SOMMARIO

ORDINI DI GRANDEZZA ED EQUAZIONI RICORSIVE

Scala degli ordini di grandezza

logaritmico	*Polinomiale (sublineare)*	*Polinomiale (lineare)*	*Polinomiale (loglineare)*	*Polinomiale (quadratico)*	*Polinomiale (cubico)*	*Esponenziale*	*n!*
lg n	$\sqrt{n}$	n	n lg n	n^2	n^3	2^n	n!

➔ ➔ ➔ ➔ ➔ ➔ ➔

Notazione:

$$\left.\begin{matrix} O(f(n)) \rightarrow \geq f(n) \\ \Omega(f(n)) \rightarrow \leq f(n) \end{matrix}\right\} \rightarrow \Theta(f(n)) \quad soluzione\ in\ ordine\ di\ grandezza$$

SOMMATORIE NOTEVOLI

$$Serie\ aritmetica \sum_{k=1}^{n} k = 1 + 2 + \cdots + k = \frac{k(k+1)}{2} = \Theta(n^2)$$

$$log \prod_{i=0}^{n} i = \log(n!) = O(n \log(n))$$

Nota: logaritmi: cambiare base corrisponde a moltiplicare per una costante, quindi si può togliere (nel calcolo ordine di grandezza)

Esempio

$$T(n) = \begin{cases} caso\ base \\ T(n-1) + \log(n) \end{cases} \quad T(n) = \log(n) + T(n-1) = \log(n) + \log(n-1) + T(n-2) + \ldots = \sum_{i=0}^{m} \log(i) = O(n \log(n))$$

$$\log(1) + \log(2) + \log(3) + \ldots + \log(n) = \log(1 * 2 * 3 * \ldots * n) = log \prod_{i=0}^{n} i = \log(n!) = O(n \log(n))$$

Esempio

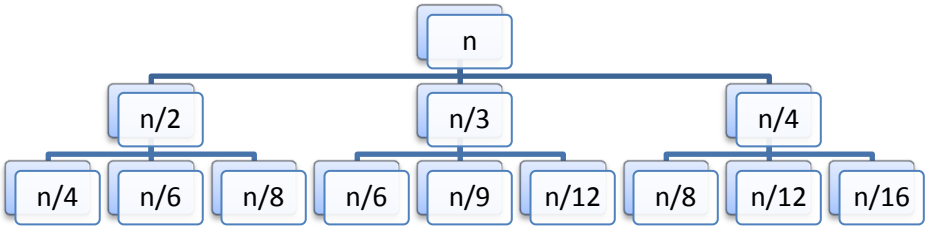
$$T(n) = \begin{cases} caso\ base \\ T(n-1) + \log(n^2) \end{cases} \quad T(n) = \log(n^2) + T(n-1) = \log(n^2) + \log((n-1)^2) + T((n-2)^2) + \ldots = \sum_{i=0}^{m} \log(i^2) = \Theta(n \log(n))$$

$$\log(1^2) + \log(2^2) + \log(3^2) + \ldots + \log(n^2) = \log(1 * 2^2 * 3^2 * \ldots * n^2) = log \prod_{i=0}^{n} i^2 = 2\ log \prod_{i=0}^{n} i = 2\ \log(n!) = \Theta(n \log(n))$$

Cit. "il logaritmo è una roba che ammazza le cose che le diamo in pasto".

Esempio

$$T(n) = \begin{cases} caso\ base \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + 1 \end{cases}$$



Si osserva che questa equazione scende più rapidamente verso i rami di destra (n, n/4, n/16, n/64, ...) che verso i rami di sinistra (n, n/2, n/4, ...).

Quindi le chiamate ricorsive massime (numero di chiamate più alto) si trovano lungo i rami di sinistra.

Quindi le chiamate ricorsive minime (numero di chiamate più basso) si trovano lungo i rami di destra.

Si può concludere che il costo dell'algoritmo si trovi tra il valore minimo (destra) e massimo (sinistra).

Allora: $T'(n) \coloneqq 3 \cdot T\left(\frac{n}{2}\right) + 1 \geq T(n) \geq 3 \cdot T\left(\frac{n}{4}\right) + 1 =: T''n$

$$\left.\begin{array}{l} T'(n) = 3 \cdot T\left(\frac{n}{2}\right) + 1 = 1 + 3 + 3 \cdot T\left(\frac{n}{4}\right) = 1 + 3 + 9 + 3 \cdot T\left(\frac{n}{8}\right) = 3^0 + 3^1 + 3^2 + 3^3 + \cdots + 3^{\log_2 n} = 3^{\log_2 n + 1} = \Theta\left(3^{\log_2 n}\right) \\ T''(n) = 3 \cdot T\left(\frac{n}{4}\right) + 1 = 1 + 3 + 3 \cdot T\left(\frac{n}{16}\right) = 1 + 3 + 9 + 3 \cdot T\left(\frac{n}{64}\right) = 3^0 + 3^1 + 3^2 + 3^3 + \cdots + 3^{\log_4 n} = 3^{\log_4 n + 1} = \Theta\left(3^{\log_4 n}\right) \end{array}\right\} \Rightarrow$$

$$T'(n) \geq T(n) \geq T''(n) \Rightarrow \Theta\left(3^{\log_2 n}\right) \geq T(n) \geq \Theta\left(3^{\log_4 n}\right)$$

$$\Rightarrow \Theta\left(3^{\log_2 n}\right) \geq T(n) \geq \Theta\left(3^{\log_4 n}\right) \Rightarrow \Theta\left(3^{\log_3 n^{\log_2 3}}\right) \geq T(n) \geq \Theta\left(3^{\log_3 n^{\log_4 3}}\right) \Rightarrow \Theta\left(n^{\log_2 3}\right) \geq T(n) \geq \Theta\left(n^{\log_4 3}\right)$$

A questo punto bisogna valutare $\log_2 3$ e $\log_4 3$: $\log_2 3 \cong 1{,}58 \quad \log_4 3 \cong 0{,}79$

Esempio

$$T(n) = \begin{cases} caso\ base \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + n \end{cases}$$

$$\left.\begin{aligned} T'(n) = 3 \cdot T\left(\frac{n}{2}\right) + n = n + 3n + 3 \cdot T\left(\frac{n}{4}\right) = n + 3n + 9n + 3 \cdot T\left(\frac{n}{8}\right) = 3^0 n + 3^1 n + 3^2 n + 3^3 n + \cdots + 3^{\log_2 n} n = \mathrm{n}\, 3^{\log_2 n + 1} = \Theta\left(\mathrm{n}\, 3^{\log_2 \mathrm{n}}\right) \\ T''(n) = 3 \cdot T\left(\frac{n}{4}\right) + n = 1 + 3n + 3 \cdot T\left(\frac{n}{16}\right) = n + 3n + 9n + 3 \cdot T\left(\frac{n}{64}\right) = 3^0 n + 3^1 n + 3^2 n + 3^3 n + \cdots + 3^{\log_4 n} n = \mathrm{n}\, 3^{\log_4 n + 1} = \Theta\left(\mathrm{n}\, 3^{\log_4 \mathrm{n}}\right) \end{aligned}\right\} \Rightarrow$$

$$T'(n) \geq T(n) \geq T''(n) \Rightarrow \Theta\left(\mathrm{n}\, 3^{\log_2 \mathrm{n}}\right) \geq T(n) \geq \Theta\left(\mathrm{n}\, 3^{\log_4 \mathrm{n}}\right)$$

$$\Rightarrow \Theta\left(\mathrm{n}\, 3^{\log_2 \mathrm{n}}\right) \geq T(n) \geq \Theta\left(\mathrm{n}\, 3^{\log_4 \mathrm{n}}\right) \Rightarrow \Theta\left(\mathrm{n}\, 3^{\log_3 \mathrm{n}^{\log_2 3}}\right) \geq T(n) \geq \Theta\left(\mathrm{n}\, 3^{\log_3 \mathrm{n}^{\log_4 3}}\right) \Rightarrow \Theta\left(n\, n^{\log_2 3}\right) \geq T(n) \geq \Theta\left(n\, n^{\log_4 3}\right)$$

A questo punto bisogna valutare $\log_2 3$ e $\log_4 3$: $\log_2 3 \cong 1{,}58$ $\log_4 3 \cong 0{,}79$

Esempio

$$T(n) = \begin{cases} caso\ base \\ T(c) + T(n - c) \quad c\ costante \end{cases}$$

$T(n) = T(c) + T(n - 2c) + T(n - 3c) + \cdots =$ $T(n - ic)$ $i \coloneqq numero\ di\ passi.$ $n - ic \leq k \rightarrow n - k \leq ic \rightarrow \frac{n - k}{c} \leq i$

La ricorsione è lunga un numero di passi necessario ad azzerare n, siccome ogni volta, da n, sottraggo un valore costante (sempre maggiore), la ricorsione sarà lunga ordine di n passi, ogni passo della ricorsione costa tempo costante, quindi l'equazione è ordine di n.

$T(n) = \frac{n - k}{c} T(c)$ *se c è costante* $\rightarrow$ $\frac{n - k}{c} T(c) = \Theta(\mathrm{n})$

Esempio

$$T(n) = \begin{cases} caso\ base \\ T(\log n) + T(n - \log n) \end{cases}$$

$T(n) = T(\log n) + T(n - \log n) + T(n - \log n) + \cdots =$ è *troppo complicata*

Esempio

$$T(n) = \begin{cases} caso\ base \\ T\left(\frac{n}{100}\right) + T\left(n - \frac{n}{100}\right) + c \quad c\ costante \end{cases}$$

$$T(n) = T\left(\frac{n}{100}\right) + T\left(n - \frac{n}{100}\right) + c = T\left(\frac{n}{100}\right) + T\left(\frac{99n}{100}\right) + c = T\left(\frac{n}{100^2}\right) + T\left(\frac{99n}{100^2}\right) + T\left(\frac{99n}{100^2}\right) + T\left(\frac{9901n}{100^2}\right) + 2c$$

i due rami della ricorsione scendono con rapidità differenti: $\mathrm{T}\left(\frac{\mathrm{n}}{100}\right)$ ha altezza $\log_{100} n$; $\mathrm{T}\left(\mathrm{n} - \frac{\mathrm{n}}{100}\right)$ ha altezza $\log_{100/99} n \rightarrow \Theta(\log n)$

Un'equazione ricorsiva esprime il valore di una funzione f(n) come combinazione di $f(n_i)$ dove $n_i < n \quad \forall i$

Esempio

$$T(n) = \begin{cases} caso\ base \\ T\left(\frac{n}{2}\right) + \log n \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + log\, n = log\, n + log\left(\frac{n}{2}\right) + log\left(\frac{n}{4}\right) + log\left(\frac{n}{8}\right) = \sum_{i=0}^{log\, n} log\left(\frac{n}{2^i}\right) = \sum_{i=0}^{log\, n} (log\, n - log\, 2^i) = \sum_{i=0}^{log\, n} log\, n - \sum_{i=0}^{log\, n} log\, 2^i =$$

$$= (log\, n + log\, n + log\, n + \cdots) - (0 + 1 + 2 + \cdots + n) = log\, n + log\, n - 1 + log\, n - 2 + log\, n - 3 + \cdots + log\, n - n - 1 = \Theta(log^2\, n)$$

Esercizio

$$T(n) = \begin{cases} caso\ base \\ T(n-1)^{101} + T(n-2)^{101} + T(n-3)^{101} \quad ???\ Nessun\ tempo\ per\ T(1)\ ??? \end{cases}$$

$$\mathrm{T(n)} = \mathrm{T(n-1)}^{101} + T(n-2)^{101} + T(n-3)^{101} =$$

$$= \mathrm{T(n-2)}^{101} + T(n-3)^{101} + T(n-4)^{101} + \mathrm{T(n-3)}^{101} + T(n-4)^{101} + T(n-5)^{101} + \mathrm{T(n-4)}^{101} + T(n-5)^{101} + T(n-6)^{101} =$$

L'albero della ricorsione, a sinistra è alto n, a destra è alto n/3.

$$\mathrm{T'}(n) = \mathrm{T(n-1)}^{101} = \sum_{\mathrm{i=0}}^{\mathrm{n}} \mathrm{i}^{101} = \frac{1 - \mathrm{i}^{\mathrm{n+1}}}{1 - \mathrm{i}} \qquad \mathrm{T''(n)} = \mathrm{T(n-3)}^{101} = \sum_{\mathrm{i=0}}^{\mathrm{n/3}} (3\mathrm{i})^{101} = 3 \sum_{\mathrm{i=0}}^{\mathrm{n/3}} (\mathrm{i})^{101} = 3\frac{1 - (\mathrm{i})^{\frac{\mathrm{n}}{3}+1}}{1 - \mathrm{i}}$$

In entrambi i casi pertanto, l'ordine di grandezza dipende da i, in quanto la serie converge o diverge se i è <=1 o >1, rispettivamente.

Esercizio

$$T(n) = \begin{cases} caso\ base \\ T(n-1)^2 + 1 \end{cases}$$

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

Esercizio

$$T(n) = \begin{cases} caso\ base \\ T(c) + T(n-c) + n^2 \end{cases}$$

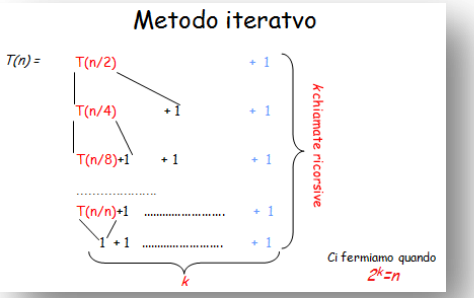
$$T(n) = T(c) + T(n-c) + n^2 = \ = \sum_{i=0}^{\frac{n}{c}} (n-ic)^2$$

$$Provo\ per\ c = 1: \sum_{i=0}^{n} (n-ic)^2 = n^2 + (n-1)^2 + (n-2)^2 + \cdots + 1 + 0 = \sum_{i=0}^{n} i^2 = 1^2 + 2^2 + 3^3 + \cdots + \left(\frac{n}{2}\right)^2 + \cdots + (n-1)^2 + n^2$$

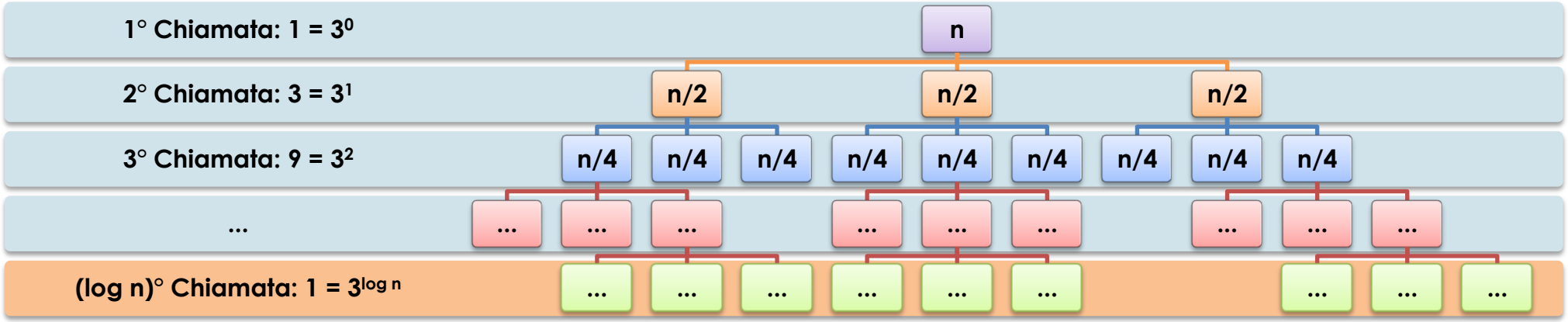
$$1^2 + 2^2 + 3^3 + \cdots + \left(\frac{n-1}{2}\right)^2 < \left(\frac{n}{2}\right)^2 \rightarrow \ \left(\frac{n}{2}\right)^2 * \frac{n}{2} = \frac{n^3}{8} \leq \sum_{i=0}^{n} i^2 \leq n^3$$

Esempio

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ 1 + 3T\left(\frac{n}{2}\right) & se\ n \neq 1 \end{cases}$$



Si può disegnare l'albero delle chiamate e ragionare su quante volte la funzione è richiamata ad ogni passaggio.



Allora risulta che $T(n) = \sum_{i=0}^{\log_2 n} 3^i = 3^0 + 3^1 + 3^2 + \ldots + 3^{\log_2 n}$ é possibile verificare che l'ultimo termine: $3^{\log_2 n}$, è maggiore di tutti i termini precedenti:

Si vuole dimostrare che: $\sum_{i=0}^{k} 3^i < 3^{k+1}$ si procede per induzione:

Passo base: $k = 0 \Rightarrow 3^0 < 3^1$ ✓

Passo di induzione: $\underbrace{\sum_{i=0}^{k} 3^i < 3^{k+1}}_{\text{(ipotesi induttiva)}} \Rightarrow \sum_{i=0}^{k+1} 3^i < 3^{k+2}$ si ha che: $\sum_{i=0}^{k+1} 3^i = \sum_{i=0}^{k} 3^i + 3^{k+1} \overset{\text{Ipotesi induttiva}}{<} 3^{k+1} + 3^{k+1} = 2\left(3^{k+1}\right) < 3\left(3^{k+1}\right) = 3^{k+2}$ ✓

TABELLA DEGLI ALGORITMI

Argomento	Nome	Cosa fa	Costo
Ordinamento	InsertionSort	Ordina un vettore	$\Theta(n^2)$ (best case $\Theta(n)$) "in place"
	MergeSort	Ordina un vettore, ricorsivo	$\Theta(n \log n)$
Ricerca	BinarySearch	Trova un elemento in un vettore	$\Theta(\log n)$
Code di priorità implementate con vettori ordinati	RicercaMassimo	Ricerca il massimo in un vettore ordinato	$\Theta(1)$
	EstrazioneMassimo	Estrae il massimo in un vettore ordinato	$\Theta(1)$
	Inserimento	Inserisce in un vettore nella posizione giusta	$\Theta(n)$
Code di priorità implementate con vettori non ordinati	RicercaMassimo	Ricerca il massimo in un vettore non ordinato	$\Theta(n)$
	EstrazioneMassimo	Estrae il massimo in un vettore non ordinato	$\Theta(n)$
	Inserimento	Inserisce in un vettore nella posizione giusta	$\Theta(1)$
Heap	Heapify	Muove un elemento, in un heap, nella posizione corretta	$\Theta(\log n)$
	Build-heap	Trasforma un vettore ordinato in un heap	$\Theta(n)$
Priority queue implementate con heap	EstrazioneMassimo	Estrae il valore massimo da un heap	$\Theta(\log n)$
	Inserimento	Inserisce un elemento in un heap (in ultima posizione poi lo porta in su fino alla sua posizione giusta)	$\Theta(\log n)$
Ordinamento	HeapSort	Trasforma un vettore in un heap, poi lo ordina	$\Theta(n \log n)$ "in place"
	QuickSort	Ordina un vettore, disponendo tutti i suoi elementi prima o dopo un elemento pivot, a seconda che siano minori o maggiori. Ricorsivo	$\Theta(n \log n)$ (worst case $\Theta(n^2)$) "in place"
	QuickSort Randomizzato	Ordina un vettore come il quicksort, ma sceglie il pivot a caso tra gli elementi disponibili.	$\Theta(n \log n)$ "in place"
	CountingSort	Ordinamento di un vettore in tempo lineare (dipendente dal numero e dalla varietà degli elementi)	$\Theta(n + k)$
	RadixSort	Ordinamento cifra per cifra (da dx a sx)	$\Theta(dn + dk)$
Selezione	Selection	Restituisce l'elemento maggiore di esattamente i-1 altri numeri (ovvero l'elemento che, se il vettore fosse ordinato, sarebbe in posizione i-esima)	$\Theta(n \log n)$
	RandSelect	Selection. Partiziona il vettore, se i < della posizione del pivot dopo la partizione, itera RandSelect sulla prima partizione, altrimenti sulla seconda.	$\Theta(n^2)$ (average case $\Theta(n)$)

Argomento	Nome	Cosa fa	Costo
Liste (pile, code)			

???	Lista semplice non ordinata	Lista semplice ordinata	Lista double-linked non ordinata	Lista double-linked ordinata
Inserimento	O(1)	O(n)	O(1)	O(n)
Ricerca	O(n)	O(n)	O(n)	O(n)
Cancellazione	O(n)	O(n)	O(n)	O(n)
Successore	O(n)	O(n)	O(n)	O(n)
Predecessore	O(n)	O(n)	O(n)	O(n)
Massimo	O(n)	O(1)	O(n)	O(1)

Hash	Search, Insert, Delete		Θ(1)
	Hash-chaining		
	Open-Addressing		
Alberi Binari di Ricerca (BST)	Inorder-tree-walk	Ordinamento, o visualizzazione ordinata	Θ(n)
Alberi Binari di Ricerca (BST)	Ricerca	Trova un elemento nel BST	Θ(h) h = altezza
	Successore / Precedessore	Trova l'elemento successore (massimo degli elemento minori di un numero).	Θ(h) h = altezza
	Minimo, Massimo	Trova l'elemento minimo (o massimo) in un BST	Θ(h) h = altezza
	Inserimento	Inserisce un elemento nella posizione (che è sempre una foglia)	Θ(h) h = altezza
	Cancellazione	Cancella un elemento (deve invertire l'elemento con il suo successore, e poi cancellarlo)	Θ(h) h = altezza
Red Black Tree	Sui RBT le operazioni hanno costo computazionale Θ(h), evitando il rischio di h = n - 1 perchè i RBT bilanciano l'albero (quasi)		
Programmazione dinamica	Matrix-multiply	Moltiplica due matrici	Θ(righe I matrice x colonne II matrice x colonne matrice risultato)
	Matrix-chain-order	Genera l'ordine secondo il quale effettuare la moltiplicazione tra matrici (per n matrici)	$\Theta(n^3)$ (spazio: $\Theta(n^2)$)
	Matrix-chain-multiply	Effettua la moltiplicazione tra matrici, a due a due, secondo l'ordine definito da Matrix-chain-order	Θ(???)
	Mem-matrix-chain	???	
	Lookup-chain	???	

Argomento	Nome	Cosa fa	Costo
Greedy	Greedy-activity-selector	Dato un insieme di attività, ordinato cronologicamente in base all'inizio di ogni attività, seleziona il sottoinsieme di attività che contiene il numero massimo di attività non contemporanee.	$\Theta(n)$
	Knapsack		
Grafi	Liste di adiacenza	Metodo di memorizzazione di un grafo con liste	Occupazione di spazio $\Theta(n + m)$
	Matrici di adiacenza	Metodo di memorizzazione di un grafo con array bidimensionale	Occupazione di spazio $\Theta(n^2)$
	Ricerca per ampiezza	Metodo di ricerca di un elemento in un grafo per cui si visitano i nodi adiacenti ad un nodo dato e si itera il procedimento "allontanandosi" di una distanza costante, di volta in volta dal nodo iniziale	$\Theta(V + E)$ *V := Vertex (nodi)* *E := Edge (archi)*
	Ricerca in profondità		$\Theta(V + E)$
Grafi - Alberi di copertura minimi	Kruskal	In un grafo pesato, individua un albero, costituito da tutti i nodi del grafo, tale che la somma dei pesi degli archi sia minima	$\Theta(E \log E)$
	Prim	Alternativo a Kruskal	$\Theta(E + V \log V)$
Grafi - Cammini minimi	Dijkstra	Individua i cammini di peso minimo da un nodo dato a tutti gli altri	$\Theta(V^2)$
	Bellman-Ford	Individua i cammini di peso minimo da un nodo dato a tutti gli altri (anche in presenza di archi con peso negativo)	$\Theta(V E)$
Vettori - Vari	Max-Gap	Individua, in un vettore, due valori tali che la loro differenza (gap) sia massima	$\Theta(n \log n)$ *È infatti sufficiente ordinare il vettore e prendere gli estremi*
Vettori - Vari	Min-Gap	Individua, in un vettore, due valori tali che la loro differenza (gap) sia minima	$\Theta(n \log n + n)$
Vettori - Vari	Ricerca di x + y = k (input)	Individua due valori, in due vettori ordinati, che sommati siano uguali ad un valore fornito in input.	$\Theta(n + m)$ *n e m := le lunghezze dei due vettori*

ALGORITMI DI ORDINAMENTO

Algoritmo	Funzionamento	Best case	Average case	Worst case	Funzione costo	Note
InsertionSort	Ogni elemento viene confrontato, uno ad uno, con i precedenti e, se necessario, si scambiano i valori.	n	n^2	n^2		INPLACE

Pseudocodice

```
insertionSort(vettore)
{
  for i=2 to vettore.size
    tmp=vettore(i);
    j=i-1;
    while (j>0 and vettore(j)>tmp)
       vettore[j+1]=vettore[j];
       j--;
    vettore[j]=tmp;
}
```

HeapSort	Ricorsivo. (1) Trasforma un elenco disordinato in un Heap, (2) poi ordina l'heap. Un heap è un albero binario bilanciato quasi completo in qui il nodo "padre" è sempre maggiore dei suoi "figli". (1) Confronta ogni elemento con i suoi "figli", se effettua uno scambio, itera ricorsivamente il confronto con il nodo scambiato e i propri figli. (2) La radice, in un heap, è sempre il maggiore, allora: sposta il primo elemento all'ultimo posto, considera l'elenco ad eccezione dell'ultimo valore, trasforma in heap tale elenco. Itera il procedimento n volte.	n log n	n log n	n log n		INPLACE

Pseudocodice

Vedi argomento heap pagina19.

Algoritmo	Funzionamento	Best case	Average case	Worst case	Funzione costo	Note
MergeSort	Ricorsivo. Si divide l'elenco in due parti, si ordina ogni singola parte, si uniscono le due parti selezionando, uno ad uno, l'elemento più piccolo tra quelli "affioranti" sulle due parti. Per ordinare le due parti si procede ricorsivamente.	n log n	n log n	n log n	$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$	

Pseudocodice

```
mergeSort(vettore,i,f)
{
  if (i<f) then
    x=(i+f)/2;
    mergeSort(vettore,i,x);
    mergeSort(vettore,x+1,f);
    merge(vettore,i,x,f);
}
```

```
merge(vettore,i,x,f)
{
  i2=i;
  i3=x+1;
  for idx=i to f
  {
    if(i2>x) then
      tmp[idx-i]=vettore[i3];
      i3++;
    if(i3>f) then
      tmp[idx-i]=vettore[i2];
      i2++;
    if(i2<=x and i3<=f) then
      if(vettore[i2]<vettore[i3]
      then tmp[idx-i]=vettore[i2]; i2++;
      else tmp[idx-i]=vettore[i3]; i3++;
  }

  for idx=i to f
    vettore[idx]=tmp[idx-i];
}
```

Algoritmo	Funzionamento	Best case	Average case	Worst case	Funzione costo	Note
QuickSort	Ricorsivo. Partiziona l'elenco in base ad un valore detto "PIVOT" in modo che tutti gli elementi minori del PIVOT si trovino prima di tutti gli elementi maggiori del PIVOT. Itera il procedimento per le due parti a sinistra e destra del PIVOT individuato.	n^2	n log n	n log n		INPLACE

Pseudocodice

```
quickSort(vettore,i,f)
{
  if(i<f) then
    x=partition(vettore,i,f);
    quickSort(vettore,i,x);
    quickSort(vettore,x+1,f);
}
```

```
partition(vettore,i,f)
{
  i2=i;
  i3=f;
  x=i;  //modificare con funzione random per scegliere un pivot a caso
  pivot=vettore[x];
  while (i2<i3)
  {
    while (vettore[i2]<pivot) i2++;
    while (vettore[i3]>pivot) i3++;
    if(i2<i3) then
      exchange(vettore[i2],vettore[i3]);
      x=i3;
  }
  xeturn x;
}
```

Algoritmo	Funzionamento	Best case	Average case	Worst case	Funzione costo	Note
CountingSort	Riempie un vettore d'appoggio di k elementi (dove k è il valore massimo all'interno del vettore di partenza) riportando, in corrispondenza di ogni indice, il numero di occorrenze del valore corrispondente all'indice. Con un ciclo aggiorna i valori nelle celle del vettore di appoggio riportando, in ogni cella, il numero di valori minori o uguali all'indice. Rilegge il vettore di partenza e, in corrispondenza di ogni valore, legge nel vettore di appoggio la posizione nella quale inserire il valore in esame.	n + k	n + k	n + k		Se k è molto alto, molto più alto di n, il costo è maggiore di n.

Pseudocodice

```
countingSort(vettore,vettoreOrdinato,k) //k:=valore massimo possibile dentro l'array
{
  for i = 1 to k              do v[i]=0;
  for i = 1 to size(vettore)  do v[vettore[i]]++;
  for i = 2 to k              do v[i]=v[i]+v[i-1];
  for i = 1 to size(vettore)
    vettoreOrdinato[v[vettore[i]]]=vettore[i];
    v[vettore[i]]--;
}
```

Algoritmo	Funzionamento	Best case	Average case	Worst case	Funzione costo	Note
RadixSort	Ordina i valori contenuti in un vettore in base ad una sola delle cifre: ordina tutti gli elementi in modo che si trovino per primi quelli con la cifra delle unità più piccola, poi itera il procedimento sulla cifra delle decine, poi delle centinaia... e così via. Definendo un "alfabeto" costituito da più di dieci cifre, si può utilizzare il radix sort per ordinare numeri di grandi dimensioni, a "blocchi" di cifre.	dn + dk	dn + dk	dn + dk		d è il numero di blocchi di cifre, k il numero di "simboli" dell'alfabeto

Pseudocodice

```
radixSort(vettore,vettoreOrdinato,k,d)  //d:=numero di cifre dei valori in array
{
  for i=1 to cifre
    countingSort(vettore,vettoreOrdinato,k,i);  //versione modificata di countingSort, che ordina solo in base alla cifra indicata (d)
}
```

ALGORITMI VARI

Max-Gap

Dato un vettore individuare i due valori tali che la differenza sia massima.

Si ordina il vettore (T = n log n), si trova il min (T = 1), si trova il max (T = 1).

5	9	12	13	15	18	22	29	40	41	52	55	59	71	78	99	112	123	131	144

Max-Gap := 144 - 5 = 139

Min-Gap

Dato un vettore individuare i due valori tali che la differenza sia minima.

Si ordina il vettore (T = n log n), si confrontano i numeri a coppie (l'idea è che, una volta ordinato il vettore, i due elementi con distanza minima saranno necessariamente vicini), quindi ogni volta che la differenza tra l'elemento i e l'elemento i-1 è minore della differenza precedente, si memorizza la posizione i.

5	9	12	13	15	18	22	29	40	41	52	55	59	71	78	99	112	123	131	144
	4	3	1	2	3	4	7	11	1	11	3	4	12	7	11	13	11	9	13

Min-Gap := 13 - 12 = 1 e 41 - 40 = 1

Ricerca, in due vettori ordinati, di due valori che, sommati, diano un valore definito

Dati due vettori ordinati individuare, se esistono, due valori che sommati sono uguali ad un valore dato.

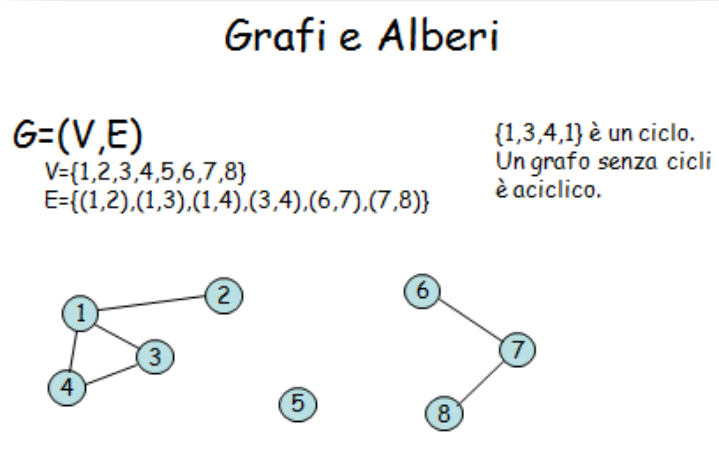
Si procede "camminando" tra i valori somma ottenuti disponendo in matrice i due vettori ordinati (uno per le righe e uno per le colonne) partendo dal valore più piccolo di un vettore e il più grande dell'altro, se la somma dei due valori è più piccola del valore cercato ci si sposta scendendo di un indice nel secondo vettore, altrimenti salendo di un indice nel primo vettore (T = n + m; dove n e m sono le lunghezze dei due vettori; cammini crescenti).

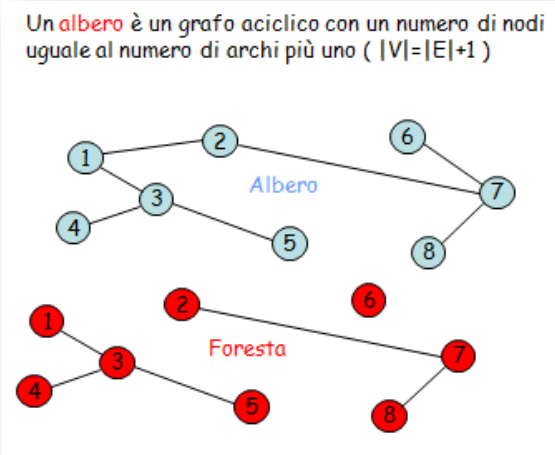
? 19	2	5	8	11	13	15
1	3	6	9	12	14	16
2	4	7	10	13	15	17
5	7	10	13	16	18	20
6	8	11	14	17	19	21
8	10	13	16	19	21	23
10	12	15	18	21	23	25

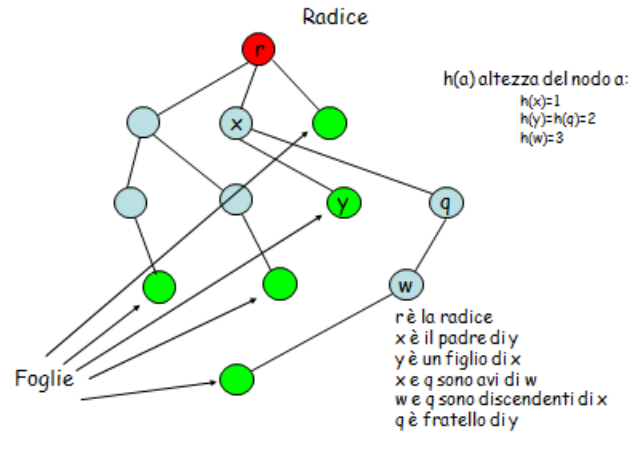
? 8	2	5	8	11	13	15
1	3	6	9	12	14	16
2	4	7	10	13	15	17
5	7	10	13	16	18	20
6	8	11	14	17	19	21
8	10	13	16	19	21	23
10	12	15	18	21	23	25

STRUTTURE DATI

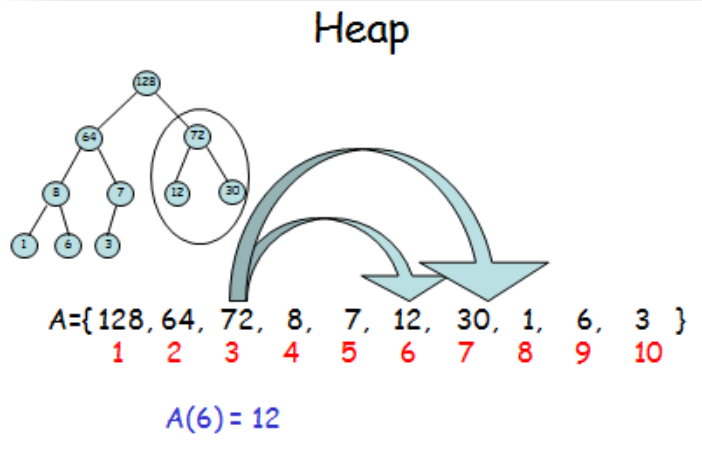
GRAFO, ALBERO, FORESTA







HEAP



Albero binario quasi completo (possono mancare alcune foglie, consecutive, a partire dall'ultima foglia di destra)

Regola: per ogni nodo: valore del nodo ≤ valore del padre del noto.

Il **massimo si trova sempre nella radice**.

Non c'è nessuna relazione tra il valore di un nodo e quello di un suo fratello.

Un heap si memorizza in un vettore seguendo la regola:

1. posizione figlio sinistro = posizione padre * 2
2. posizione figlio destro = (posizione padre * 2) + 1

Inserimento in un Heap

Inserisce l'elemento nella prima posizione libera del vettore (prima foglia libera dell'heap), poi la fa "risalire" lungo il ramo fino alla giusta posizione.

```
heapInsert(vettore, nuovoElemento)
{
  vettore.heapSize++;
  V(vettore.heapSize)=nuovoElemento;
  heapMoveNewChild(vettore);
}

heapMoveNewChild(vettore,i)
{
  if (i==1) then return;
  if (V[i]>V[i/2]) then
  {
    exchange(V[i],V[i/2]);
    heapMoveNewChild(vettore,i/2);
  }
}
```

Estrazione massimo da un Heap

Copia il valore della testa in una variabile. Mette in testa il valore dell'ultima foglia e applica heapify alla nuova testa per riordinare gli elementi. Riduce la lunghezza dellheap.

```
heapExtract(vettore)
{
  tmp=vettore[1];
  vettore[1]=vettore[vettore.heapSize];
  heapify(vettore,1);
  return tmp;
}
```

BuildHeap

Trasforma un vettore disordinato in un heap.

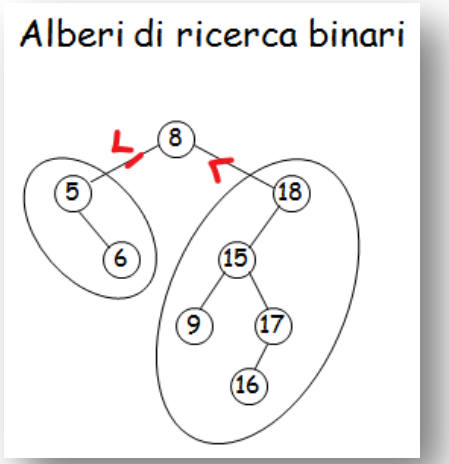
```
buildHeap(vettore, nuovoElemento)
//A partire da metà vettore, verso l'inizio, applica heapify a tutti i nodi.
{
  for (i=vettore.heapSize/2,i>0;i--)
    heapify(vettore,i);
}

heapify(vettore,i)
//Se l'elemento è maggiore di uno dei suoi figli lo scambia con il figlio
//poi esegue ricorsivamente heapify sul figlio scambiato
{
  l=left(i);
  r=right(i);
  max=i;
  if(V[l]>V[max]) then max=l;
  if(V[r]>V[max]) then max=r;
  if(max!=i) then
  {
    exchange(V[i],V[max]);
    heapify(vettore,max);
  }
}
```

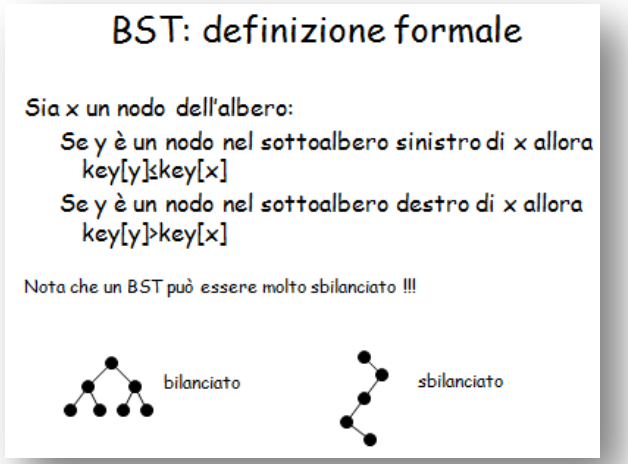
HeapSort

L'elemento in testa è sempre il più grande. Allora scambia l'elemento in testa con l'ultimo elemento, poi ricostruisce un heap di dimensione -1; in questo modo il secondo elemento più grande è in testa; lo scambia con il penultimo e itera il procedimento.

```
heapSort(vettore)
//Il vettore in input è già un heap
{
  for (i=vettore.heapSize;i>1;i--)
  {
    exchange(vettore[1],vettore[i]);
    vettore.heapSize--;
    heapify(vettore);
  }
}
```

BINARY SEARCH TREE (ALBERO DI RICERCA BINARIO)



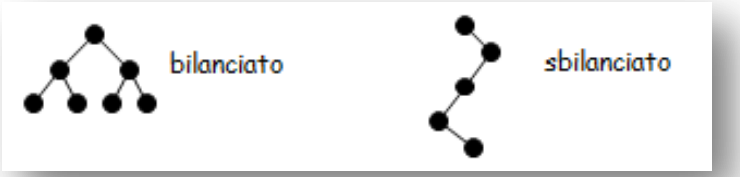


In un albero di ricerca binario:

- tutti i figli di sinistra sono minori del padre
- tutti i figli di destra sono maggiori del padre.

Negli alberi di ricerca le operazioni hanno un costo h = altezza dell'albero.

Problema: siccome il BST può essere sbilanciato, nel caso pessimo h = n - 1, quindi il costo diventa lineare (n).



Inserimento in un BST

L'inserimento di un nuovo elemento viene sempre effettuato in una foglia, si segue tutto il percorso fino all'individuazione della foglia giusta.

```
bstInsert(nodo,elemento)
{
  if(nodo.valore<elemento)
  then
    if (nodo.right=NIL) then nodo.right.valore=elemento;
    else bstInsert(nodo.right,elemento);
  else
    if (nodo.left=NIL) then nodo.left.valore=elemento;
    else bstInsert(nodo.left,elemento);
}
```

Ordinamento di un BST

Sfrutta le caratteristiche del BST e lavora ricorsivamente.

```
bstSort(nodo,*vettore)
//Copia in un vettore tutti i nodi appartenenti al sottoalbero radicato in
//nodo (compreso)
{
  if (nodo!=NIL) then
  {
    bstSort(nodo.left);
    vettore.size++;
    vettore[vettore.size]=nodo;
    bstSort(nodo.right);
  }
}
```

Cancellazione in un BST (successore)

Si possono verificare 3 casi:
1) l'elemento non ha figli: si può cancellare
2) l'elemento ha un figlio: si scambia l'elemento con il figlio e si cancella
3) l'elemento ha 2 figli: si scambia il successore dell'elemento con l'elemento si cancella l'elemento nella nuova posizione (ricorsivo)

```
bstDelete(nodo,elemento)
{
  if(nodo.left=NIL and nodo.right=NIL) then delete(nodo);
  if(nodo.left=NIL and nodo.right!=NIL) then
    exchange(nodo,nodo.right);
    delete(nodo.right);
  if(nodo.left!=NIL and nodo.right=NIL) then
    exchange(nodo,nodo.left);
    delete(nodo.left);
  if(nodo.left!=NIL and nodo.right!=NIL) then
    exchange(nodo,successore(nodo.right));
//nodo e il suo successore si scambiano di posto
//il successore ha, al massimo, un figlio
//bstDelete "sa" come eliminarlo
    bstDelete(nodo,elemento)
}

successore(nodo)
{
  if(nodo.left=NIL)
  then return nodo;
  else return successore(nodo.left);
}
```

Ricerca in un BST

Sfrutta le caratteristiche del BST e lavora ricorsivamente.

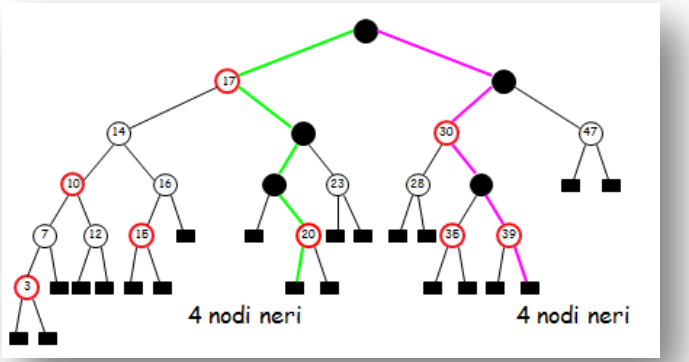
```
bstSearch(nodo,elemento)
//cerca l'elemento nel sottoalbero radicato in nodo
{
  if (nodo==NIL or nodo==elemento) then return nodo;
  if (nodo.valore<elemento)
  then return bstSearch(nodo.right,elemento);
  else return bstSearch(nodo.left,elemento)
}
```

RED-BLACK TREES

Sono alberi binari di ricerca con alcune proprietà aggiuntive che permettono il mantenimento del bilanciamento dell'albero a seguito delle operazioni eseguite sull'albero. *L'albero si considera bilanciato quando la differenza tra l'altezza minima e l'altezza massima delle sue foglie è minore o uguale ad una determinata costante.*

Proprietà:

a) Ogni nodo è **rosso** oppure è **nero**
b) Ogni foglia è **nera** (proprietà non fondamentale, serve per comodità)
c) Un nodo **rosso** ha sempre figli **neri**
d) Preso un ogni nodo, tutti i cammini dal nodo alle foglie hanno lo stesso numero di nodi **neri**.



Con questa proprietà l'albero non può che essere "quasi bilanciato".

Osservazioni:

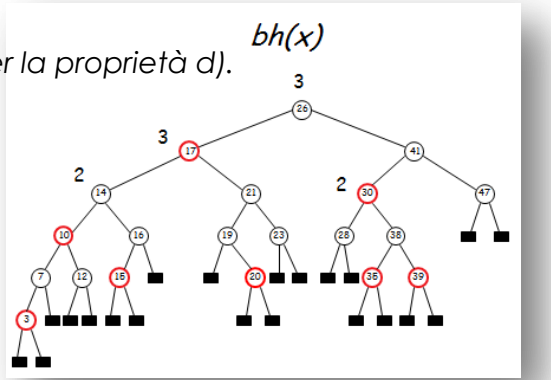
- Se il Red-Black Tree è costituito da soli nodi **neri**, allora diventa un albero completo perché tutti i cammini dalla radice alle foglie hanno la stessa lunghezza (h = log n).
- Ogni ramo (cammino dalla radice ad una foglia) può avere al massimo tanti nodi **rossi** quanti nodi **neri** (-1) (h = 2 log n).

(1) Teorema: un Red-Black Tree con n nodi interni è alto al massimo $2 \log_2 (n + 1)$

Definizioni necessarie:

h (x) := altezza di un nodo

bh(x) := altezza **nera** di un nodo = numero di nodi **neri** (escluso x) nel cammino da x ad una foglia.
(non è importante la foglia, infatti tutti i cammini da un nodo alle foglie hanno lo stesso numero di nodi neri per la proprietà d).



Altezza dell'albero = h (radice). Altezza nera dell'albero = bh (radice)

Osservazione: l'altezza **nera** di un figlio è sempre minore o uguale all'altezza **nera** di un nodo, in particolare:

1. se il figlio è **rosso** ha la stessa altezza nera del padre
2. se il figlio è **nero** ha altezza nera del padre -1.

Lemma: il numero di nodi interni di un sotto albero radicato in x è maggiore o uguale a $2^{bh(x)}-1$

Dimostrazione del lemma (induzione)

Caso base: h(x) = 0

Se h(x) = 0 ⇒ bh(0), inoltre x è una foglia quindi il numero d nodi interni è 0.

Induzione: h(x) > 0

Se x ha due figli (L := figlio sinistro; R := figlio destro), si hanno due casi:

i. se L è **rosso** bh(L) = bh (x)

ii. se L è **nero** bh(L) = bh (x) - 1

Lo stesso ragionamento vale per R.

Naturalmente l'altezza dei figli è minore dell'altezza del padre, pertanto si può applicare l'ipotesi induttiva e si ha che:

il numero dei nodi interni dell'albero radicato in L è maggiore o uguale a $2^{bh(L)}-1$ (stesso ragionamento per R).

Quindi: $2^{bh(L)} - 1 \geq 2^{bh(x)} - 1$ e $2^{bh(R)} - 1 \geq 2^{bh(x)} - 1$ ⇒

⇒ nodi di dell'albero radicato in L + nodi dell'albero radicato in R + 1 (il nodo x) = nodi dell'albero radicato in x

⇒ $2^{bh(x)} - 1 + 2^{bh(x)} - 1 + 1 = 2^{bh(x)}-1$.

(1) Dimostrazione del Teorema

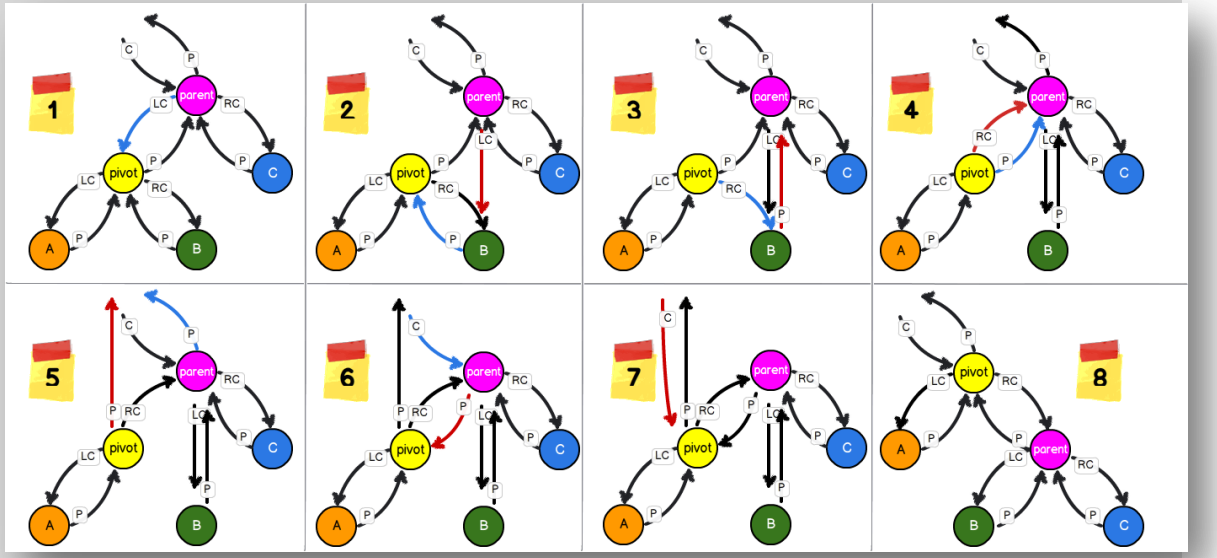
Se n è il numero di nodi di un BRT, per il lemma, $n \geq 2^{bh(radice)} - 1$

Inoltre: bh (radice) ≥ h (radice) / 2 allora, $n \geq 2^{bh(radice)} - 1 \geq 2^{h/2} - 1$ da cui:

Applicando il logaritmo si ha che: $\log_2 (n + 1) \geq \log_2 (2^{h/2}) = h / 2$ ⇒ **$h \leq 2 \log_2 (n + 1)$** ■

Affinché le operazioni di inserimento e cancellazione in un RBT mantengano le proprietà è necessario applicare delle rotazione sull'albero.

```
//Effettua una rotazione, richiede in input il puntatore al RBT e l'elemento che deve essere ruotato (chiamato pivot)
//Se l'elemento è un figlio sinistro, rotation effettua una rotazione a destra
//Se l'elemento è un figlio destro, rotation effettua una rotazione a sinistra
rotation(T, pivot) {
  //Cerca l'elemento pivot
  pivot=findNode(T,pivot);  //findNode trova il nodo all'interno dell'albero
  //Individua se il nodo è un figlio sx oppure un figlio dx
  if (pivot->parent->leftChild=pivot) then
  {
    //figlio sinistro
   pivot->parent->leftChild  = pivot->rightChild;      //1->2
   pivot->rightChild->parent = pivot->parent;          //2->3
   pivot->rightChild         = pivot->parent;          //3->4
   pivot->parent             = pivot->parent->parent; //4->5
   pivot->rightChild->parent = pivot;                  //5->6
   if (pivot->parent->rightChild = pivot->rightChild)
   {
     pivot->parent->rightChild = pivot;                //6->7
   }
   else
   {
     pivot->parent->leftChild = pivot;                 //6->7
   }
  }
  else
  {
    //figlio destro
   ...
  }
}
```



Inserimento

> caso zio rosso

> caso zio nero

Cancellazione

> sempre cancellazione di elementi con un solo figlio (come nel BST)

Esercizio da vecchi compiti (17/2/2010 - Esercizio 1)

Sia A un albero binario i cui nodi contengono un campo COLORE che può assumere il valore NERO o AZZURRO. Scrivere una funzione RICORSIVA che prenda in input A e restituisca il numero di nodi NERI dell'albero. La funzione deve essere ricorsiva e possibilmente scritta in pseudo codice. Analizzare il costo computazionale della procedura proposta.

```
blackNodes(A)
{
//Se il nodo ha figlio sinistro, conta il numero di nodi neri del sottoalbero radicato in A->figlioSx
  if (A->figlioSx=NIL) then
    s = 0                          // <- caso base per il figlio sinistro
  else
    s = blackNodes(A-figlioSx)     // <- chiamata ricorsiva sinistra

//Se il nodo ha figlio destro, conta il numero di nodi neri del sottoalbero radicato in A->figlioDx
  if (A->figlioDx=NIL) then
    d = 0                          // <- caso base per il figlio destro
  else
    d = blackNodes(A-figlioDx)     // <- chiamata ricorsiva destra

//Se il nodo è nero, il numero di nodi neri dell'albero radicato in A è uguale al numero di nodi neri dei due sottoalberi Sx e Dx di A + 1
  if (A->COLORE=NERO) then
    return d+s+1
  else
    return d+s
}

L'algoritmo blackNodes ha costo computazione O(n) perché deve viene eseguito per ogni nodo, e per ogni nodo deve valutare se sia NERO oppure
AZZURRO.
```

BST e RBT conclusioni

Quando serve una struttura dati buona per una ricerca, si possono utilizzare i BST.

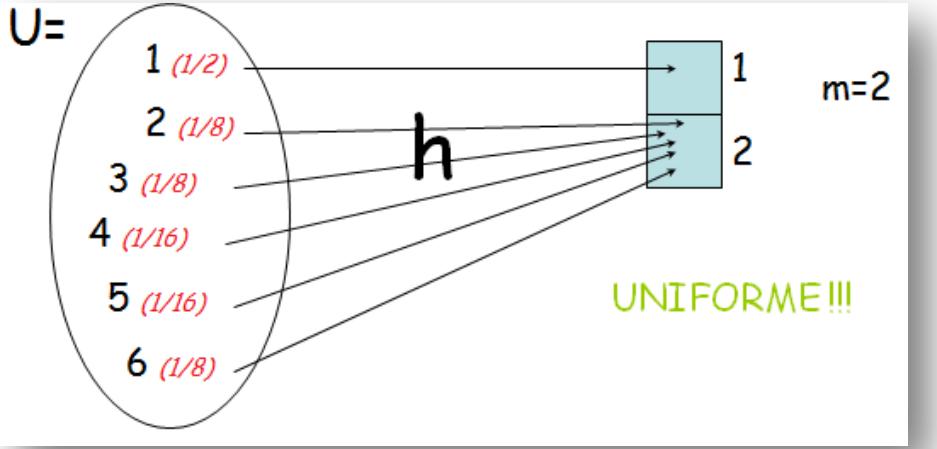
Quando però è necessario anche mantenere il bilanciamento dell'albero (ad esempio per migliorare tempi di ricerca) si possono utilizzare i RBT.

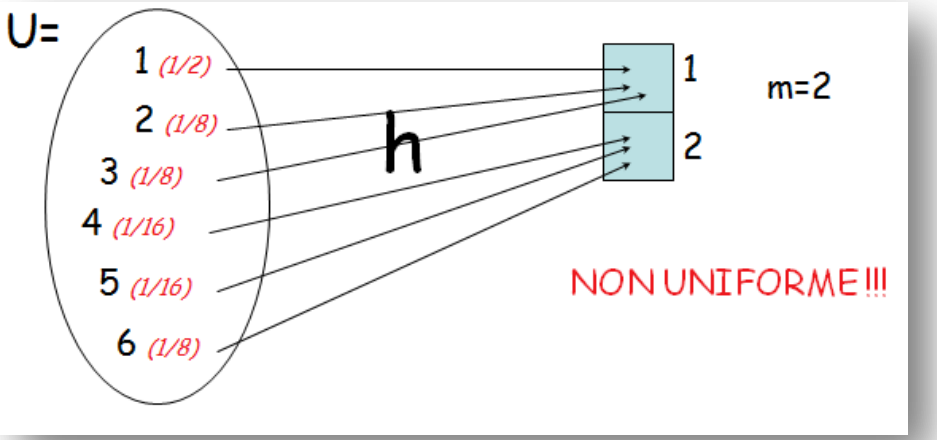
HASH

Ottimizzazione delle risorse tempo/spazio

Una funzione hash deve essere deterministica: a partire dagli stessi valori di input deve produrre sempre gli stessi risultati.

Una "buona" funzione hash deve sembrare "random" e "uniformare" la distribuzione dei risultati (attenzione: distribuzione dei risultati, non delle chiavi).



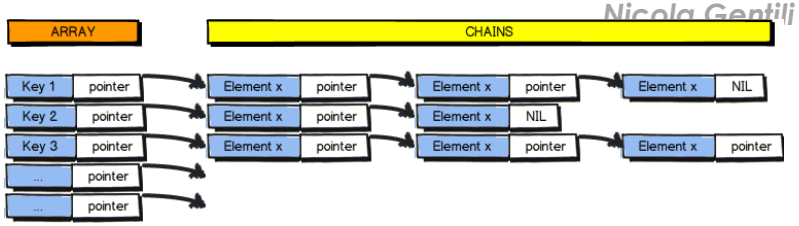


Gestione delle collisioni: minimizzazione delle collisioni; per fare questo si devono distribuire al meglio i valori delle funzioni hash per distribuire al meglio lo spazio delle chiavi sugli indici.

Hash "chaining"

Gestione delle collisioni con "chaining" (array di liste):

1. La funzione hash genera l'indice a partire da una chiave
2. in ogni posizione è "attaccata" una lista di elementi.



Il Load factor è la lunghezza media delle catene che, con una buona distribuzione delle chiavi è costante:
m chiavi, n indici disponibili -> lunghezza media delle catene (load factor): m / n.

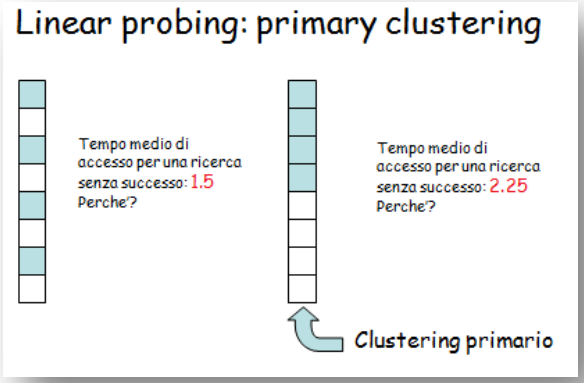
Funzione	Note	Costo
Ricerca	La ricerca in un hash chaining costa "load factor"+1(funzione hash) tempo.	Load factor + 1
Inserimento	L'inserimento ha tempo costante: funzione hash + inserimento in testa.	Tempo costante
Cancellazione	Cancellazione: ricerca + cancellazione.	Load factor + 1

Open addressing

Gestione delle collisioni con algoritmo iterativo di ricerca: la funzione hash calcola l'indice, se la posizione è occupata si calcola una nuova posizione e si itera il calcolo fino all'individuazione di una posizione libera.

Open addressing è una famiglia di metodi.

Linear probing: h'(k) è la funzione di hash, h(k,i) = (h'(k)+i) mod m



("mod m" riporta un qualsiasi valore individuato all'interno del range delle posizioni disponibili)

Con la gestione **open addressing linear probing** si presenta il problema dei cluster di posizioni: quando si formano gruppi di posizioni consecutive occupate il tempo di accesso per la ricerca aumenta.
Linear probing prova ad occupare posizioni successive consecutive a quelle occupate.

Linear probing: $h(k,i) = (h'(k)+i) \bmod m$

Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_2 \neq 0$

Double hashing: $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

PROGRAMMAZIONE DINAMICA

Il metodo di programmazione divide et impera prevede la suddivisione di problemi in sottoproblemi indipendenti e l'applicazione dello stesso metodo sui sottoproblemi ottenuti ricorsivamente fino ad ottenere problemi banali di cui si conosce la soluzione.

Il metodo della programmazione dinamica è simile al divide et impera (top-down), si tiene però traccia delle soluzioni perché può capitare di dover risolvere lo stesso sottoproblema più volte (bottom-up).

E' un metodo utile quando esistono sottoproblemi in comune.

ESERCIZIO: prendere 3 matrici non quadrate e dimensionarle in modo da moltiplicarle l'una per l'altra
(es: 3x5 * 5x19 * 19x42)... e calcolare le operazioni svolte svolgendo i calcoli AxB xC oppure Ax BxC

Caso (AxB)xC

Moltiplico AxB:	3x5x19 operazioni	=	285	ottengo una matrice 3x19
Moltiplico (AxB)xC:	3x19x42 operazioni	=	2394	ottengo una matrice 3x42
	Totale operazioni	=	2679	

Caso Ax(BxC)

Moltiplico BxC: 5x19x42 operazioni		=	3990	ottengo una matrice 5x42
Moltiplico Ax(BxC): 3x5x42 operazioni		=	630	ottengo una matrice 3x42
	Totale operazioni	=	4620	

Nel caso delle matrici, si potrebbe provare, iterativamente, a verificare tutte le combinazioni di parentesizzazioni e valutarne la complessità in termini di moltiplicazioni di scalari. Questa strada non è percorribile per la complessità computazionale generata.
Allora si può ragionare in questo modo: immaginando di avere la paretesizzazione ottima P(n), si sa che questa è costituita dalla parentesizzazione dei primi n-k elementi e dei secondi k elementi. Ora: P(n-k) e P(k) sono rispettivamente parentesizzazioni ottime per i due sottoproblemi di n-k e k dimensioni, infatti, se così non fosse, si avrebbe (ad esempio) una parentesizzazione migliore di P(k) che permetterebbe di ottenere una parentesizzazione P'(n) migliore di P(n), che per ipotesi è ottima.

Proprietà necessaria per poter applicare la tecnica della programmazione dinamica:
"la soluzione ottima del problema contiene la soluzione ottima di ogni sottoproblema".

Questo codice calcola contemporaneamente le due tabelle.

```
Matrix-chain-order(p)
  n = length(p)-1                    per ogni possibile combinazione di matrici (nell'ordine)
  for i = 1 to n do                    imposta a 0 il costo (sottosequenze di dimensione 1)
    m[i,i] = 0                       effettua il ciclo principale (sottosequenze lunghe da 2 a n):
  for l = 2 to n do                    per ogni indice successivo alla posizione attuale (inizio sottosequenza)
    for i = 1 to n-l+1 do                imposta il valore di j (fine sottosequenza)
      j = i+l-1                          imposta il costo di moltiplicazione al massimo
      m[i,j] = ∞                         calcola la parentesizzazione ottima:
      for k = i to j-1 do                  calcola il costo (numero di moltiplicazioni) per 2 matrici (dimensioni p_{i-1}×p_k e p_k×p_j)
        q = m[i,k]+m[k+1,j]+p_{i-1}p_kp_j  se il costo è minore di quello finora individuato
        if q<m[i,j]                          lo registra nella matrice dei costi
          then m[i,j) = q                    registra nella matrice delle parentesizzazioni il valore di k (divisione delle matrici)
               s[i,j] = k              ritorna le matrici m (dei costi) e s (delle parentesizzazioni)
  return m,s
```

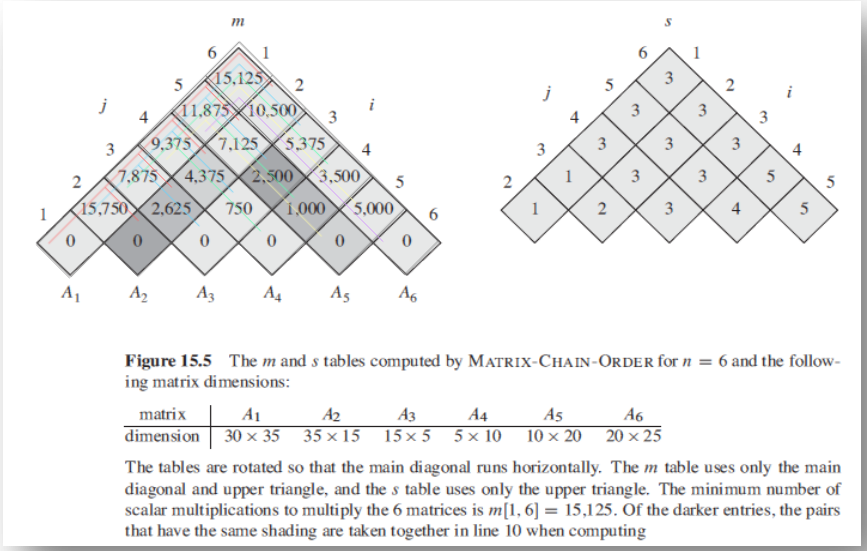


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

Pseudo codice

```
Matrix-chain-multiply(A,s,i,j)
  if j>i then X = Matrix-chain-multiply(A,s,i,s[i,j])
              Y = Matrix-chain-multiply(A,s,s[i,j]+1,j)
              return Matrix-multiply(X,Y)
               else return A_i
```

Parentesizzazione ottima dell'esempio = $((A_1(A_2A_3))((A_4A_5)A_6))$

Costo computazionale:

Matrix-chain-order	tempo $\Theta(n^3)$	spazio $\Theta(n^2)$
Matrix-chain-multiply	tempo esercizio	spazio esercizio

Matrix − chain − order trova le soluzioni per i sottoproblemi di lunghezza 1 (n) e di lunghezza da 2 a n $\left(\binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2-n}{2}\right)$,

ovvero: $n + \frac{n^2-n}{2} = \frac{n^2+n}{2} = \Theta(n^2)$. Pertanto la quantità di memoria necessaria a registrare le due matrici m e s è $\Theta(n^2)$. Si ha inoltre che l'algoritmo calcola $\Theta(n^2)$ soluzioni per i sottoproblemi e deve effettuare, per ognuno di essi, al massimo n-1 scelte. In totale $\Theta(n^3)$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

Print-Optimal-Parens stampa la corretta sequenza per la moltiplicazione di matrici ottimizzata.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

Matrix-Multiply moltiplica due matrici.

La versione "banale" della moltiplicazione di matrici che ha un costo $\Omega(4^n / n^{3/2})$.

Esercizio: provare a scrivere il codice Matrix-chain-multiply con metodo iterativo anziché ricorsivo.

Passi fondamentali della programmazione dinamica

1. Caratterizzazione della struttura di una soluzione ottima
2. Definizione ricorsiva del valore di una soluzione ottima
3. Calcolo del valore di una soluzione ottima con strategia bottom-up
4. Costruzione di una soluzione ottima a partire dalle informazioni già calcolate.

Caratteristiche del problema per applicare la programmazione dinamica

Sottostruttura ottima.

Una soluzione ottima per il problema contiene al suo interno le soluzioni ottime dei sottoproblemi

Sottoproblemi comuni.

Un problema di ottimizzazione ha sottoproblemi comuni quando un algoritmo ricorsivo richiede di risolvere più di una volta lo stesso sottoproblema

ALGORITMI GREEDY

"Greedy", termine inglese: avido.

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).

Selezione di attività

$S=\{1,\dots,n\}$ insieme di attività. Ogni attività ha un tempo di inizio s_i e un tempo di fine f_i.

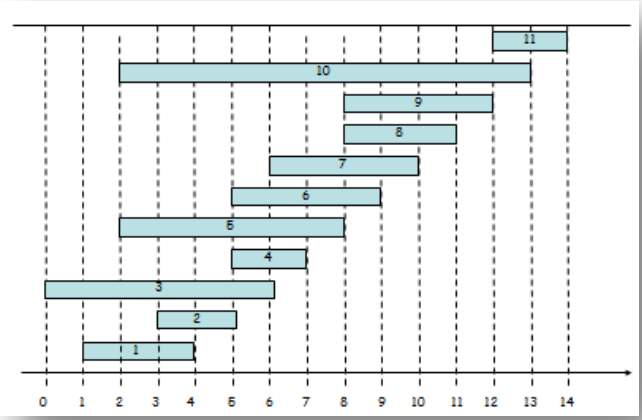
Problema: Selezionare un sottoinsieme S' di attività in modo tale che:

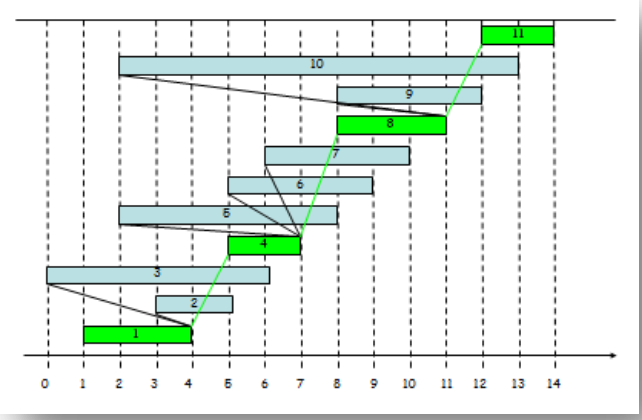
1. se i e j appartengono a S' allora: $s_i \geq f_j$ oppure $s_j \geq f_i$.
2. La cardinalita di S' è massimizzato.

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).

PROBLEMA DELLA SELEZIONE DI ATTIVITÀ

Dato un elenco di attività (ordinato in base al tempo di fine) in un determinato intervallo di tempo, individuare il numero massimo di attività non sovrapposte.





Pseudocodice

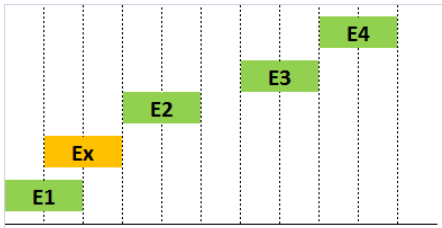
```
Greedy-activity-selector(s,f)
    n=length(s)
    A={1}
    j=1
    for i=2 to n
      do if s_i≥f_j then A=A∪{i}
                         j=i
    return A
```

$\Theta(n)$

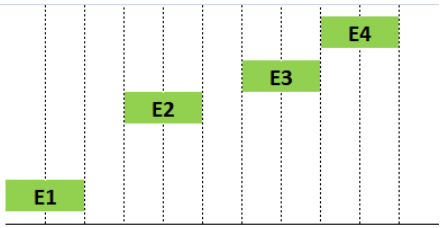
Dimostrazione correttezza del pseudocodice greedy: si vuole dimostrare che l'algoritmo sia "ottimo" (funziona bene nel minor tempo).

Si dimostra per induzione:

Passo base: il primo elemento fa parte di almeno una soluzione ottima.
Il passo base di dimostra per assurdo: se affermiamo che il primo elemento non fa parte di nessuna soluzione ottima, allora una soluzione ottima contiene un elemento che finisce dopo E1. Si nota però che togliendo questo primo elemento (x) della soluzione ottima si ha che E1 entra a far parte di una soluzione ottima, quindi per assurdo si ha che **il primo elemento fa sempre parte di una soluzione ottima.**



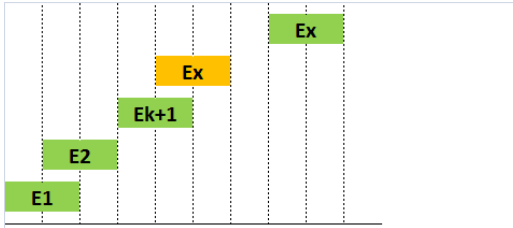
Soluzione ottima: Ex, E2, E3, E4



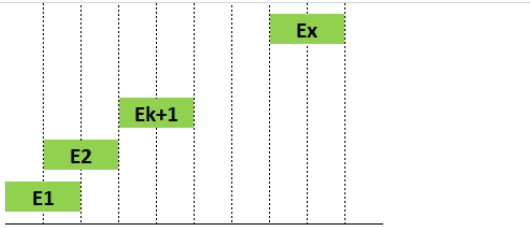
Soluzione ottima: E1, E2, E3, E4

Passo di induzione: come il passo base, per assurdo, su elemento k+1

Ovvero: se E1, E2, ..., Ek fanno parte di una soluzione ottima, ipotizzo che il successivo Ek+1 non faccia parte di una soluzione ottima,
Allora, scegliendo un'altro elemento Ex (successivo a Ek, che finisce dopo Ek+1) che fa parte di una soluzione ottima, si nota che, eliminando tale elemento, Ek+1 diventa parte di una soluzione ottima; che è un assurdo.



Soluzione ottima: E1, E2, ..., Ex, E2, E3, E4



Soluzione ottima: E1, E2, ..., Ek+1, E2, E3, E4

Nota: non sempre una soluzione di ottimo "globale" si ottiene con una serie di soluzioni "localmente" ottime. Nel problema del commesso viaggiatore nessuna politica decisionale greedy fornisce una soluzione finale ottima.

Programmazione dinamica Vs Algoritmi greedy

Non sempre è possibile risolvere un problema con programmazione dinamica o con algoritmi greedy (commesso viaggiatore).

Non sempre i problemi che soddisfano la proprietà della sottostruttura ottima possono essere risolti con entrambi i metodi.

Ci sono problemi che non possono essere risolti con algoritmi greedy ma possono essere risolti con programmazione dinamica

Se un problema può essere risolto con algoritmi greedy è inutile scomodare la programmazione dinamica.

Proprietà della scelta greedy.

Ad ogni passo l'algoritmo compie una scelta in base ad una certa politica ed alle scelte compiute fino a quel momento. Cosi facendo ci si riduce ad un sottoproblema di dimensioni più piccole. Ad ogni passo si calcola un pezzo della soluzione.

Sottostruttura ottima.

Come nel caso della programmazione dinamica, anche per applicare un algoritmo greedy occorre che la soluzione ottima contenga le soluzioni ottime dei sottoproblemi.

PROBLEMA DELLO ZAINO: KNAPSACK

Knapsack 1 (0-1)

Un ladro, durante una rapina, si trova davanti a n oggetti. Ogni oggetto ha un valore v_i e un peso w_i. Il ladro ha uno zaino che può contenere fino al peso massimo W. Il ladro deve scegliere quali oggetti rubare per massimizzare il valore complessivo degli oggetti rubati.

Gli oggetti sono "interi" non frazionabili.

Knapsack

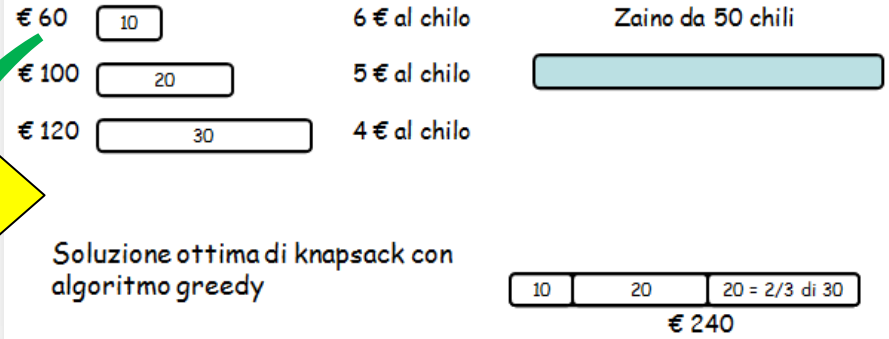
Gli oggetti sono frazionabili: il ladro può prendere un pezzo, grande a suo piacimento.

Knapsack: soluzione greedy

Idea: il ladro prende la quantità più grande possibile dell'oggetto *i* tale per cui v_i/w_i (valore per unità di peso) è massimo. Dopodiche', se nello zaino c'è ancora posto, ripete l'operazione.

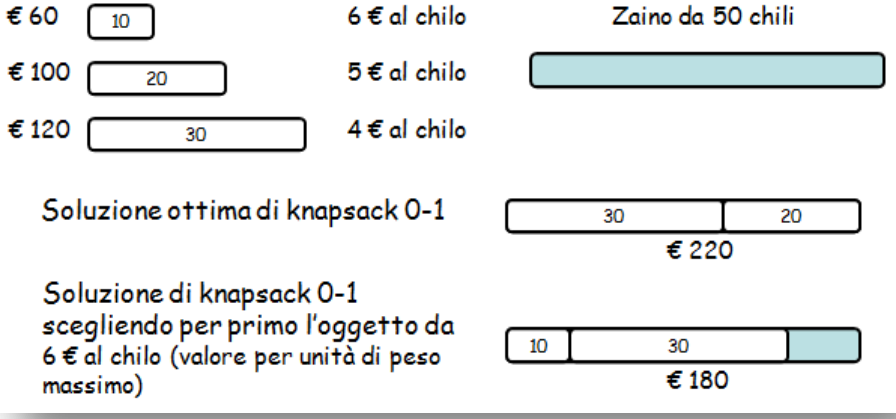
Esercizio. Dimostrare che l'algoritmo è greedy ed è corretto.

Knapsack: soluzione greedy



L'algoritmo Greedy nel caso "0-1" non è ottimo.

Knapsack 0-1: nessuna soluzione greedy



Knapsack 0-1: Soluzione con programmazione dinamica

Prima di decidere se inserire nello zaino l'oggetto *i* bisogna controllare il valore della sottosoluzione ottima di due altri sottoproblemi su *n-1* oggetti:

il sottoproblema nel quale l'oggetto *i* è inserito nello zaino e lo zaino ha capienza $W-w_i$

il sottoproblema nel quale l'oggetto *i* non è inserito nello zaino e lo zaino ha ancora peso W

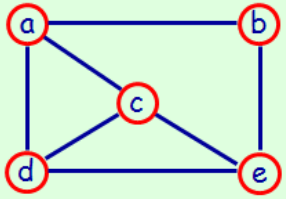
Esercizio: risolvere knapsack 0-1 con la programmazione dinamica

ALGORITMI ELEMENTARI SU GRAFI

Grafo: definizione

Un **grafo** $G = (V,E)$ è composto da:
- V: insieme di **vertici**
- $E \subset V \times V$: insieme di **archi** *(edge)* che connettono i **vertici**

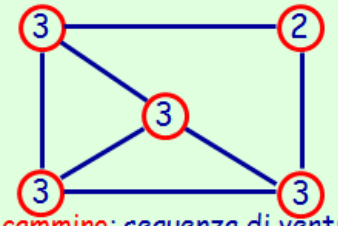
un **arco** $a = \{u,v\}$ è una coppia di vertici
Se (u,v) è ordinato allora **G** è un **grafo diretto**



V = {a,b,c,d,e}
A = { (a,b),(a,c),(a,d)
(b,e),(c,d)
(c,e),(d,e)}

Terminologia

vertici adiacenti: connessi da un arco
grado (di un **vertice**): num. di vertici adiacenti



$$\sum_{v \in V} \deg(v) = 2(\text{num di archi})$$

cammino: sequenza di vertici $v_1, v_2, \ldots v_k$ tale che ogni coppia di vertici consecutivi v_i e v_{i+1} sia adiacente

Primo problema: come memorizzare un grafo con strutture dati disponibili?

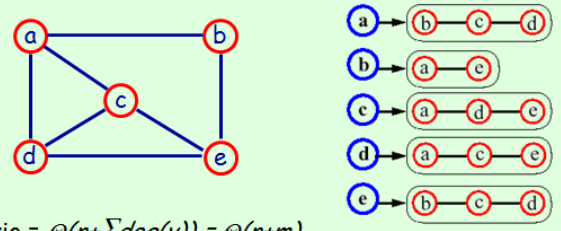
Metodo 1: liste di adiacenza

Ogni nodo è un elemento di una lista. Da ogni nodo parte una lista contenente tutti i nodi adiacenti.

Questa rappresentazione è molto utile quando si devono fare una serie di operazioni su tutti i nodi adiacenti ad un determinato nodo.

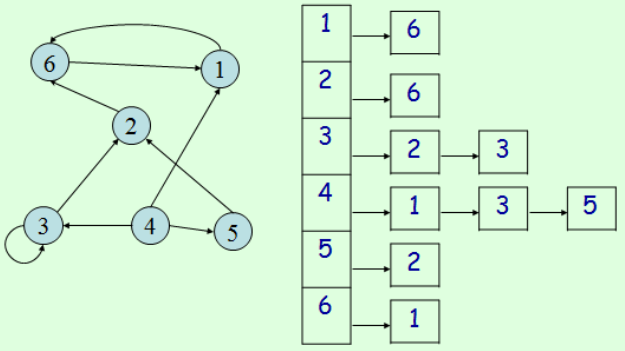
Rappresentazione dei grafi - Liste di adiacenza

Lista di adiacenza di un vertice v: sequenza di tutti i vertici adiacenti a v
Il grafo può essere rappresentato dalle liste di adiacenza di tutti i suoi vertici

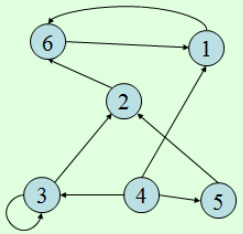


Spazio = $\Theta(n+\Sigma deg(v)) = \Theta(n+m)$

Rappresentazione dei grafi - Liste di adiacenza



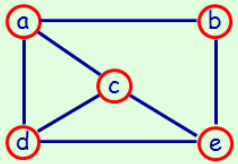
Rappresentazione dei grafi – matrice di adiacenza



	1	2	3	4	5	6
1	0	0	0	0	0	1
2	0	0	0	0	0	1
3	0	1	1	0	0	0
4	1	0	1	0	1	0
5	0	1	0	0	0	0
6	1	0	0	0	0	0

Rappresentazione dei grafi – matrice di adiacenza

Una matrice di adiacenza è una matrice M di variabili booleane con una cella per ogni coppia di vertici
M[i,j] = vero (oppure 1) – c'è l'arco (i,j) nel grafo
M[i,j] = falso (oppure 0) – non c'è l'arco (i,j) nel grafo
Spazio = $\Theta(n^2)$



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

Metodo 2: matrice di adiacenza

Ogni cella rappresenta l'adiacenza tra il nodo indicato in "coordinata x" e il nodo indicato in "coordinata y".

Maggior spazio utilizzato!

Quando un grafo ha poche connessioni logiche, si dice "sparso". Quando è sparso, memorizzare in una matrice di adiacenza tutti gli archi, determina uno spreco enorme di spazio.

Esempio di grafi giganteschi: tutti i social network (facebook...).

Il tempo di esecuzione di un qualsiasi algoritmo su un grafo dipende dal metodo di memorizzazione (Struttura dati) utilizzato.

VISITA DEL GRAFO

Metodo 1: Breadth First Search

1. Visito un nodo, segno che è visitato, poi visito i suoi adiacenti non visitati(dovrò scegliere un ordinamento per visitarli)
2. Procedo ricorsivamente su 1 per ogni nodo adiacente al nodo di partenza.

Quindi vengono visitati i nodi nell'ordine: nodo di partenza, nodi a distanza 1, nodi a distanza 2, ...

Utile per individuare il cammino più corto (in grafi con archi non pesati: tutti gli archi hanno peso uguale).

Si utilizza una struttura dati Q (coda) e si colorano i nodi con i colori: bianco, grigio, nero secondo le seguenti regole.

```
BFS(G,s)
foreach vertice u ∈ V[G]-{s} do
   color[u] = white
   d[u] = ∞
   π[u] = NIL
color[s] = gray
d[s] = 0
π[s] = NIL
Q = {s}
while Q ≠ ∅ do
   u = head[Q]
   foreach v ∈ Adj[u] do
      if color[v] == white
      then color[v] = gray
           d[v] = d[u] + 1
           π[v] = u
           Enqueue(Q,v)
   Dequeue(Q)
   color[u] = black
```

Inizializza tutti i vertici

Inizializza BFS con *s*

Gestisci tutti i figli di *u* prima di passare ai figli dei figli

1. All'inizio tutti i nodi sono bianchi
2. Coloro di grigio il nodo di partenza e lo inserisco in Q
3. Inserisco i nodi adiacenti (bianchi) in Q e li coloro di grigio
4. Coloro di nero il nodo di partenza o lo tolgo da Q
5. itero il procedimento.

Nota: in Q memorizzo anche la distanza = 1 + distanza nodo precedente

In π memorizzo il padre (nodo di arrivo) di ogni nodo, in questo modo sono in grado di ricostruire sempre il cammino da un nodo ad un altro. In particolare si memorizza il "percorso all'indietro" perché è unico.

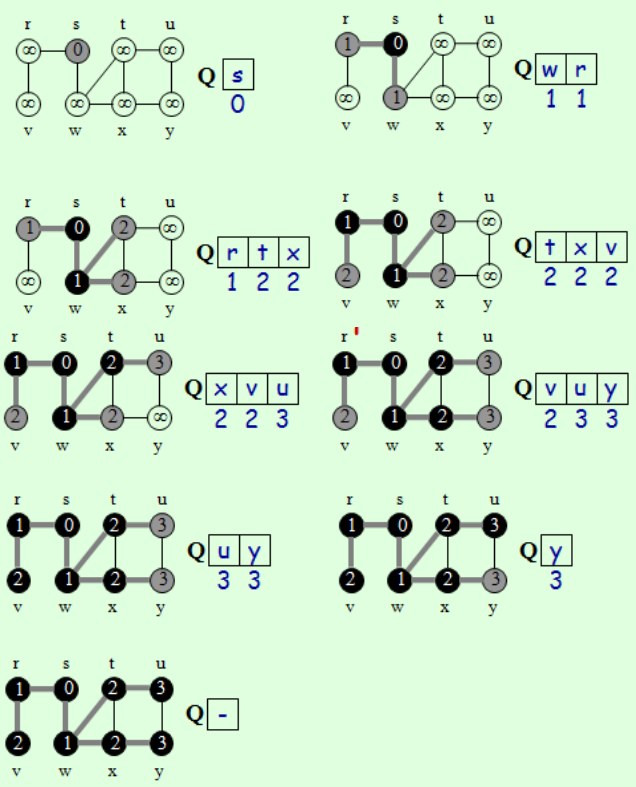
In questo caso si può implementare il grafo con una lista di adiacenza; la coda può essere implementata con un vettore perché so esattamente quando può essere lunga al massimo: numero di elementi del grafo.

Costo O(V + E):

- Tutti i vertici vengono inseriti e rimossi in Q. Θ (V)
- Tutti gli archi sono descritti (e ricercati dall'algoritmo) per individuare i vertici adiacenti di ogni nodo da inserire in Q. O (E)

Proprietà della BFS

- ✓ Dato un grafo G(V, E), la BFS scopre tutti i vertici raggiungibili da un vertice origine
- ✓ Calcola la distanza minimia dal vertice origine ad ogni vertice raggiungibile
- ✓ Calcola un albero breadth-first che contiene tutti i vertici raggiungibili
  Per ogni vertice raggiungibile, il cammino nell'albero breath-first da s a v corrisponde ad un cammino minimo.



Se, anziché utilizzare una coda (Q) si utilizza una pila (S) e si utilizzano le funzioni push e pop anziché enqueue e dequeue, allora si effettua la visita in profondità (DFS) anziché in ampiezza (BFS).

Metodo 2: Depth First Search

1. Visito un nodo, segno che è visitato, poi visito i il suo primo adiacente non visitato(dovrò scegliere un ordinamento per visitarli)
2. Procedo ricorsivamente su 1 per sul nodo adiacente visitato
3. Visito i successivi adiacenti..

```
DFS(G)
 foreach vertice u ∈ V[G] do
    color[u] = white
    π[u] = NIL
 time = 0
 foreach vertice u ∈ V[G] do
    if color[u] == white
    then DFS-Visit(u)
```

Inizializza tutti i vertici

```
DFS-Visit(u)
 color[u] = gray
 time++; d[u] = time
 foreach v ∈ Adj[u] do
    if color[v] == white
    then π[v] = u
         DFS-Visit(v)
 color[u] = black
 time++; f[u] = time
```

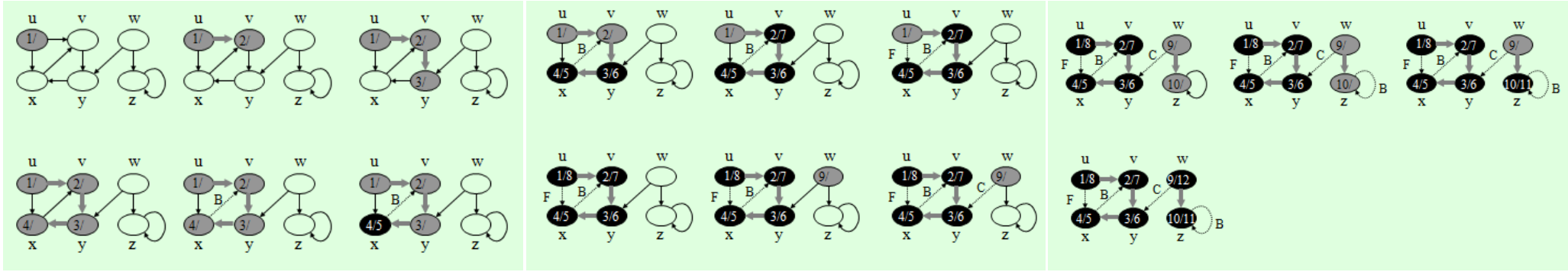
Visita ricorsivamente tutti i figli

Questo frammento di pseudocodice effettua la visita in profondità.

Inoltre, a differenza del frammento di pseudocodice precedente effettua una visita completa del grafo: anche nel caso in cui sia "disconnesso", ovvero nel caso in cui vi siano parti del grafo non collegate da archi.

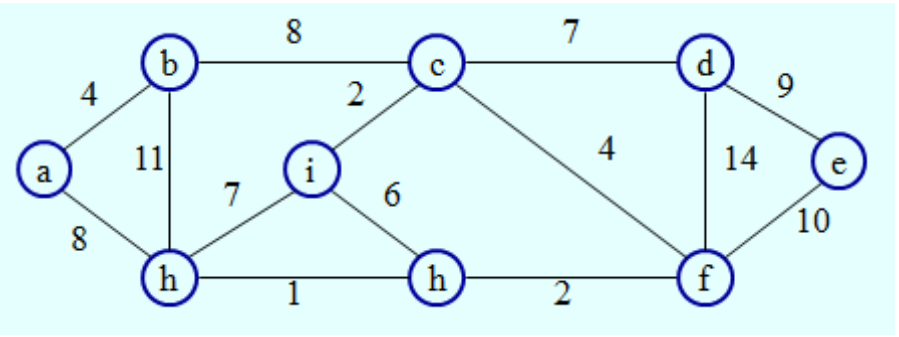
Si osservi che questo frammento di codice non contiene né pile, né liste, né altre strutture per memorizzare l'elenco di visita dei nodi. Questa struttura è implicita (viene utilizzato lo stack di sistema) in quando si utilizzano chiamate ricorsive.

Costo Ω(V + E)



ALBERI DI COPERTURA MINIMI (MINUMUM SPANNING TREE)

In un grafo, non orientato, con archi pesati, si vuole individuare un **albero** (quindi senza cicli) **di copertura** (albero che colleghi tutti i nodi del grafo), tale che la somma dei pesi degli archi sia minima.



L'albero di copertura minima è un indicatore della fault tolerance di una rete.

Due algoritmi: Kruskal, Prim.

Algoritmo di Kruskal

Idea: ordinare gli archi in ordine di peso (algoritmo greedy).

```
MST-Kruskal(G,w)
  A = ∅
  foreach vertice v ∈ V[G] do
    Make-Set(v)
  ordina gli archi di E(G) per pesi non decrescenti
  foreach (u,v) ∈ E[G] in ordine di peso non decrescente do
    if Find-Set(u) ≠ Find-Set(v)
    then A = A U {(u,v)}
    Union(u,v)
  return A

Make-Set(v) - Crea un insieme con unico membro V
Find-Set(v) - Restituisce il rappresentante dell'insieme contenente v
Union(u,v)  - Unisce i due insiemi che contengono u e v
```

Per dimostrare che è ottimo è sufficiente capire che, presi due insiemi disgiunti, di vertici del grafo appartenenti ad un qualche cammino minimo di copertura, unendoli con l'arco di peso minimo che li connette, si ottiene un sottografo appartenente ad un MST.

Complessità O(E lg E) = O(V) + O(E lg E) + O(E):

- Inizializzazione: O(V)
- Ordinamento degli archi: O(E lg E)
- Operazione nella foresta di insiemi disgiunti: O(E)

Algoritmo di Prim

Idea: parto da un nodo, scelgo l'arco di frontiera più leggero, collego il nodo ad esso connesso.

Arco di frontiera: arco tra un nodo scelto e i nodi da scegliere.

```
MST-Prim(G,w,r)
  Q = V[G]
  foreach u ∈ Q do key[u] = ∞
  key[r] = 0
  π[r] = nil
  while Q ≠ Ø do
    u = Extract-Min(Q)
    foreach v ∈ Adj(u) do
      if v ∈ Q and w(u,v) < key[v]
      then π[v] = u
           key[v] = w(u,v)

w(u,v) - Calcola il peso del cammino da u a v
```

Complessità: T(V,E) = O(V) + O(V log V) + O(E) = O(E + V log V)

CAMMINI MINIMI

Dato un grafo si vuole calcolare il **cammino minimo** da un nodo (**sorgente**) ad un nodo (**destinazione**) tale che il peso complessivo degli archi sia minimo.

A parità di costo computazionale è possibile calcolare il cammino minimo da un nodo a tutti gli altri.

Caso semplice: tutti gli archi hanno peso uguale.
In questo caso è sufficiente utilizzare la visita in ampiezza del grafo (BFS) (costo: O (|V| + |E|) (ovvero: numero di nodi + numero di archi).

Caso complesso: archi con pesi diversi.
In questo caso, la visita BFS non funziona perché trova i cammini con meno archi, e non tiene conto del peso degli archi.

Ipotesi: non esistono cicli con pesi negativi.

```
Initialize-Single-Source(G, s)
1 foreach v ∈ V[G] do
2    d[v] = ∞
3    p[v] = nil
4 d[s] = 0
```

```
Relax(u, v, w)
1 if d[v] > d[u] + w(u,v)
2 then d[v] = d[u] + w(u,v)
3       p[v] = u
```

Rilassamento: operazione che associa ad un nodo il peso minore tra il peso di due cammini che arrivano al nodo stesso.

Il tempo di esecuzione di Dijkstra varia in base all'implementazione della coda Q.

1) **vettore ordinato**
v^3 : ??

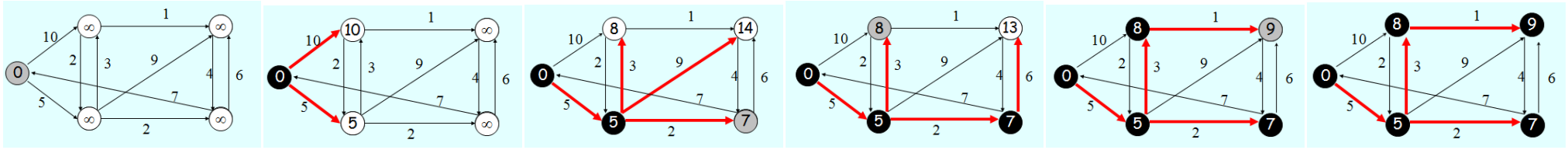
2) **vettore disordinato**
v^2 : ??

3) **heap**
$v^2 \log v$: ??

Algoritmo di Dijkstra

Mantiene un insieme S che contiene i vertici v il cui peso del cammino minimo da s, $\delta(s,v)$, è già stato determinato.

```
Dijkstra(G,w,s)
1. Initialize-Single-Source(G,s)
2. S = ∅
3. Q = V[G]
4. while Q ≠ 0 do
5.   u = Extract-Min(Q)
6.   S = S ∪ {u}
7.   foreach v ∈ Adj[u] do
8.     Relax(u,v,w)
```



Algoritmo di Bellman-Ford

Possibili archi con peso negativo.
Restituisce un booleano che dice se esiste un ciclo di peso negativo (nessuna soluzione) oppure produce l'albero dei cammini minimi.

```
BellmanFord(G,w,s)
1 Initialize-Single-Source(G,s)
2 for i = 1 to |V[G]−1| do
3   for (u,v)∈ E[G]  do
4     Relax(u,v,w)
5 for (u,v) ∈ E[G]  do
6   if d[v] > d[u] + w(u,v)
7   then return false
8 return true
```

Questo algoritmo funziona anche con pesi negativi, ma si ferma se individua clicli negativi.

Dimostrazione di correttezza

i) Se esiste un cammino minimo, l'algoritmo di Bellman-Ford, rilassando tutti i nodi, in un qualche momento rilassa anche i nodi del cammino minimo. Dopo aver rilassato tutti gli archi l'algoritmo ha individuato il cammino minimo.

ii) Se esiste un ciclo negativo, ad ogni ciclo di rilassamento, si abbassa il valore del cammino minimo.

Quindi, dopo aver rilassato tutti i nodi (con n-1 cicli di rilassamento); se al n-esimo ciclo esiste almeno un arco da rilassare allora esiste un ciclo negativo.

Costo dell'algoritmo

1 c: 1
2 c: v-1
3 c: e
4 c: 1
5 c: e
6 c: 1
7
8

Costo: (v-1) * e = (v-1) * v^2 = v^3

ESERCIZI

ESERCIZIO: MAGGIORANZA

Problema: dato n vettore di interi l'algoritmo deve restituire l'elemento di maggioranza := elemento che compare almeno n/2 + 1 volte nell'array.

Soluzione

i. Si imposta un contatore a 0
ii. Si legge il primo elemento del vettore, lo si memorizza e si incrementa il contatore
iii. L'attuale elemento è l'unico candidato ad essere elemento di maggiornanza
iv. Si legge il successivo elemento del vettore
v. Se il successivo è uguale si incrementa il contatore
vi. Se il successivo è diverso si decrementa il contatore
vii. Se il contatore è 0 si torna al punto iii altrimenti al punt iv
viii. Alla fine del ciclo se il contatore è maggiore di 0, l'elemento che si sta confrontando è candidato ad essere elemento di maggioranza.
ix. Effettuo un ciclo di verifica in cui conto le occorrenze dell'elemento candidato
x. Se le occorrenze sono > n/2 il candidato è un elemento di maggioranza.

L'idea è che se alla fine il contatore è maggiore di 0 significa che più della metà degli elementi (dell'ultimo blocco) sono uguali all'ultimo candidato.

| c1 | c2 | c3 | c4 | c5 |

ESERCIZIO DEL POZZO (O POZZO UNIVERSALE)

Dato un grafo, un nodo P del grafo è un **pozzo** se tutti i nodi del grafo hanno archi verso X e non esistono archi in uscita da P.

Problema: Avendo la matrice di adiacenza dei nodi del grafo, verificare se esiste un pozzo, in tempo lineare.

	N1	N2	N3	P	N5	N6
N1				1		
N2				1		
N3				1		
P	0	0	0	0	0	0
N5				1		
N6				1		

i. Si parte dalla prima cella (gialla)
ii. Si scorre la diagonale principale
iii. Se c'è un **1** significa che N1 non è un pozzo (perché c'è un arco entrante) e si passa al successivo elemento della diagonale
iv. Se c'è uno **0** si inizia a scorrere verso il basso
v. Se si individuano tutti **1** significa che o l'elemento è un pozzo oppure non esistono pozzi (gli elementi precedenti e successivi hanno nodi in uscita)
vi. Allora si controllano tutti gli archi del nodo individuato.
vii. Se, scendendo, si individua uno **0**, si prosegue sulla diagonale dall'elemento con lo **0** in poi.

	N1	N2	N3	P	N5	N6
N1	1			1		
N2		1		1		
N3			1	1		
P	0	0	0	0	0	0
N5				1		
N6				1		

	N1	N2	N3	N4	N5	N6
N1	1					
N2		0				
N3		1				
N4		0		1		
N5					1	
N6						1

E su un arco non orientato???

ESERCIZIO: DETERMINARE IL NUMERO DI CAMMINI DI LUNGHEZZA H+1 IN UN GRAFO

Numero di cammini da un nodo ad un altro di lunghezza h						
	N1	N2	N3	N4	N5	N6
N1	12	4	7	4	21	5
N2						
N3						
N4						
N5						
N6						

×

Matrice di adiacenza degli archi						
	N1	N2	N3	N4	N5	N6
N1	0					
N2	0					
N3	1					
N4	0					
N5	0					
N6	1					

=

Numero di cammini da un nodo ad un altro di lunghezza h + 1						
	N1	N2	N3	N4	N5	N6
N1	12					
N2						
N3						
N4						
N5						
N6						

Se M è la matrice di adiacenza, gli elementi della matrice M^n sono i cammini lunghi n da un nodo ad un altro.

$M^n + M^{n-1} + M^{n-2} + ... + M$ corrisponde alla matrice di tutti i cammini da lunghi n o meno da un nodo ad un altro.

$M^n + M^{n-1} + M^{n-2} + ... + M$ è il polinomio caratteristico della matrice.

ESERCIZIO: DIAMETRO DI UN GRAFO

Idee:

- Visita in ampiezza: però la visita in ampiezza dipende dal nodo da cui parto
- Allora, visita in ampiezza da qualsiasi nodo e memorizzo la distanza massima
- Con il prodotto tra matrici M x M x M x M x .... riesco a trovare, di volta in volta, la matrice di adiacenza con cammini lunghi n... quando la matrice non ha nessuno 0 tra i suoi elementi, allora n-1 è il diametro

Probabilmente si impiega meno tempo facendo le visite in ampiezza tra tutti i nodi.

Trovare un algoritmo migliore per calcolare il diametro.

ESERCIZIO: DATI DUE NODI A E B IN UN GRAFO NON ORIENTATO, DETERMINARE IL NUMERO DI NODI EQUIDISTANTI DA A A B

Un nodo C è equidistante da A e B se i cammini minimi da A a C e da B a C hanno la stessa lunghezza.

Visita in ampiezza da A. Poi visita in ampiezza da B. Quando in un nodo scrivo lo stesso valore ho trovato un nodo equidistante.

ESERCIZIO: DATO UN ALBERO BINARIO CALCOLARE RICORSIVAMENTE LA SUA ALTEZZA

h(p)

if p = NIL then return 0

else

return max(**h**(p.left(),**h**(p.right))+1

ESERCIZIO: DATO UN ALBERO BINARIO CALCOLARE RICORSIVAMENTE IL NUMERO DI NODI

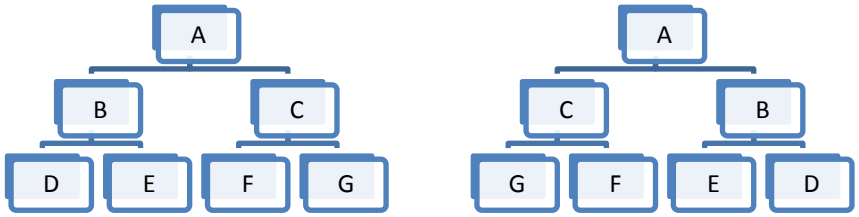
n(p)

if p = NIL then return 0

else

return n(p.left) + n(p.right) +1

ESERCIZIO: DATO UN ALBERO BINARIO INVERTIRNE SPECULARMENTE GLI ELEMENTI



mirror(p)

```
if p != NIL
  tmp=p.left;
  p.left=p.right;
  p.right=p.left;
  mirror(p.left);
  mirror(p.right);
```

ESERCIZIO: DATO UN ALBERO BINARIO TROVARE IL NUMERO DI NODI A DISTANZA DISPARI DALLA RADICE

```
T := puntatore all'albero

conta ( T )

if T = NIL return ( 0 )

else

  return 1 + conta ( T->left->left ) + conta ( T->left->right ) + conta ( T->right->left ) + conta ( T->right->right );
```

per contare quelli di lunghezza dispari: contaDispari (T) = conta (T.left) + conta (T.right)

ESERCIZIO: DATO UN GRAFO NON ORIENTATO TROVARE IL NUMERO MINIMO DI NODI TALI CHE COPRANO TUTTI GLI ARCHI

Tipo di esercizio: edge covering (vedere anche vertex covering). cercare P vs NP

Definizione: grado del nodo (degree) := numero di archi che partono dal nodo.

Se ordino i nodi per grado decrescente e inserisco i nodi nella copertura, prima o poi copro tutti gli archi. Però non è ottimo perché non è detto che trovi la copertura minima.

Trovare l'algoritmo ottimo.

	N1	N2	N3	N4	N5	N6
N1						
N2						
N3						
N5						
N5						
N6						

In questi casi si possono utilizzare algoritmi di approssimazione che sono sicuramente peggiori dell'algoritmo ottimo, però garantiscono una soluzione ottima con una costante moltiplicativa accettabile...

ESERCIZIO: TROVARE UNA FUNZIONE PER TROVARE IL PREDECESSORE IN UN ALBERO BINARIO DI RICERCA. POI UTILIZZARE QUESTA FUNZIONE PER ORDINARE L'ALBERO BINARIO DI RICERCA.

1. trovo il massimo nel BST (ultimo nodo a destra) e lo metto in ultima posizione del vettore

2. cerco il predecessore e lo metto in penultima posizione del vettore

Costo: $h(T) * n = n * n = n^2$

ESERCIZIO: DATO UN VETTORE DI INTERI POSITIVI, TROVARE DUE NUMERI TALI CHE | X - 3 Y | È MINIMO

2	4	7	11	14	16	20	21

(problema simile a min-gap)

Ordino il vettore → O n log n

2	4	7	11	14	16	20	21

Creo un vettore "copia" moltiplicato per 3

6	12	21	33	42	48	60	63

Per ogni elemento del primo vettore, cerco l'elemento più vicino nel secondo vettore.

PROBLEMA "2SAT"

Data una funzione booleana in formula normale congiuntiva con 2 letterali per clausola, descrivere un algoritmo che verifichi se la funzione è soddisfacibile, ovvero se esiste una combinazione delle variabili booleane che rendono la funzione verificata.

L'algoritmo deve avere un costo computazionale polinomiale.

Formula Normale Congiuntiva con 2 letterali per clausola: (X or Y) and (not X or Y) and ...

L'idea è di utilizzare un grafo per rappresentare i cammini trai i nodi, dove i nodi rappresentano le combinazioni "variabile - stato".

(X, V); (X, F); ...; (Y, V); (Y, F); ...

Con una visita (costo V + E) posso verificare se esistono cammini che vanno da un nodo di tipo (X, V) a (X, F) o viceversa (assurdi).

Nel caso in cui esista un cammino che non unisce due nodi di questo tipo, allora quel cammino rappresenta un assegnamento (soluzione) che verifica la funzione.

Devo fare una visita (V + E) per ogni nodo, i nodi sono 2 V, dove n sono le variabili booleane. Il costo complessivo è 2 2 V (V + E) = 4 V (V + E)

DATO UN GRAFO G = (V, E) NON ORIENTATO, PESATO, DIMOSTRARE CHE SE ESISTE UN ARCO CON PESO MINIMO, ALLORA OGNI MINIMO ALBERO DI COPERTURA CONTIENE L'ARCO.

e := arco con peso minimo

Dimostrazione per assurdo.

Supponendo per assurdo che non sia vera l'ipotesi: $\forall$ **T** $\in$ MST, **e** $\notin$ **T**

Allora, un qualsiasi albero di copertura che non contiene l'arco **e** copre tutti i nodi del grafo; pertanto aggiungendo l'arco **e** a **T**, si crea un ciclo.

A questo punto è possibile eliminare un arco del ciclo (che parta da uno dei due nodi connessi da **e**), definiamo tale arco **f**.

Allora il peso complessivo del MST **T** = peso (MST **T**) - peso (**f**) + peso (**e**) ma peso (**f**) $\geq$ peso (**e**) $\Rightarrow$ **e** $\in$ T , T $\in$ MST **che è assurdo**

Il teorema è dimostrato.

DATO UN GRAFO G = (V, E) NON ORIENTATO, PESATO, DIMOSTRARE CHE SE ESISTE UN ARCO CON PESO MINIMO, ALLORA NON ESISTE NESSUN MINIMO ALBERO DI COPERTURA CHE CONTIENE L'ARCO.

e := arco con peso minimo

Dimostrazione per assurdo.

Supponendo per assurdo che non è vera l'ipotesi: $\exists$ **T** $\in$ MST, **e** $\notin$ **T**

Allora, un qualsiasi albero di copertura che non contiene l'arco **e** copre tutti i nodi del grafo; pertanto aggiungendo l'arco **e** a **T**, si crea un ciclo.

A questo punto è possibile eliminare un arco del ciclo (che parta da uno dei due nodi connessi da **e**), definiamo tale arco **f**.

Allora il peso complessivo del MST **T** = peso (MST **T**) - peso (**f**) + peso (**e**) ma peso (**f**) > peso (**e**) $\Rightarrow$ **e** $\in$ T1 , T1 $\in$ MST

ma T1 è minore di T, **che è assurdo** perché T era, per ipotesi, un MST.

Il teorema è dimostrato.

Valgono le due proposizioni precedenti anche per il secondo arco (in ordine di peso)?

Nel caso di peso strettamente crescenti. Nota: i pesi devono essere tutti distinti.

ESERCIZIO: DATO UN GRAFO ORIENTATO, DESCRIVERE UN ALGORITMO PER EFFETTUARE L'ORDINAMENTO TOPOLOGICO DEI NODI DEL GRAFO. (CFR. GRAFO DELLE PRECEDENZE).

Ordinamento topologico: ordinare i nodi in modo tale che, disegnando gli archi, siano tutti orientati in avanti.

Dimostrare che, dato un grafico aciclico, è possibile effettuare l'ordinamento.

Si può utilizzare la visita in profondità.

Concetto: ogni nodo ha un momento di "inizio visita" appena lo prendo in esame e di "fine visita" quando ho torno al nodo dopo aver visitato tutti i nodi (non ancora visitati) da esso raggiungibili.

Effettuo una visita in profondità e registro l'ordine dei nodi in base a "fine visita" (ovvero ogni volta che finisco la visita di un nodo lo inserisco in un array o lista). L'array conterrà l'ordinamento topologico (crescente o decrescente a seconda che si inseriscano i nodi in testa o in coda).

Si utilizzano pertanto i tempi di inizio e fine visita generati dalla DST.

PROBLEMI INTRATTABILI (ESPONENZIALI)

Edge covering, vertex covering, knapsack binario, nodi tricolorabili, commesso viaggiatore, ...

PROBLEMI FAMOSI

è sempre possibile colorare un grafo planare con 4 colori in modo che i nodi siano sempre di colori distinti? SI

(è stato dimostrato con un calcolatore...., primo caso in cui un computer ha dimostrato un teorema che l'uomo non sa dimostrare)

GLOSSARIO

Funzione deterministica

Funzione che, a partire da valori di input fissi dati, restituisce sempre lo stesso valore.

Funzione randomizzata

Funzione che, a partire da valori di input fissi dati, restituisce valori differenti.

Grafo planare

Un grafo che si può disegnare senza incrociare nessun arco. (le cartine geografiche sono grafi planari)

DOMANDE

1) Algoritmo di PRIM: non mi è chiaro il procedimento né il calcolo del costo.

2) Algoritmo di Dijkstra: idem

3) Che differenza c'è tra un albero minimo di copertura e i cammini minimi da un nodo a tutti gli altri?

4) Cosa fa la funzione Relax sulla coda?

5) Selection

   a. Perché il costo di ordinamento delle colonne è n?
   b. Perché 7/10 n?
   c. 5° slide, cos'è "c"?