

# Domande svolte di Algoritmi e Strutture Dati

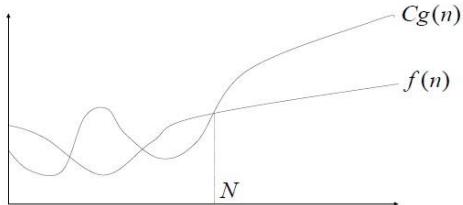
LASCIATE OGNI SPERANZA VOI CHE STUDIATE  
ROMAN KISS, DAVIDE MORA, CRISTINA TEDESCO

## Spiegare il concetto di complessità di un programma e descrivere le notazioni asintotiche (O, teta, omega, o, w)

La complessità può essere valutata in tempo e in spazio: quella in **tempo** è legata alla stima del tempo impiegato dall'algoritmo, confrontato con algoritmi diversi e ottimizzate le parti più importanti. Quella in **spazio** è legata alla stima della dimensione dell'input.

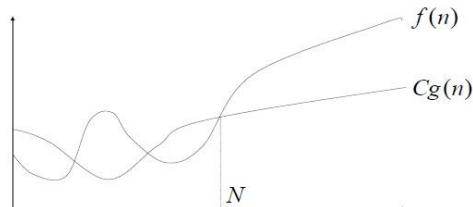
### Notazione asintotica O: limite superiore (1)

$O(g(n)) = \{f(n) : \text{esistono } C > 0 \text{ ed } N \text{ tali che } 0 \leq f(n) \leq Cg(n) \text{ per ogni } n \geq N\}$



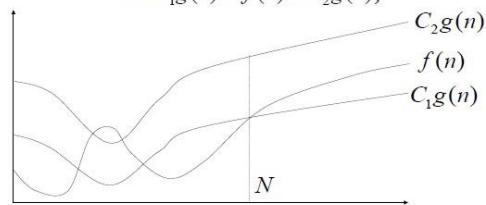
### Notazione asintotica Ω: limite inferiore (1)

$\Omega(g(n)) = \{f(n) : \text{esistono } C > 0 \text{ ed } N \text{ tali che } f(n) \geq Cg(n) \geq 0 \text{ per ogni } n \geq N\}$



### Notazione asintotica Θ : limite stretto (1)

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) = \{f(n) : \text{esistono } C_1, C_2 > 0 \text{ ed } N \text{ tali che per ogni } n \geq N \ 0 \leq C_1g(n) \leq f(n) \leq C_2g(n)\}$



Se  $f(n) = O(g(n))$  è il tempo calcolo richiesto da un algoritmo  $A$ ,  $O(g(n))$  è un **limite superiore asintotico** per la complessità in tempo di  $A$ .

Se  $f(n) = \Omega(g(n))$  è il tempo calcolo richiesto da un algoritmo  $A$ ,  $\Omega(g(n))$  è un **limite inferiore asintotico** per la complessità in tempo di  $A$ .

### Notazione asintotica o

- Usata per denotare il **limite superiore non asintoticamente stretto**.
  - $o(g(n)) = \{f(n) : \text{per qualsiasi costante } c > 0, \text{ esiste una costante } n_0 > 0 \text{ tale che } 0 \leq f(n) < cg(n) \text{ per ogni } n \geq n_0\}$
- $f(n) = o(g(n))$  si legge  $f(n)$  è "o piccolo di g di n"
  - Esempio:  $2n = o(n^2)$  e  $2n^2 \neq o(n^2)$
- Nella notazione o la funzione  $f(n)$  diventa insignificante rispetto a  $g(n)$  quando  $n$  tende all'infinito:

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

### Notazione asintotica ω

- Usata per denotare il **limite inferiore non asintoticamente stretto**.
  - $\omega(g(n)) = \{f(n) : \text{per qualsiasi costante } c > 0, \text{ esiste una costante } n_0 > 0 \text{ tale che } 0 \leq cg(n) < f(n) \text{ per ogni } n \geq n_0\}$
- $f(n) = \omega(g(n))$  si legge  $f(n)$  è "omega piccolo di g di n"
  - Esempio:  $n^2/2 = \omega(n)$  e  $n^2/2 \neq \omega(n^2)$
- Nella notazione ω la funzione  $g(n)$  diventa insignificante rispetto a  $f(n)$  quando  $n$  tende all'infinito:

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

Ho un insieme valori  $S$  di cardinalità  $n$  e devo determinare se un certo valore  $x$  appartiene all'insieme.

Qual è il costo della operazione (caso peggiore) nel caso di implementazione dell'insieme con:

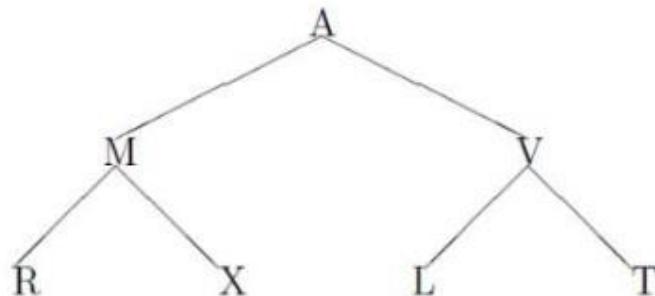
- (a) **Lista linkata;**
- (b) **Albero binario;**
- (c) **Albero binario di ricerca;**
- (d) **Albero binario di ricerca bilanciato.**

Definire l'attraversamento

- **Inorder**
- **Preorder**
- **Postorder**

In un albero binario e scrivere le sequenze risultanti dell'attraversamento del seguente albero.

- (a) Il caso peggiore si ha quando tutte le chiavi collidono e la tabella consiste in una unica lista di lunghezza  $n$ . In questo caso il tempo di calcolo è  $O(n)$
  - (b)  $h=O(\log(n))$  dove  $n$  è la dimensione dell'Input.
  - (c)  $h=O(n)$
  - (d)  $h=O(\log n)$
- Visita in **preordine**: si visita prima la radice e poi si effettuano le chiamate ricorsive sul figlio sinistro e destro (radice + sottoalberi da sinistra a destra)
  - Visita **inordine (simmetrica)**: si effettua prima la chiamata ricorsiva sul figlio sinistro, poi si visita la radice ed, infine, si effettua la chiamata ricorsiva sul figlio destro (non definita per alberi generici)
  - Visita in **postordine**: si effettuano prima le chiamate ricorsive sul figlio sinistro e destro e poi si visita la radice (visito i sottoalberi da sinistra a destra e poi radice)



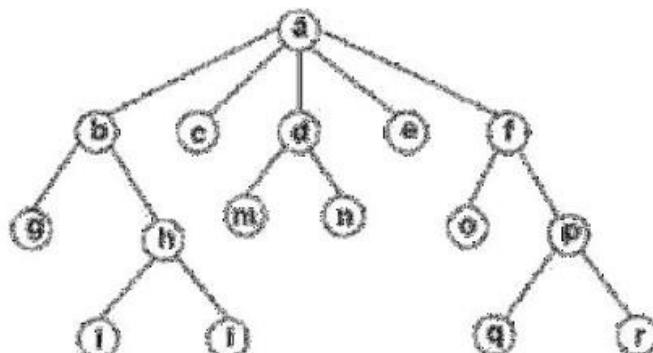
A-M-R-X-V-L-T

R-M-X-A-L-V-T

R-X-M-L-T-V-A

Elencare la sequenza di visita dei nodi del seguente albero, supponendo di visitare l'albero secondo l'ordine di visita preordine, postordine e inordine.

a-b-g-h-i-l-c-d-m-n-e-f-o-p-q-r  
g-i-l-h-b-c-m-n-d-e-o-q-r-p-f-a  
g-b-i-h-l-a-c-m-d-n-e-o-f-q-p-r



## ADT stack e code: definizioni, operazioni ed esempi di applicazione. Spiegare

i concetti di:

- (a) pila;
- (b) coda;
- (c) coda di priorità
- (d) coda circolare
- (e) procedura ricorsiva

- (a) Il termine stack o pila, in informatica, indica un tipo di dato astratto che viene usato in diversi contesti per riferirsi a strutture dati, le cui modalità d'accesso ai dati in essa contenuti seguono una modalità *LIFO* (Last In First Out), ovvero tale per cui i dati vengono estratti (letti) in ordine rigorosamente inverso rispetto a quello in cui sono stati inseriti (scritti).
- (b) In informatica per **coda** si intende una struttura dati di tipo *FIFO*, **First In First Out** (il primo in ingresso è il primo ad uscire). Un esempio pratico sono le code che si fanno per ottenere un servizio, come pagare al supermercato o farsi tagliare i capelli dal parrucchiere: idealmente si viene serviti nello stesso ordine con cui ci si è presentati.
- (c) Nella teoria delle code, una **coda di priorità** è una struttura dati astratta, simile ad una coda o ad una pila, ma si differisce da queste in quanto ogni elemento inserito all'interno della coda possiede una sua "priorità". In una coda di priorità, ogni elemento avente priorità più alta, viene inserito prima rispetto ad un elemento avente priorità più bassa. In particolare, l'elemento con priorità più alta si trova in testa alla coda, quello con priorità più bassa si troverà, appunto, in coda.
- (d) La coda circolare è una coda provvista di sentinella, ossia un elemento fittizio che ha il puntatore (nell'ultimo elemento) che punta al primo, quindi la coda continua a ripetersi.
- (e) Nella logica matematica e nell'informatica, le **funzioni ricorsive** sono una classe di funzioni dai numeri naturali ai numeri naturali che sono "calcolabili" in un qualche senso intuitivo. Infatti nella teoria della calcolabilità si mostra che le funzioni ricorsive corrispondono precisamente a quelle funzioni che possono essere calcolate tramite una macchina di Turing.

**Nell'ambito della struttura dati lista, perché è vantaggioso utilizzare la sentinella? Giustificare la risposta mostrando anche un esempio che illustri chiaramente i vantaggi derivanti dall'uso della sentinella.**

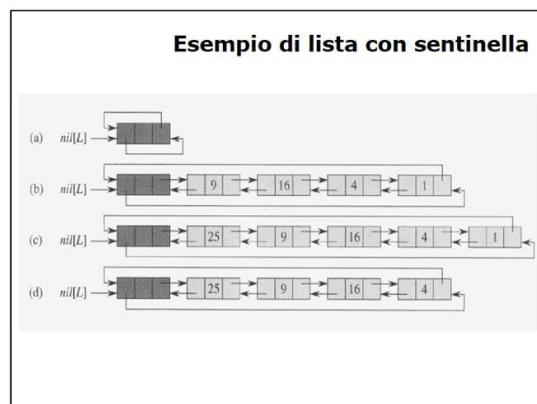
La sentinella è un elemento fittizio che consente di semplificare le condizioni al contorno. Nella lista è posta fra testa e coda e aiuta nelle operazioni di modifica, inserimento e cancellazione.

Esempio:

Supponiamo di fornire alla lista  $L$  un oggetto  $\text{nil}[L]$  che rappresenta  $\text{NIL}$ . i riferimenti a  $\text{NIL}$  nel codice si sostituiscono con il riferimento alla sentinella  $\text{nil}[L]$ . la lista circolare doppiamente concatenata diventa una **lista circolare doppiamente concatenata con sentinella**.

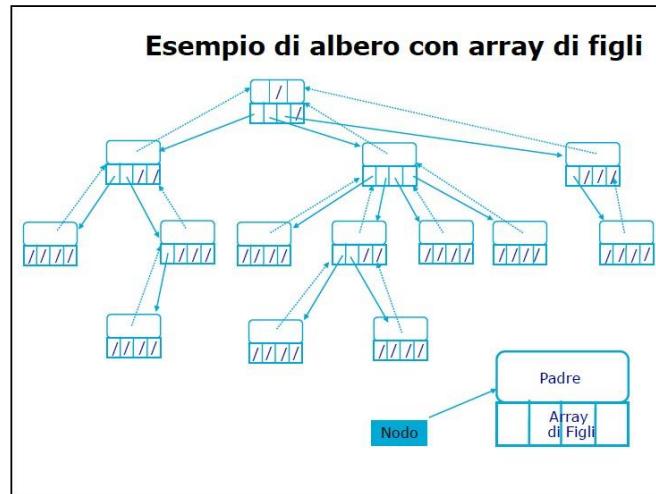
La sentinella è posta fra testa e coda. Il campo  $\text{next}[\text{nil}[L]]$  punta alla testa della lista. Il campo  $\text{prev}[\text{nil}[L]]$  punta alla coda. Il campo  $\text{next}$  della coda ed il campo  $\text{prev}$  della testa puntano entrambi a  $\text{nil}[L]$ .

$\text{Next}[\text{nil}[L]]$  pu essere usato invece di  $\text{head}[L]$ . Una lista vuota contiene solo la sentinella con  $\text{next}[\text{nil}[L]]$  e  $\text{prev}[\text{nil}[L]]$  impostati a  $\text{nil}[L]$ .



## Descrivere come possono essere rappresentati alberi in cui ogni nodo ha un numero arbitrario di figli.

In un albero binario ogni elemento ha 3 campi, che sono "P" [memorizza il puntatore al padre], "left" [memorizza il puntatore al figlio sinistro] e "right" [memorizza il puntatore al figlio destro]. Possiamo avere anche "root" [che punta alla radice dell'albero T]. Se un albero ha un numero arbitrario di figli, allora si estende lo schema in modo da avere al più k figli. Al posto di "left e right" abbiamo child1, child2, etc.. Ogni nodo x ha 2 puntatori che diventano "**left-child [x]**", che punta al figlio più a sinistra di x, e "**right-sibling[x]**", che punta al fratello di x immediatamente a destra.



## Definire la struttura dati lista e descrivere l'implementazione basata su puntatori e cursori, evidenziando le equivalenze tra essi.

Una lista è un insieme di oggetti posti in ordine lineare. L'ordine è determinato da un puntatore in ogni oggetto. Una lista doppiamente concatenata è un oggetto con un campo:

- chiave **key**

- puntatore **next** al prossimo elemento
- puntatore **prev** all'elemento precedente

Una lista può essere:

- **singolarmente concatenata** (manca prev)
- **doppiamente concatenata**
- **ordinata** (la testa è l'elemento minimo e la coda l'elemento massimo)
- **non ordinata**
- **circolare** (il puntatore prev della testa punta alla coda ed il puntatore next della coda punta alla testa).

Le operazioni da implementare sono:

- **List-search(L,k)**: restituisce un puntatore al primo elemento con valore k
- **List-insert(L,x)**: inserisce elemento x davanti alla lista concatenata
- **List-delete(L,x)**: rimuove un elemento x da L

È possibile realizzare una lista in due modi:

Realizzazione con Puntatori: Ogni elemento della lista L è un oggetto con un campo chiave Key e un puntatore Next al prossimo elemento. Esistono diversi tipi di liste: Singolarmente Concatenata (ha i campi Key e Next), Doppiamente Concatenata (campi Key, Prev e Next), Ordinata (l'elemento minimo è la testa e quello massimo in coda), Non Ordinata (elementi presentati in qualsiasi ordine), Circolare (il puntatore Prev delle testa punta alla coda, l'elemento Next della coda punta alla testa). Inoltre è possibile utilizzare una Sentinella per rendere il Delete più semplice.

Realizzazione con Cursori: È possibile realizzare puntatori con cursori, cioè non variabili interne, il cui valore è interpretato come un indice di un vettore. Questo vettore simula la memoria disponibile per i puntatori, che deve essere gestita esplicitamente nella realizzazione. Per far questo si deve definire un vettore SPAZIO che contiene tutte le liste, ognuna individuata dal proprio cursore iniziale; e contiene tutte le celle libere, organizzate anch'esse in una lista detta "listalibera". Lo SPAZIO sarà generalmente diviso in tre campi precedente, elemento e successivo.

Otteniamo quindi un equivalenza tra puntatori e cursori, ovvero p.next equivale a SPAZIO[p].next e p.prev equivale a SPAZIO[p].prev.

## In che cosa consiste l'approccio "divide et impera"? Nominare un algoritmo che se ne serve.

L'algoritmo che se ne serve è il quicksort, e la tecnica consiste nel prevedere tre passi ad ogni livello di ricorsione:

- **Divide** : il problema viene diviso in un certo numero di sottoproblemi
- **Impera** : i sottoproblemi vengono risolti in modo ricorsivo (se i problemi sono piccoli si possono risolvere in maniera semplice)
- **Combina** : le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale.

## Si descriva in modo sintetico e preciso cosa è uno heap e come può essere mantenuto attraverso un vettore.

In informatica, un heap è una struttura dati basata sugli alberi che soddisfa la "proprietà di heap": se A è un genitore di B, allora la chiave di A è ordinata rispetto alla chiave di B conformemente alla relazione d'ordine applicata all'intero heap. Di conseguenza, gli heap possono essere suddivisi in "max heap" e "min heap". Gli heap sono essenziali negli algoritmi della teoria dei grafi (come l'algoritmo di Dijkstra) o negli algoritmi di ordinamento come l'heapsort. Un'implementazione molto comune di un heap è l'heap binario, basato su un albero binario completo (come quello in figura). L'heap è, inoltre, una delle implementazioni più efficienti di un tipo di dato astratto chiamato coda di priorità. Si può mantenere uno heap attraverso un vettore utilizzando un Max-Heapify, ossia il più piccolo dei figli deve scendere in modo che il sottoalbero diventi un Max Heap.

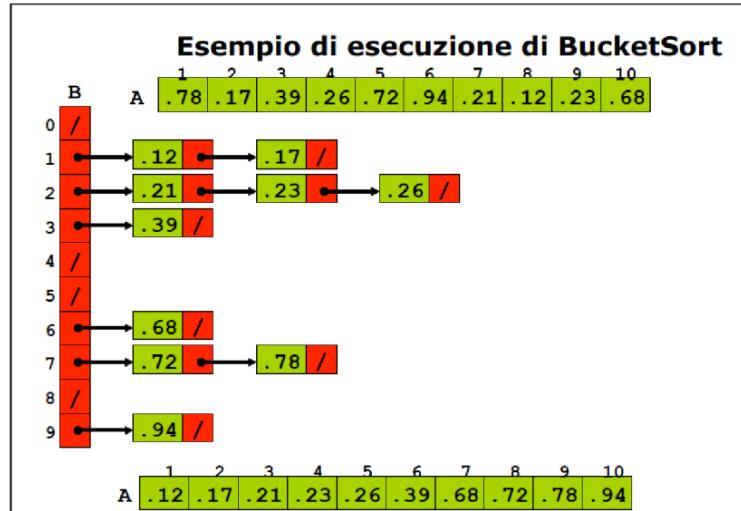
**Mostrare una sequenza di 10 numeri interi che determinano il caso peggiore per l'algoritmo QuickSort e mostrare una sequenza della stessa lunghezza che ne determina il caso migliore. Si richiede di discutere il motivo per cui le sequenze date determinano rispettivamente il caso peggiore e migliore.**

Il caso migliore si verifica quando l'algoritmo di partizionamento determina due sottoproblemi perfettamente bilanciati, entrambi di dimensione  $n/2$ ; in questo caso il tempo di esecuzione è  $\Theta(n \log n)$ . Il caso peggiore si verifica quando lo sbilanciamento è totale, cioè quando l'algoritmo di partizionamento restituisce una partizione di lunghezza  $n-1$  e una di lunghezza 0; in questo caso il tempo di esecuzione è  $\Theta(n^2)$ .

Per quanto riguarda gli esempi, basta prendere 10 numeri, mettere gli indici  $i, j$ , e nel caso migliore facendo scorrere gli indici, tutto sarà bilanciato, mentre nel caso peggiore, ad ogni passata, bisognerà sostituire il numero del puntatore  $i$  con il numero del puntatore  $j$ .

**Descrivere l'algoritmo di ordinamento BucketSort. Quale è la complessità sia nel caso peggiore sia nel caso migliore?**

Assume che i valori da ordinare siano numeri reali in un intervallo semiaperto  $[a,b)$  che per semplicità di esposizione assumiamo sia l'intervallo  $[0,1)$ . Per ordinare un array  $A[1..n]$  divide l'intervallo in  $n$  parti uguali e usa un array  $B[0..n-1]$  di liste (i bucket) mettendo in  $B[k]$  gli  $A[i]$  che cadono nella  $k$ -esima parte dell'intervallo. Dopo di che riordina ciascuna lista e infine ricopia in  $A$  tutti gli elementi delle liste. Nel caso peggiore in cui tutti gli elementi vanno a finire in un'unica lista abbiamo:  $O(n^2)$  Nel caso migliore in cui gli elementi si distribuiscono uno per lista abbiamo:  $O(n)$



Si supponga di avere una variante dell'algoritmo MergeSort che suddivide l'array da ordinare in  $k$  parti, le ordina applicando ricorsivamente questa variante del MergeSort e unifica le  $k$  parti tramite una variante del Merge. Qual è la complessità di tale variante? Si richiede di fornire una risposta dettagliata e precisa.

Concettualmente, l'algoritmo funziona nel seguente modo:

- 1 ) Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti:
  - 2 ) La sequenza viene divisa (*divide*) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda)
  - 3 ) Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo (*impera*) Le due sottosequenze ordinate vengono fuse (*combina*). Per fare questo, si estraе ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata.
- La complessità è  $O(N)$ : ad ogni selezione il vettore complessivo cresce di 1 e c'è un costo iniziale di copia del vettore appoggio ( $O(N)$ ). Ovviamente noi dobbiamo sempre tenere conto della grandezza delle  $K$  parti, che andranno a differire a seconda della grandezza del nostro array.

**Descrivere l'algoritmo di ordinamento counting sort. Qual è la sua complessità?**

L'algoritmo conta il numero di occorrenze di ciascun valore presente nell'[array](#) da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervallo di valori. Il numero di ripetizioni dei valori inferiori indica la posizione del valore immediatamente successivo. Si calcolano i valori massimo,  $\text{Max}(A)$ , e minimo,  $\text{Min}(A)$ , dell'array e si prepara un array ausiliario  $C$  di dimensione pari all'intervallo dei valori con  $C[i]$  che rappresenta la frequenza dell'elemento  $i + \text{Min}(A)$  nell'array di partenza  $A$ . Si visita l'array  $A$  aumentando l'elemento di  $C$  corrispondente. Dopo si visita l'array  $C$  in ordine e si scrivono su  $A$ ,  $C[i]$  copie del valore  $i + \text{Min}(A)$ .

L'algoritmo esegue tre [iterazioni](#), due di lunghezza  $n$  (pari alla lunghezza dell'array da ordinare) per l'individuazione di  $\text{Max}(A)$  e  $\text{Min}(A)$  e per il calcolo delle occorrenze dei valori, e una di lunghezza  $k$  (pari a  $\text{Max}(A) - \text{Min}(A) + 1$ ) per l'impostazione delle posizioni finali dei valori: la [complessità](#) totale è quindi  $O(n+k)$ .

**Illustrare il funzionamento degli algoritmi: counting sort e radix sort. Quale complessità hanno?**

Input Data	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>4</td><td>2</td><td>2</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>2</td><td>4</td><td>2</td></tr></table>	0	4	2	2	0	0	1	1	0	1	0	2	4	2																					
0	4	2	2	0	0	1	1	0	1	0	2	4	2																							
Count Array	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>3</td><td>4</td><td>0</td><td>2</td></tr></table>	0	1	2	3	4	5	3	4	0	2																									
0	1	2	3	4																																
5	3	4	0	2																																
Sorted Data	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td><td>2</td><td>4</td><td>4</td></tr></table>	0	0	0	0	0	1	1	1	2	2	2	2	4	4																					
0	0	0	0	0	1	1	1	2	2	2	2	4	4																							
Radix sort	Counting sort $\rightarrow$ complessità = $O(n+k)$																																			
<table border="1" style="display: inline-table; vertical-align: middle;"><tr><th>1<sup>o</sup> PASSO</th><th>2<sup>o</sup> PASSO</th><th>3<sup>o</sup> PASSO</th><th>4<sup>o</sup> PASSO</th><th></th></tr><tr><td>253</td><td>10</td><td>5</td><td>5</td><td>5</td></tr><tr><td>346</td><td>253</td><td>10</td><td>10</td><td>10</td></tr><tr><td>1034</td><td>1034</td><td>127</td><td>1034</td><td>127</td></tr><tr><td>10</td><td>5</td><td>1034</td><td>127</td><td>253</td></tr><tr><td>5</td><td>346</td><td>346</td><td>253</td><td>346</td></tr><tr><td>127</td><td>127</td><td>253</td><td>346</td><td>1034</td></tr></table>	1 <sup>o</sup> PASSO	2 <sup>o</sup> PASSO	3 <sup>o</sup> PASSO	4 <sup>o</sup> PASSO		253	10	5	5	5	346	253	10	10	10	1034	1034	127	1034	127	10	5	1034	127	253	5	346	346	253	346	127	127	253	346	1034	$\rightarrow$ complessità = $O(nk)$
1 <sup>o</sup> PASSO	2 <sup>o</sup> PASSO	3 <sup>o</sup> PASSO	4 <sup>o</sup> PASSO																																	
253	10	5	5	5																																
346	253	10	10	10																																
1034	1034	127	1034	127																																
10	5	1034	127	253																																
5	346	346	253	346																																
127	127	253	346	1034																																

## Caratterizzare il problema del dizionario e discutere la applicazione di tecniche di hash per la sua soluzione.

Un dizionario è una struttura dati **ASTRATTA** utilizzata per mantenere un insieme di  $S$  elementi, ognuno con un valore associato chiamato **CHIAVE**, su cui vengono supportate le seguenti operazioni:

1) **INSERT (S,x)** => inserisce l'elemento  $x$  nell'insieme  $S$ . 2)

**DELETE (S,x)** => cancella l'elemento  $x$  dall'insieme  $S$ . 3)

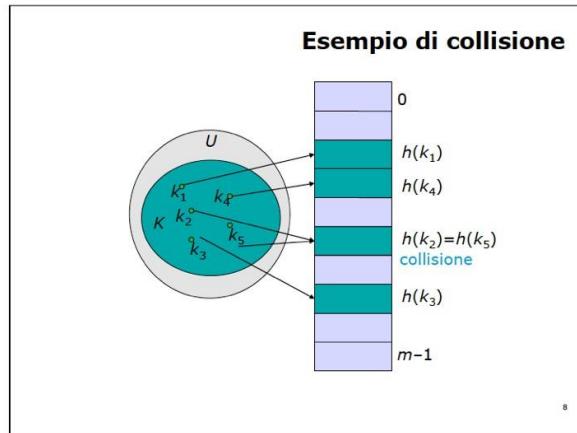
**SEARCH (S,x)** => trova l'elemento  $x$  nell'insieme  $S$ .

## Discutere il problema delle collisioni e illustrare possibili tecniche per la loro gestione.

Siccome  $|U| > m$  esisteranno molte coppie di chiavi distinte  $k_1 \neq k_2$  tali che  $h(k_1) = h(k_2)$ , diremo in questo caso che vi è una **collisione** tra le due chiavi  $k_1$  e  $k_2$ . Nessuna funzione hash pu' evitare le collisioni. Le funzioni hash devono minimizzare la probabilità delle collisioni e prevedere un meccanismo per gestire le collisioni.

Per risolvere il problema delle collisioni:

- Gli elementi con lo stesso valore hash  $h$  vengono memorizzati in una lista concatenata.
- Si memorizza un puntatore alla testa della lista nello slot  $A[h]$  della tabella hash.
- Operazioni:
- **insert**: in testa
- **search, delete**: richiedono di scandire la lista alla ricerca della chiave.



Si consideri una tabella di hash con  $m$  celle,  $n$  chiavi memorizzate e dove le collisioni sono gestite con liste. Qual è la complessità delle operazioni di inserimento, cancellazione e ricerca? Come varia la complessità delle operazioni di inserimento, cancellazione e ricerca se le liste vengono mantenute ordinate?

	Array non ordinato	Array ordinato	Lista	Alberi (abr, rb, ...)	Performance ideale
insert()	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
lookup()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
remove()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$

## Discutere la differenza tra hashing interno ed esterno ed i criteri che si conoscono per la gestione delle collisioni (ispezione lineare, ispezione quadratica, hashing doppio).

La realizzazione di un dizionario con tabelle hash si basa sul concetto di ricavare direttamente dal valore della chiave la posizione della chiave stessa. Oltre ad una buona funzione hash  $H(k)$  che mi restituirà il valore della posizione della chiave nel dizionario, occorre un metodo di scansione che permetta di posizionare e reperire le chiavi che hanno trovato la propria posizione occupata (le collisioni inevitabili). I metodi di scansione si distinguono in esterni e interni, a seconda che le chiavi siano memorizzate all'esterno o all'interno del vettore  $D$  contenente il dizionario.

**Metodi esterni:** liste di trabocco

**Metodi interni:** scansione: lineare, quadratica, pseudocasuale, hashing doppio

In generale i metodi esterni sono più efficienti di quelli interni.

**Scansione esterna, liste di trabocco**

La lista di trabocco è un metodo di scansione interno. L'idea è quella di associare ad ogni posizione  $H(k)$  del dizionario una lista  $V[H(k)]$ , detta appunto di trabocco. Le liste di trabocco hanno un duplice vantaggio, ovvero nessun limite alla capacità del dizionario ed evitare completamente degli agglomerati di chiavi.

**Scansione interna**

Scansione interna significa, in caso di collisione, scandire le posizioni libere in  $D$  per trovarne una libera per la chiave  $k$ . Supponiamo che sia  $F_i$  la funzione che permette di trovare, con l' $i$ -esima applicazione, l' $i$ -esima posizione occupata partendo da  $H(k)$ . In sostanza  $i$  sta ad indicare il numero di volte che si verifica una collisione, per uno stesso indirizzo del vettore  $D$ , ottenuto dalla funzione hash.

**Scansione lineare:**  $F_i = (H(k) + h i) \bmod m$ , dove  $h$  è primo con  $m$ , ed indica la distanza tra due successive posizioni esaminate nella scansione. Se  $h=1$  si ottiene la scansione a passo unitario. La divisione in modulo rende la scansione circolare;

**Scansione quadratica:**  $F_i = (H(k) + h i + i(i - 1)/2) \bmod m$ , con  $m$  primo;

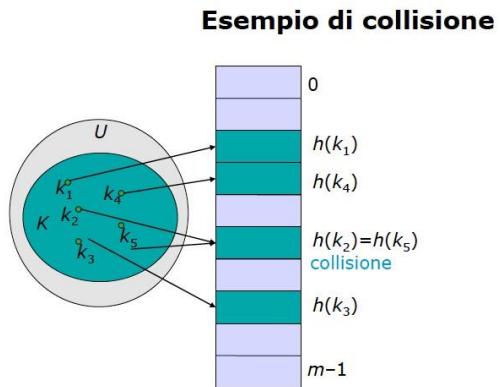
**Scansione pseudocasuale:**  $F_i = (H(k) + r_i) \bmod m$ , con  $r_i$  generato in modo pseudocasuale tra 0 e  $m - 1$ ;

**Scansione hashing doppio:**  $F_i = (H(k) + i F(k)) \bmod m$ , con  $F(k)$  altra funzione hash diversa da  $H(k)$ .

## Dire cosa sono le tabelle di hash e descrivere quando si verificano delle collisioni e quali meccanismi si possono prevedere per la loro gestione.

Per quanto concerne i meccanismi di collisione, vedi domande precedenti, mentre per la definizione di tabella di hash sappiamo che in informatica una hash table, in italiano tabella hash è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore. Viene usata per l'implementazione di strutture dati astratte associative come Map o Set. L'hash table è molto utilizzata nei metodi di ricerca nominati Hashing. L'hashing è un'estensione della ricerca indicizzata da chiavi che gestisce problemi di ricerca nei quali le chiavi di ricerca non presentano queste proprietà. Una ricerca basata su hashing è completamente diversa da una basata su confronti: invece di muoversi nella struttura data in funzione dell'esito dei confronti tra chiavi, si cerca di accedere agli elementi nella tabella in modo diretto tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella.

Si richiede di fornire un esempio di funzione di hash e di illustrane il funzionamento. Cosa succede in caso di collisioni?



Nell'ambito delle tecniche di hashing, dire cosa si intende per ispezione lineare e quadratica. Si richiede di fare un esempio.

**ISPEZIONE LINEARE** = La funzione hash  $h(k,i)$  si ottiene da una funzione hash ordinaria  $h'(k)$  ponendo  $h(k,i) = (h'(k) + i) \bmod m$ . L'esplorazione inizia dalla cella  $h(k,0) = h'(k)$  e continua con le celle  $h'(k)+1, h'(k)+2\dots$  fino ad arrivare alla cella  $m-1$ , dopo di che si continua con le celle  $0, 1, \dots$  fino ad aver percorso circolarmente tutta la tabella. I vantaggi sono che è facile da realizzare, mentre gli svantaggi sono che la prima cella ispezionata determina l'intera sequenza e quindi ci sono solo  $m$  sequenze di ispezione distinte; presenta un problema noto come **addensamento primario**: • si formano lunghe file di celle occupate che aumentano il tempo medio di ricerca

- uno slot vuoto preceduto da  $i$  slot pieni ha probabilità  $(i+1)/m$  di essere riempito.

**ISPEZIONE QUADRATICA** = La funzione hash  $h(k,i)$  si ottiene da una funzione hash ordinaria  $h'(k)$  ponendo  $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$  dove  $c_1$  e  $c_2$  sono due costanti con  $c_2 \neq 0$ . I valori di  $m$ ,  $c_1$  e  $c_2$  non possono essere qualsiasi ma debbono essere scelti opportunamente in modo che la sequenza di ispezione percorra tutta la tavola. Se per due chiavi  $k$  e  $l$ ,  $h'(k) = h'(l)$  allora le due sequenze di ispezione coincidono. Questo porta ad un **addensamento secondario** meno grave dell'addensamento primario. L'addensamento secondario è dovuto al fatto che il valore iniziale  $h'(k)$  determina univocamente la sequenza di ispezione e pertanto abbiamo soltanto  $m$  sequenze di ispezione distinte.

**Cosa si intende per hashing doppio?**

Per hashing doppio intendiamo che se facendo l'hash di una chiave si incontra una collisione, allora si somma all'indice ottenuto il risultato di una nuova funzione hash (generalmente diversa dalla prima e che ha come parametro l'indice ottenuto precedentemente), e si tenta l'inserimento nel nuovo indice così ottenuto, riapplicando la seconda funzione sino a che non si trovi una casella libera. Si hanno  $\Theta(m^2)$  sequenze di ispezione distinte, e questo riduce notevolmente i fenomeni di addensamento e rende il comportamento della funzione hash molto vicino a quello ideale dell'hash uniforme.

**Quante foglie ha un albero binario completo di altezza  $h$ ?**

Il numero di foglie è pari al numero di nodi  $-1$ , quindi  $2^{h+1}$

**Quanti nodi sono contenuti in un albero binario completo di altezza  $h$ ?**

Vanno da un minimo di  $2^h$  nodi, a un massimo di  $2^{h+1}-1$  nodi.

**Dare la definizione di altezza e profondità di un nodo in un albero. Illustrare inoltre un esempio di albero specificando l'altezza e la profondità dei suoi nodi.**

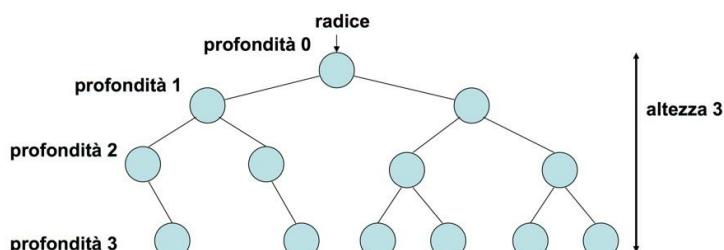
Per la definizione di **altezza**, vedi domanda precedente, mentre per la **profondità** sappiamo che  $h=\text{profondità del più lungo cammino}$ .

Esempio (figura a lato):

Altezza: 3

Profondità: 3

- In un albero binario **la profondità** di un nodo è la lunghezza del percorso dalla radice al nodo (cioè il numero di archi tra la radice e il nodo).
- La profondità maggiore di un nodo all'interno di un albero è **l'altezza** dell'albero.

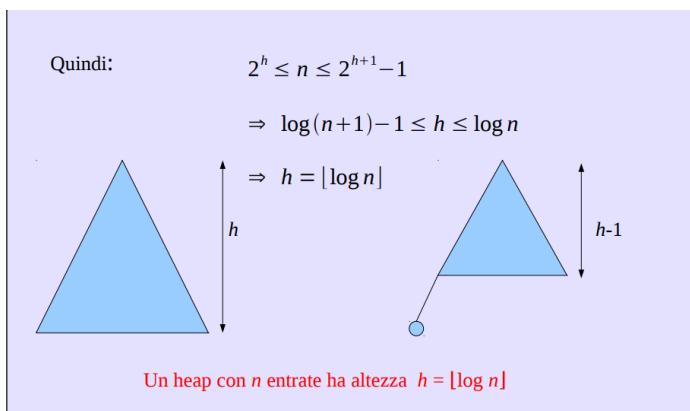


**Quali sono le operazioni che si possono effettuare su un albero binario di ricerca? Qual è la complessità di tali operazioni?**

Le operazioni che possono essere effettuate sono:

- **Inorder-tree-walk(x)**: restituisce l'elenco ordinato delle chiavi presenti nell'albero. Complessità  $T(n) = T(k)+d+T(n-k-1)$
- **Tree-search(x,k)**: restituisce il puntatore al nodo con chiave k (se esiste), oppure nil. Complessità:  $O(h)$
- **Tree-minimum(x)**: restituisce l'elemento con chiave minima. Complessità:  $O(h)$
- **Tree-maximum(x)**: restituisce l'elemento con chiave massima. Complessità:  $O(h)$
- **Tree-successor(x)**: restituisce il successore del nodo x. Complessità:  $O(h)$
- **Tree-predecessor(x)**: restituisce il predecessore del nodo x. Complessità:  $O(h)$
- **Tree-insert(T,z)**: inserisce il nuovo elemento z nell'albero binario T. Complessità:  $O(h)$
- **Tree-delete(T,z)**: cancella l'elemento z da T. Complessità:  $O(h)$

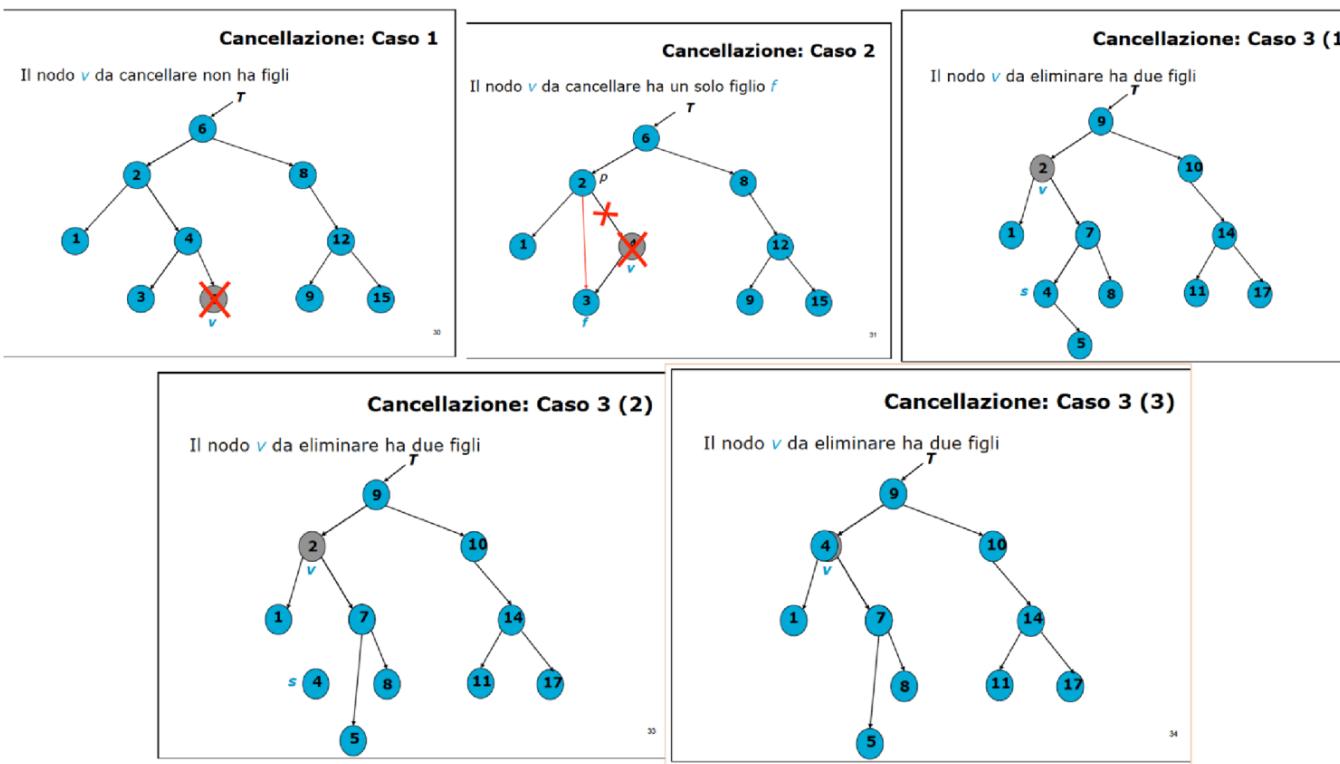
**Dato un albero binario con N nodi, quale può essere la sua altezza minima? e quella massima?**



**Dire cosa si intende per albero binario di ricerca**

Un albero binario di ricerca in contesto informatico, è un Albero binario in cui i valori dei figli di un nodo sono ordinati, usualmente avendo valori minori di quelli del nodo di partenza nei figli a sinistra e valori più grandi nei figli a destra.

Descrivere come viene eseguita l'operazione di cancellazione di un nodo all'interno di un albero binario di ricerca. La descrizione deve illustrare chiaramente tutti i casi che si possono presentare. Mostrare inoltre un esempio di cancellazione per ognuno dei casi descritti.



Scrivere (pseudo-codice) la procedura di ritrovamento del valore minimo in un albero binario di ricerca.

```
Tree-Minimum(x)
while left[x] ≠ nil do x
  ← left[x]
return x
```

Dire cosa rappresenta il fattore di bilanciamento di un nodo in un albero binario di ricerca.

Il fattore di bilanciamento  $\beta(v)$  di un nodo  $v$  è la massima differenza di altezza fra i sottoalberi di  $v$

Es: l'albero perfetto è un albero dove  $\beta(v)=0$  per ogni nodo  $v$

In questo caso si parla di bilanciamento in altezza.

Dire cosa si intende per fattore di bilanciamento di un nodo  $v$  ed illustrare (tramite un esempio) come viene preservato il bilanciamento negli alberi AVL.

Per la definizione vedi domanda precedente, mentre per il bilanciamento, vengono fatte le opportune rotazioni a Sinistra e Destra in modo che l'albero sia sempre bilanciato.

Dato un albero binario, cosa si intende per visita in preordine e postordine?

Visita in **preordine**: si visita prima la radice e poi si effettuano le chiamate ricorsive sul figlio sinistro e destro (radice + sottoalberi da sinistra a destra)

Visita in **postordine**: si effettuano prima le chiamate ricorsive sul figlio sinistro e destro e poi si visita la radice (visito i sottoalberi da sinistra a destra e poi radice)

## Cosa vuol dire albero bilanciato?

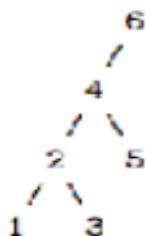
In informatica, un albero bilanciato è un albero binario di ricerca la cui altezza, grazie a particolari condizioni che la sua struttura deve soddisfare, rimane limitata. Queste condizioni implicano delle operazioni di inserimento ed eliminazione più complesse rispetto a quelle di semplici alberi binari, ma garantiscono che esse vengano eseguite in  $O(\log n)$ .

**Dato un albero binario di ricerca, l'operazione di cancellazione su tale albero è commutativa?**

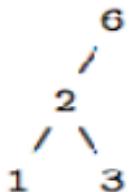
**Si richiede di giustificare la risposta mostrando anche un esempio.**

Per commutativa si intende che la cancellazione del valore  $x$  e poi del valore  $y$  produca lo stesso risultato che rimuovere prima il valore  $y$  e poi  $x$ . Negli ABR tale operazione NON è commutativa.

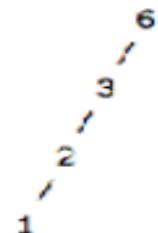
Esempio:



Rimuovendo prima il valore 5, poi il valore 4 si ottiene questo risultato:



Se invece rimuovo prima il valore 4, poi 5 ottengo questo risultato diverso:

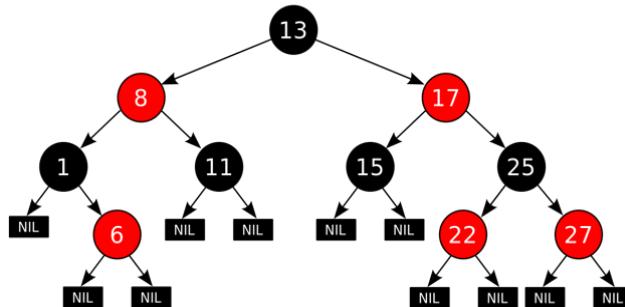


## Spiegare le proprietà principali degli alberi ROSSO-NERI.

1. La radice è nera;
2. I nodi nil sono neri;
3. Se un nodo è **rosso**, allora entrambi i suoi figli sono neri;
4. Ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri.

**Descrivere le proprietà che devono essere soddisfatte da un albero rosso-nero e fornire un esempio.**

Le proprietà sono quelle della domanda precedente. L'esempio nella figura a fianco.



**Spiegare le proprietà principali degli alberi rosso-neri e descrivere le operazioni di inserimento e le conseguenti operazioni per il mantenimento delle proprietà, mostrando degli esempi.**

Le proprietà sono le solite, mentre per le operazioni di inserimento:

Si Ricerca la posizione usando la stessa procedura usata per gli alberi binari di ricerca

- Coloriamo il nuovo nodo di rosso
- Se l'albero risultante non soddisfa le quattro proprietà che lo caratterizzano le si devono risistemare
- RB-Insert-Fixup ricolore i nodi ed effettua delle rotazioni.

Possono essere violate delle proprietà:

- La proprietà 2 (i nodi nil sono neri) è soddisfatta perché il nodo inserito è rosso ed i due figli sono la sentinella nil[T] che è nera.
- La proprietà 4 (ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri) è soddisfatta perché il nodo inserito sostituisce la sentinella (nera) ed esso è rosso con figli neri.
- Le proprietà che possono essere violate sono la 1 (la radice è nera) e la 3 (se un nodo è rosso entrambi i figli sono neri):
  - la 1 viene violata se il nodo inserito è la radice.
  - la 3 viene violata se il padre del nodo inserito è rosso.

Per mantenere tutto in modo corretto si provvede a ribilanciare/effettuare le opportune rotazioni.

**Dire cosa rappresenta il fattore di bilanciamento di un nodo in un albero binario di ricerca.**

**Descrivere inoltre le proprietà che devono essere soddisfatte da un albero rosso-nero e fornire un esempio di albero rosso-nero valido e un esempio di albero rosso-nero non valido.**

Il fattore di bilanciamento  $\beta(v)$  di un nodo  $v$  è la massima differenza di altezza fra i sottoalberi di  $v$

- es: l'albero perfetto è un albero dove  $\beta(v)=0$  per ogni nodo  $v$  In questo caso si parla di bilanciamento in altezza.

Le proprietà sono sempre le stesse delle domande precedenti, e se un albero rispetta TUTTE queste proprietà, viene definito valido, altrimenti, in violazione di anche solo una di queste proprietà, viene definito (l'albero) invalido.

**Dare la definizione di bilanciamento perfetto e bilanciamento AVL. Esistono implicazioni fra i due? Perchè si cerca di bilanciare gli alberi? Bilanciamento perfetto e bilanciamento AVL. Descrivere i concetti indicando se esistono implicazioni fra di essi fornendo anche esempi (o controesempi).**

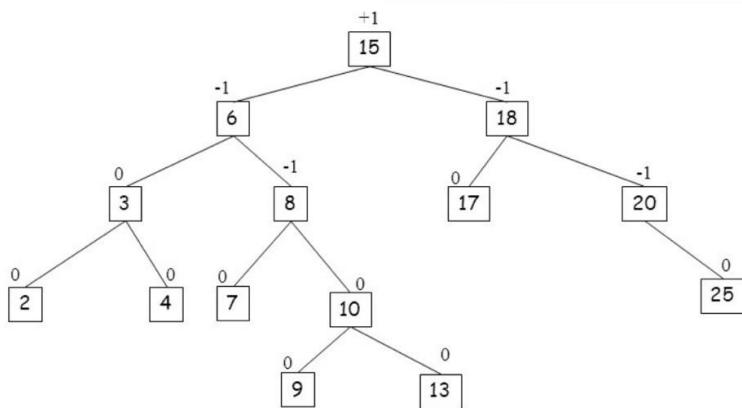
### Bilanciamento perfetto

• Un albero binario perfettamente bilanciato di  $n$  nodi ha altezza  $\lfloor \log n \rfloor + 1$ . Se ogni nodo ha 0 oppure 2 figli allora si ha  $nf = ni + 1$  che dove  $nf$  è il numero di foglie ed  $ni$  il numero di nodi interni. Le foglie sono circa il 50% dei nodi. Generalizzando, per alberi di aritÀ  $k$  si ha  $nf = (k-1)ni + 1$ . Il costo delle operazioni di ricerca/inserimento/cancellazione è pertanto  $O(\log n)$ . Ripetuti inserimenti e cancellazioni possono distruggere il bilanciamento con un conseguente degrado delle prestazioni.

### Ribilanciamento AVL

- Si ricalcolano i fattori di bilanciamento solo nel ramo interessato dall'inserimento/cancellazione (gli altri fattori non possono mutare), dal basso verso l'alto. Se appare un fattore di bilanciamento pari a  $+/ - 2$  occorre ribilanciare per mezzo di rotazioni. Si distinguono i seguenti casi:
  - **DD** inserimento nel sottoalbero destro di un figlio destro
  - **SD** inserimento nel sottoalbero destro di un figlio sinistro
  - **DS** inserimento nel sottoalbero sinistro di un figlio destro
  - **SS** inserimento nel sottoalbero sinistro di un figlio sinistro

**Dire cosa si intende per bilanciamento AVL e mostrare un esempio di albero AVL con il coefficiente di bilanciamento relativo a ciascun nodo.**



Per la definizione di bilanciamento AVL vedere domande precedenti.

**Si richiede di descrivere tutte le differenze tra B-alberi e B+-alberi.**

### • B+-Alberi:

I nodi foglia sono collegati tra loro in modo da formare una catena ordinata in base al valore della chiave. Supporta le interrogazioni basate su intervalli efficientemente. Sono molto usati nei DBMS relazionali.

### • B-Alberi:

Non c'è un collegamento tra i nodi foglia. I nodi intermedi usano due puntatori per ogni valore della chiave  $ki$ :

- 1) Un puntatore al blocco che contiene le tuple corrispondenti alla chiave  $ki$ .
- 2) Un puntatore al sottoalbero ch contiene le chiavi più grandi di  $ki$  e minori di  $ki+1$ .

**Descrivere il concetto di B-albero, indicandone le proprietà. Discutere inoltre le operazioni di aggiunta di elementi.**

Gli **B-Alberi**, sono delle [strutture di dati](#)/metodi che permettono la rapida localizzazione dei file ([Records](#) o [keys](#)), specie nei [database](#), riducendo il numero di volte che un utente necessita per accedere alla memoria in cui il dato è salvato. Essi derivano dagli [alberi di ricerca](#), in quanto ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto a ogni chiave appartenente ai sottoalberi alla sua destra; inoltre, la loro struttura ne garantisce il [bilanciamento](#): per ogni nodo, le altezze dei sottoalberi destro e sinistro differiscono al più di una unità. Questo è il vantaggio principale dei B-Tree, e permette di compiere operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati logaritmamente.

Operazione preliminare, che deve essere opportunamente implementata, per poter realizzare una funzione per l'inserimento di una chiave in un B-Albero è l'operazione di divisione di un nodo pieno. Un nodo di un B-Albero si definisce  *pieno* se contiene esattamente  $2t - 1$  chiavi: essendo pieno, in fase di inserimento di una chiave, essa non pu , per la definizione stessa di B-Albero, essere eventualmente inserita all'interno di esso. L'operazione di divisione viene effettuata in corrispondenza

della chiave mediana  $keyt[y]$  del nodo  $y$  pieno. Successivamente alla divisione, il nodo pieno  $y$  viene suddiviso in due nodi differenti ciascuno con  $t - 1$  chiavi. In concreto, la chiave mediana del nodo  $y$  viene spostata nel padre del nodo  $y$  (non pieno). L'operazione di divisione di un nodo, chiaramente, aumenta l'altezza dell'albero. L'operazione di inserimento di una chiave viene realizzata grazie ad una visita dell'albero che, sfruttando la procedura di splitting del nodo, evita che essa venga inserita in un nodo già pieno. Al primo passo della procedura di inserimento si verifica se la radice del B-Albero sia piena: in tal caso essa viene divisa all'altezza della chiave mediana; quest'ultima diverrà l'unica chiave di un nuovo nodo radice; a questo punto si pu effettuare la procedura vera e propria di inserimento mediante un'apposita funzione ricorsiva che si occupa di inserire la chiave nella posizione corretta. Nel caso in cui la radice dell'albero, invece, non sia piena si pu procedere direttamente con l'inserimento. A tal scopo si possono implementare due procedure: B-Tree-Insert che si occupa di verificare se la radice sia piena o meno e B-Tree-Insert-Nonfull che si occupa di effettuare la visita ricorsiva dell'albero per inserire la chiave nella corretta corrispondenza. Quest'ultima procedura viene invocata comunque dalla prima procedura, ma se la radice è piena viene preliminarmente effettuato il suo split.

**Quando può cambiare l'altezza di un B-albero? Motivare la risposta fornendo anche degli esempi.**

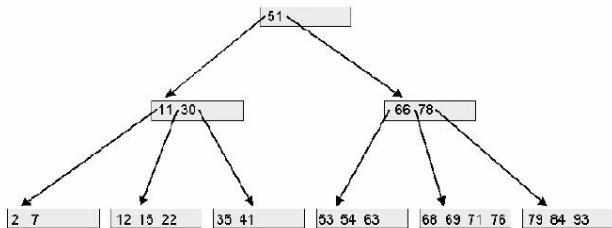
L'altezza di un B-Albero è il numero di nodi che compaiono in un cammino dalla radice ad un nodo foglia. È possibile stabilire con buona approssimazione l'altezza media di un B-Albero conoscendo il numero di chiavi presenti, potendo così stimare i costi di ricerca

Si denota con  $n_{min}$  e  $n_{max}$  il numero minimo e massimo, rispettivamente, di nodi che un B-Albero di altezza  $h$  può contenere.

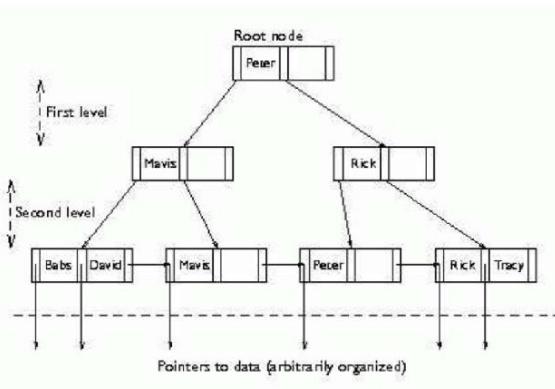
Si richiede di spiegare la seguente affermazione: Dal confronto tra i B-alberi ed i B+-alberi si evince che la ricerca di una singola chiave è in media più costosa in un B+-albero. Si richiede inoltre di mostrare un esempio sia di B-albero sia di B+-albero.

Il tempo di esecuzione dell'algoritmo è, banalmente,  $O(th) = O(t \log_t n)$ , in un B-albero, mentre in un B+-albero è SEMPRE necessario raggiungere un nodo foglia per ottenere il puntatore ai dati.

←Esempio B-albero:



Esempio B+-albero: →



Quali proprietà deve soddisfare un B-albero di ordine m?

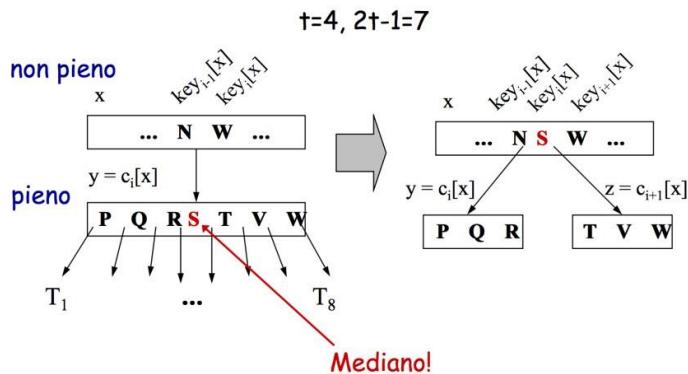
Un B-albero di ordine m deve soddisfare le seguenti proprietà:

- ogni nodo contiene al più  $m - 1$  elementi
- ogni nodo contiene almeno  $\lceil \frac{m}{2} \rceil - 1$  elementi, la radice può contenere anche un solo elemento.
- ogni nodo non foglia contenente  $j$  elementi ha  $j + 1$  figli
- ogni nodo ha una struttura del tipo:  $p_0(k_1, r_1)p_1(k_2, r_2)p_2 \dots p_{j-1}(k_j, r_j)p_j$  dove  $j$  è il numero degli elementi del nodo.
- $k_1 \dots k_j$  sono chiavi ordinate,  $k_1 < \dots < k_j$
- nel nodo sono presenti  $j + 1$  riferimenti ai nodi figli  $p_0 \dots p_j$  e  $j$  riferimenti ai file dati  $r_1 \dots r_j$
- per ogni nodo non foglia  $k(p_i)$  ( $i = 1, \dots, j$ ) è l'insieme delle chiavi memorizzate nel sottoalbero di radice  $p_i$  le quali soddisfano le seguenti relazioni:
  - $\&y' k(p_0), y < k_1$
  - $\&y' k(p_i), k_i < y < k_{i+1}, i = 1, \dots, j-1$
  - $\&y' k(p_j), y > k_j$

Dato un B-albero, quando si rende necessario applicare una operazione di split? Si richiede di mostrare un esempio.

Durante l'operazione di inserimento, se la radice risulta piena, viene preliminarmente effettuato uno split. Questa procedura evita che la chiave inserita, venga posizionata in un nodo già pieno. (Vedi figura a lato)

## Split di un nodo



Dato un B-albero di ordine  $m$  che contiene  $n$  valori, qual è il numero di accessi a disco nel caso peggiore e migliore (usare la notazione O-grande) quando si eseguono le seguenti operazioni?

(a) trovare un dato valore (b) inserire un dato valore (c) cancellare un dato valore.

Nel caso (a) = caso migliore  $\Rightarrow O(th) = O(t \log_t n)$ , caso peggiore  $\Rightarrow \Omega(h) = \Omega(\log_t n)$ .

Nel caso (b) = La complessità dell'algoritmo di inserimento in un B-Albero va valutata in funzione del

numero di accessi al disco sia per la lettura dei nodi che per la scrittura. Supponendo che l'altezza del B-Albero sia  $h$  la procedura B-Tree-Insert effettua  $O(h)$  accessi al disco. Il tempo di esecuzione è pari a  $O(th) = O(t \log_t n)$

**Nel caso (c) =** La complessità dell'algoritmo di cancellazione in termini di accesso al disco è pari a  $O(h)$ , mentre la complessità temporale è  $O(th) = O(t \log_t n)$ .

**Descrivere i possibili approcci all'implementazione dei grafi, fornendo esempi e discutendo vantaggi e svantaggi.**

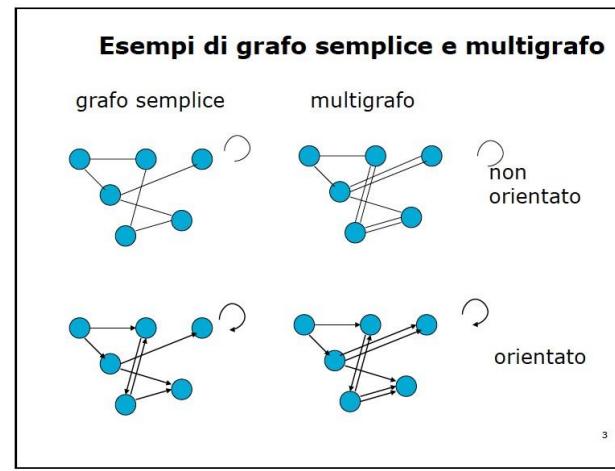
I grafi sono un potente strumento per la rappresentazione di problemi complessi. La soluzione di moltissimi problemi pu` essere ricondotta alla soluzione di opportuni problemi su grafi. Un grafo  $G = (V, E)$  è costituito da un insieme di vertici  $V$  ed un insieme di archi  $E$  ciascuno dei quali connette due vertici in  $V$  detti estremi dell'arco. Un grafo è orientato quando vi è un ordine tra i due estremi degli archi. In questo caso il primo estremo si dice coda ed il secondo testa. Un cappio è un arco i cui estremi coincidono. Un grafo non orientato è semplice se non ha cappi e non ci sono due archi con gli stessi estremi. Un grafo orientato è semplice se non ci sono due archi con gli stessi estremi. In caso contrario si parla di multigrafo.

Dare la definizione di sottografo indotto da un insieme di vertici.

Un sottografo del grafo  $G = (V, E)$  è un grafo  $G' = (V', E')$  tale che  $V' \subseteq V$  e  $E' \subseteq E$ . Il sottografo di  $G = (V, E)$  indotto da  $V' \subseteq V$  è il grafo  $G' = (V', E')$  tale che  $E' = \{uv : uv \in E \text{ e } u, v \in V'\}$ .

## Dire cosa si intende per multigrafo ed illustrare un esempio.

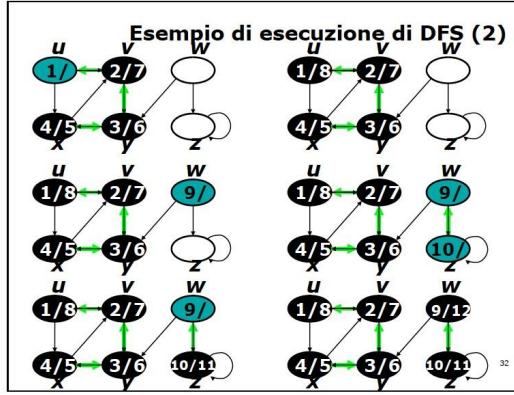
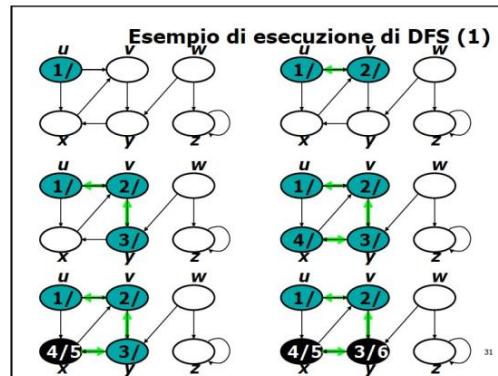
Un grafo orientato è semplice se non ci sono due archi con gli stessi estremi. In caso contrario si parla di multigrafo.



Descrivere le tecniche DFS e BFS per attraversare grafi e scrivere (pseudocodice) gli algoritmi per la BFS e la DFS. Che costo hanno? Indicare esempi di applicazione delle due tecniche di attraversamento per la risoluzione di problemi. **DFS(G)**

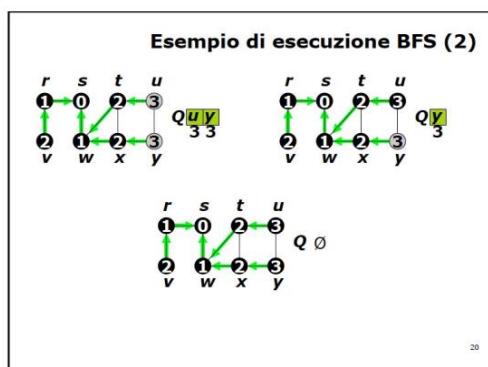
for "ogni  $v \in V[G]$ " do  $\text{color}[v] \leftarrow \text{bianco}$   $\pi[v] \leftarrow \text{nil}$   
 $t \leftarrow 0$  /\* variabile globale \*/ for "ogni  $v \in V[G]$ " do  
if  $\text{color}[v] = \text{bianco}$  then  
 $\text{DFS-Visit}(v)$   $\text{DFS-Visit}(u)$   $\text{color}[u] \leftarrow \text{grigio}$   $t \leftarrow t + 1$   
 $d[u] \leftarrow t$  for "ogni  $v \in \text{Adj}[u]$ " do if  $\text{color}[v] = \text{bianco}$  then  $\pi[v] \leftarrow u$   $\text{DFS-Visit}(v)$   $\text{color}[u] \leftarrow \text{nero}$   
 $f[u] \leftarrow t \leftarrow t + 1$

Poiché la somma delle lunghezze di tutte le liste delle adiacenze è  $\Theta(|E|)$  l'intero algoritmo ha complessità  $O(|V| + |E|)$ .

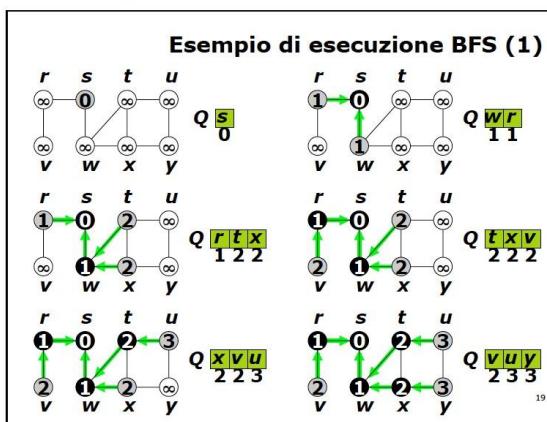


## BFS( $G, s$ )

for "ogni vertice  $v \in V[G]$ " do  $\text{color}[v] \leftarrow \text{bianco}$   
 $d[v] \leftarrow \infty$   $\pi[v] \leftarrow \text{nil}$   $\text{color}[s] \leftarrow \text{grigio}$   $d[s] \leftarrow 0$   
 $\pi[s] \leftarrow \text{nil}$   
 $Q \leftarrow \emptyset$



$\text{Enqueue}(Q, s)$  while not  $\text{Empty}(Q)$  do  
 $u \leftarrow \text{Dequeue}(Q)$   
for "ogni  $v \in \text{Adj}[u]$ " do if  $\text{color}[v] = \text{bianco}$  then  
 $\text{color}[v] \leftarrow \text{grigio}$ ,  $d[v] \leftarrow d[u] + 1$ ,  $\pi[v] \leftarrow u$   
 $\text{Enqueue}(Q, v)$   
 $\text{color}[u] \leftarrow \text{nero}$



La somma delle lunghezze di tutte le liste è  $O(m)$  quindi si conclude che la complessità è  $O(n+m)$ .

**Si descriva in breve il funzionamento dell'algoritmo di visita in ampiezza (breadth-first) di un grafo, indicando in particolare quale complessità temporale ha e di quale struttura dati conviene servirsi nell'eseguirlo.**

Dato un grafo  $G = (V, E)$  ed un vertice particolare  $s \in V$  detto sorgente, la visita in ampiezza (breadthfirst search) parte da  $s$  e visita sistematicamente il grafo per scoprire tutti i vertici che sono raggiungibili da  $s$ . Calcola la distanza di ogni vertice del grafo dalla sorgente  $s$  (lunghezza minima di un cammino dalla sorgente al vertice). Produce anche un albero BF (breadth-first tree) i cui rami sono cammini di lunghezza minima. La visita in ampiezza espande uniformemente la frontiera tra i vertici scoperti e quelli non ancora scoperti. Scopre tutti i vertici a distanza  $k$  da  $s$  prima di scoprire quelli a distanza  $k+1$ . Per tenere traccia del lavoro svolto, i vertici sono colorati di:

- **bianco** (vertici non ancora raggiunti)
- **grigio** (vertici raggiunti che stanno sulla frontiera)
- **nero** (vertici raggiunti che non stanno sulla frontiera)

I vertici adiacenti ad un vertice nero possono essere soltanto neri o grigi. I vertici adiacenti ad un vertice grigio possono essere anche bianchi. L'algoritmo costruisce un albero che all'inizio contiene soltanto la radice  $s$ . Quando viene scoperto un vertice bianco  $v$  a causa di un arco  $uv$  che lo connette ad un vertice  $u$  scoperto precedentemente, il vertice  $v$  e l'arco  $uv$  vengono aggiunti all'albero. Il vertice  $u$  viene detto padre di  $v$ .

**Dare la definizione di ordinamento topologico e descrivere un approccio per trovare un ordinamento topologico di un grafo orientato.**

Un ordinamento topologico di un grafo orientato aciclico (DAG)  $G = (V, E)$  è un ordinamento lineare dei suoi vertici tale che:

- per ogni arco  $uv \in E$  il vertice  $u$  precede il vertice  $v$ .
- per transitività, ne consegue che se  $v$  è raggiungibile da  $u$ , allora  $u$  compare prima di  $v$  nell'ordinamento.
- L'ordinamento topologico si usa per determinare un ordine in cui eseguire un insieme di attività in presenza di vincoli di precedenze.

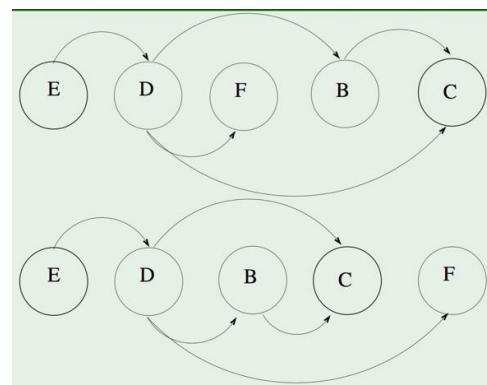
Il possibile approccio è:

**Soluzione diretta:**

- trovare ogni vertice che non ha alcun arco incidente in ingresso
- stampare tale vertice e rimuoverlo insieme ai suoi archi
- Ripetere la procedura finché tutti i vertici risultano rimossi
- **Soluzione basata su DFS**

**L'ordine topologico di un grafo è univoco? Motivare la risposta illustrando un esempio.**

L'ordinamento topologico non e' univoco. Esempio Illustrato →



**Definire cosa si intende per componente fortemente connessa.**

Un grafo orientato si dice fortemente connesso se esiste almeno un cammino da ogni vertice  $u$  ad ogni altro vertice  $v$ . Le componenti fortemente connesse di un grafo orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di mutua accessibilità. Una **componente fortemente connessa** di un grafo diretto  $G$  è un sottografo massimale di  $G$  in cui esiste un cammino orientato tra ogni coppia di nodi ad esso appartenenti. Le componenti fortemente connesse formano una partizione di  $G$  poiché un nodo non pu trovarsi contemporaneamente in due componenti fortemente connesse, di conseguenza un grafo diretto è fortemente connesso se e solo se ha una sola componente connessa. Due vertici di  $G$  sono fortemente connessi se e solo se fanno parte dello stesso ciclo orientato.

## Definire il concetto di componente fortemente connessa in un grafo orientato e illustrare un approccio per la sua risoluzione.

Le componenti fortemente connesse formano una partizione di  $G$  poiché un nodo non può trovarsi contemporaneamente in due componenti fortemente connesse, di conseguenza un grafo diretto è fortemente connesso se e solo se ha una sola componente connessa. Una componente fortemente connessa (ccf) di un grafo orientato  $G = (V, E)$  è un insieme massimale di vertici  $U \subseteq V$  tale che per ogni  $u, v \in U$  esiste un cammino da  $u$  a  $v$  ed un cammino da  $v$  ad  $u$ . Le componenti connesse sono calcolate in tre fasi:

- si usa la visita in profondità in  $G$  per ordinare i vertici in ordine di tempo di completamento  $f$  decrescente (come per l'ordinamento topologico)
- si calcola il grafo trasposto  $GT$  del grafo  $G$
- si esegue una visita in profondità in  $GT$  usando l'ordine dei vertici calcolato nella prima fase nel ciclo principale.

## Definire il concetto di ordinamento topologico di un grafo orientato. Descrivere inoltre il funzionamento della “soluzione diretta” per la determinazione dell’ordine topologico.

Un ordinamento topologico di un grafo orientato aciclico (DAG)  $G = (V, E)$  è un ordinamento lineare dei suoi vertici tale che:

- per ogni arco  $uv \in E$  il vertice  $u$  precede il vertice  $v$
- per transitività, ne consegue che se  $v$  è raggiungibile da  $u$ , allora  $u$  compare prima di  $v$  nell'ordinamento
- L'ordinamento topologico si usa per determinare un ordine in cui eseguire un insieme di attività in presenza di vincoli di precedenze.

Per quanto riguarda il funzionamento della soluzione diretta:

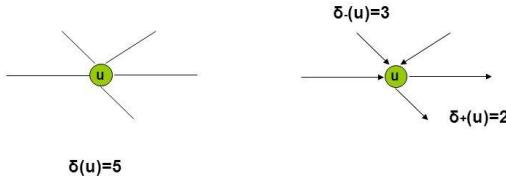
- trovare ogni vertice che non ha alcun arco incidente in ingresso
- stampare tale vertice e rimuoverlo insieme ai suoi archi
- Ripetere la procedura finché tutti i vertici risultano rimossi
- Soluzione basata su DFS

## Definire il concetto di grado di un nodo nel caso in cui il nodo faccia parte di un grafo orientato. Mostrare anche un esempio.

Il grado  $\delta(v)$  del vertice  $v$  è il numero di archi incidenti in  $v$ . Se il grafo è orientato  $\delta(v)$  si suddivide in un grado entrante  $\delta^-(v)$  che è il numero di archi entranti in  $v$  ed un grado uscente  $\delta^+(v)$  che è il numero di archi uscenti da  $v$ . Se  $u = uv \in E$  diciamo che il vertice  $v$  è adiacente al vertice  $u$ . Se il grafo non è orientato la relazione di adiacenza è simmetrica.

### Grado di un nodo

- Il grado di un nodo  $u$  corrisponde al **numero di archi incidenti** con  $u$ .
- In un grafo diretto si può anche calcolare il **grado uscente** o il **grado entrante** di  $u$ , rispettivamente il numero di archi uscenti da  $u$  o entranti in  $u$ .



**Dato un grafo orientato  $G$ , dire cosa si intende per chiusura transitiva di  $G$ . Descrivere a parole un possibile algoritmo per il calcolo della chiusura transitiva.**

Dato un grafo orientato  $G=(V,E)$  con l'insieme di vertici  $V=\{1,\dots,n\}$ , vogliamo sapere se esiste un cammino da  $i$  a  $j$  per tutte le coppie  $i,j \in V$ . La chiusura transitiva di  $G$  è definita come il grafo  $G^*=(V,E^*)$  dove:  $E^*=\{(i,j) : \text{esiste un cammino da } i \text{ a } j \text{ in } G\}$ .

Possibile algoritmo:

1. Assegno un peso pari a 1 ad ogni arco in  $E$  ed applico Floyd-Warshall e se esiste un cammino da  $i$  a  $j$  allora  $d_{ij} < n$ ; altrimenti  $d_{ij} = \infty \Rightarrow$  complessità  $\Theta(n^3)$ .
2. Sostituisco le operazioni di  $\min$  e  $+$  dell'algoritmo di Floyd-Warshall con le operazioni logiche  $\vee$  (or) e  $\wedge$  (and). Per  $i,j,k=1,\dots,n$ , definiamo  $t_{ij}^{(k)}$  pari a 1 se esiste in  $G$  un cammino da  $i$  a  $j$  con tutti i vertici intermedi in  $\{1,\dots,k\}$ ; 0 altrimenti

$t_{ij}^{(k)}$  è definito in modo ricorsivo come segue:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E \\ 1 & \text{se } i = j \text{ o } (i,j) \in E \end{cases}$$

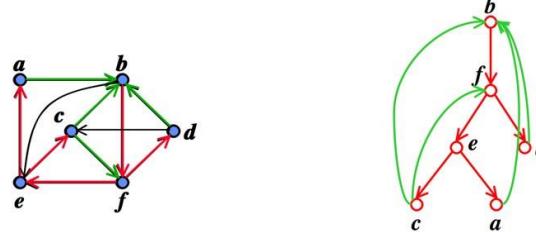
Per  $k \geq 1$ :

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

**Data la foresta generata da una visita DFS su un grafo  $G$ , dire cosa si intende per: arco in avanti, arco di attraversamento, arco d'albero e arco di ritorno.**

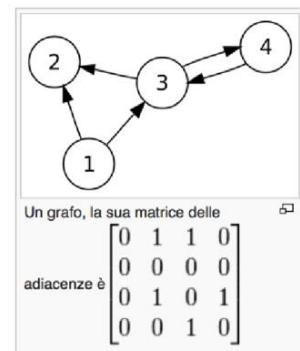
Se un arco  $(u,v)$  in  $G$  ha un flusso tale che  $0 < f(u,v) < c(u,v)$ , allora l'arco  $(u,v)$  si sdoppia in  $G_f$

- **Arco in avanti** con  $cf(u,v) = c(u,v) - f(u,v)$
- **Arco all'indietro** con  $cf(v,u) = c(v,u) - f(v,u) = c(v,u) + f(u,v)$  Sia  $G$  la foresta DF generata da DFS sul grafo  $G$ .
- **Arco d'albero:** gli archi della foresta  $G_{\Pi}$ , tali che l'arco  $(u,v) \in E_{\Pi}$  se  $v$  è stato scoperto esplorando l'arco  $(u,v)$ .
- **Arco di ritorno:** gli archi  $(u,v)$  che connettono un vertice  $u$  con un antenato  $v$  nell'albero DF.



**Dire cosa si intende per matrice di adiacenza di un grafo. Dato quindi un grafo  $G$  e la sua matrice di adiacenza, dire qual è la complessità delle seguenti operazioni (giustificare la risposta).**

Dato un qualsiasi grafo la sua **matrice** delle adiacenze è costituita da una matrice binaria quadrata che ha come indici di righe e colonne i nomi dei vertici del grafo. Nel posto  $(i,j)$  della matrice si trova un 1 se e solo se esiste nel grafo un arco che va dal vertice  $i$  al vertice  $j$ , altrimenti si trova uno 0.  $O(n^2)$  è la complessità, essendo lo spazio richiesto e richiedendo ogni vertice  $O(1)$  tempo.

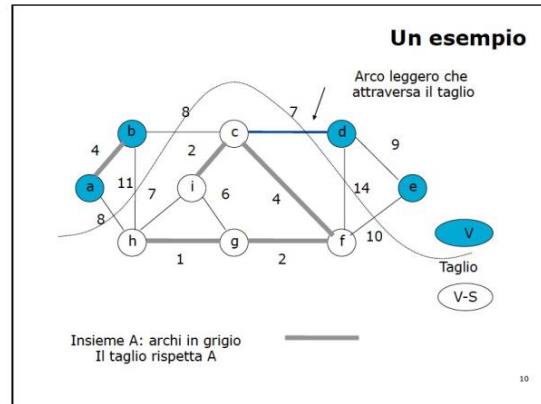


**Dato un taglio di un grafo non orientato, dire cosa si intende per arco sicuro e fornire un esempio.**

L'idea è di **accrescere** un sottoinsieme  $A$  di archi di un albero di copertura aggiungendo un arco alla volta. Ad ogni passo si determina un arco che può essere aggiunto ad  $A$  mantenendo la proprietà per  $A$  di essere un sottoinsieme di archi di un albero di copertura. Un arco di questo tipo è detto **arco sicuro**.

**Dato un grafo non orientato, dare la definizione di arco leggero rispetto ad un taglio. Cos'è il taglio? Si richiede di fornire un esempio.**

**Un taglio**  $(S, V \setminus S)$  di un grafo non orientato  $G = (V, E)$  è una partizione di  $V$  in due sottoinsiemi disgiunti. **Un arco**  $(u, v)$  attraversa il taglio se  $u \in S$  e  $v \in V \setminus S$ . **Un taglio** rispetta un insieme di archi  $A$  se nessun arco di  $A$  attraversa il taglio. **Un arco** che attraversa un taglio è **leggero** nel taglio se il suo peso è minimo fra i pesi degli archi che attraversano un taglio.



**Sia  $G$  un grafo con  $n$  nodi ed  $m$  archi. Dire se l'affermazione "tutte le foreste generate da differenti visite in profondità hanno lo stesso numero di alberi" è vera o falsa, giustificando la risposta data.**

Si ricordi che, dato un grafo  $G$  con  $k$  componenti connesse, qualsiasi foresta ricoprente (che sia generata da una visita in profondità o no) è formata esattamente da  $k$  alberi, uno per ciascuna componente connessa. Da questa semplice osservazione si deduce che la risposta al quesito è affermativa.

**Un cammino minimo pu contenere un ciclo? Giustificare la risposta.**

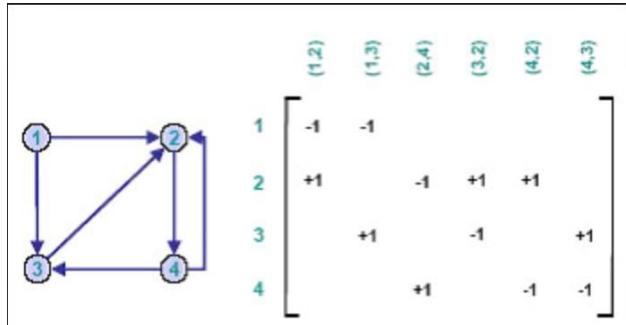
No, un cammino minimo NON pu contenere un ciclo dal momento che il cammino minimo è il percorso con meno nodi da una radice a una foglia, e un ciclo sarebbe come minimo un ripetersi di un nodo, quindi risulta impossibile per un cammino minimo contenere un ciclo intero, mentre, pur una sola volta, pu contenere una parte del ciclo, ma i nodi di quest'ultimo devono essere attraversati una volta sola, altrimenti andrebbe a decadere la definizione stessa di cammino minimo.

**Che differenza c'è tra un albero ed un grafo?**

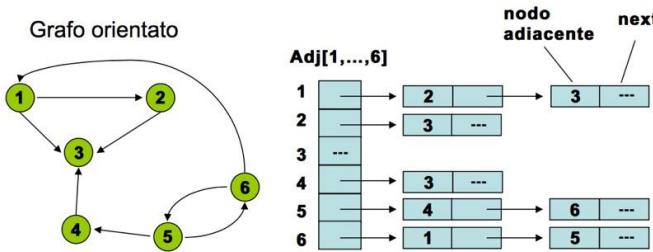
Naturale applicazione dell'ADT Grafo è data dalla rappresentazione delle relazioni che intercorrono tra più oggetti. Un esempio classico di applicazione di un grafo è la rappresentazione delle strade che pongono in comunicazione tra loro un insieme di siti. Gli alberi sono una specializzazione dell'ADT Grafo. Mediante gli alberi rappresentiamo comunque relazioni che intercorrono tra più oggetti, ma più specificatamente rappresentiamo relazioni di tipo gerarchico.

Si richiede di descrivere la rappresentazione di grafi mediante matrici, liste e vettori di adiacenza.

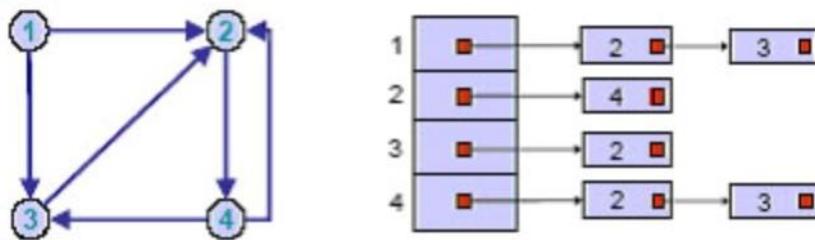
## Matrice →



## Liste di adiacenza ➔



## Vettori di adiacenza →



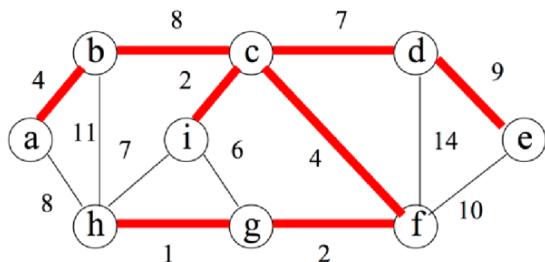
Dato un grafo non orientato  $G$  cosa si intende per circuito? Mostrare un esempio.

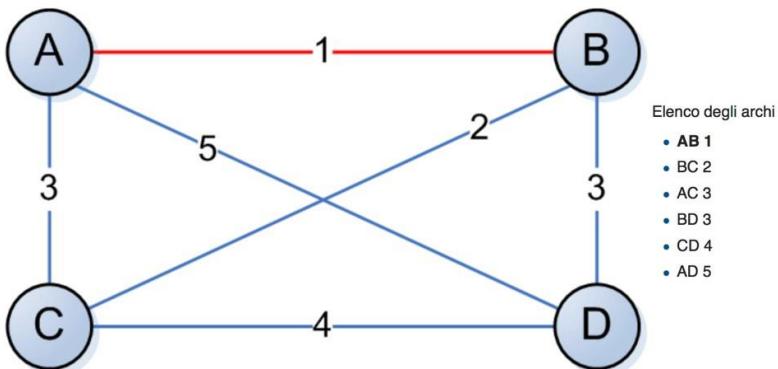
Un "percorso" di lunghezza  $n$  in  $G$  è dato da una sequenza di vertici  $v_0, v_1, \dots, v_n$  (non necessariamente tutti distinti) e da una sequenza di archi che li collegano  $(v_0, v_1), (v_1, v_2), \dots, (v_{n-1}, v_n)$ . I vertici  $v_0$  e  $v_n$  si dicono *estremi* del percorso. Un percorso con i lati a due a due distinti tra loro prende il nome di *cammino*. Un cammino chiuso ( $v_0 = v_n$ ) si chiama *circuito* o *ciclo*.

Definire in modo preciso un albero di copertura minimo e descrivere il funzionamento dell'algoritmo di Kruskal . Qual è la complessità di questo algoritmo?

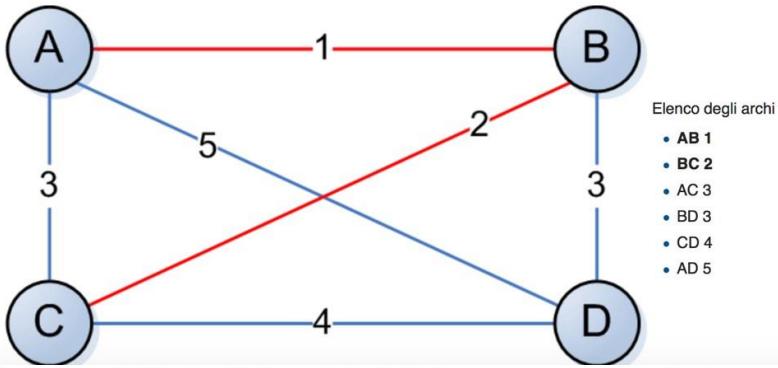
Partiamo dalla determinazione di un sottografo aciclico  $T \subseteq E$  che connetta tutti i vertici in modo da minimizzare il peso totale  $w(T) = \sum_{(u,v) \in T} w(u,v)$ . Dato che  $T$  è aciclico e collega tutti i vertici deve essere un albero. Tale albero è chiamato albero di copertura minima o minimum spanning tree MST. Si consideri un grafo non orientato e connesso dove  $V$  rappresenta il numero di vertici (o nodi) ed  $E$  il numero di spigoli (o archi). Ad ogni spigolo è associato un peso (o distanza): lo scopo dell'algoritmo è quello di trovare un albero ricoprente di peso minimo, cioè quello in cui la somma dei pesi sia minima. L'algoritmo può essere applicato solo se si dispone di due o più vertici.

L'algoritmo di Kruskal si basa sulla seguente semplice idea: ordiniamo gli archi in ordine crescente di costo e successivamente li analizziamo singolarmente, inseriamo l'arco nella soluzione se non forma cicli con gli archi precedentemente selezionati. Notiamo che ad ogni passo, se abbiamo più archi con lo stesso costo, è indifferente quale viene scelto. Il [costo computazionale](#) dell'algoritmo è nel caso peggiore  $O(VE)$  dove  $E$  è il numero di archi ed  $V$  il numero di vertici.



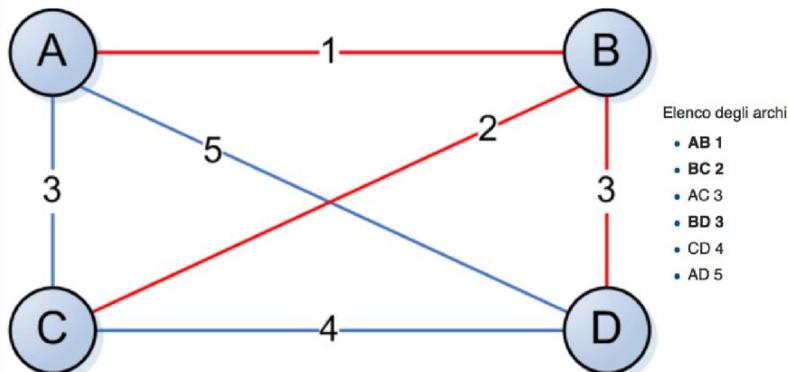


Procediamo considerando l'arco BC (costo 2) e, analogamente a quanto appena fatto, lo evidenziamo nel grafico e lo segniamo nell'elenco:



Il prossimo arco dell'elenco sarebbe AC (costo 3) ma lo saltiamo perché creerebbe un circolo chiuso in quanto collegherebbe insieme il nodo A e il nodo C che sono già collegati attraverso B e lo scopo del problema è creare una rete in cui tutti i nodi sono collegati ma utilizzando il minor cavo possibile.

Si procede dunque con BD (costo 3) colorando l'arco di rosso ed evidenziandolo nell'elenco:



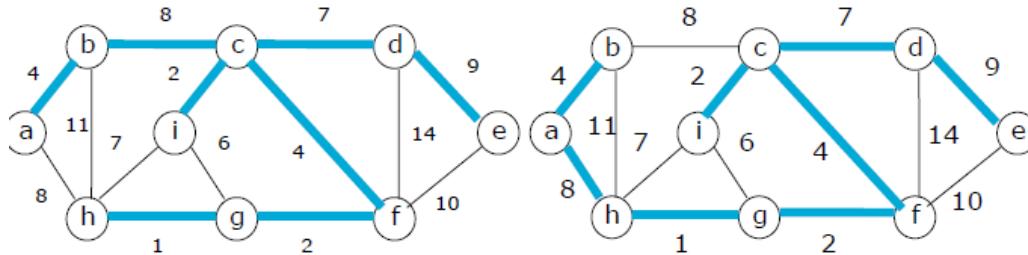
Ora tutti i nodi sono collegati alla rete quindi il problema è concluso. Procedendo con l'elenco, infatti, si potrebbe verificare che gli altri archi rimanenti creerebbero solamente circoli chiusi.

### Descrivere il problema della determinazione dell'albero di copertura minimo, ed illustrare l'algoritmo di Kruskal. Cosa si intende per arco sicuro?

Il problema dell'albero di copertura/connesione minimo è un problema di notevole importanza: l'obiettivo è quello di determinare come interconnettere diversi elementi fra loro minimizzando certi vincoli sulle connessioni

**ESEMPIO** → Progettare circuiti elettronici minimizzando la quantità di filo elettrico per collegare fra loro diversi componenti.

L'albero di copertura minimo è l'albero il cui peso totale sia minimo.



Per caratterizzare gli archi sicuri dobbiamo introdurre alcune definizioni:

- Un **taglio**  $(S, V \setminus S)$  di un grafo non orientato  $G = (V, E)$  è una partizione di  $V$  in due sottoinsiemi disgiunti
- Un arco  $(u, v)$  **attraversa** il taglio se  $u \in V \setminus S$  e  $v \in S$
- Un taglio **rispetta** un insieme di archi  $A$  se nessun arco di  $A$  attraversa il taglio
- Un arco che attraversa un taglio è **leggero** nel taglio se il suo peso è minimo fra i pesi degli archi che attraversano un taglio.

La nostra regola per riconoscere gli archi sicuri è definita dal seguente teorema:

Sia  $G = (V, E)$  un grafo connesso non orientato con una funzione peso  $w$  a valori reali definita in  $E$ .

Sia  $A$  un sottoinsieme di  $E$  che è contenuto in qualche albero di connessione minima per  $G$ , sia  $(S, V - S)$  un taglio qualsiasi di  $G$  che rispetta  $A$  e sia  $(u, v)$  un arco leggero che attraversa  $(S, V - S)$ .

Allora, l'arco  $(u, v)$  è sicuro per  $A$ .

L'algoritmo di Kruskal si occupa di ingrandire sottoinsiemi disgiunti di un albero di copertura minima connettendoli fra di loro fino ad avere l'albero complessivo.

Si individua un arco sicuro scegliendo un arco  $(u, v)$  di peso minimo tra tutti gli archi che connettono due alberi distinti (componenti connesse) della foresta.

Nell'algoritmo ad ogni passo si aggiunge alla foresta un arco con il peso minore.

**Dato un grafo non orientato e connesso  $G$ , dire cosa si intende per albero di copertura di costo minimo. Qual'è la complessità dell'algoritmo di Kruskal? Perché?**

Dato un grafo  $G = (V, E)$  non orientato, connesso e pesato sugli archi, un albero di copertura minima di  $G$  è un albero di copertura di  $G$  di costo minimo. Abbrevieremo il nome in MST, dall'inglese Minimum Spanning Tree.

L'algoritmo di Kruskal si occupa di ingrandire sottoinsiemi disgiunti di un albero di copertura minima connettendoli fra di loro fino ad avere l'albero complessivo.

Lo pseudocodice consiste in:

```

AlbConnMinKruskal(G)    G grafo pesato sugli archi
    A ← Ø
    for "ogni  $u \in V[G]$ " do
        MakeSet( $u$ )
    "ordina gli archi in ordine di costo crescente"
    for "ogni  $a = uv \in E[G]$  in ordine di costo" do
        if FindSet( $u$ ) ≠ FindSet( $v$ ) then
            Union( $u, v$ )
            A ← A ∪ {( $u, v$ )}
    return A

```

Complessità:

Il primo ciclo for richiede  $O(|V|)$

L'ordinamento degli archi richiede  $O(|E| \log |E|)$

L'ultimo ciclo for richiede  $O(|E| \alpha(|E|, |V|))$

Pertanto la COMPLESSITÀ' è  $O(|E| \log |E|)$  e siccome  $\log |E| = O(\log |V|^2) = O(\log |V|)$  essa è anche uguale a  $O(|E| \log |V|)$

Definire formalmente il problema di determinare l'albero di copertura minima indicando chiaramente Cos'è l'input e cos'è l'output. Si richiede inoltre di descrivere il funzionamento di un algoritmo per la risoluzione di tale problema, mostrandone anche il funzionamento su un grafo di esempio.

### Il Problema dell'Albero di Copertura Minima, 1

Sia dato un grafo non orientato e connesso  $G=(V,E)$ ,  $|V|=n$ ,  $|E|=m$ , e ad ogni arco  $[i,j]$  in  $E$  sia assegnato un costo  $c_{ij}$ .

Per ogni sottoinsieme  $E'$  non proprio di  $E$ , il costo complessivo  $c(E')$  associato ad  $E'$  è così definito:

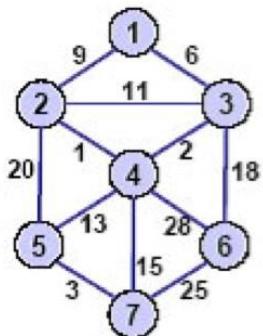
$$c(E') = \sum_{[i,j] \in E'} c_{ij}.$$

Il Problema dell'Albero di Copertura Minima (Minimum Spanning Tree - MST) di  $G$  consiste nell'individuare un albero  $T_G = (V, E')$  che "ricopra"  $G$  e che fra tutti gli alberi ricoprenti  $G$  sia un albero di costo totale minimo.

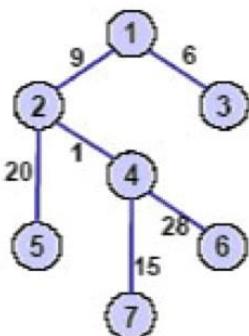
### Il Problema dell'Albero di Copertura Minima, 2

Alcune semplici considerazioni utili alla completa comprensione del problema:

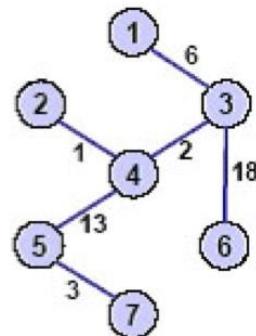
- l'insieme dei nodi dell'albero  $T_G = (V, E')$  coincide con l'insieme dei nodi del grafo  $G$ , per cui l'insieme degli archi  $E'$  di  $T_G$  ha cardinalità pari a  $|V|-1=n-1$ .
- affermare che  $T_G = (V, E')$  sia un albero di copertura minima per  $G$  vuole dire affermare che  $c(E') \leq c(\hat{E})$ , per ogni  $\hat{E}$  albero di copertura di  $G$ .
- se  $G=(V,E)$  non è connesso, il problema MST non ammette soluzione.



grafo pesato



soluzione  
ammissibile  
79



soluzione  
ottima  
43

Quale problema risolve l'algoritmo di Prim? Qual è la sua complessità? Può essere usato per qualsiasi grafo?

L'algoritmo di Prim procede mantenendo in  $A$  un singolo albero. L'albero parte da un vertice arbitrario  $r$  (la radice) e cresce fino a quando non ricopre tutti i vertici. Ad ogni passo viene aggiunto un arco leggero che collega un vertice in  $V_A$  con un vertice in  $V \setminus V_A$  – dove  $V_A$  è l'insieme di nodi raggiunti da archi in  $A$ . Correttezza:

- $(V_A, V \setminus V_A)$  è un taglio che rispetta  $A$  (per definizione)
- Per il corollario, gli archi leggeri che attraversano il taglio sono sicuri.

I vertici sono aumentati con un puntatore  $p$  al padre nell'albero in costruzione, un campo colore che è bianco inizialmente e diventa nero quando il vertice viene aggiunto all'albero in costruzione, un campo key che contiene il costo minimo di un arco che connette il vertice ad uno dei vertici già raggiunti dall'albero in costruzione. Per quanto riguarda la complessità, dipende da come viene implementata la coda  $Q$ : supponiamo che sia implementata tramite un minheap:

- l'inizializzazione richiede  $O(|V|)$
- il ciclo while viene eseguite  $|V|$  volte e poiché ExtracMin richiede  $O(\lg|V|)$ , il tempo totale per queste operazioni è  $O(|V|\lg|V|)$
- il ciclo for interno al while viene eseguite  $O(|E|)$  volte e l'operazione di decremento della chiave richiede  $O(\lg|V|)$ . Tempo totale:  $O(|V|\lg|V| + |E|\lg|V|)$ .

**In un albero di copertura minima (MST), il cammino da un nodo ad un altro nodo è un cammino minimo tra i due nodi?**

Un albero di copertura o albero di connessione o albero di supporto di un grafo, connesso e con archi non orientati, è un albero che contiene tutti i vertici del grafo, ma degli archi ne contiene soltanto un sottoinsieme, cioè solo quelli necessari per connettere tra loro tutti i vertici con uno e un solo cammino. Questa definizione fa capire con chiarezza che in un MST il cammino da un nodo a un altro nodo è anche il minimo fra essi, quindi la risposta alla domanda è affermativa.

**Descrivere il problema dell'esistenza degli archi con pesi negativi quando e perchè creano problemi nella definizione e ritrovamento del cammino minimo (illustrare un esempio).**

**Descrivere inoltre un algoritmo per il calcolo dei cammini minimi da sorgente unica e discuterne la complessità.**

Gli archi di peso negativo non creano problemi nella ricerca dei cammini minimi da una sorgente  $s$ . Se ci sono cicli di costo negativo raggiungibili da  $s$ , il peso dei cammini non è ben definito: un vertice raggiungibile da  $s$  con un cammino  $p$  passante per un vertice  $v$  di un ciclo negativo, implica esistenza di cammini da  $s$  a  $v$  di costi sempre minori il costo di cammino minimo  $\delta(s, v)$  non è definito e poniamo  $\delta(s, v) = -\infty$ .

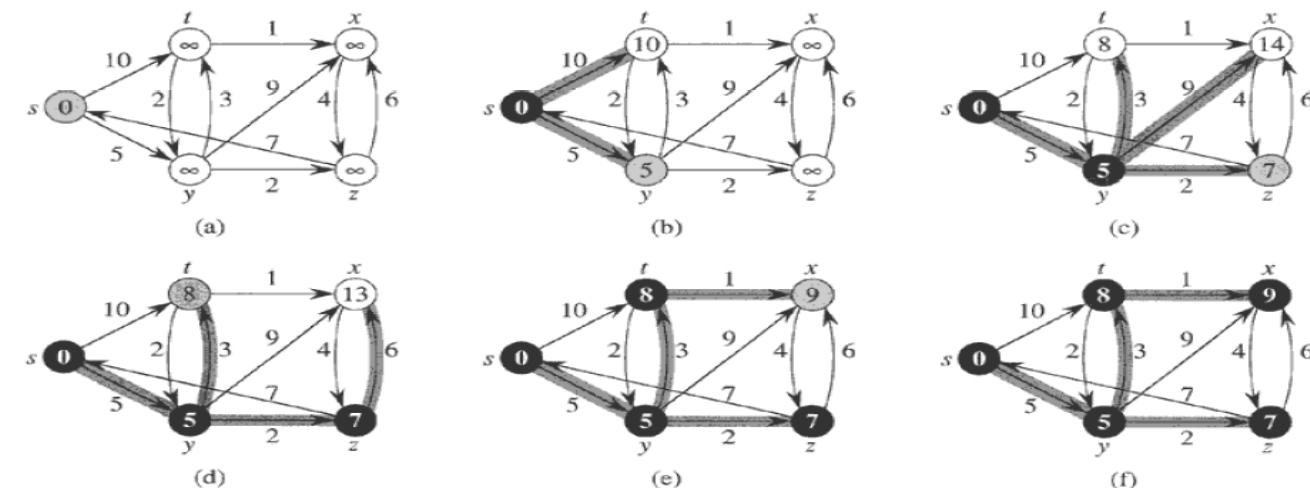
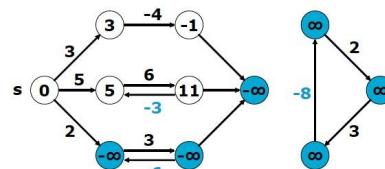
**Perchè i cammini semplici sono gli unici cammini da considerare quando si affronta il problema del cammino più breve?**

Un cammino si dice semplice se composto da nodi distinti. Sfruttando questa definizione, si può arrivare facilmente a capire che siccome il cammino minimo ha solo la peculiarità di non ripassare da un nodo e arrivare il prima possibile alla destinazione, considerando solo il cammino semplice non mi devo preoccupare di ripassare dallo stesso nodo.

**Dire che problemi risolve l'algoritmo di Dijkstra e che limitazioni ha. Perchè ci sono queste limitazioni? Quanto costa l'algoritmo?**

Risolve il problema dei cammini minimi a sorgente unica (indicata con  $s$ ) in un  $G = (V, E)$  orientato e con pesi sugli archi non negativi (per ogni arco  $(u, v) \in E$ , il peso  $w(u, v) \geq 0$ ). Mantiene un insieme di nodi  $S$  i cui pesi dei cammini minimi da  $s$  sono già stati determinati. Seleziona ripetutamente il nodo in  $u \in V - S$  con la minima stima del cammino minimo e rilassa tutti gli archi che escono da  $u$ . Il tempo richiesto dall'algoritmo è dell'ordine di  $O(n^2 + n) = O(n^2)$ .

**Esempio con archi di peso negativo**



## A cosa serve l'algoritmo di Dijkstra? Pu sempre essere utilizzato? Motivare la risposta

Sull'utilità vedi domanda precedente. Per quanto concerne l'utilizzo, per definizione l'algoritmo di dijkstra non pu avere archi negativi. Se il peso degli archi è negativo il ciclo è mal definito in quanto per ogni "giro" si "guadagna" e quindi il ciclo non avrebbe fine! Quindi bisogna imporre nelle precondizioni che gli archi non possano essere negativi: infatti l'algoritmo impone:  $C_e \geq 0$  per ogni e appartenente ad  $E$ .

**Il percorso più breve in un grafo orientato pesato è il percorso la cui somma degli archi visitati per raggiungere un vertice  $v$  partendo da un vertice  $s$  è più piccola. Dire se le seguenti affermazioni sono vere o false e nel caso in cui siano false, fornire un contro-esempio.** (a) Se tutti gli archi in un grafo orientato pesato hanno pesi diversi, esiste un unico percorso breve. (b) L'algoritmo di Dijkstra per la ricerca del percorso breve trova sempre il percorso breve con il numero minimo di archi.

- (a) L'affermazione è falsa, dal momento che potrei scegliere i nodi che hanno peso minore (tutti), e in questo caso sarebbe unico, ma non sempre è possibile, spesso sono costretto a fare una scelta di percorso, e esistono casi in cui, proprio per la differenza di tutti gli archi a livello di peso, ho più percorsi brevi.
- (b) Anche questa affermazione è falsa, dal momento che esistono condizioni per l'utilizzo dell'algoritmo di Dijkstra, come quella che non devono esserci archi con peso negativo. Inoltre questo algoritmo è stato studiato per calcolare i cammini minimi, ma non è detto che ci sia il minimo numero di archi.

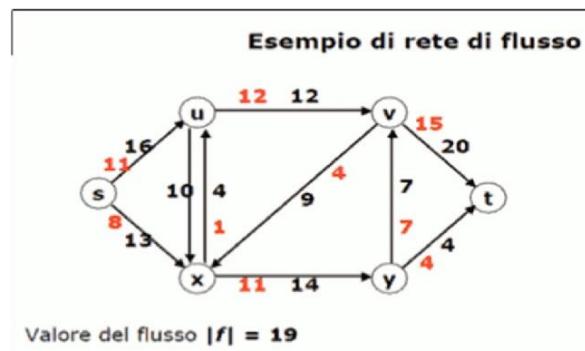
## Dire a cosa serve l'algoritmo di Floyd-Warshall. Che complessità ha?

L'algoritmo di Floyd-Warshall calcola il cammino minimo per tutte le coppie di un grafo pesato e orientato con una complessità  $O(|N|^3)$ . L'idea alla base di questo algoritmo è un processo iterativo che, scorrendo tutti i nodi, ad ogni passo  $h$  si ha in una matrice  $A$  nella posizione  $[i,j]$  la distanza - pesata - minima dal nodo di indice  $i$  a quello  $j$  attraversando solo nodi di indice minore o uguale a  $h$ . Se non vi è collegamento allora nella cella c'è infinito. Ovviamente alla fine (con  $h = \text{numero di nodi}$ ) leggendo la matrice si ricava la distanza minima fra i vari nodi del grafo.

**Nell'ambito delle reti di flusso, fornire la definizione formale delle proprietà di vincolo sulla capacità, antisimmetria e conservazione del flusso e dire cosa rappresentano.**

Una rete di flusso è un grafo orientato in cui ogni arco ha una capacità positiva quindi maggiore o uguale di zero. Il flusso è una funzione a valori reali che deve soddisfare delle proprietà:

- **vincolo di capacità:** il flusso non pu superare la capacità dell'arco,  $f(u,v) \leq c(u,v)$ ;
- **antisimmetria:** il flusso da  $u$  a  $v$  posso vederlo come un flusso negativo  $f(u,v) = -f(v,u)$ ;
- **conservazione del flusso:** il flusso non pu perdersi lungo il tragitto.



**Nell'ambito delle reti di flusso, cosa dice la proprietà di conservazione del flusso? Mostrare un esempio.**

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v), \quad \forall u \in V \setminus \{s, t\}$$

Per ogni nodo il flusso entrante deve eguagliare quello uscente. Ci non vale per la sorgente, che "crea" flusso, ed il pozzo, che "consuma" flusso.

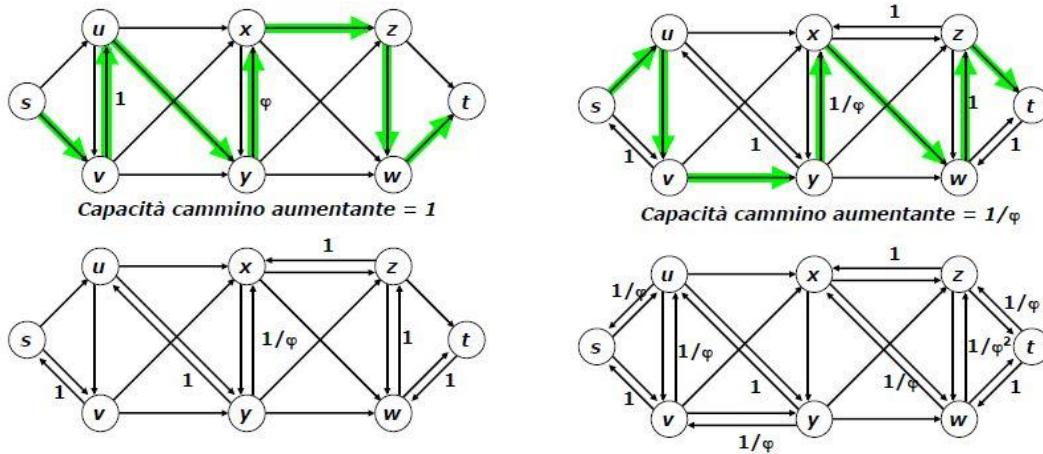
**A cosa serve l'algoritmo di Ford-Fulkerson? Termina sempre tale algoritmo? Si richiede di motivare la risposta.**

L'algoritmo di Ford-Fulkerson permette di trovare il flusso massimo che attraversa un grafo da un punto ad un altro di questo. Si definisce un metodo e non un algoritmo perché può avere diverse implementazioni eseguite in tempi diversi per tutte basate su tre concetti:

- cammini aumentanti: cammino non saturo che posso utilizzare ancora per trasmettere;
- rete residua: grafo con archi di capacità residua diversa da zero. Gli archi sdoppiano se non sono saturi;
- taglio di una rete di flusso: partiziono l'insieme dei nodi in due insiemi; sorgente e pozzo devono stare in due sottoinsiemi diversi. La capacità del taglio è la capacità degli archi che stanno in tutti e due i sottoinsiemi

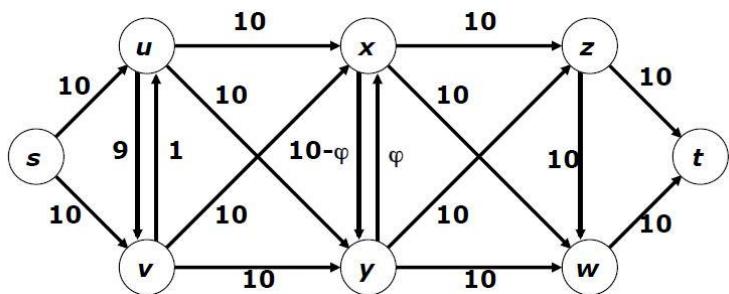
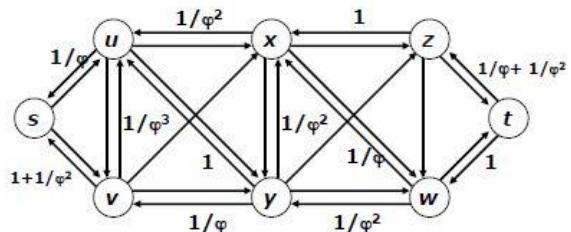
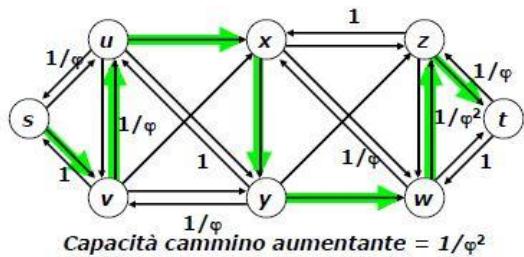
L'algoritmo NON termina sempre, dal momento che ci sono percorsi che possono essere usati infinite volte, senza cambiare le loro capacità residue

L'algoritmo di Ford-Fulkerson serve per trovare i flussi massimi. Partendo con il flusso  $f(u,v) = 0$  a ogni iterazione trova un cammino aumentante  $p$  in  $G_f$  e incrementa il flusso  $f$  lungo  $p$  della capacità residua  $c_f(p)$ . Ogni arco residuo nel cammino può essere un arco della rete originale o l'opposto di un arco della rete originale. Successivamente il flusso viene aggiornato sommando il flusso quando l'arco residuo è un arco originale e sottraendo il flusso nell'altro caso. Solitamente il problema del flusso massimo si presenta con le capacità degli archi espresse da numeri interi, tuttavia se si dovesse presentare con capacità reali abbiamo anche la possibilità che non termini. Andando più nello specifico se fosse con capacità razionali bisognerebbe effettuare un'appropriata conversione di scala



per trasformarle tutte in interi, mentre se fosse con capacità irrazionali l'algoritmo potrebbe anche non terminare: il flusso aumenterebbe di una quantità sempre minore a ogni passo senza raggiungere mai il massimo e a volte neanche convergerebbe a tale massimo. Un esempio di irrazionali potrebbe essere: dove  $\phi$  è il rapporto aureo per cui  $\phi-1=1/\phi$

L'algoritmo NON termina sempre, dal momento che ci sono percorsi che possono essere usati infinite volte, senza cambiare le loro capacità residue

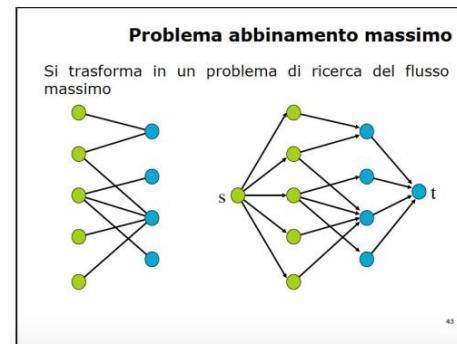
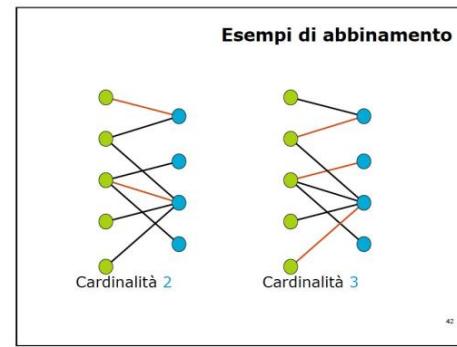


Procedendo in questo modo il flusso aumentante procede al limite:

$$\lim_{n \rightarrow \infty} \sum_{i=0}^{n-1} 1/\varphi^i = \sum_{i=0}^{\infty} 1/\varphi^i = \varphi^2 < 20$$

**Si richiede di descrivere il problema dell'abbinamento massimo in un grafo bipartito e di mostrare tramite un esempio come è possibile trasformarlo in uno di flusso massimo.**

Un grafo non orientato  $G = (V, E)$  si dice bipartito se i suoi vertici si possono ripartire in due sottoinsiemi  $S$  e  $D$  tali che ogni arco abbia estremi appartenenti a sottoinsiemi distinti ( $E \subseteq S \times D$ ). Se  $xy$  è un arco diciamo che  $x$  e  $y$  si possono accoppiare. Il problema chiede di trovare un insieme massimo di coppie distinte.



**Nell'ambito delle strutture dati per insiemi disgiunti, dire cosa si intende per euristica dell'unione pesata. Si richiede inoltre di scrivere la procedura Unione con tale euristica.**

Consiste nello scegliere sempre la lista più corta per aggiornare i puntatori al rappresentante. Occorre memorizzare la lunghezza di ciascuna lista in un opportuno campo `lung` del rappresentante. Si aggiunge ad ogni nodo un campo booleano `rp` per distinguere il rappresentante dagli altri nodi. Nel rappresentante possiamo mettere il campo `lung` al posto del campo `rappr` che risulta inutile. La procedura `Unione` è la seguente: `Union(x, y)` if not `rp[x]` then `x ← rappr[x]` if not `rp[y]` then `y ← rappr[y]` if `lung[x] < lung[y]` then `z ← x, x ← y, y ← z` `lung[x] ← lung[x] + lung[y]` `rp[y] ← false, rappr[y] ← x, z ← succ[y]` while `z ≠ y` do `rappr[z] ← x, z ← succ[z]` `z ← succ[x], succ[x] ← succ[y], succ[y] ← z`

**Nell'ambito della struttura dati astratta insiemi disgiunti, cosa si intende per euristica dell'unione per rango? Si richiede di mostrare un esempio di esecuzione dell'operazione `union` che sfrutta l'euristica menzionata.**

Per ogni nodo  $x$  manteniamo un campo `rank` che è un limite superiore all'altezza del sottoalbero di radice  $x$  (numero di archi nel cammino più lungo fra  $x$  e una foglia discendente). `Union` mette la radice con rango minore come figlia di quella di rango maggiore. Esempio: `Union(x,y)`  $x \leftarrow \text{FindSet}(x)$   $y \leftarrow \text{FindSet}(y)$

```
Link(x, y) Link(x,
y)
if rank[x] > rank[y] then
parent[y] ← x else
parent[x] ← y if rank[x]
= rank[y] then
rank[y] ← rank[y] + 1
```

## Descrivere il tipo di dato astratto sottoinsiemi disgiunti e descrivere le strutture dati che possono essere usate per rappresentare gli insiemi disgiunti.

Le strutture dati per insiemi disgiunti servono a mantenere una collezione  $S = \{S_1, S_2, \dots, S_k\}$  di insiemi disgiunti. Ogni insieme della collezione è individuato da un rappresentante che è un particolare elemento dell'insieme. Le operazioni che possono essere effettuate sugli insiemi disgiunti sono: [Make-Set\(x\)](#): aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento x. Si richiede che x non compaia in nessun altro insieme della struttura.

[Find-Set\(x\)](#): ritorna il rappresentante dell'insieme che contiene x.

[Union\(x,y\)](#): riunisce i due insiemi contenenti x ed y in un'unico insieme.

**Rappresentazione:** con [Liste](#) - Il modo più semplice per rappresentare una famiglia di insiemi disgiunti è usare una lista circolare per ogni insieme. Ogni nodo ha un puntatore al nodo successivo ed un puntatore al nodo della lista che viene assunto come rappresentante; con [Foreste](#) - Una rappresentazione più efficiente si ottiene usando foreste di insiemi disgiunti. Ogni insieme è rappresentato con un albero i cui nodi, oltre al campo info hanno soltanto un campo parent che punta al padre.

## Discutere il problema della ricorsione: cosa si intende per ricorsione, le diverse tipologie di ricorsione e come sono strutturate le procedure ricorsive. Descrivere inoltre la tecnica di tail recursion.

È ricorsivo quello che pu essere definito in termini di se stesso.

- fattoriale  $\rightarrow ! 0! = 1!$   
 $! n! = n(n-1)! \quad n > 0$
- numeri di fibonacci  $\rightarrow ! f_0 = 0, f_1 = 1$   
 $! f_n = f_{n-1} + f_{n-2}, n \neq 2$

La potenza di una definizione ricorsiva è legata alla possibilità di definire un insieme di oggetti infinito mediante una descrizione finita.

Si parla di **ricorsione diretta** quando una procedura è espressa in termini di se stessa:  $p(n)$   
 $if(n > 0) \text{ then } p(n-1);$

...

Si parla di **ricorsione indiretta** quando una procedura  $p$  è espressa in termini di una procedura a sua volta definita direttamente o indirettamente in termini di  $p$ :  $p(n) q(m)$

$if(n < 1) \text{ then } \dots \text{ if}(m > 0) \text{ then } p(n-2)$   
 $else q(n-1);$

La struttura di una procedura ricorsiva è costituita da:

- base della ricorsione
- passo ricorsivo

Funziona se e solo se le sue variabili interne sono istanziate nuovamente per ogni attivazione. L'introduzione della ricorsione introduce il problema potenziale di ricorsioni infinite. In pratica, mentre un loop infinito effettivamente non ha mai termine, la ricorsione "infinita" prima o poi determina l'esaurimento della memoria disponibile per lo stack del processo che la esegue e la distruzione di quest'ultimo. In pratica è necessario che la ricorsione termini e che il numero di passi della ricorsione (uso dello stack) sia modesto. Si parla di ricorsione di coda, o tail, quando la chiamata ricorsiva è l'ultima istruzione eseguita nella funzione. Questo tipo di algoritmo ricorsivo è possibile trasformarlo semplicemente in una versione iterativa, che di solito è più efficiente, in quanto non occorre mantenere lo stato della funzione una volta calcolata come è stato fatto nell'esempio precedente.

## Dire cosa si intende per ricorsione diretta e ricorsione indiretta e fornire degli esempi.

Un metodo si dice ricorsivo quando all'interno della propria definizione compare una chiamata direttamente al metodo stesso. Questa forma di ricorsione si chiama **ricorsione diretta**. Un esempio di ricorsione diretta è:

```
public static int numeroTriangolare(int n) {  
    int result; if (n == 1)  
        result = 1; // caso base  
    else  
        result = n + numeroTriangolare(n - 1); // ricorsione  
  
    return result;  
  
}
```

Condizioni come  $(n == 1)$  si chiamano clausole di chiusura o casi base perché garantiscono che la ricorsione termini. Esistono due requisiti che sono basilari per essere sicuri che la ricorsione funzioni:

- ogni invocazione ricorsiva deve semplificare in qualche modo l'elaborazione
- devono esistere casi speciali che gestiscono in modo diretto le elaborazioni più semplici. Il metodo `numeroTriangolare` chiama se stesso con valori di ampiezza sempre più piccoli; l'ampiezza prima o poi diventa uguale a 1 che è il caso speciale che vale 1.

Occorre per fare molta attenzione: cosa succede se chiedete l'area del triangolo di ampiezza -1? Per evitare ciò il metodo `numeroTriangolare` dovrebbe restituire 0 se l'ampiezza è minore di 0. In questo modo il metodo `numeroTriangolare` riuscirebbe sempre a concludere la propria elaborazione.

```
public class MyRecursiveMethods {  
  
    // altri metodi  
  
    public static int numeroTriangolare(int n) {  
        int result;  
        if (n < 0)  
            result = 0; // situazione anomala  
  
        else if (n == 1)  
            result = 1; // caso base  
  
        else  
            result = n + numeroTriangolare(n - 1); // ricorsione  
  
        return result;  
  
    }  
  
    // altri metodi  
}
```

Si parla di **ricorsione indiretta** quando nella definizione di un metodo compare la chiamata ad un altro metodo il quale direttamente o indirettamente chiama il metodo iniziale. Un esempio di ricorsione indiretta:

```
public class MyRecursiveMethods {  
  
    ... // altri metodi
```

```

// notare che si restituisce direttamente il risultato,
// evitando l'uso della variabile locale result    public
static int ping(int n) {
    if (n < 1)
        return 1;    else
    return pong(n - 1); // chiamata di pong
}

public static int pong(int n) {
    if (n < 0)
return 0;    else
    return ping(n/2); // chiamata di ping
}

... // altri metodi
}

```

Notare che, nell'esempio sopra, i metodi sono cooperanti nel senso che si invocano ripetutamente a vicenda (indirettamente), dando luogo ad un caso particolare di ricorsione indiretta, detto ricorsione mutua.

**E' vero che le chiamate ricorsive non possono mai essere eliminate? Motivare la risposta.**

Esiste una semplice tecnica per eliminare la ricorsione? Si, la **tail recursion**.

Una procedura p presenta tail recursion quando:

- Ha un parametro x passato per valore
- La sua ultima istruzione è una chiamata di p (si noti che questo non vuol dire che la chiamata a p sia l'ultima cosa scritta) ricorsiva con valore y del parametro di chiamata.

Si pu eliminare la chiamata ricorsiva sostituendola con un assegnamento di y a x ed un goto alla prima istruzione della procedura.

**In che cosa consiste l'approccio “divide et impera”? Nominare un algoritmo che se ne serva.**

Un problema viene suddiviso in sotto-problemi indipendenti, che vengono risolti ricorsivamente (topdown). Il *divide et impera* rappresenta un approccio molto efficace per la risoluzione di vari problemi computazionali. In particolare si parla di algoritmi *divide et impera*. Questi algoritmi dividono ricorsivamente un problema in due o più sotto-problemi sino a che questi ultimi diventino di semplice risoluzione, quindi, si combinano le soluzioni al fine di ottenere la soluzione del problema dato. Questo approccio permette di affrontare in modo "semplice" problemi anche molto difficili (non complessi, dove la complessità sia irriducibile o non-lineare), inoltre la natura del "divide" permette di parallelizzare la computazione aumentandone l'efficienza su sistemi distribuiti o multiprocessore. Questo tipo di approccio è tipicamente detto top down. Un programma sviluppato secondo questa tecnica è sostanzialmente diviso in tre parti:

- Divide*: in questa parte si procede alla suddivisione dei problemi in problemi di dimensione minore;
- Impera*: nella seconda parte i problemi vengono risolti in modo ricorsivo. Quando i sottoproblemi arrivano ad avere una dimensione sufficientemente piccola, essi vengono risolti direttamente tramite il caso base;
- Combina*: l'ultima fase del paradigma prevede di ricombinare l'output ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.

In informatica questo principio viene applicato in modo diffuso per la risoluzione di molteplici problemi. Le applicazioni più conosciute, sono due dei più usati algoritmi di ordinamento, il quick sort e il merge sort.

**Descrivere le differenze principali tra la tecnica di programmazione divide-et-impera e la tecnica di programmazione dinamica. Nell'ambito della tecnica di programmazione dinamica, cosa si intende con sottostruttura ottima?**

Divide-et-impera → Tecnica ricorsiva, approccio top-down (problemi divisi in sottoproblemi).

Vantaggioso solo quando i sottoproblemi sono indipendenti, altrimenti, gli stessi sottoproblemi possono venire risolti più volte.

Programmazione dinamica → Tecnica iterativa, approccio bottom-up. Vantaggiosa quando ci sono sottoproblemi in comune.

Sottostruttura ottimale → E' possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande. Le decisioni prese per risolvere un problema rimangono valide quando esso diviene un sottoproblema di un problema più grande.

**Si richiede di descrivere sia i passi di cui si compone la tecnica divide-et-impera sia la relazione di ricorrenza che descrive la complessità di algoritmi basati su tale tecnica.** Passi divide et impera: 1. Divide: il problema viene diviso in un certo numero di sottoproblemi. 2. Impera: i sottoproblemi vengono risolti in modo ricorsivo. 3. Combina: le soluzioni vengono combinante per generare la soluzione del problema originale. La complessità pu essere espressa mediante una relazione di ricorrenza che pu essere risolta attraverso tre metodi: sostituzione, albero di ricorsione, metodo dell'esperto. Osservando la relazione di ricorrenza, il bilanciamento della dimensione dei sottoproblemi risulta la chiave di volta per ottenere algoritmi efficienti. La complessità pu essere espressa tramite una relazione di ricorrenza che pu essere risolta applicando uno dei tre modi seguenti:

- Metodo di sostituzione
- Metodo dell'albero di ricorsione
- Metodo dell'esperto

**Dire cosa sono i codici di Huffman e mostrare un esempio.**

Codici di Huffman - vengono usati nella compressione dei dati, permettono un risparmio compreso tra il 20 ed il 90% a seconda del tipo di file. Sulla base delle frequenze con cui ogni carattere appare nel file, l'algoritmo Greedy di Huffman trova un modo ottimale di associare ad ogni carattere una sequenza di bit detta parola codice.

#### Esempio di codice a lunghezza fissa

Sia dato un file di 120 caratteri con frequenze:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5

Usando un codice a lunghezza fissa occorrono 3 bit per rappresentare 6 caratteri. Ad esempio:

carattere	a	b	c	d	e	f
cod. fisso	000	001	010	011	100	101

Per codificare il file occorrono  $120 \times 3 = 360$  bit

#### Esempio di codice di lunghezza variabile

Possiamo fare meglio con un codice a lunghezza variabile che assegna codici più corti ai caratteri più frequenti. Ad esempio con il codice

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. var.	0	101	100	111	1101	1100

Bastano  $57 \times 1 + 49 \times 3 + 14 \times 4 = 260$  bit

**Dire cosa si intende per problema decisionale e problema di ricerca e fornire un esempio per entrambi i tipi di problemi.**

**Problemi decisionali:** Il dato di ingresso soddisfa una certa proprietà?

Esempio: Stabilire se un grafo è connesso

**Problemi di ricerca:** Spazio di ricerca (insieme di soluzioni possibili) e soluzione ammissibile (soluzione che rispetta certi vincoli).

Esempio: posizione di una sottostringa in una stringa.

**Nell'ambito della tecnica di programmazione dinamica, cosa si intende con sottostruttura ottima?**

In informatica, un problema possiede una sottostruttura ottimale se è possibile costruire efficientemente una soluzione ottimale a partire dalle soluzioni ottimali dei suoi sottoproblemi. Questa proprietà è necessaria per poter utilizzare tecniche di programmazione dinamica o algoritmi greedy. Un esempio di sottostrutture ottimali è la ricerca del cammino minimo tra due vertici in un grafo, come illustrato nella figura accanto: prima si trova il cammino minimo dal vertice di arrivo a tutti i vertici vicini a quello di partenza (ripetutamente e con lo stesso metodo); poi si sceglie il cammino che minimizza il peso totale degli archi. Solitamente, un problema con sottostruttura ottimale viene risolto con un algoritmo greedy. Nel caso in cui il problema possieda anche sottoproblemi sovrapponibili si può usare un algoritmo dinamico.

**Nell'ambito della tecnica di programmazione greedy, cosa si intende per soluzione locale ottima? La scelta locale ottima garantisce l'ottenimento di una soluzione globalmente ottima?**

**Motivare la risposta.**

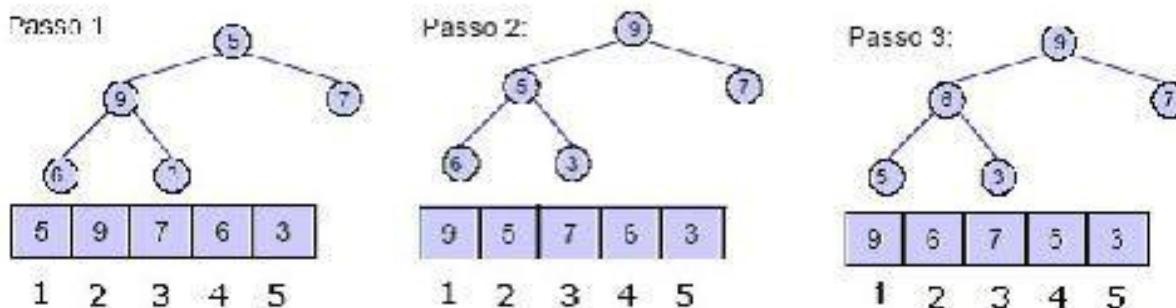
Un algoritmo greedy è un algoritmo che cerca di ottenere una soluzione ottima da un punto di vista globale attraverso la scelta della soluzione più golosa (aggressiva o avida, a seconda della traduzione preferita del termine greedy dall'inglese) ad ogni passo locale. Questa tecnica consente, dove applicabile (infatti non sempre si arriva ad una soluzione ottima), di trovare soluzioni ottimali per determinati problemi in un tempo polinomiale (cfr. Problemi NP-Completi, cioè problemi di soluzione non deterministica polinomiale).

**Cosa vuol dire che un problema A si riduce in tempo polinomiale in un problema B?**

Dati A e B due problemi decisionali, si dice che A si riduce in tempo polinomiale a B se esiste una funzione f di trasformazione tale che: f è computabile in tempo polinomiale con un algoritmo deterministico e x è un dato di ingresso per cui A ha risposta Sì se e solo se f(x) è un dato di ingresso per cui B ha risposta sì.

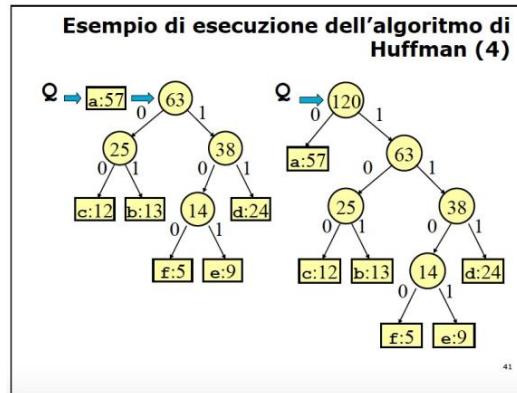
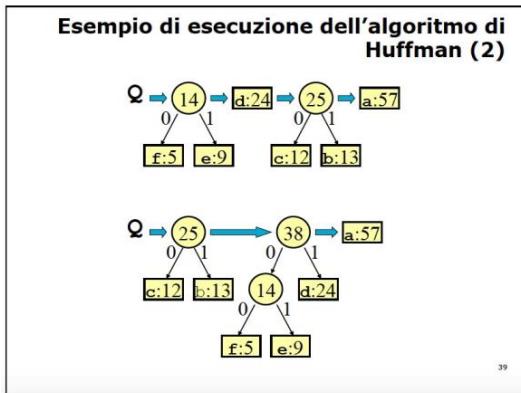
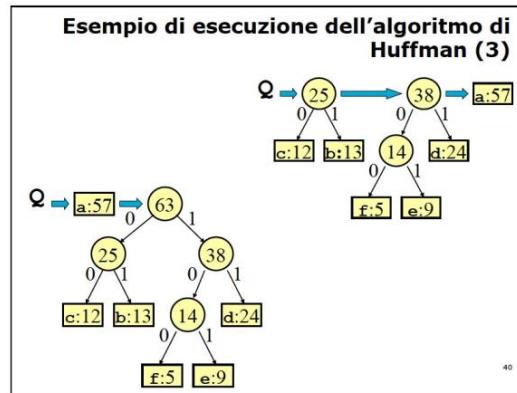
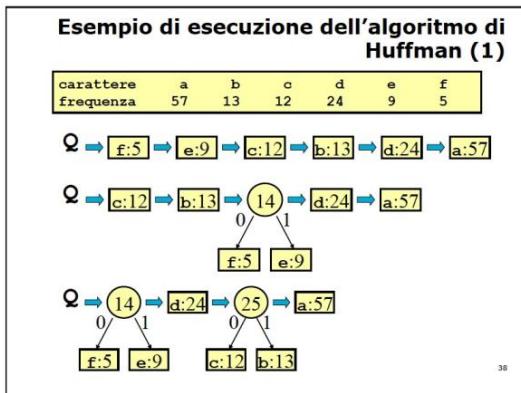
**Descrivere (e mostrare un esempio di funzionamento) la procedura “restauraheap”.**

La funzione restauraheap serve a sistemare quella selezione di A, che rappresenta lo heap, coinvolta dall'estrazione di un massimo.



## Si richiede di descrivere il funzionamento dell'algoritmo greedy per il calcolo di un codice di Huffman.

Principio del codice di Huffman → Minimizzare la lunghezza dei caratteri che compaiono più frequentemente. Assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero. Un codice è progettato per un file specifico, si ottiene la frequenza di tutti i caratteri, si costruisce il codice, si rappresenta il file tramite il codice, si aggiunge al file una rappresentazione del codice. Assumendo che la coda Q venga realizzata con uno heap, le operazioni Insert ed ExtractMin richiedono tempo  $O(\log n)$ . Pertanto l'intero algoritmo richiede tempo  $O(n \log n)$  (dove  $n$  è il numero di caratteri dell'alfabeto).



Si richiede di definire la struttura dati dizionario e di descrivere come pu essere realizzato. Un dizionario è un caso particolare di insieme, in cui gli elementi di un dizionario sono detti chiavi. In un dizionario è possibile: verificare l'appartenenza di una chiave; inserire una nuova chiave; cancellare una chiave. Essendo il dizionario un caso particolare di insieme, la sua specifica è la stessa del tipo insieme limitata agli operatori di creazione, inserisci, appartiene e cancellazione.

**Realizzazione dizionario con vettore ordinato** - Si realizza il dizionario con un vettore nel quale un cursore punta all'ultima posizione occupata e le chiavi sono memorizzate in posizioni contigue e in ordine crescente a partire dalla prima posizione.

**Realizzazione dizionario con tabella hash** - La realizzazione di un dizionario con tabelle hash si basa sul concetto di ricavare direttamente dal valore della chiave la posizione della chiave stessa.

Dato un grafo con  $n$  nodi e  $m$  archi, si richiede di indicare in modo preciso lo spazio di memoria necessario per la rappresentazione del grafo nei seguenti casi:

1. Grafo orientato con matrice di adiacenza
2. Grafo non orientato con matrice di adiacenza
3. Grafo orientato con liste di adiacenza
4. Grafo non orientato con liste di adiacenza

1.  $O(n^2)$
2.  $O(n^2)$
3.  $O(n+m)$
4.  $O(n+2m)$

**Spiegare vantaggi e svantaggi della rappresentazione di grafi tramite matrici di adiacenza o liste di adiacenza. Quando è da preferisci una rappresentazione piuttosto che l'altra?**

La rappresentazione di un grafo  $G=(V,E)$  mediante **liste delle adiacenze** è costituita da una lista  $\text{Adj}[u]$  per ogni vertice  $u \in V$ .

La lista contiene i vertici adiacenti al vertice  $u$  (cioè tutti i vertici  $v$  tali che  $uv \in E$ ).

Sono necessari:

- $|V|$  puntatori alle cime delle liste  $\text{Adj}[u]$
- $|E|$  elementi delle liste se il grafo è orientato
- $2|E|$  elementi delle liste se il grafo non è orientato

La rappresentazione di un grafo  $Gg=(V,E)$  mediante **matrice delle adiacenze** assume che i vertici siano numerati  $1, 2, \dots, |V|$  in modo arbitrario.

La rappresentazione è quindi costituita anche da una matrice booleana  $A = (a_{i,j})$  tale che:

$$a_{i,j} = \begin{cases} 1 & \text{se } ij \in E \\ 0 & \text{se } ij \notin E \end{cases}$$

È quindi necessario una matrice  $A$  di  $|V| \times |V|$  valori booleani.

### CONFRONTO:

#### 1. **Liste di adiacenza**

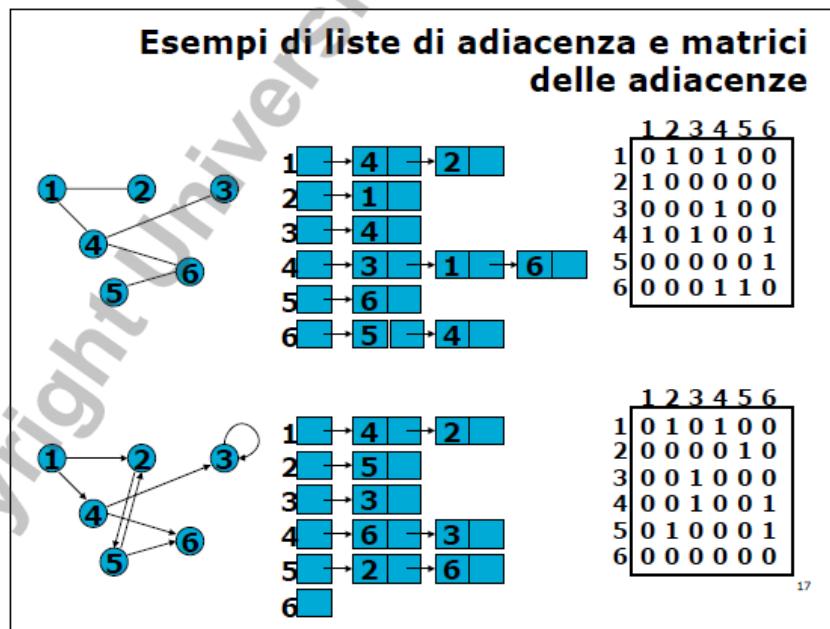
- a. La quantità di memoria richiesta (sia per i grafi orientati che non orientati) è  $O(|V|+|E|)$ ;
- b. Non c'è un modo veloce per determinare se un arco  $uv$  è presente nel grafo  
(L'operazione può richiedere  $O(|V|)$  perché dobbiamo scandire tutta la lista con tutti i vertici)

#### 2. **Matrice di adiacenza:**

- a. La quantità di memoria richiesta (sia per i grafi orientati che non orientati) è  $(O|V|^2)$
- b. Quando il grafo è sparso la matrice è peggiore
- c. Per determinare se un arco  $uv$  è presente nel grafo si richiede tempo costante (accesso diretto)

Dato un grafo, descrivere la sua rappresentazione tramite liste di adiacenza e matrice di adiacenza, illustrando un esempio.

### ESEMPIO



Si richiede di enunciare in modo preciso e completo il teorema dell'esperto, indicando chiaramente i tre casi possibili, quando si applicano e qual è la soluzione dell'equazione di ricorrenza.

Date le costanti  $a \geq 1$  e  $b > 1$  e la funzione  $f(n)$  sia  $T(n)$  una funzione definita sugli interi non negativi della ricorrenza  $T(n) = aT(n/b) + f(n)$  dove  $n/b$  rappresenta  $\lfloor n/b \rfloor$  o  $\lceil n/b \rceil$  allora  $T(n)$  può essere asintoticamente limitata nei seguenti modi:

1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  per qualche costante  $\epsilon > 0$ , allora  $T(n) = \Theta(n^{\log_b a})$
2. Se  $f(n) = \Theta(n^{\log_b a})$ , allora  $T(n) = \Theta(n^{\log_b a} \lg n)$
3. Se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  per qualche costante  $\epsilon > 0$ , e se  $af(n/b) \leq cf(n)$  per qualche costante  $c < 1$  e per ogni  $n$  sufficientemente grande, allora  $T(n) = \Theta(f(n))$

Descrivere in cosa consiste una *rete di flusso* fornendo, in particolare, la definizione formale di flusso ed anche di flusso entrante, uscente e netto. Descrivere inoltre i tre elementi base su cui si fonda l'algoritmo di Ford-Fulkerson per il calcolo del flusso massimo.

Una rete di flusso è un grafo orientato  $G = (V, E)$  ai cui archi è associata una capacità  $c(u, v) \geq 0$   
In una rete di flusso si distingue un vertice  $s$  detto sorgente ed un vertice  $t$  detto pozzo

Assumeremo che ogni vertice sia raggiungibile dalla sorgente e che il pozzo sia raggiungibile da ogni vertice.

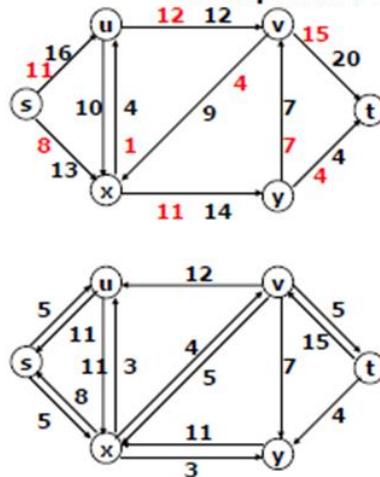
**Nell'ambito del calcolo del flusso massimo in una rete di flusso, cosa si intende per rete residua?**

È una delle 3 parti fondamentali dell'algoritmo di ford fulkerson.

### Rete residua

- Dato un flusso  $f$  in una rete di flusso  $G = (V, E)$  la **capacità residua** di un arco  $uv$  è la quantità  $c_f(u, v) = c(u, v) - f(u, v)$
- La **rete residua**  $G_f = (V, E_f)$  è la rete con archi  $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$  e capacità  $c_f(u, v)$
- Se un arco  $(u, v)$  in  $G$  ha un flusso tale che  $0 < f(u, v) < c(u, v)$ , allora l'arco  $(u, v)$  si sdoppia in  $G_f$ 
  - arco in avanti con  $c_f(u, v) = c(u, v) - f(u, v)$
  - arco all'indietro con  $c_f(v, u) = c(v, u) - f(v, u) = c(v, u) + f(u, v)$

### Esempio di rete residua



13

### Proprietà additiva dei flussi

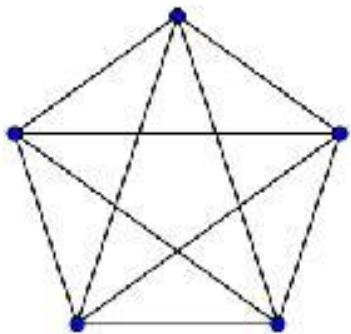
Sia  $f$  un flusso in una rete di flusso  $G = (V, E)$ , sia  $G_f = (V, E_f)$  la rete residua e sia  $g$  un flusso in  $G_f = (V, E_f)$

Allora  $f + g$  è un flusso in  $G = (V, E)$  di valore

$$|f + g| = |f| + |g|$$

**Si richiede di descrivere il problema della cricca e di mostrare un esempio con  $k=5$ .**

Una cricca è un insieme  $V$  di vertici in un grafo non orientato  $G$ , tale che, per ogni coppia di vertici in  $V$ , esiste un arco che li collega.



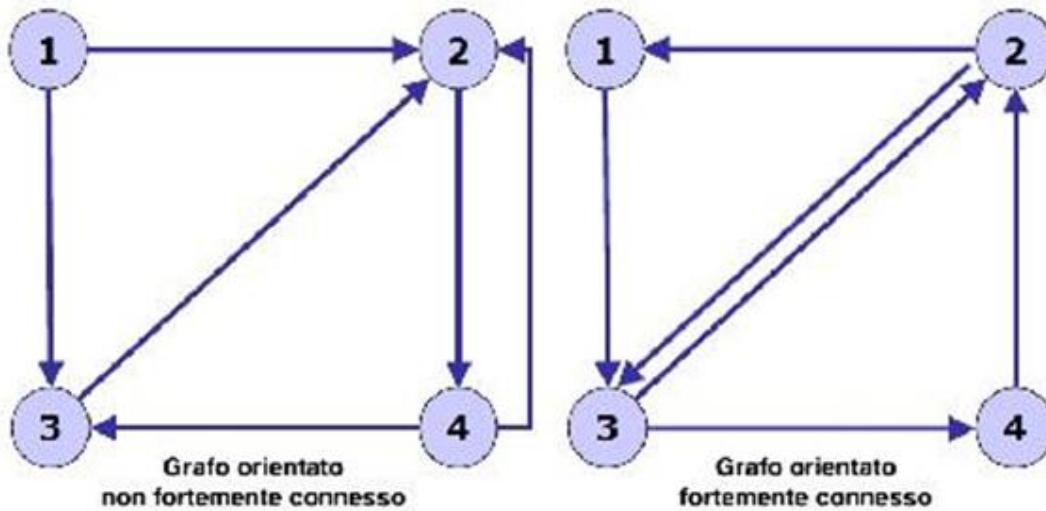
**Dire cosa si intende per algoritmo non deterministico e definirlo in modo formale.**

Un algoritmo non deterministico è un algoritmo che, al momento di effettuare una decisione, effettua sempre quella migliore, ossia quella che porta alla soluzione corretta.

**Dire cosa si intende per grafo orientato fortemente connesso e mostrare un esempio.**

Un grafo orientato si dice fortemente connesso se per ogni coppia di nodi  $u$  e  $v$  esiste almeno un cammino da  $u$  a  $v$  ed almeno un cammino da  $v$  ad  $u$ .

Nel grafo non fortemente connesso esiste un cammino da 1 a 4 ma non viceversa.



**Definire la struttura dati Mfset e dire in cosa consiste la tecnica di compressione dei percorsi mostrando anche un esempio.**

L'Mfset (Merge-Find Set) è una struttura dati derivante dal concetto di insieme delle parti, per cui dato un insieme finito di elementi a volte risulta utile partizionarli in insiemi disgiunti. L'algoritmo Mfset è quindi utile per le operazioni di Ricerca e Unione possibili su questa struttura dati.

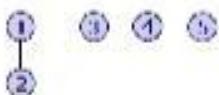
La tecnica di compressione dei percorsi consiste nel rendere figlio della radice ogni nodo che viene incontrato dalla funzione ricerca nel percorso di risalita dal generico nodo  $x$  alla rad.

**Situazione iniziale:**

Mfset con 5 parti

① ② ③ ④ ⑤

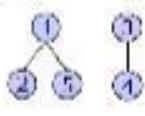
(1) fondi(1,2,5)



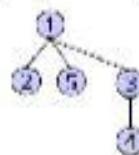
(2) fondi(3,4,5)



(3) fondi(1,5,5)



(4) fondi(5,4,5)

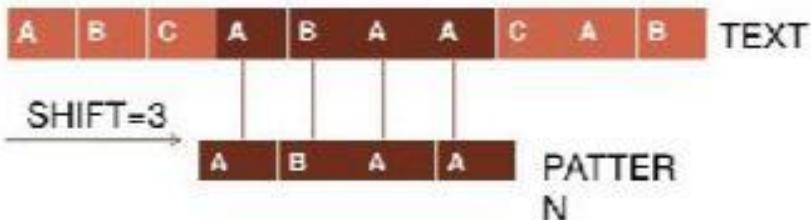


## Dire in cosa consiste il problema dello String Matching e quale è la complessità dell'algoritmo KMP

Il problema dello String Matching consiste nel trovare almeno una occorrenza di una sequenza P di m caratteri, detta pattern, all'interno di un'altra sequenza T di n caratteri detta testo.

Le due sequenze o stringhe P e T sono formate da caratteri tratti dallo stesso insieme A, di cardinalità finita detto "alfabeto", e sono tale che m non supera n.

L'algoritmo KMP (Knuth–Morris–Pratt) ha complessità  $O(n + m)$



## In che cosa consiste il problema dello String Matching approssimato? Si richiede di mostrare un esempio.

Il problema di String Matching Approssimato consiste nel ricercare un pattern P all'interno di un testo T ammettendo errori (o differenze) tra T e P. I possibili errori sono:

- i corrispondenti caratteri in P e in T sono diversi
- un carattere in P non compare in T
- un carattere in T non compare in P

## Si richiede di descrivere il problema che consiste nel trovare un'occorrenza k-approssimata di un pattern P nel testo T. Mostrare un esempio di occorrenza 2-approssimata.

Lo String Matching approssimato consiste nel cercare un pattern P all'interno di un testo T ammettendo errori tra T e P. Il problema quindi consiste nel trovare un'occorrenza k-approssimata di P in T.

k-approssimata saranno quindi il numero degli errori.

T = abcdefghi

P = dafghj

k = 2

## Nell'ambito della teoria della complessità, si richiede di definire le classi P e NP.

Nell'ambito della teoria della complessità, la classe P è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi deterministici, mentre la classe NP è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi non deterministici.

Ovviamente la classe P è contenuta nella classe NP, poiché ogni algoritmo deterministico è anche un algoritmo non deterministico (che semplicemente non usa mai choice), ma non si sa se P sia propriamente contenuta in NP oppure se le due classi coincidono.

**Nell'ambito della teoria della complessità dire cosa si intende per certificato polinomiale.**

Un certificato polinomiale è un algoritmo che, data una presunta soluzione del problema, verifica in tempo polinomiale che tale soluzione sia effettivamente una soluzione che dà risposta affermativa.

**Dato un grafo orientato  $G$  cosa si intende per albero  $T$  di copertura DFS? Quando un arco di  $T$  si definisce arco all'indietro, in avanti o di attraversamento?**

Un albero di copertura è un albero formato dagli  $n$  nodi e da  $n-1$  archi del grafo. Un albero di copertura DFS è un albero composto dagli archi che durante una visita DFS connettono un nodo marzato ad uno non marzato; nodi ed archi formano un albero radicato  $T$ .

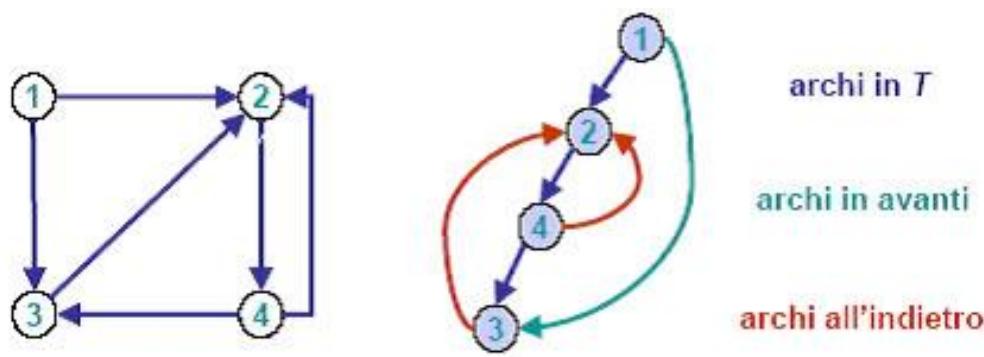
Se il grafo è orientato la DFS partiziona gli archi in 4 sottoinsiemi:

Archi in  $T$ : archi che collegano un nodo marzato ad uno non marzato

Archi all'indietro: archi esaminati passando da un nodo di  $T$  ad un altro loro antenato

Archi in avanti: archi esaminati passando da un nodo di  $T$  ad un altro loro discendente in  $T$

Archi di attraversamento: archi fra nodi che stanno sullo stesso livello in  $T$



**Si richiede di enunciare il Teorema di Cook-Levin e di descrivere le sue conseguenze.**

Il teorema risponde alla domanda: "Esiste un particolare problema decisionale in NP tale che se si dimostrasse la sua appartenenza a P, allora dovrebbe risultare sicuramente  $P = NP$ ?"

"Ogni problema in NP si riduce in tempo polinomiale al Dominio Limitato"

Di conseguenza:

$P=NP$  se e solo se Dominio Limitato  $\in P$

**A cosa serve l'algoritmo di Prim? Qual'è la sua complessità?**

L'algoritmo Prim serve a determinare come interconnettere diversi elementi fra loro minimizzando certi vincoli sulle connessioni.

Esempio: Progettare circuiti elettronici minimizzando la quantità di filo elettrico per collegare fra loro i diversi componenti

L'algoritmo di Prim procede mantenendo in  $A$  un singolo albero

L'albero parte da un vertice arbitrario  $r$  (la radice) e cresce fino a quando non ricopre tutti i Vertici

Ad ogni passo viene aggiunto un arco leggero che collega un vertice in  $VA$  con un vertice in  $V \setminus VA$  – dove  $VA$  è l'insieme di nodi raggiunti da archi in  $A$

Correttezza ( $VA, V \setminus VA$ ) è un taglio che rispetta  $A$  (per definizione)

Per il corollario, gli archi leggeri che attraversano il taglio sono sicuri

I vertici sono aumentati con:

- Un puntatore  $p$  al padre nell'albero in costruzione

- Un campo colore che è bianco inizialmente e diventa nero quando il vertice viene aggiunto all'albero in costruzione
- Un campo key che contiene il costo minimo di un arco che connette il vertice ad uno dei vertici già raggiunti dall'albero in costruzione

Dipende da come viene implementata la coda Q: supponiamo che sia implementata tramite un minheap:

L'inizializzazione richiede  $O(|V|)$

Il ciclo while viene eseguite  $|V|$  volte e poiché ExtractMin richiede  $O(\lg|V|)$ , il tempo totale per queste operazioni è  $O(|V|\lg|V|)$

Il ciclo for interno al while viene eseguite  $O(|E|)$  volte e l'operazione di decremento della chiave richiede  $O(\lg|V|)$

Tempo totale:  $O(|V|\lg|V| + |E|\lg|V|)$

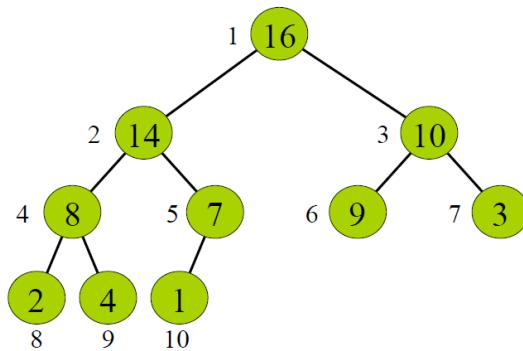
**Cosa è un max-heap? Si richiede di fornire un esempio.**

E' una proprietà dello Heap, la quale impone che per ogni nodo i diverso dalla radice si verifichi:

$A[\text{parent}(i)] \geq A[i]$

### Esempio di Max-Heap

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Un Heap è una struttura dati composta da un array che possiamo considerare come un albero binario, ogni nodo dell'albero corrisponde ad un elemento dell'array. Gli heap possono essere suddivisi in Max-Heap e Min-Heap, in un max heap le chiavi/radici di ciascun nodo sono sempre maggiori o uguali di quelle dei figli, la chiave del valore massimo appartiene alla radice

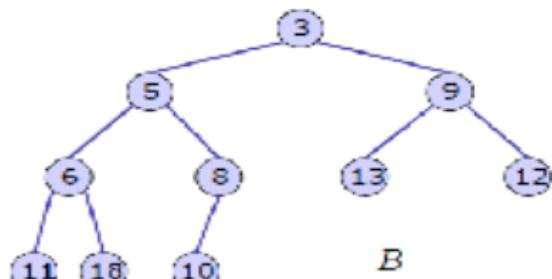
**Si richiede di descrivere la realizzazione di una coda di priorità con uno heap e di mostrare un esempio sul quale si richiede anche di eseguire l'operazione di cancellamin.**

Una coda di priorità è una struttura dati che serve a mantenere un insieme  $S$  di elementi, ciascuno con un valore associato detto chiave.

È simile ad una coda, ma si differisce da questa in quanto ogni elemento inserito all'interno della coda possiede una sua "priorità".

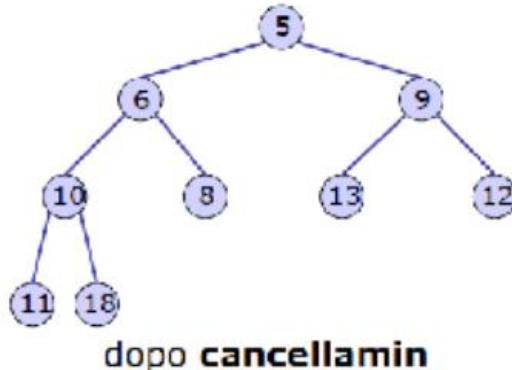
In una coda di priorità, ogni elemento avente priorità più alta, viene inserito prima rispetto ad un elemento avente priorità più bassa. In particolare, l'elemento con priorità più alta si trova in testa alla coda, quello con priorità più bassa si troverà, appunto, in coda.

Prendiamo in esempio il seguente albero:



Per eseguire “cancellamin” si cancella la foglia di livello massimo più a destra (nell’esempio il valore 10)

se ne copia l’ elemento nella radice,



e si fa “scendere” tale elemento lungo un percorso radice-foglia,

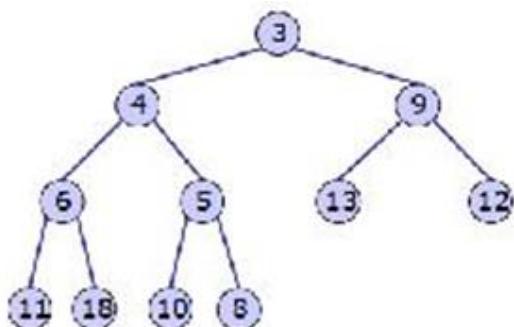
scambiando col minimo degli elementi contenuti nei figli.

la relativa coda di priorità è:

Priorità	Valore
1	3
2	5
3	9
4	6
5	8
6	13
7	12
8	11
9	18
10	10

Per eseguire “inserisci” si inserisce l’elemento come foglia più a destra del livello massimo (in modo da mantenere verificate le proprietà 1 e 2) e si fa “salire” il nuovo nodo con scambi padrefiglio lungo un cammino foglia – radice fino a verificare la proprietà 3. Nel seguente esempio viene inserito l’elemento 4.

Per eseguire “inserisci” si inserisce l’elemento come foglia più a destra del livello massimo (in modo da mantenere verificate le proprietà 1 e 2) e si fa “salire” il nuovo nodo con scambi padrefiglio lungo un cammino foglia – radice fino a verificare la proprietà 3. Nel seguente esempio viene inserito l’elemento 4.



**dopo inserimento di 4**

**Quale `e il costo di operazioni insiemistiche (unione, intersezione, differenza) con insiemi implementati con liste linkate? Può cambiare qualcosa se le liste sono ordinate?**  
**Si richiede di definire formalmente la notazione “o piccolo” e di fare un esempio.**

I costi delle operazioni insiemistiche con insiemi implementati con le liste sono  $O(nm)$  se non sono ordinate, mentre  $O(n)$  se sono ordinate

**Si richiede di definire formalmente la notazione “o piccolo” e di fare un esempio.**

Usata per denotare il limite superiore non asintoticamente stretto.

$f(n)=o(g(n))$  si legge  $f(n)$  è "o piccolo di  $g$  di  $n$ "

Esempio:  $2n=o(n^2)$  e  $2n^2 \neq o(n^2)$

Nella notazione o la funzione  $f(n)$  diventa insignificante rispetto a  $g(n)$  quando  $n$  tende all'infinito

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

**Data una rete di flusso, quali proprietà deve soddisfare la funzione  $f$  di flusso?**

### **Definizione formale di flussi (1)**

Sia  $G = (V, E)$  una rete di flusso con una funzione capacità  $c$ . Sia  $s$  la sorgente della rete e sia  $t$  il pozzo

Un flusso in  $G$  è una funzione a valori reali  $f: V \times V \rightarrow \mathbb{R}$  che soddisfa le seguenti tre proprietà:

- **vincolo sulla capacità:**  $f(u, v) \leq c(u, v)$ ;
- **antisimmetria:**  $f(u, v) = -f(v, u)$ ;
- **conservazione del flusso:** per ogni  $u \in V \setminus \{s, t\}$ :

$$\sum_{v \in V} f(u, v) = 0$$

**In cosa consiste la tecnica *tail recursion*? Si richiede di mostrare un esempio.**

Una tecnica semplice per eliminare la Ricorsione è la Tail Recursion.

Una procedura p presenta tail recursion quando:

- o ha un parametro x passato per valore
- o la sua ultima istruzione è una chiamata di p (si noti che questo non vuol dire che la chiamata a p sia l'ultima cosa scritta - vedi esempio lucido seguente) ricorsiva con valore y del parametro di chiamata

Si può eliminare la chiamata ricorsiva sostituendola con un assegnamento di y a x ed un goto alla prima istruzione della procedura

### Esempi di Procedure con Chiamate Terminali

---

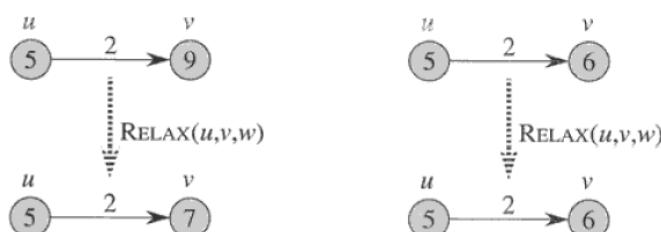
```
p(...)  
istr1  
istr2  
.....  
q(...)//* terminale */  q(...)//* non terminale */  
istrn  
  
p(...)  
istr1  
if(...)  
then  
  istr2  
  q(...)//* terminale */  
else  
  istr3
```

---

**Nell'ambito degli algoritmi per il calcolo dei cammini minimi, in cosa consiste il *processo di rilassamento* di un arco  $(u,v)$ ?**

Il processo di rilassamento di un arco  $(u,v)$  consiste nel verificare se passando per  $u$  è possibile migliorare la stima del cammino minimo per  $v$  e in caso affermativo nell'aggiornare i valori sia di  $d[v]$  sia di  $\pi[v]$ .

### Esempio di rilassamento



## Cosa si intende per hashing uniforme semplice?

Supponiamo che la funzione hash  $h(k)$  distribuisce uniformemente le  $n$  chiavi tra le  $m$  liste

### Hash uniforme semplice:

ogni chiave da inserire nella tavola è estratta da  $U$ , con una certa distribuzione di probabilità, in modo casuale ed indipendente dalle chiavi inserite in precedenza

la funzione hash è tale che ogni chiave ha la stessa probabilità  $1/m$  di essere messa in una qualsiasi delle  $m$  celle

## Cosa si intende per tecnica di Backtrack? Mostrare un esempio di algoritmo basato su tale tecnica.

La tecnica di Backtrack viene sfruttata per la progettazione di algoritmi e si basa sul concetto di costruzione ed eventuale distruzione di parte della soluzione, più semplicemente possiamo dire che l'algoritmo prova a fare qualcosa se il risultato non è quello previsto annulla tutto e riprova.

Un algoritmo è di tipo BackTrack quando in esso sono previsti strumenti per la costruzione della soluzione che per la distruzione di parte di essa.

Questo algoritmo viene sfruttato nella tecnica di String Matching.

## Si richiede di definire il problema della massima diversità. Quale è la complessità dell'algoritmo Greedy più efficiente per la soluzione di tale problema? L'uso di un heap migliora la complessità (giustificare la risposta)?

La tecnica di progettazione di algoritmi di tipo Greedy, si basa sulla semplice strategia dell'ingordo, ovvero, quella di compiere, ad ogni passo, la scelta migliore nell'immediato piuttosto che adottare una strategia a lungo termine.

Si supponga di avere monete da 11, 5 e 1 centesimo, e di dover dare un resto di 15 centesimi. L'ingordo giudicherà la moneta da 11 cent più "appetibile" delle altre, e darà un resto con una moneta da 11 e quattro da 1, utilizzando cinque monete, mentre la soluzione ottima è data da tre monete da 5 centesimi.

Il metodo greedy si può applicare a quei problemi di ottimizzazione in cui occorre selezionare un sottoinsieme  $S$  "ottimo" di oggetti.

## Descrivere le quattro fasi che caratterizzano la progettazione di un algoritmo.

**classificazione del problema:** decisionale, di ricerca, di ottimizzazione.

Si cerca di verificare l'appartenenza di un problema ad una classe più generale avente caratteristiche comuni.

**Caratterizzazione della soluzione:** individuare proprietà matematiche nella soluzione.

La caratterizzazione matematica della soluzione, quando è possibile, suggerisce algoritmi di soluzione, talvolta semplici

**tecnica di progetto:** divide et impera, backtracking, greedy, PD, ricerca locale.

Esistono delle tecniche di progetto di algoritmi che possono rendere gli algoritmi più efficienti.

**impiego di opportune strutture dati** per migliorare l'efficienza dell'algoritmo.

L'impiego di opportune strutture di dati per organizzare l'input del problema può migliorare l'efficienza di un dato algoritmo.

## **A cosa serve l'algoritmo di Pape-D'Esopo? Cosa si intende per grafo planare? Quale è la sua complessità?**

L'algoritmo di Pape-D'Esopo, per il calcolo dei cammini minimi, usa come insieme S una dequeue (double ended queue). Una dequeue è una coda che ammette anche l'inserzione in testa, e non solo in coda. Il generico nodo u verrà inserito la prima volta in coda ed in testa le volte successive. Più precisamente, se  $d[u] == \text{MAXINT}$ , allora u è inserito in coda, altrimenti in testa. L'idea dell'inserimento in testa è quella di sfruttare immediatamente il miglioramento dell'etichetta affinché esso si propaghi ai nodi vicini.

L'algoritmo, nel caso pessimo, ha complessità superpolinomiale!!

Tuttavia, si è sperimentalmente verificato essere uno dei più efficienti su grafi che modellano reti di comunicazione stradali, ovvero su grafi sparsi e planari. Un grafo è planare quando, disegnato su un piano, le linee corrispondenti a due archi distinti non si sovrappongono mai.

## **Cosa è il perno utilizzato nell'algoritmo Quicksort? Quale è la complessità computazionale nel caso peggiore e medio di tale algoritmo?**

Il perno, o pivot, è l'elemento chiave dell'algoritmo di Quicksort. Gli elementi minori dell'array si porranno a sinistra del pivot mentre gli elementi maggiori a destra, l'operazione di ordinamento si reitera sugli insieme risultati fino al completamento dell'ordinamento.

Nel caso pessimo il pivot è sempre elemento più piccolo, e quindi la porzione A con elementi minori di x è vuota, mentre quella con elementi maggiori o uguali ne contiene  $n - 1$ . Supponiamo che questo sbilanciamento si verifichi in ogni chiamata ricorsiva. Il partizionamento costa  $\Theta(n)$  in termini di tempo, applichiamo il teorema delle ricorrenze lineari di ordine costante il tempo di esecuzione è:  $O(n^2)$

Il caso migliore si verifica quando l'algoritmo di partizionamento determina due sottoproblemi perfettamente bilanciati (ad esempio: 5,6,7,8,9,1,2,3,4), entrambi di dimensione  $n/2$ ; in questo caso il tempo di esecuzione è  $O(n \log n)$

## **Nell'ambito delle tabelle di hash, cosa si intende per indirizzamento aperto?**

Con la tecnica di indirizzamento aperto tutti gli elementi vengono memorizzati nella tavola, la funzione di hash non individua una singola cella ma una sequenza esaustiva di esse. L'inserimento di un nuovo elemento avviene nella prima cella libertà che si incontra nella sequenza.

## **Descrivere il funzionamento dell'algoritmo di Moore che fa uso di uno heap modificato. Qual è la complessità dell'algoritmo?**

Viene utilizzato per risolvere problemi di scheduling, in cui si hanno dei programmi da eseguire su un processore e si vuole trovare l'ordine di esecuzione ottimo in base ad un prefisso criterio. Ovvero dato il seguente problema: dati  $n$  programmi  $p$  tali che ciascun richiede unità di tempo per la sua esecuzione e deve essere completato entro una certa scadenza, trovare un ordine  $S$  in cui eseguire tutti i programmi in modo da minimizzare il numero di programmi per i quali la scadenza non è rispettata. L'algoritmo di Moore risolve questo tipo di problema. Una realizzazione poco accorta dell'algoritmo porta ad una complessità di  $O(n^2)$ , ma può essere migliorata attraverso l'uso di una coda di priorità  $Q$  implementata con uno heap modificato (per trattare programmi), ottenendo una complessità  $O(n \log n)$ .

Nel caso pessimo avremo  $O(n)$  eliminazioni  $\Rightarrow$  algoritmo è  $O(n^2)$