almeno una sorgente e un pozzo

Un grafo diretto aciclico DAG è un grafo diretto  $G$  che non contiene cicli (diretti).

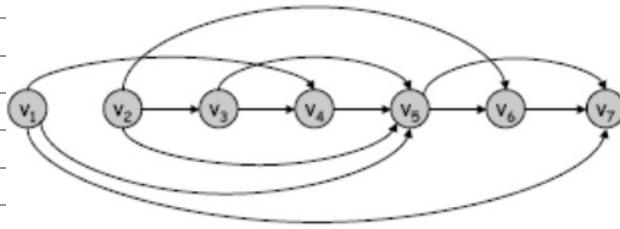
Un ordinamento topologico di un grafo diretto  $G = (V, E)$  è una funzione  $\sigma: V \rightarrow \{1, \dots, n\}$

tale che per ogni arco  $(u, v) \in E$   $\sigma(u) < \sigma(v)$



potro: solo  
archi entranti

sorgente: solo  
archi uscenti



ordinamento topologico

Un grafo diretto  $G$  ammette un ordinamento topologico sse  $G$  è un DAG.

Dim.

$\Rightarrow$  per assurdo: sia  $\sigma$  un ordinamento topologico di  $G$  e sia  $\langle v_0, v_1, \dots, v_k = v_0 \rangle$  un ciclo allora  $\sigma(v_0), \sigma(v_1) < \dots < \sigma(v_k) = \sigma(v_0)$

$\Leftarrow$  diamo un algoritmo costruttivo: fai una visita DFS e restituisci i nodi in ordine decrescente rispetto ai tempi di fine visita post(v)

## Ordinamento Topologico (grafo G)

$\text{top} = n$  numero vertici

$L \leftarrow$  lista vuota

è uguale alla visita DFS ( $O(n+m)$ )

chiama visitaDFS ma:

quando hai finito di visitare un nodo  $v$  (quando imposta  $\text{post}(v)$ ):

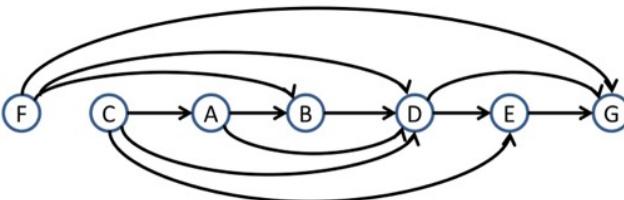
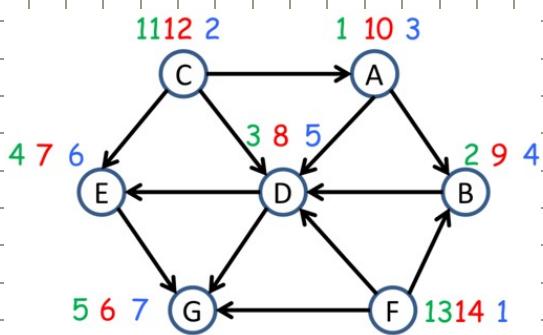
$\sigma(v) = \text{top}$

$\text{top} = \text{top} - 1$

aggiungi  $v$  in testa alla lista  $L$

return  $L$  e  $\sigma$

es:



$\text{pre}(v)$   $\text{post}(v)$   $\sigma(v)$

correttezza

per ogni coppia di nodi  $u$  e  $v$ , gli intervalli  $[\text{pre}(u), \text{post}(u)]$  e  $[\text{pre}(v), \text{post}(v)]$  o sono disgiunti o l'uno è contenuto nell'altro, non ci possono essere archi all'indietro (ho un dico)

quindi solo:

$\begin{bmatrix} & & & \end{bmatrix}_{\substack{u \\ v \\ v \\ v}}$  in avanti dove  $\text{post}(v) < \text{post}(u)$

$\begin{bmatrix} & & & \end{bmatrix}_{\substack{v \\ u \\ v \\ v}}$  all'indietro

$\begin{bmatrix} & & \end{bmatrix}_{\substack{v \\ v \\ u \\ u}} \begin{bmatrix} & & \end{bmatrix}_{\substack{u \\ u \\ v \\ v}}$  trasversali: dove  $\text{post}(v) < \text{post}(u)$

## Algoritmo alternativo

algoritmo Ordinamento Topologico (grafo  $G$ )  $\rightarrow$  lista

$$\hat{G} \leftarrow G$$

$ord \leftarrow$  lista vuota di vertici

while (esiste un vertice  $v$  senza archi entranti in  $\hat{G}$ ) do

    appendi  $v$  come ultimo elemento di  $ord$

    rimuovi da  $\hat{G}$  il vertice  $v$  e tutti i suoi archi uscenti

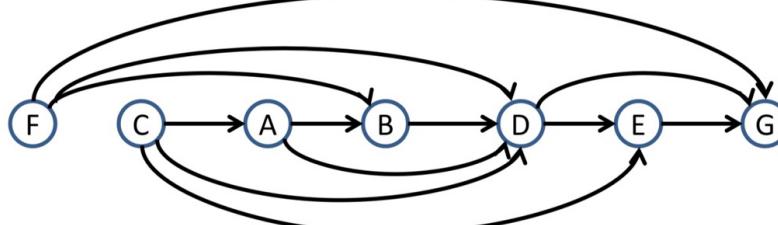
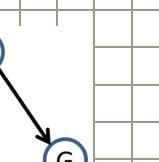
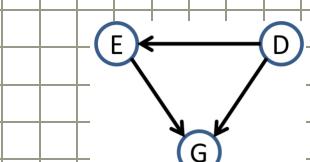
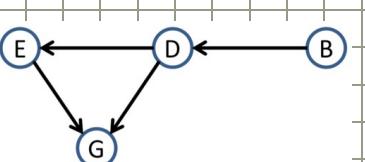
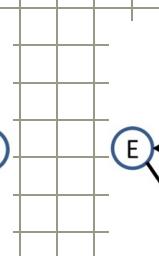
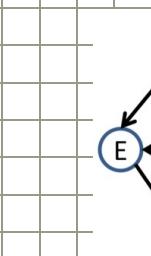
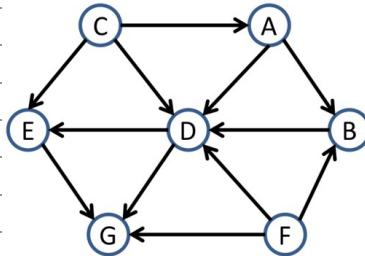
(\*) if ( $\hat{G}$  non è diventato vuoto) then

        errore il grafo  $G$  non è aciclico

return  $ord$

(\*) altrimenti in  $\hat{G}$  ogni vertice deve avere almeno un arco entrante, e quindi posso trovare un ciclo percorrendo archi entranti a ritroso, e quindi  $G$  non può essere aciclico

es:

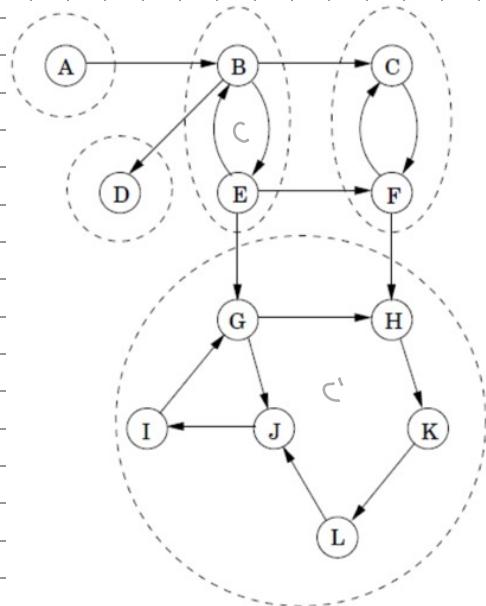


Componenti: fortemente connesse (sono gradi diretti)

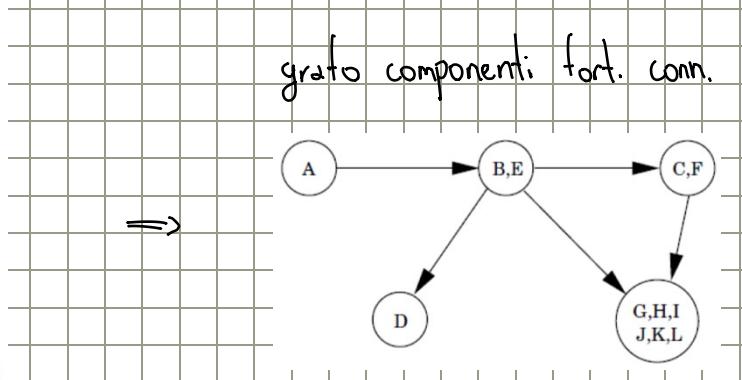
Una componente fortemente connesse di un grafo  $G = (V, E)$  è un insieme massimale di vertici  $C \subseteq V$  tale che per ogni coppia di nodi  $u$  e  $v$  in  $C$ ,  $u$  è raggiungibile da  $v$  e  $v$  è raggiungibile da  $u$ .

massimale: se si aggiunge un qualsiasi vertice a  $C$  la proprietà non è più vera.

es:



grafo componenti fort. conn.



è sempre un DAG

Per calcolare le componenti fortemente connesse si usano tre proprietà:

Proprietà 1: se si esegue la procedura visitaDFS ricorsiva a partire da un nodo  $v$  la procedura termina dopo che tutti i nodi raggiungibili da  $v$  sono stati visitati.

Idea: eseguire una visita a partire da un nodo di una componente pozzo, "eliminare" la componente e ripetere.

Per trovare una componente pozzo:

Proprietà 2: Se  $C$  e  $C'$  sono due componenti e c'è un arco da un nodo in  $C$  verso uno in  $C'$ , allora il più grande valore post() in  $C$  è maggiore del più alto valore post() di  $C'$ .

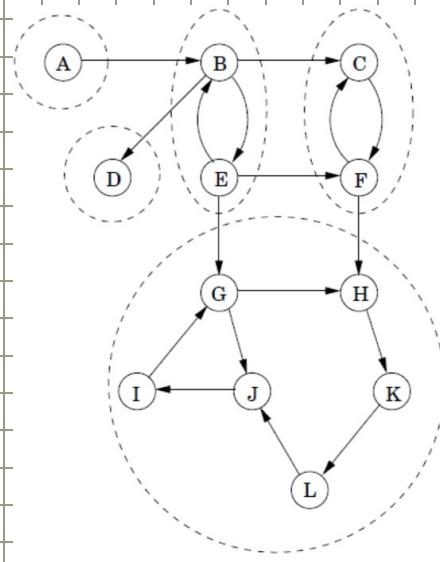
dim: se la DFS visita prima  $C'$  di  $C$ : banale. se visita prima  $C$ , allora si ferma dopo che ha raggiunto tutti i nodi di  $C$  e  $C'$  e termina su un nodo di  $C$ . dev tornare indietro

inizia a finire c'è poi  
inizia a finire C

Proprietà 3: il nodo che riceve da una visitaDFS il valore più grande di post() appartiene a una componente sorgente

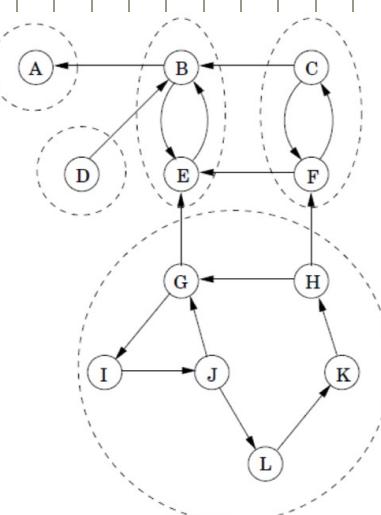
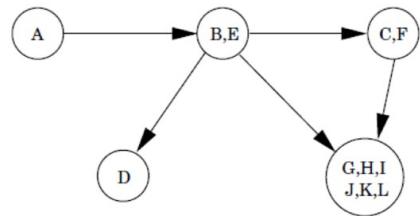
Idea: Dato che avevamo bisogno di una componente puro ci basterà invertire gli archi.

es:

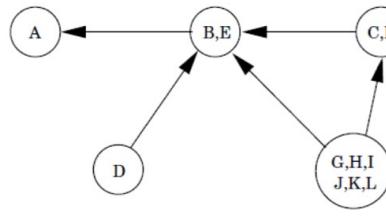


$G$

le componenti  
fortemente  
connesse sono  
le stesse



$G''$



visita DFS (grafo  $G$ )

calcola  $G^T$  (modifica archi per farli reversed)

esegui DFS( $G^T$ ) per trovare valori post(v)

return CompConnesse( $G$ )

il tempo è uno visita dfs (1,2)

$\Theta(n+m)$

CompConnesse (grafo  $G$ )

for each nodo v do

imposta v come non marcato

Comp  $\leftarrow \emptyset$

for each nodo v in ordine decrescente di post(v) do

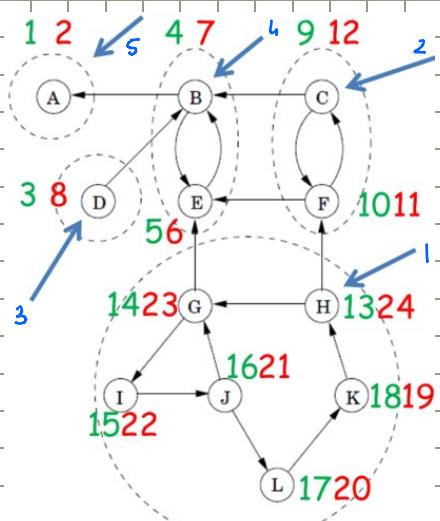
if (v è non marcato) then

$T \leftarrow$  albero vuoto

visita DFS ricorsiva ( $v, T$ )

aggiungi  $T$  a Comp

return Comp



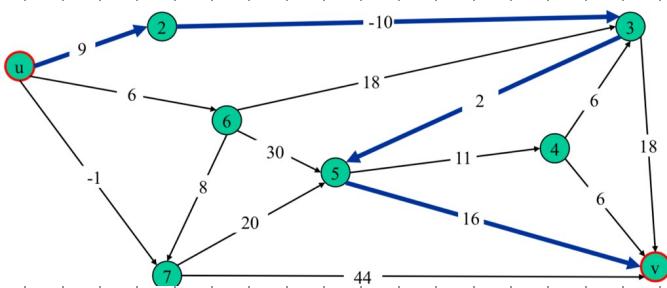
## Lezione 16

Cammini minimi a singola sorgente in grafi (senza pesi negativi):

Sia  $G = (V, E, w)$  un grafo orientato o non con pesi  $w$  reali sugli archi. Il costo o lunghezza di un cammino  $\pi = \langle v_0, v_1, \dots, v_k \rangle$  è:

$$w(\pi) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

Un cammino minimo tra una coppia di vertici  $x$  e  $y$  è un cammino avente costo minore o uguale a quello di ogni altro cammino tra gli stessi vertici. Il cammino minimo non è necessariamente unico.



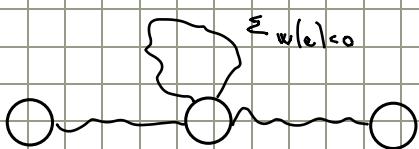
La distanza  $d_G(u, v)$  da  $u$  a  $v$  in  $G$  è il costo di un qualsiasi cammino minimo da  $u$  a  $v$

$$d_G(u, v) = 12$$

Un problema tipico: dati  $u$  e  $v$  trovare un cammino minimo (e/o distanza) da  $u$  a  $v$ .  
es: Maps dove gli incroci sono nodi e la strada archi e il peso può essere la lunghezza, il tempo di percorrenza, ...

Non esiste sempre un cammino minimo fra due nodi,  
se non esiste nessun cammino da  $u$  a  $v$ :  $d(u, v) = +\infty$

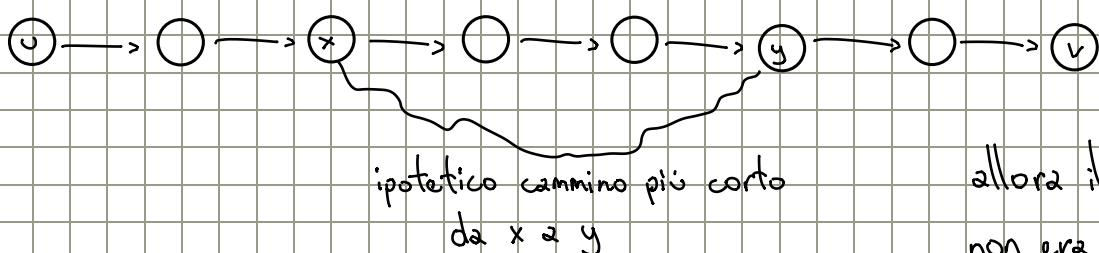
se c'è un cammino che contiene un ciclo (raggiungibile) il cui costo è negativo:  $d(u, v) = -\infty$



se  $G$  non contiene cicli negativi, esistono cammini minimi semplici (non contiene nodi ripetuti)

Ogni sottocammino di un cammino minimo è un cammino minimo.

dim: tecnica cut & paste



allora il cammino da  $u$  a  $v$  non era minimo!

## Disuguaglianza triangolare

per ogni  $u, v, x \in V$  vale:  $d(u, v) \leq d(u, x) + d(x, v)$ , il cammino da  $u$  a  $v$  che passa per  $x$  è un cammino nel grafo e quindi il suo costo è almeno il costo del cammino minimo da  $u$  a  $v$ .

Problema del calcolo dei cammini minimi a singola sorgente:

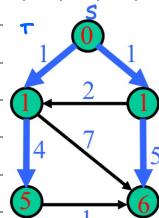
Due varianti:

- Dato  $G = (V, E, w)$ ,  $s \in V$ , calcola le distanze di tutti i nodi da  $s$ , ovvero  $d_s(v) \forall v \in V$ .
  - Dato  $G = (V, E, w)$ ,  $s \in V$ , calcola l'albero dei cammini minimi di  $G$  radicato in  $s$ .

## Albero dei cammini minimi (o Shortest Path Tree - SPT)

T'è un albero dei cammini minimi con sorgente  $s$  di un grafo  $G = (V, E, w)$  se:

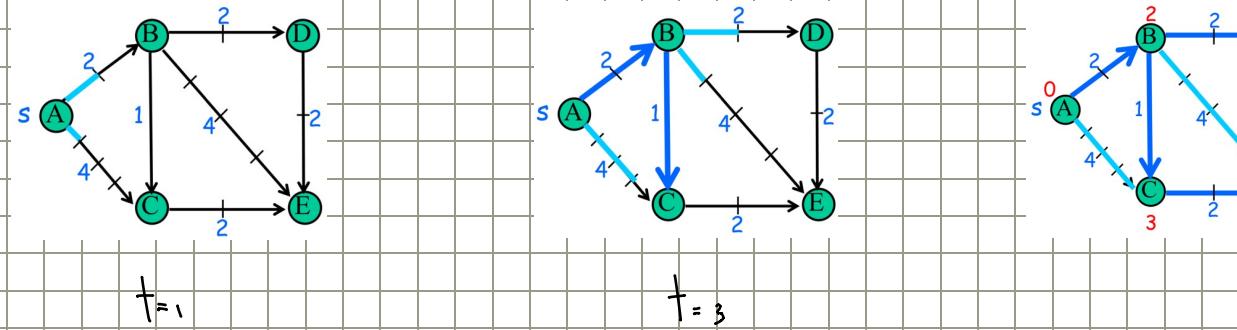
- $T$  è un albero radicato in  $s$ .
  - per ogni  $v \in V$ , vale:  $d_T(s, v) = d_G(s, v)$ .



# Algoritmo di Dijkstra

Assunzione: tutti gli archi hanno peso non negativo, ovvero ogni arco  $(u, v)$  del grafo ha peso  $w(u, v) \geq 0$ . (per grafi non pesati si utilizza la visita BRF)

Ideaz intuitiva dell'algoritmo: pompare acqua nella sorgente.



L'algoritmo di Dijkstra utilizza un approccio **greedy** (goloso).

1. mantiene per ogni nodo  $v$  una **stima** (per eccesso)  $D_{sv}$  alla distanza d( $s, v$ ).

Inizialmente, unica stima finita  $D_{ss} = 0$ .

2. mantiene un insieme  $X$  di nodi per cui le stime sono esatte; e anche un albero  $T$  dei cammini minimi verso nodi in  $X$  (albero nero). Inizialmente  $X = \{s\}$ ,  $T$  non ha archi.

3. ad ogni passo aggiunge a  $X$  il nodo  $v$  in  $V - X$  la cui stima è minima; aggiunge a  $T$  uno specifico arco (arancione) entrante in  $v$ .

4. aggiorna le stime guardando i nodi adiacenti a  $v$ .

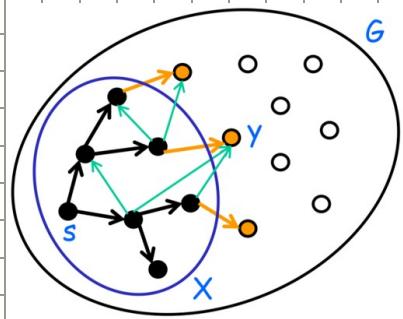
I nodi da aggiungere progressivamente a  $X$  (e quindi a  $T$ ) sono mantenuti in una coda di priorità, associati ad un unico arco (arancione) che li connette a  $T$ .

La stima per un nodo  $y \in V - X$  è:

$$D_{sy} = \min\{D_{sx} + w(x, y) : (x, y) \in E, x \in X\}$$

L'arco che fornisce il minimo è l'arco arancione.

Se  $y$  è in coda con arco  $(x, y)$  associato, e se dopo aver aggiunto  $v \in T$  troviamo un arco  $(v, y)$  tale che  $D_{sv} + w(v, y) < D_{sx} + w(x, y)$ , allora rimpiazziamo  $(x, y)$  con  $(v, y)$  ed aggiorniamo  $D_{sy}$ .



nodi bianchi: nodi per i quali non è stato scoperto nessun cammino; stima:  $+\infty$

nodi arancioni: nodi "scoperti"; hanno stima  $< +\infty$  sono mantenuti in una coda di priorità insieme al "miglior" arco entrante

algoritmo Dijkstra (grafo  $G$ , vertice  $s$ )  $\rightarrow$  albero

for each (vertice  $v$  in  $G$ ) do  $D_{sv} \leftarrow +\infty$

$\hat{T} \leftarrow$  albero formato dal solo nodo  $s$ ;  $X \leftarrow \emptyset$

Coda Priorità  $S$

$D_{ss} \leftarrow 0$ ;  $S.insert(s, 0)$

while ( $\text{not } S.isEmpty()$ ) do

$v \leftarrow S.deleteMin(); X \leftarrow X \cup \{v\}$

for each (arco  $(v, u)$  in  $G$ ) do

if ( $D_{sv} = +\infty$ ) then

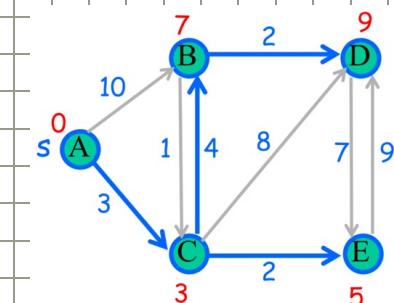
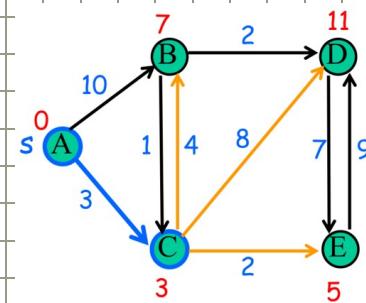
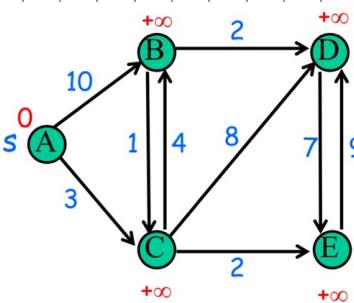
$S.insert(u, D_{sv} + w(v, u))$ ;  $D_{sv} \leftarrow D_{sv} + w(v, u)$ ; rendi  $v$  padre di  $u$  in  $\hat{T}$

else if ( $D_{sv} + w(v, u) < D_{su}$ ) then

$S.decreaseKey(u, D_{su} - D_{sv} - w(v, u))$ ;  $D_{su} \leftarrow D_{su} + w(v, u)$ ; rendi  $v$  padre di  $u$  in  $\hat{T}$

return  $\hat{T}$

es.



- Correttezza

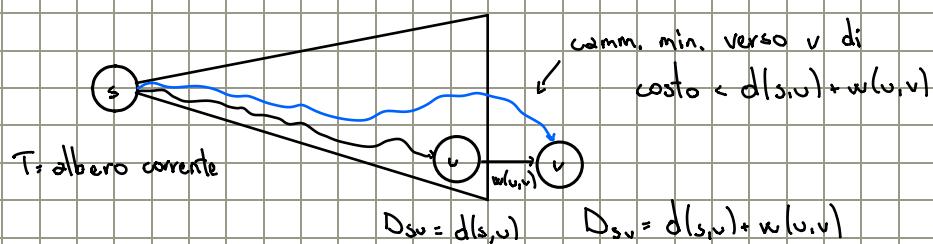
Lemma

Quando il nodo  $v$  viene estratto dalla coda con priorità vale:

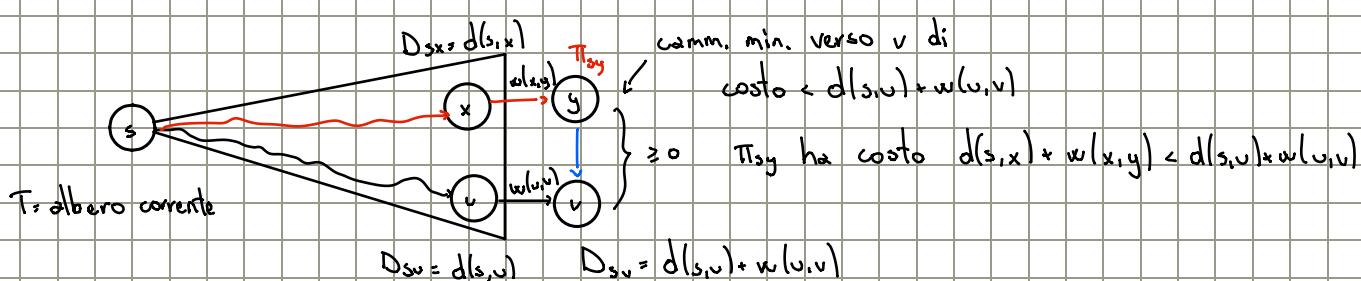
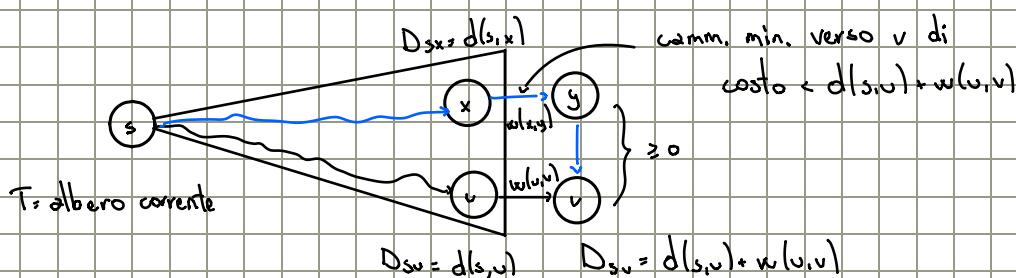
- $D_{sv} = d(s, v)$  (stima esatta)
- il cammino da  $s$  a  $v$  nell'albero corrente ha costo  $d(s, v)$  (cammin. min. in  $T$ )

dim (per assurdo)

Sia  $v$  il primo nodo per cui l'alg sbaglia, sia  $(v, v)$  l'arco aggiunto all'albero corrente



$(x, y)$ : primo arco del cammino t.c.  $x \in T$  e  $y \notin T$



$$\Rightarrow D_{sy} \leq d(s, x) + w(x, y) < d(s, v) + w(v, v)$$

assurdo: l'alg avrebbe estratto  $y$  e non  $v$  (se  $y = v$ ,  $v$  avrebbe avuto una stima più piccola)

- Complessità

algoritmo Dijkstra (grafo  $G$ , vertice  $s$ )  $\rightarrow$  albero

for each (vertice  $v$  in  $G$ ) do  $D_{sv} \leftarrow +\infty$

$\hat{T} \leftarrow$  albero formato dal solo nodo  $s$ ;  $X \leftarrow \emptyset$

Coda Priorità  $S$

$D_{ss} \leftarrow 0$ ;  $S.insert(s, 0)$

while ( $\text{not } S.isEmpty()$ ) do

$u \leftarrow S.deleteMin(); X \leftarrow X \cup \{u\}$

for each (arco  $(u, v)$  in  $G$ ) do

if ( $D_{sv} = +\infty$ ) then

$S.insert(v, D_{su} + w(u, v))$ ;  $D_{sv} \leftarrow D_{su} + w(u, v)$ ; rendi  $u$  padre di  $v$  in  $\hat{T}$

else if ( $D_{su} + w(u, v) < D_{sv}$ ) then

$S.decreaseKey(v, D_{sv} - D_{su} - w(u, v))$ ;  $D_{sv} \leftarrow D_{su} + w(u, v)$ ; rendi  $u$  padre di  $v$  in  $\hat{T}$

return  $\hat{T}$

Se si escludono le operazioni sulla coda di priorità: ogni nodo viene inserito ed estratto una sola volta e per ogni nodo guarda gli archi uscenti quindi  $O(m+n)$  con il costo delle operazioni sulla coda con priorità:

Supponendo che il grafo  $G$  sia rappresentato tramite liste di adiacenza, e supponendo che tutti i nodi siano connessi ad  $s$ , avremo  $n$  insert,  $n$  deleteMin e al più  $m$  decreaseKey al costo di:

	Insert	DelMin	DecKey
Array non ord.	$O(1)$	$O(n)$	$O(1)$
Array ordinato	$O(n)$	$O(1)$	$O(n)$
Lista non ord.	$O(1)$	$O(n)$	$O(1)$
Lista ordinata	$O(n)$	$O(1)$	$O(n)$
Heap binario	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Binom.	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Fibon.	$O(1)$	$O(\log n)^*$ (ammortizzata)	$O(1)^*$ (ammortizzata)

array non ord:  $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$

array ord:  $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(mn)$

liste non ord:  $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$

liste ord:  $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(mn)$

heap binario o binomiale:  $n \cdot O(\log n) + n \cdot O(\log n) + O(m) \cdot O(\log n) = O(m \log n)$

heap di fibonacci:  $n \cdot O(1) + n \cdot O(\log n)^* + O(m) \cdot O(1) = O(m + n \log n)$

tempo della soluzione migliore:  $O(m + n \log n)$

Osservazione sulla `decreaseKey`:

Le complessità computazionali esposte per la `decreaseKey` sono valide supponendo di avere un puntatore diretto all'elemento, quindi si deve mantenere un puntatore fra il nodo  $v$  nell'array dei nodi della lista di adiacenza del grafo e l'elemento nella coda di priorità associato al nodo  $v$ ; tale puntatore viene inizializzato nella fase di inserimento all'interno della coda.