

Insertion Sort

AUTORE: Denny Acciaro
[\[acciariogennaro@gmail.com\]](mailto:acciariogennaro@gmail.com)

Fonti

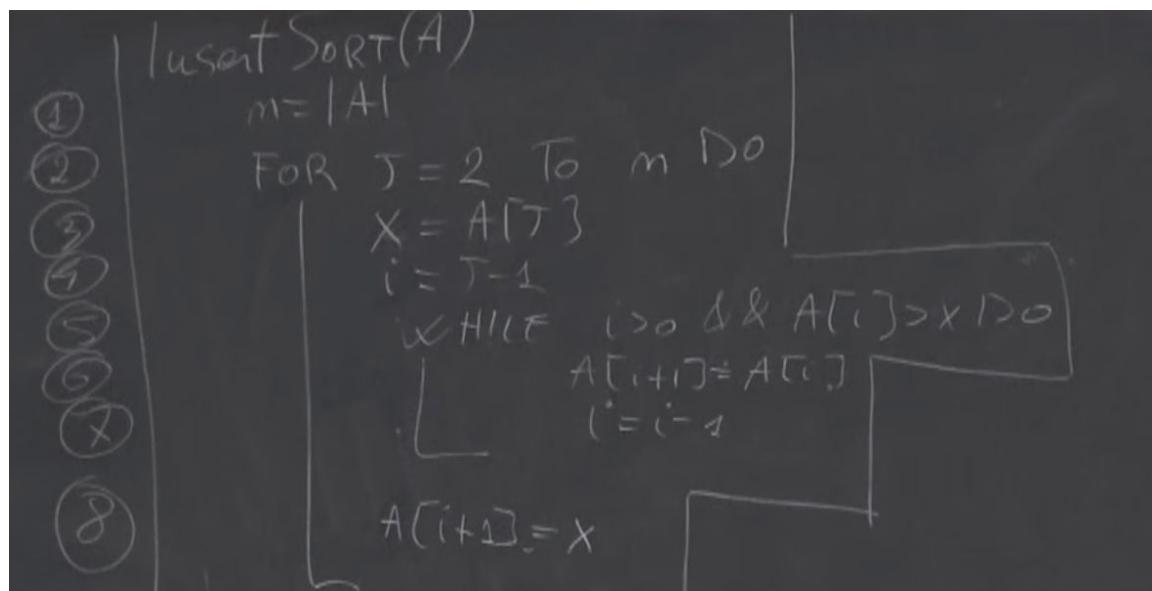
1. <https://www.youtube.com/watch?v=oqA-c7gbahk> (**Lezione 7**)
 2. <https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort>
-

Introduzione	1
Codice	2
Analisi	2
Caso peggiore	5
Caso migliore	6
Caso medio	6
Conclusioni finali	7

Introduzione

Insertion Sort è un algoritmo di ordinamento che agisce su una sequenza di elementi su cui si può definire un ordinamento. Considerando un' istanza grande n il primo elemento di quest'ultima è già ordinato (caso semplice, istanza base) da qui si può iniziare a definire una parte ordinata e una ancora da ordinare. Si procede iterativamente a considerare il secondo, posizionandolo di conseguenza rispetto a quelli già ordinati (finché trova elementi più grandi continua a scorrere la parte ordinata fino a giungere al primo minore o uguale) una volta posizionato si considera il primo elemento della sequenza disordinata e si trova il suo posto in quella ordinata finché non si giunge a una sequenza tutta ordinata.

Codice



Analisi

La riga 1 effettua un'operazione non elementare che si può interpretare come risultato di un'altra funzione che scorre tutta la sequenza A calcolandone la lunghezza. Per questo è una funzione lineare e quindi può essere descritto da $\Theta(n)$.

La riga 1 viene eseguita una volta sola.

La riga 2 è la testa di un ciclo for ed in quanto tale viene eseguita sempre una volta in più rispetto al suo corpo.

In questo caso ci sono $n-2+1$ ($= n-1$) elementi tra 2 ed n, quindi la riga 2 verrà eseguita 1 volta in più a questo valore, quindi viene eseguita n volte.

Dato che il numero di operazioni elementari eseguite nella riga 2 sono costanti ed indipendenti da n possiamo astrarre ciò ad una certa costante c_2 perché semplicemente ha poco impatto nell'analisi asintotica essendo una costante moltiplicativa.

Lo stesso discorso si applica alla riga 3 che avrà un c_3 operazioni elementari e verrà eseguita $n-1$ volte (perché fa parte del corpo di un for che viene eseguito n volte).

Questo comportamento viene applicato alla riga 4 e alla riga 8 che verranno eseguite anch'esse $n-1$ volte e che avranno rispettivamente le costanti c_4 e c_8 ad indicare il loro numero di operazioni fondamentali.

Il vero problema di quest'analisi è rappresentato dal ciclo while e dal suo corpo.

Dal punto di vista di numero di operazioni fondamentali, anche qui possiamo astrarre banalmente questi tre valori in tre costanti: c5, c6 e c7 (rispettivamente per la riga 5, 6 e 7).

Per quanto riguarda il numero di esecuzioni del ciclo while, possiamo immaginare di fissare un certo valore del ciclo esterno (che è discriminato dall'indice j) e ci chiediamo quante volte viene eseguito il ciclo while, con j fissato.

Purtroppo però considerando solo j non sappiamo quante volte la condizione del ciclo while sarà vera perché dipende da come evolve la variabile i .

Per questo motivo dobbiamo analizzare cosa fa questo ciclo:

- 1) i viene inizializzata ad $j-1$
- 2) per ogni iterazione del quel while i viene decrementato fino al massimo di $i=0$

Nel caso peggiore, ossia nel caso in cui dobbiamo inserire il valore nella prima posizione dell'array, la testa ciclo while verrà eseguito j volte perché se l'elemento $A[j]$ (disordinato) lo voglio andare a mettere nella parte ordinata e $A[j]$ è minore di tutti gli elementi della parte ordinata (\Rightarrow deve essere messo in posizione 0) si hanno un numero di iterazioni pari ad j , semplicemente perché la lunghezza della parte ordinata è j .

Nel caso migliore, invece, ossia quando la condizione del while è falsa inizialmente la testa del ciclo verrà eseguita 1 volta sola.

In altre parole, il numero di iterazioni della riga 5 in dipendenza di j (ossia, fissando j) sarà compreso tra 1 (valore minimo) e j (valore massimo).

$$1 \leq \#5(j) \leq j$$

Il numero di iterazioni della riga 5 sarà sempre più grande di 1 rispetto al numero di iterazioni della riga 6 e della riga 7 (perché è la testa del ciclo).

Segue che:

$$\#7(j) = \#6(j) = \#5(j) - 1$$

Da tutto ciò si nota che il numero di iterazioni del ciclo while dipende della variabile j .

Ad esempio se $j = 2$, la riga 5 può essere eseguita una o due volte.

Oppure se $j = 10$, la riga può essere eseguita da una a dieci volte.

Questo però non ci basta per determinare il numero di esecuzioni preciso, che oltre a dipendere da j , dipende anche dai valori contenuti nell'array stesso (input).

Quello che possiamo fare adesso è definire un certo insieme di parametri, uno per ogni valore di j . Poniamo quindi T_j = al numero di esecuzioni della riga 5 per un j fissato

$$T_j = \#5(j)$$

Fissato j posso sapere che la riga 5 viene eseguita T_j volte ma dato che T_j è compreso tra 1 e n , posso esprimere il numero di esecuzioni come $t_2 + t_3 + t_4 + \dots + t_n$

$$T_2 + T_3 + T_4 + \dots + T_n = \sum_{j=2}^n T_j$$

Ne consegue che la riga 6 e la riga 7 verrà eseguita il numero di volte della riga 5 meno uno, ossia ad ogni iterazione di j (che è l'indice del for esterno) il corpo viene eseguito 1 volta in meno, per questa ragione le righe 6 e 7 vengono eseguite un numero di volte pari ad

$$\sum_{j=2}^n T_j - 1$$

Adesso possiamo definire l'espressione di tempo dell'algoritmo

Riga	Numero di Operazioni	Numero di esecuzioni
1	$\Theta(n)$	1
2	c_2	n
3	c_3	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=2}^n t_j$
6	c_6	$\sum_{j=2}^n t_j - 1$
7	c_7	$\sum_{j=2}^n t_j - 1$
8	c_8	$n-1$

Moltiplicando il numero di operazioni e il numero di esecuzioni avremo un tempo pari ad

$$\Theta(n) + C_2 n + (C_3 + C_4 + C_8)(n-1) + \\ (C_6 + C_7) \cdot \sum_{j=2}^n (t_j - 1) + C_5 \sum_{j=2}^n t_j$$

Da questa espressione è chiaro che non è possibile estrarre un'unica funzione che dipende unicamente da n , perché diverse sequenze di lunghezza n possono generare casi molto differenti, in base determinano valori diversi dei parametri T_j . In altre parole, è chiaro che (fissato n) i valori che non sono delle righe 5, 6 e 7 sono calcolabili, mentre quest'ultimi dipendono anche della qualità dell'input.

Possiamo cercare di raggruppare le sequenze in ingresso in "classi" che vengono definite in questo modo:

- 1) Classe delle istanze che determinano il comportamento peggiore
- 2) Classe delle istanze che determinano il comportamento migliore
- 3) Classe delle istanze che non determinano il comportamento né peggiore né migliore

Tornando alle righe 5,6,7: più grande sarà valore delle due sommatorie, più grande sarà il suo tempo d'esecuzione. Dobbiamo trovare adesso degli input che minimizzano (massimizzano) le quantità delle due sommatorie per avere le sequenze del caso migliore (peggiore).

Caso peggiore

Esiste una sequenza in cui ogni iterazione del for esterno, il while esegue tutte le esecuzioni possibili (che al massimo sono j , perché $1 \leq \#(J) \leq J$)?

Sì, esiste ed è una sequenza di ordine in modo decrescente.

Nel caso peggiore $t_j = j \quad \forall 2 \leq j \leq n$

Si noti che dev'essere **per ogni j (indice del ciclo for)**!

Sostituiamo quindi il valore $T_j=j$ nell'espressione generale trovata precedentemente:

$$\sum_{j=2}^n t_j = \sum_{j=2}^n j = \left(\sum_{j=2}^n j \right) - 1 = \sum_{j=2}^{n-1} j = \frac{n(n-1)}{2}$$

Ciò avviene perché:

- (1) \rightarrow (2) : questo è vero per la tipologia di input che abbiamo in considerazione (caso peggiore).
- (2) \rightarrow (3) : vero sempre

(3) \rightarrow (4) : vero perché le espansioni di (3) e (4) sono uguali

(4) \rightarrow (5) : vero banalmente

Quindi le due sommatorie daranno un contributo al $T(n)$ generale pari ad $\Theta(n^2)$ e dato che la parte non dipendente da j è lineare e il contributo delle sommatorie è quadratico, concludiamo che nel caso peggiore l'algoritmo impiega tempo quadratico.

$$T_{CP}(n) = \Theta(n^2)$$

Caso migliore

Sempre dall'analisi precedente abbiamo notato che $1 \leq t_j \leq J$ (quindi T_j al minimo vale 1) per trovare il caso migliore dobbiamo chiederci:

Esiste una sequenza in input che forza T_j ad essere 1. per ogni j ?

Sì, la sequenza ordinata in ordine non decrescente.

Questo avviene perché la testa del ciclo while viene eseguita una volta sola per ogni j , perché appunto non deve ordinare.

Nel caso migliore $t_j = 1 \quad \forall 2 \leq j \leq n$

Sostituiamo quindi il valore $t_j=1$ nell'espressione generale trovata precedentemente:

$$\sum_{j=2}^n T_j = \sum_{j=2}^n 1 = n - 1$$

(1) \rightarrow (2) : vero perché sommo sempre 1

(2) \rightarrow (3) : sommo 1 da $j=2$ fino a n volte $\rightarrow n-1$ volte

Nota che stiamo considerando solo la sommatoria della testa del ciclo while, perché la sommatoria del corpo del while da contributo zero, perché non viene mai eseguito.

Ne consegue che $T(n)$ è la somma di 4 termini lineari; quindi

$$T_{CM}(n) = \Theta(n)$$

Caso medio

Infine ci chiediamo: tutte le altre istanze di problemi che non sono né migliori né peggiori, hanno un andamento più lineare, più quadratico oppure stanno nel "mezzo"?

Per far ciò calcoliamo la media aritmetica fissando sempre un j , sommiamo tutti i valori di T_j e poi dividiamo per j . La somma quindi è pari a $j(j+1)/2$ e la dobbiamo dividere per j .

Quindi avremo $(j+1)/2$, ciò significa che possiamo aspettarci un comportamento del genere: T_j è in media la metà della sequenza ordinata; per alcune sequenze sarà di più, per altre sarà meno, ma facendone una stima stiamo su quest'ordine.

Quindi dato che $T_j = j/2$, ma questo se viene sostituito nella funzione $T(n)$ ci restituisce esattamente il caso peggiore diviso 2 [nda.ricorda che il caso peggiore aveva $T_j=j$]

Questo però dato che asintoticamente i coefficienti costanti non “importano” concludiamo che:

$$T_{CMedio}(n) = \Theta(n^2)$$

Conclusioni finali

Caso migliore	$T_{CP}(n) = \Theta(n)$
Caso peggiore	$T_{CM}(n) = \Theta(n^2)$
Caso medio	$T_{CMedio}(n) = \Theta(n^2)$

Questo algoritmo è semplice da realizzare ma ha una **forte dipendenza** della qualità della sequenza dell’input. Infatti non è un algoritmo ottimale per l’ordinamento ma è utile solo nel caso è necessario ordinare sequenze già più o meno ordinate.

Merge Sort

AUTORE: Denny Acciaro
[acciarogennaro@gmail.com]

Fonti

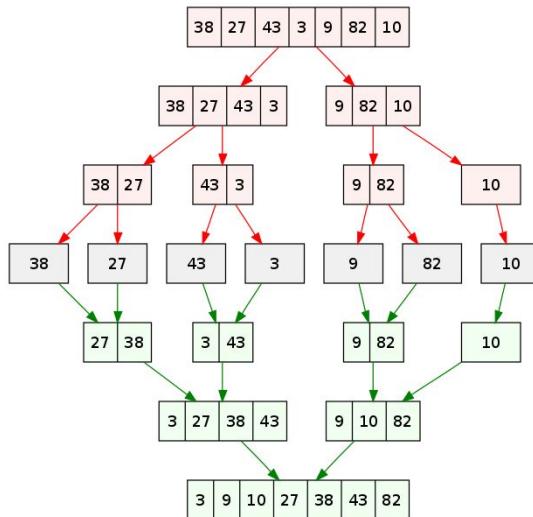
1. https://www.youtube.com/watch?v=GUZHvu_ySsM (Lezione 8)
2. <https://www.youtube.com/watch?v=NKauNYP7IuI> (Lezione 9)
3. <https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort>

Codice	2
Analisi	3
Condizioni per la terminazione	3
Dimostrazione: $p < r \Rightarrow p \leq q < r$?	4
Analisi asintotica	5
Conclusioni finali	7

Il merge sort si basa sul decomporre il problema in sotto-problemi (strettamente più piccoli rispetto al problema originale) e risolvere questi problemi ricorsivamente fino ad arrivare ad un sotto-problema facilmente risolvibile.

Una volta ottenute le varie soluzioni dei sotto-problemi, vengono ricomposte fino a trovare la soluzione del problema originario.

Graficamente avremo una situazione del genere:



Per implementare questa idea dobbiamo sostanzialmente risolvere due problemi:

- 1) Come effettuare la divisione delle istanze
- 2) Come effettuare il merge

Codice

```
MergeSort(A, p, r)
if p < r then
    q =  $\frac{p+r}{2}$ 
    MergeSort(A, p, q)
    MergeSort(A, q+1, r)
    Merge(A, p, q, r)
```

dove:

- A = array da ordinare
- p = inizio della sottosequenza da ordinare
- r = fine della sottosequenza da ordinare

La parte del “divide” viene espressa da $q = p+r/2$, mentre la parte del “impera” è espressa dalle due chiamate ricorsive a MergeSort stesso.

Analisi

Condizioni per la terminazione

Essendo un algoritmo ricorsivo è necessario studiarlo per capire se termina, in particolare si noti che:

- 1) **Tutte le sottosequenze di ordine 1 o 0, sono già ordinate.** Per questa ragione l'algoritmo entra in funzione se e solo se la lunghezza della sottosequenza da ordinare è maggiore strettamente di 1; questa condizione viene applicata dall'if, infatti:
 - a) lunghezza della sottosequenza da ordinare = numero di elementi tra p e $r = r-p+1$
 - b) poniamo questa lunghezza $> 1 \Rightarrow r-p+1>1$
 - c) $r-p>1-1$
 - d) $r-p>0$
 - e) $r>p$, ossia, $p < r$

Quindi tramite questa condizione determiniamo il caso base e se in una iterazione mi trovo davanti ad una sottosequenza che non è il caso base, vado avanti.

- 2) **La variabile q rappresenta il punto medio della lunghezza della sottosequenza.** Tramite esso, riusciamo a dividere la sequenza da ordinare in due sotto sequenze; la prima va da p a q (lato sinistro) e la seconda va da $q+1$ ad r (lato destro).

Da queste due osservazioni possiamo creare il seguente sistema dal quale possiamo dire che il valore q deve essere compreso tra p e r in questo modo:

$$\begin{cases} p < r \\ q = \frac{p+r}{2} \end{cases} \rightarrow p \leq q < r$$

Nota bene che non è possibile ammettere l'uguaglianza tra q ed r perché altrimenti la seconda chiamata ricorsiva andrebbe da $r+1$ ad r , effettuando la chiamata su una sequenza vuota invece se q fosse uguale a p non avrei problemi perché la prima chiamata ricorsiva lavorerebbe su un range che va da p a p (quindi un elemento) e l'altra chiamata ricorsiva lavorerebbe da $p+1$ ad r ossia il resto degli elementi.

Quindi ne consegue che la proprietà che bisogna garantire affinché la terminazione dell'algoritmo è garantita è proprio:

$$p \leq q < r, \text{ ossia } p \leq (p+r)/2 < r$$

Dimostrazione: $p < r \Rightarrow p \leq q < r$?

Sapendo che $p < r$ è possibile dimostrare che $p \leq q < r$ sia vero?

Quindi:

Ipotesi: $p < r$

Tesi: $p \leq q < r$

- 1) Prendiamo l'ipotesi e aggiungendo p ad ambo i membri:

$$p < r \Rightarrow 2p < p + r$$

- 2) Adesso se replica gli stessi passaggi, ma invece di aggiungere p aggiungo r , otteniamo la verifica anche della seconda parte del sistema:

$$p < r \Rightarrow p + r < 2r$$

- 3) Dato che valgono entrambe, mettiamole a sistema eabbiamo che:

$$\begin{cases} 2p < p + r \\ p + r < 2r \end{cases} \Rightarrow 2p < p + r < 2r$$

- 4) Dividendo tutto per due abbiamo che vale:

$$p < \frac{p+r}{2} < r$$

- 5) Dato che la nostra tesi è

$$p \leq \frac{p+r}{2} < r$$

e siamo arrivati al punto 4 possiamo dire che la dimostrazione è finita perchè se vale il punto 4 (dove p è strettamente minore di q) allora sicuramente vale il minore o l'uguaglianza tra p e q .

Quindi in questo modo abbiamo verificato la prima parte del sistema della tesi.

Quindi questa dimostrazione ci conferma che:

$$p < q \Rightarrow p \leq p + r < r$$

e questo ci garantisce che entrambe le chiamate ricorsive MergeSort terminino ossia che ogni chiamata all'algoritmo effettua chiamate ricorsive strettamente più piccole e siccome queste dimensioni sono interi arriverò, prima o poi, arriverò alla dimensione 1 o 0 che sono un caso base.

Analisi asintotica

Fissiamo n pari lunghezza della dimensione della sequenza su cui effettuare il MergeSort, ossia: $n = r - p + 1$

Per calcolare l'equazione di ricorrenza del Merge Sort ,leggendo l'algoritmo, possiamo distinguere due casi:

- se la condizione $p < r$ è falsa (terminare l'algoritmo)
- se la condizione $p < r$ è vera (e dobbiamo eseguire il corpo dell'if)

La condizione è falsa se la dimensione dell'input è ≤ 1 ed il tempo di esecuzione è $\Theta(1)$ perché l'unica cosa che bisogna eseguire è la testa dell'if.

Se la condizione è vera la dimensione sarà per forza ≥ 2 ; bisogna quindi considerare i contributi di tutto il corpo dell'if:

- 1) Il confronto e l'assegnazione di q hanno tempo costante, che verrà sintetizzato con $\Theta(1)$.
- 2) Il merge delle due sottosequenze viene espresso con un valore pari a $T_M(n)$
- 3) Le due sottochiamate ricorsive hanno un valore pari a $T_{MS}(n/2)$ perché la lunghezza delle sequenze in input delle due chiamate vale $n/2$. Ciò implica che queste due chiamate impiegheranno lo stesso tempo che impiega merge sort stesso su un input di $n/2$, perciò $T_{MS}(n/2)$.

The handwritten recurrence relation is:

$$T_{MS}(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 1 + T_M(n) + T_{MS}\left(\frac{n}{2}\right) + T_{MS}\left(\frac{n}{2}\right) & \text{se } n \geq 2 \end{cases}$$

Annotations above the relation indicate: $f_{ret} = 1$ and T_{MS} returns f_{ret} .

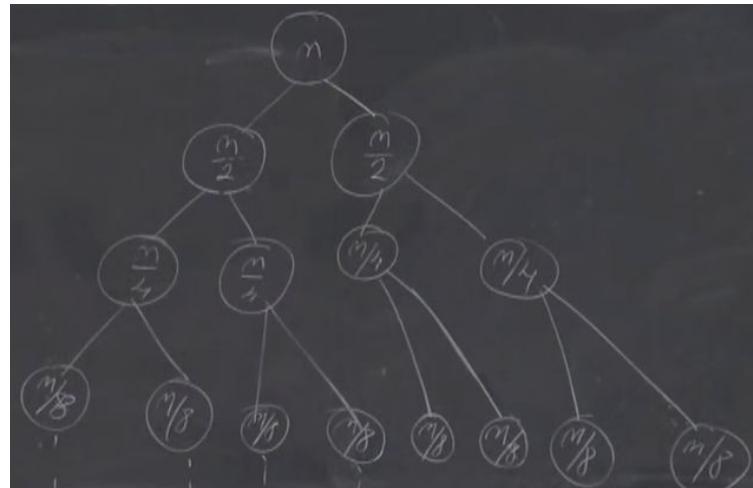
Infine, essendo T_{MS} una funzione che viene usata due volte su parametri uguali $(n/2)$, posso semplificare la somma come $2 * T_{MS}(n/2)$ e dato che il tempo dell'algoritmo di merge è maggiore o uguale ad un tempo costante può assorbire tutte le operazioni costanti.

NB. Per capire questo concetto pensa a questo esempio: $n+1 = \Theta(n)$
In termini asintotici possiamo far sparire il +1

Assumendo che il merge abbia tempo lineare avremo che l'equazione di ricorrenza del MergeSort è pari a:

$$T_{MS}(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2 \cdot T_{MS}\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

Descriviamo l'albero delle chiamate ricorsive, (quello che c'è scritto al centro del nodo è la lunghezza della sequenza sul quale quel nodo esegue MergeSort).



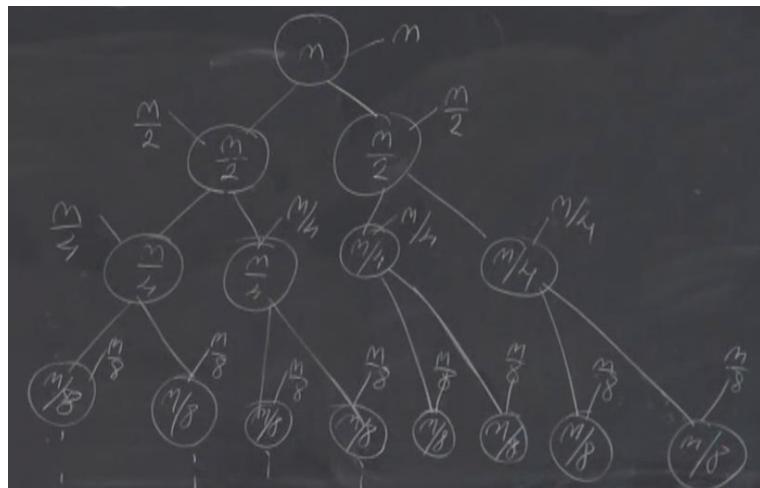
Ovviamente si assume che la radice non faccia parte del caso base, altrimenti terminerebbe subito. Non essendolo quindi genera due chiamate ricorsive che ricevono in input $n/2$. A loro volta, questi due figli, se la dimensione in input è diversa dal caso base, genereranno altri due figli a testa che riceveranno in input la metà di quello che ha ricevuto il loro padre, ossia $n/4$. Ciò continua finché non si raggiunge il caso base.

Qualsiasi sia n in ingresso sicuramente l'albero è finito, ed è dimostrato perché l'algoritmo termina sempre, quindi ogni ramo raggiungerà una foglia.

Si noti che il numero di nodi corrisponde al numero di volte in cui l'algoritmo viene ripetuto, perché ad ogni chiamata ricorsiva il codice dell'algoritmo viene eseguito.

Leggendo ancora l'equazione di ricorrenza dell'algoritmo (quando non è il caso base) vediamo che, escludendo le chiamate ricorsive, **ogni nodo avendo n in input contribuisce con un valore n** (per il tempo del merge).

Per questa ragione possiamo estendere l'albero di prima assegnando ad ogni nodo il proprio contributo (che sarà appunto uguale al valore in input del nodo stesso):



Bisogna adesso **sommare** tutte queste etichette esterne per avere il tempo di esecuzione dell'algoritmo.

Si noti che in questo albero ogni nodo di ogni livello contribuisce con la stessa quantità e la somma di ogni livello contribuisce di n , perché:

(livello 0: n ; livello 1: $n/2+n/2=n$; ecc..)

Quindi ogni livello ha un costo computazionale pari a $\Theta(n)$.

Dato che all'i-esimo livello avremo che la dimensione dell'input sarà pari a: $\frac{n}{2^i}$

Ci serve sapere adesso il numero di livelli, ossia qual è l'altezza dell'albero che sarà pari al livello in cui i nodi ricevono in ingresso un'istanza di dimensione 1 (perché è il caso base).

Quindi mettendo a sistema queste due informazioni avremo che

$$i - \text{esimo livello e' il livello delle foglie} \Leftrightarrow \frac{n}{2^i} = 1$$

ossia se la dimensione dell'input è uguale al caso base.

Quindi per dedurre l'altezza dell'albero, che per definizione è la dalla massima profondità raggiunta dalle sue foglie, dobbiamo risolvere l'equazione per i , avendo così il livello in cui ci sono le foglie, per cui avremo:

$$\frac{n}{2^i} = 1 \rightarrow i = \log_2(n)$$

Ne consegue che la funzione di tempo in forma chiusa sarà uguale ad

$$T_{MS}(n) = \sum_{i=0}^{\log_2 n} n$$

ossia sommo al variare dei livelli da 0 al massimo livello ($\log_2 n$) dei contributi di ciascun livello. Dato che n non dipende dall'indice i , possiamo raccoglierlo a fattor comune, avremo così:

$$T_{MS}(n) = \sum_{i=0}^{\log_2 n} n = n \cdot \sum_{i=0}^{\log_2 n} 1$$

che sarà uguale a $n(\log_2 n + 1)$ che tramite le proprietà della notazione asintotica, possiamo descriverla come $\Theta(n \log_2 n)$

$$T_{MS}(n) = \sum_{i=0}^{\log_2 n} n = n \cdot \sum_{i=0}^{\log_2 n} 1 = n \cdot \log_2 n + 1 = \Theta(n \cdot \log_2 n)$$

La cosa interessante del MergeSort è il **tempo di esecuzione dell'algoritmo è indipendente dalle istanze specifiche**, dipende solo della dimensione; non si comporta in maniera diversa in base a sequenze diverse della stessa lunghezza, come a differenza dell'Insertion Sort. In questo caso, quindi caso medio, peggiore e migliore coincidono sempre e sono pari a

$$\Theta(n \log_2 n)$$

Conclusioni finali

MergeSort è un algoritmo ottimale sia nel caso peggiore, ma anche nel caso medio.

Selection Sort

AUTORE: Denny Acciaro
[\[acciarogennaro@gmail.com\]](mailto:acciarogennaro@gmail.com)

Fonti

- <https://www.youtube.com/watch?v=d3kgKtiVIVY>

Il selection sort si basa sulla seguente idea: data una sequenza, per averla ordinata sappiamo che alla posizione n (l'ultima) si deve trovare il massimo tra elementi della sequenza, il selection sort quindi va a cercare il massimo della sequenza e lo pone in posizione n . Una volta effettuato ciò, "separo" la parte che contiene il massimo (ordinata) con la parte che va dalla posizioni 0 alla $n-1$ (disordinata)

Ripeto questo concetto quindi alla sottosequenza da 0 a $n-1$, sapendo appunto che nella posizione $n-1$ ci deve andare il massimo della sottosequenza da 0 a $n-1$.

Ripeto ciò finché la parte ordinata non copre tutta la sequenza originaria.

Codice

The image shows handwritten pseudocode on a chalkboard. On the left, the `SelectionSort(A, n)` function is defined. It initializes $J = n$, calls `FindMax(A, J)` to get $i = \text{FindMax}(A, J)$, and enters a loop where it swaps $A[i]$ and $A[J]$ using `SWAP(A, i, J)`, decrements J by 1, and then calls `FindMax(A, J)` again. On the right, the `FindMax(A, J)` function is defined. It initializes $\text{mex} = 1$ and iterates from $i = 2$ to J , checking if $A[i] > A[\text{mex}]$. If true, it updates $\text{mex} = i$. Finally, it returns mex .

SelectionSort()

- 1) Fissa j alla massima posizione della sequenza
- 2) Richiama la funzione `FindMax` la quale restituisce la posizione del massimo della sequenza fino a j , questa posizione viene salvata in i .
- 3) Finché ci sono elementi, effettua lo scambio tra gli elementi in posizione i e j .
- 4) Decrementiamo j (in questo modo riduciamo il numero degli elementi della sequenza).
- 5) Ricalcoliamo i per la prossima iterazione.

'A pane e pummaruol diventa: trova il primo massimo, fa lo scambio, decrementa; trova il prossimo massimo, fa lo scambio, decrementa; ecc..

FindMax

- 1) Fisso il massimo alla prima posizione
- 2) Controllo per ogni valore che va dalla seconda all'ultima posizione siano maggiori di max

Analisi

Dato che FindMax effettua un solo ciclo for che va da 2 a j, è lineare su j, quindi $T_{FM}(j) = \Theta(j)$.

Per quanto riguarda SelectionSort() :

- 1) $j=n$ ha complessità di $\Theta(1)$.
- 2) Effettua una chiamata su FindMax, aggiungendo un contributo pari ad $\Theta(n)$
- 3) La testa del ciclo ha un costo di $\Theta(n)$ perché viene eseguita n volte e ogni volta che la eseguo ha un tempo costante, perché è solo un confronto.
- 4) Lo swap() viene effettuato $n-1$ volte (perché è il corpo di un ciclo) e ogni volta che viene fatto ha tempo costante (tre operazioni a tempo costante) $\rightarrow \Theta(n)$
- 5) Ovviamente $j=j-1$ ha tempo costante e viene effettuato $n-1$ volte $\rightarrow \Theta(n)$
- 6) La ricerca del nuovo massimo viene effettuata $n-1$ volte (perché fa parte del corpo del while) e ogni volta che viene eseguito ha un costo di $\Theta(n)$, quindi

$$\sum_{j=1}^{n-1} \Theta(j) = \Theta\left(\sum_{j=1}^{n-1} j\right) = \Theta(n^2)$$

Quindi abbiamo sommato tutti i contributi: $\Theta(1) + 4 * \Theta(n) + \Theta(n^2) \rightarrow T_{SS}(n) = \Theta(n^2)$

Considerazioni

- 1) Questo algoritmo non è il miglior algoritmo per il sorting
- 2) È completamente insensibile rispetto alla "qualità" della sequenza, ossia il suo tempo è sempre $\Theta(n^2)$ indipendentemente della sequenza, quindi non esiste un caso medio, peggiore e migliore.

Appunti sugli Alberi

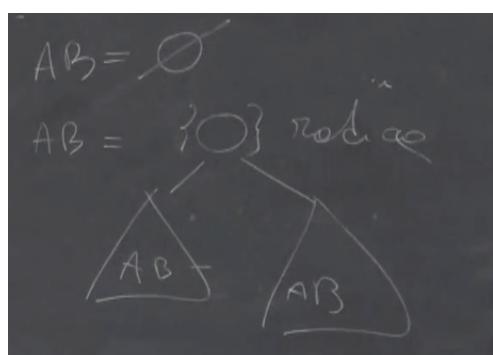
AUTORE: Denny Acciaro
acciariogennaro@gmail.com

Albero binario	1
Albero binario pieno	2
Da h ad n	2
Da n ad h	2
Albero binario completo	4

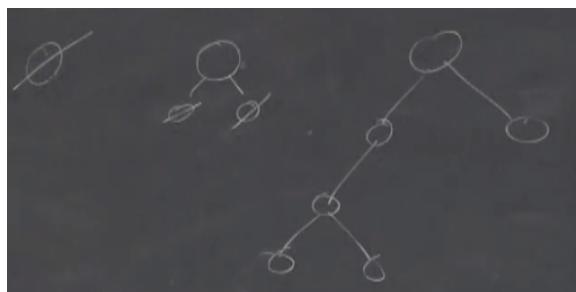
Albero binario

Un albero binario è un insieme di elementi che può essere

- 1) O l'insieme vuoto
- 2) Oppure, un insieme che può essere partizionato in 3 sottoinsiemi disgiunti di cui:
 - a) Un insieme che contiene un solo elemento detto **radice**
 - b) Altri due insiemi che sono sotto-alberi della radice che sono a loro volta alberi binari.



Alcuni esempi di alberi binari:

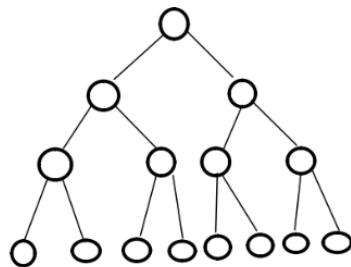


Una **foglia** è un nodo dell'albero senza un figlio.

Definiamo l'**altezza** di un albero come la massima profondità di una foglia.

Albero binario pieno

E' un albero in cui ogni nodo interno ha due figli e **tutte le foglie sono alla stessa profondità**, graficamente avremo una cosa del genere



Data una certa altezza h esiste uno ed un solo albero pieno ad esso associato.

Gli alberi pieni hanno inoltre una funzione biettiva che descrive il rapporto tra l'altezza ed il numero di nodi:

- 1) Da h ad n

Se si ha un altezza h , esistono $2^{(h+1)} - 1$ nodi.

- 2) Da n ad h

Se si hanno n nodi, è possibile ricavare l'altezza h svolgendo i seguenti calcoli:

- a) $n = 2^{(h+1)} - 1$
- b) Applico il logaritmo ad ambo i membri: $\log_2 n = \log_2 (2^{(h+1)} - 1)$
- c) Dato che $(2^{(h+1)} - 1) < (2^{(h+1)})$ anche i loro logaritmi saranno strettamente minori fra loro: $\log_2 (2^{(h+1)} - 1) < \log_2 (2^{(h+1)})$
- d) Ma $\log_2 (2^{(h+1)}) = h+1$, per definizione di logaritmo
- e) Un'altra cosa che possiamo notare è che la quantità $\log_2 (2^{(h+1)} - 1)$ ($=\log_2 n$) è maggiore strettamente di $\log_2 (2^h)$ ($=h$) quindi possiamo dire che $h \leq \log_2 (2^{(h+1)} - 1)$
 - i) $h \leq \log_2 n = \log_2 (2^{(h+1)} - 1) < \log_2 (2^{(h+1)}) = h+1$
- f) Quindi ne conseguе che :
 - i) $h \leq \log_2 n < h+1$, dove h (e $h+1$) sono entrambi interi, ma probabilmente $\log_2 n$ non sarà un intero
- g) Per concludere, se prendo la parte intera di $\log_2 n$ è h perché è il più grande intero che è minore o uguale del numero, quindi
 - i) $h = (\text{int}) \log_2 n$

$$h = \lfloor \log_2 n \rfloor$$

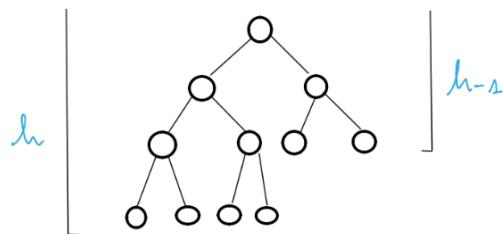
Albero binario completo

L'albero binario pieno ha delle buone proprietà però è strettamente legato al numero di nodi, il quale dev'essere obbligatoriamente pari a $2^{(h+1)} - 1$.

Partendo da un qualsiasi numero di nodi non è sempre possibile far rispettare tale condizione; perciò si giunge ad una sorta di compromesso, ossia gli alberi binari completi.

Definiamo un albero binario completo come un albero binario in cui ogni nodo interno ha 2 figli tranne al più 1, tutte le foglie stanno a profondità h o $h-1$.

Non vale più la proprietà che fissata l'altezza h esiste un solo albero associato, perciò è impossibile determinare il numero di nodi dato h .



Possiamo però dire che un albero completo di altezza h avrà sicuramente meno nodi di un albero pieno di altezza h e avrà almeno un nodo in più rispetto all'albero pieno di altezza $h-1$.

Quindi se n è il numero di nodi dell'albero binario completo avremo che:

$$2^{(h-1+1)} - 1 \leq n \leq 2^{(h+1)} - 1$$

HeapSort

AUTORE: Denny Acciaro
[acciarogennaro@gmail.com]

fonti:

- <https://www.youtube.com/watch?v=qlgalineyxq>
- <https://www.youtube.com/watch?v=URoFMjosM80>
- <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/3-heapsort-1.pdf>
- <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/4-heapsort-2.pdf>
- <https://www.geeksforgeeks.org/heap-sort/>

Heap	1
Codice	3
CostruisciHeap	4
Heapify	5
HeapSort	6
Analisi Asintotica	7
Heapify	7
CostruisciHeap	7
HeapSort	7
Conclusioni finali	13

HeapSort nasce analizzando un grave problema del SelectionSort.

Quest'ultimo infatti, ad ogni iterazione va a cercare il massimo della sequenza; una volta trovato, pone questo massimo nella posizione n della sequenza e considera la sequenza che va da 0 a n-1.

A questo punto però ri-esegue la ricerca del massimo su questa sotto-sequenza e questo non è molto furbo, perchè non tiene alcuna traccia dei confronti effettuati precedentemente per il calcolo del primo findMax.

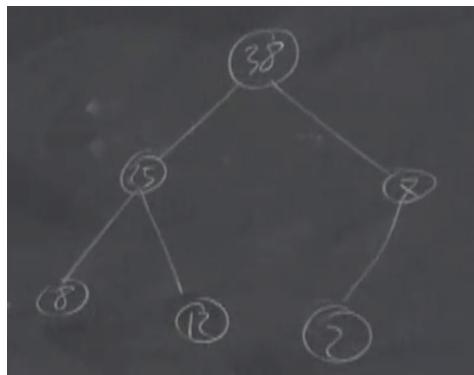
Per risolvere questo problema utilizziamo una struttura dati chiamata heap.

Heap

Un heap è un albero binario completo tale che per ogni nodo i e per ogni figlio j del nodo i vale la seguente proprietà:

$$\text{valore}(i) \geq \text{valore}(j)$$

Ad esempio:

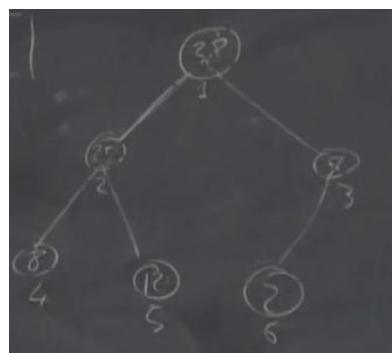


perchè $38 > 25$, $38 > 8$, $25 > 8$, $25 > 12$, $8 > 2$.

Con questa struttura dati stiamo codificando un sottoinsieme di relazione di ordine dagli elementi. Si noti che l'informazione codificata è parziale perché tra due elementi che sono entrambi figli di uno stesso nodo non abbiamo alcuna informazione, ad esempio tra 25 e 8 non esiste un arco che ci dice chi è il maggiore tra i due valori.

Quindi la posizione all'interno di un nodo contenuto in un albero heap ci dà informazione solo dei nodi che sono contenuti lungo il percorso del nodo stesso.

Un albero heap può essere codificato attraverso un'array, per far ciò si assegnano le posizioni dell'array all'albero nel seguente modo: la testa dell'heap va nella posizione 1, i due figli della testa avranno rispettivamente le posizioni 2 (sinistra) e 3 (destra), allo stesso modo i figli della posizione 2 avranno il più basso numero disponibile (4 e 5) e così via, graficamente:



L'array quindi sarà così:

1	2	3	4	5	6
38	25	8	8	12	2

Ne consegue che data la posizione i , i due figli del nodo i si trovano rispettivamente nella posizione $2i$ (figlio sinistro) e $2i+1$ (figlio destro).

Dato che, per definizione di heap, il padre deve essere più grande dei figli vale la seguente condizione nell'array della codifica dell'albero heap:

$$A[i] \geq A[2 \cdot i] \quad e \quad A[i] \geq A[2 \cdot i + 1]$$

Si noti un'importante proprietà: **in uno heap si può accedere all'elemento massimo in tempo costante** perché è nella radice dell'albero binario.

Codice

Per effettuare il sort utilizzeremo 3 funzioni:

Heapify(A,i): ripristina la proprietà di Heap al sottoalbero radicato in i assumendo che i suoi sottoalberi destro e sinistro siano già degli Heap

CostruisciHeap(A): produce uno Heap a partire dall'array A non ordinato

HeapSort(A): ordina l'array A (in place).

```
HeapSort(A,n)
| CostruisciHeap(A,n)
| for (i = n down to 2) do
|   | swap(A,1,heapsiz)
|   | heapsiz = heapsiz - 1
|   | heapify(A,1)
```

```
Heapify(A,i)
| max = i
| l = 2*i
| r = 2*i+1
| if l ≤ heapsiz AND A[l]>A[max] then
|   | max = l
| if r ≤ heapsiz AND A[r]>A[max] then
|   | max = r
| if max != i then
|   | swap(A,i,max)
|   | heapify(A,max)
```

```
CostruisciHeap(A,n)
| heapsiz = n
| for i= [n/2] downto 1 do
|   | heapify(A,i)
```

CostruisciHeap

HeapSort si basa sul concetto che i due sottoalberi che si creano sono alberi heap. Ovviamente un generico array A è estremamente improbabile che sia un albero heap a priori, ragion per cui la prima cosa che facciamo è trasformare l'array A in un array A' il quale è un albero heap.

Per risolvere ciò utilizziamo la funzione CostruisciHeap() la quale si basa di questo concetto:

Un albero binario di un singolo nodo è, per definizione, un albero heap.

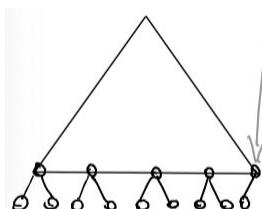
Per questa ragione possiamo considerare le foglie dell'albero derivante dall'array A come degli alberi heap.

Quindi per trasformare l'array A in un albero heap possiamo fare questo ragionamento:

- 1) Prendiamo un sottoalbero binario formato dalle foglie + il padre di queste, al più, due foglie.
- 2) Dato che le foglie sono heap, possiamo richiamare la funzione per rendere il sottoalbero preso in un heap.
- 3) Applichiamo questo per tutti i padri di tutte le foglie.
- 4) Allo stesso modo possiamo rendere heap i sottoalberi composti dai due sottoalberi heap che sono stati appena creati ed il padre di questi due sottoalberi.
- 5) Iterando in questo modo, arriveremo in cui il padre sarà la radice dell'albero originale ed in questo modo avremo trasformato A in albero heap.

In altre parole, se partiamo dai padri delle foglie e trasformiamo in heap la struttura composta dal padre stesso più le sue foglie e poi andiamo fino alla radice, aggiungendo i padri alle strutture già create mantenendo l'ordine heap fino a quando arriveremo fino alla radice, avremo un albero heap.

Per far ciò dobbiamo partire dal penultimo livello dell'albero, o più precisamente, dobbiamo partire dal nodo interno all'estrema destra **faseista**:



Questo nodo si trova precisamente nella posizione $n/2$ arrotondata per difetto, perché corrisponde all'ultima posizione possibile affinché il nodo possa avere dei figli.

Una dimostrazione banale di ciò, può essere fatta empiricamente considerando il nodo successivo, $(n/2)+1$, quest'ultimo non può non essere una foglia perché le posizioni dei suoi eventuali figli sia destro che sinistro andrebbero oltre la dimensione n dell'array, infatti:

- 1) il figlio sinistro di un nodo i si trova in nella posizione 2^*i
 - a) $2 * ((n/2)+1) = n+2$ che è maggiore di n
- 2) il figlio destro di un nodo i si trova in nella posizione 2^*i+1
 - a) $2 * ((n/2)+1)+1 = n+3$ che è maggiore di n

Quindi la posizione $(n/2)+1$ non può non essere una foglia ed l'ultimo nodo che può esserlo è $n/2$ arrotondato per difetto.

Per questa ragione l'algoritmo che trasforma l'array A in A' è un semplice for che va da $(\text{int})n/2$ a 1 che chiama heapify

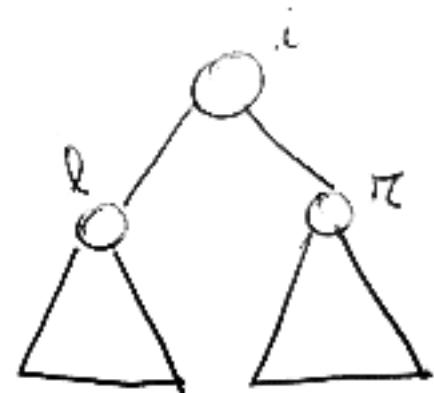
```
CostruisciHeap (A, n)
|   heapsize = n
|   for i= [n/2] downto 1 do
|       |   heapify(A, i)
```

Heapify

Heapify confronta la radice presa in input (i) con la radice del sottoalbero heap sinistro (l) e con quella del sottoalbero heap destro (r) e di collocare il massimo tra questi tre elementi al posto della radice, effettuando uno scambio se necessario (se lo scambio non viene eseguito, significa che è già un albero heap).

Se invece, avviene lo scambio nasce un problema: effettuando uno scambio possono modificare le relazioni di ordine all'interno di uno dei due sottoalberi (perché se effettuiamo uno scambio, ad esempio, tra i ed l significa che $A[i] < A[l]$, ma non sappiamo se il valore di i è nella giusta posizione come radice del sottoalbero heap)

Per risolvere questo semplicemente richiamo in modo ricorsivo heapify, il quale verrà ripetuto fino al raggiungimento delle foglie.



```
Heapify(A, i)
|   max = i
|   l = 2*i
|   r = 2*i+1
|   if l ≤ heapsize AND A[l]>A[max] then
|       |   max = l
|   if r ≤ heapsize AND A[r]>A[max] then
|       |   max = r
|   if max != i then
|       |   swap(A, i, max)
|       |   heapify(A, max)
```

HeapSort

Come già detto precedentemente HeapSort nasce andando a sviluppare il problema che ha SelectionSort nella ricerca del massimo.

La prima cosa che esegue è la costruzione dell'albero heap dato l'array A, attraverso l'algoritmo CostruisciHeap.

Una volta costruito l'albero heap, sappiamo che il valore maggiore si trova in radice (quindi nella posizione 1 dell'array) e che dovrà essere messo nell'ultima posizione dell'array proprio perché stiamo ordinando in senso crescente.

Perciò effettueremo lo scambio tra $A[1]$ ed $A[\text{heapsize}]$, (heapsize viene settato in CostruisciHeap ad n).

Una volta che sappiamo che l'ultimo elemento è nella parte ordinata, possiamo ridurre heapsize in modo da separare appunto la parte ordinata dell'array a quella disordinata. Dato che abbiamo effettuato uno scambio, è necessario richiamarsi $\text{heapify}()$ per "aggiustare" l'albero heap, in modo che il prossimo massimo si ritrovi di nuovo nella radice. Questo ragionamento viene iterato da n fino a 2 perché..?

```
HeapSort (A, n)
|  CostruisciHeap (A, n)
|  for (i = n down to 2) do
|    |    swap (A, 1, heapsize)
|    |    heapsize = heapsize - 1
|    |    heapify (A, 1)
```

Analisi Asintotica

Heapify

Heapify viene chiamato sulla radice dell'albero (il quale ha n elementi) e dato che heapify impiega tempo proporzionale all'altezza dell'albero su cui opera possiamo concludere che nel caso peggiore, ossia quando bisogna eseguire heapify fino al livello delle foglie bisognerà eseguire tante volte quanto è l'altezza dell'albero di n elementi, che sappiamo è pari ad $\log_2 n$, quindi possiamo concludere che è limitato superiormente da $O(n \cdot \log_2 n)$, ossia "peggiore di $\log_2 n$ non può essere".

CostruisciHeap

Semplicemente guardando l'algoritmo, possiamo notare che c'è un for che viene seguito $n/2$ volte che richiama heapify()

- 1) Il ciclo contribuisce per $n/2$ volte
- 2) Heapify viene eseguito $\log_2 n$ volte, al massimo

Quindi il contributo di CostruisciHeap è anch'esso limitato superiormente da $O(n \cdot \log_2 n)$

HeapSort

Abbiamo visto che CostruisciHeap() è $O(n \cdot \log_2 n)$, mentre per il ciclo for possiamo subito notare che :

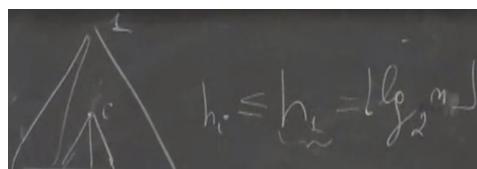
- 1) Ciclo for viene eseguito n volte
- 2) L'operazione di swap è costante
- 3) L'operazione di decremento di heapsize è costante
- 4) Heapify viene eseguito $\log_2 n$ volte, al massimo

Il contributo del ciclo for è limitato superiormente da $O(n \cdot \log_2 n)$ e quindi ci troviamo in questa situazione:

$$T_{CH}(n) = O(n \cdot \log_2 n)$$
$$T_{HS}(n) = O(n \cdot \log_2 n)$$

Cerchiamo adesso di sviluppare questi per trovare un limite asintotico stretto, partendo dal **CostruisciHeap**:

- 1) Ognuno degli alberi su cui è applicato heapify(A, i) ha un'altezza inferiore rispetto all'albero radicato in 1, che a sua volta è pari a base di $\log_2 n$ (quindi ciascuno albero radicato su un nodo interno non può avere altezza maggiore di $\log_2 n$)



- 2) Sappiamo anche il tempo di esecuzione di un heapify su un albero di altezza h_i è un $\Theta(h_i)$ che è pari a sua volta ad $O(\log_2 n)$.

$$T_{hy}(h_i) = \Theta(h_i) = O(\log_2 n)$$

- 3) Quindi heapify ha un costo (alla peggio) di $O(\log_2 n)$ e viene eseguita (base di $n/2$) volte otteniamo il limite superiore pari a $O(n \cdot \log_2 n)$
- 4) Ma il nostro scopo è quello di trovare un limite asintotico stretto e per fare ciò possiamo analizzare le chiamate ad heapify le quali hanno tutte altezza diverse
- 5) Infatti, la maggioranza degli alberi su cui chiamiamo heapify ha altezza molto bassa perché l'albero di partenza è un albero completo.
Questo lo possiamo enunciare pensando al fatto che chiamiamo heapify a partire dai padri delle foglie che, per loro definizione, hanno altezza 1.
Questi padri sono pari ad $n/4$, ossia la metà del ciclo for esegue heapify che ha un costo pari a $\Theta(1)$
- 6) Andiamo adesso a notare i padri dei padri delle foglie, "i nonni"; Essi hanno altezza 2 e ce ne sono $(n/4)/2$ (la metà dei padri); ossia ci sono $n/8$ nonni, che quindi che ci costano $\Theta(2)$.

- 7) In generale, quindi, al livello i -esimo avremo $\frac{n}{2^{i+1}}$ sottoalberi, e per ciascuno di essi il suo costo sarà pari a $\Theta(i)$
- 8) Quindi il costo di CostruisciHeap sarà la somma al variare dell'altezza dell'albero da 1 all'altezza massima (h) dei contributi del livello i -esimo

$$T_{CH}(n) = \sum_{i=1}^h \left(\frac{n}{2^{i+1}} \cdot \Theta(i) \right)$$

- 9) h è l'altezza dell'albero che contiene n nodi che sappiamo essere pari alla base di $\log_2 n$

$$T_{CH}(n) = \sum_{i=1}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{i+1}} \cdot \Theta(i) \right) = \sum_{k=1}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{k+1}} \cdot \Theta(k) \right) =$$

- 10) L'operazione Θ possiamo estrarlo fuori.

$$\begin{aligned} T_{CH}(n) &= \sum_{i=1}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{i+1}} \cdot \Theta(i) \right) = \sum_{k=1}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{k+1}} \cdot \Theta(k) \right) \\ &= \Theta \left(\sum_{i=1}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{i+1}} \cdot i \right) \end{aligned}$$

- 11) L'incognita n che si trova come argomento della sommatoria non dipende dall'indice della sommatoria, quindi è un fattore moltiplicativo che è sempre lo stesso per tutti i termini che può essere raccolto a fattor comune; in particolare raccogliamo $n/2$ perché anche il 2 può essere raccolto perché $2^{(i+1)}$ è uguale a $2 \cdot 2^i$; quindi avremo:

$$\begin{aligned} T_{\text{ch}}^{(n)} &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\frac{n}{2^i} \cdot \mathcal{O}(i) \right) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\frac{n}{2^i} \cdot \mathcal{O}(i) \right) \\ &= \mathcal{O}\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \frac{n}{2^i} \cdot i\right) = \mathcal{O}\left(\frac{n}{2} \left| \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(i \cdot \left(\frac{1}{2}\right)^i\right) \right| \right) \end{aligned}$$

- 12) Ragioniamo adesso sulla parte evidenziata nel rettangolo: quella è una sommatoria è una somma parziale di termini provenienti della serie $i \cdot x^i$ dove $x=1/2$
NB, ricordati $x=1/2$, è super importante!

- 13) La formula chiusa di questa sommatoria può essere dedotta da ciò pensando al fatto che uno degli x moltiplicati da i viene portato fuori; avremo quindi:

$$i \cdot x^i = x \cdot i \cdot x^{i-1}$$

- 14) Prendiamo adesso la parte di "destra" della moltiplicazione, $i \cdot x^{i-1}$; Questo non è nient'altro che la derivata rispetto ad x di x^i

$$i \cdot x^{i-1} = \frac{d}{dx}(x^i)$$

Infatti se derivo x^i troverò proprio $i \cdot x^{i-1}$

- 15) Da tutto questo pippone, posso enunciare che la sommatoria da 0 a k di $i \cdot x^i$ è uguale alla sommatoria, da 0 fino a k , di $x \cdot i \cdot x^{i-1}$; quest'ultima abbiamo visto nel passo 14 è uguale a $d/dx(x^i)$, quindi sostituisco ed avrò:

$$\sum_{i=0}^k i \cdot x^i = \sum_{i=0}^k \left(x \cdot \frac{d}{dx}(x^i) \right)$$

Nb. da 0 a k è una convenzione a caso, non è importante.

- 16) La prima cosa che possiamo osservare è che l' x indicato da zio Benny qui:

$$\sum_{i=0}^k i \cdot x^i = \sum_{i=0}^k \left(x \cdot \frac{d}{dx}(x^i) \right) = x \cdot \sum_{i=0}^k \frac{d}{dx}(x^i)$$

derivate of x^i



Derivative:

$$\frac{d}{dx}(x^i) = i x^{i-1}$$



Questo x non dipende dall'indice della sommatoria e quindi può essere raccolto a fattor comune, quindi:

- 17) La derivata è un operatore lineare, quindi distribuisce con le somme; ossia se io sommo tante derivate il risultato è pari alla derivata della somma e viceversa.
Quindi possiamo distribuire la nostra sommatoria nel seguente modo:

$$\sum_{i=0}^{\infty} i \cdot x^i = \sum_{i=0}^{\infty} \left(x \cdot \frac{d}{dx} (x^i) \right) = x \cdot \sum_{i=0}^{\infty} \frac{d x^i}{dx} = x \cdot \frac{d}{dx} \left(\sum_{i=0}^{\infty} x^i \right)$$

- 18) La cosa checazza è che ora siamo di fronte ad una sommatoria di cui conosciamo la forma chiusa e potremmo effettuare la derivata di questa forma chiusa.
- 19) Dato che k era un valore a caso, possiamo benissimo porlo uguale ad infinito e le uguaglianze tra sommatorie varrebbero lo stesso

$$\sum_{i=0}^{\infty} i \cdot x^i = \sum_{i=0}^{\infty} \left(x \cdot \frac{d}{dx} (x^i) \right) = x \cdot \sum_{i=0}^{\infty} \frac{d x^i}{dx} = x \cdot \frac{d}{dx} \left(\sum_{i=0}^{\infty} x^i \right)$$

- 20) La forma chiusa della sommatoria da i all'infinito di x^i (quella che sta nella derivata) quando x è compreso tra 0 e 1 è uguale ad $1/(1-x)$

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{1}{1-x}$$

$0 < x < 1$

Nb. se $x \geq 1$ la sommatoria è infinita

- 21) Quindi adesso possiamo sostituire il risultato che abbiamo appena calcolato nella derivata del punto 19, da ciò otteniamo

$$x \cdot \frac{d}{dx} \left(\sum_{i=0}^{\infty} x^i \right) = x \cdot \frac{d}{dx} \left(\frac{1}{1-x} \right)$$

- 22) La derivata è "semplice" da calcolare perché

$$\frac{1}{1-x} = (1-x)^{-1}$$

la derivata di $(1-x)^{-1}$ rispetto ad x è uguale all'esponente ($\Rightarrow -1$) moltiplicato $(1-x)$ elevato al vecchio esponente meno uno ($\Rightarrow (1-x)^{-2}$) per la derivata dell'argomento che stiamo derivando, ossia $(1-x)$, la cui derivata è -1 :

$$\therefore (-1) \cdot (1-x)^{-2} \cdot (-1) =$$

$-1 * -1 = 1 \rightarrow$ avremo $(1-x)^{-2}$, quindi:

$$\frac{d}{dx} \left(\frac{1}{1-x} \right) = \frac{d}{dx} \left((1-x)^{-1} \right) = (-1) \cdot (1-x)^{-2} \cdot (-1) = \frac{1}{(1-x)^2}$$

- 23) Quindi abbiamo calcolato la derivata che si trova nel punto 21 e possiamo sostituirla a quella equazione:

$$x \cdot \frac{d}{dx} \left(\sum_{i=0}^{\infty} x^i \right) = x \cdot \boxed{\frac{d}{dx} \left(\frac{1}{1-x} \right)} = \frac{x}{(1-x)^2}$$

Quando x è compreso tra 0 e 1

- 24) Dato che la nostra x è $\frac{1}{2}$ (vedi punto 12) possiamo sostituire e avremo che:

Input:

$$\frac{x}{(1-x)^2} \text{ where } x = \frac{1}{2}$$

Result:

2

Questo significa che l'ultima sommatoria del punto 11 (quella messa nel rettangolo) è limitata superiormente da 2 perché parte da 1 e si finisce a $\log_2 n$ ed è sicuramente più piccola della stessa sommatoria che inizia da 0 e finisce ad infinito che vale 2.

25) Quindi posso concludere dicendo che :

$$\begin{aligned}
 T_{CH}(n) &= \sum_{i=1}^h \left(\frac{n}{2^{i+1}} \cdot \Theta(i) \right) = \sum_{i=1}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{i+1}} \cdot \Theta(i) \right) = \Theta\left(\sum_{i=1}^{\lfloor \log_2 n \rfloor} \left(\frac{n}{2^{i+1}} \cdot i \right)\right) \\
 &= \Theta\left(\sum_{i=1}^{\lfloor \log_2 n \rfloor} \left(i - \left(\frac{i}{2}\right)^i \right)\right) = \Theta\left(\frac{n}{2} * \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2}\right) = \Theta(n)
 \end{aligned}$$

26) Quindi possiamo aggiornare la tabella iniziale in questo modo:

$$\begin{aligned}
 T_{CH}(n) &= \Theta(n) \\
 T_{HS}(n) &= O(n \log n)
 \end{aligned}$$

27) Abbiamo quindi valutato il tempo di CostuiscoHeap e siamo passati da un $O(n \log n)$ ad un $\Theta(n)$, rendendo molto più preciso l'analisi computazionale;

28) HeapSort è composto da CostruiscoHeap + un ciclo for che effettua $n-1$ chiamate su heapify passandoci come parametri ($A, 1$). Invece CostruiscoHeap chiama heapify con i parametri (A, i).

Per come è strutturato l'algoritmo CostruiscoHeap la sua variabile i varia dal basso verso l'alto e il numero di radici che trova si dimezza sempre di più andando verso l'alto. HeapSort fa esattamente il contrario e chiama heapify sempre e solo della radice.

Ciò significa che HeapSort effettuerà $n/2$ chiamate (che sarebbero le foglie) a heapify su un albero di altezza $h-1$. A questo valore bisogna aggiungere il costo di tutte le altre chiamate che HeapSort fa ad heapify che possiamo generalizzare con un qualsiasi valore k

$$\frac{n}{2}(h-1) + K$$

Ma dato che $h = \log_2 n$, sostituiamo:

$$\frac{n}{2}(\log_2 n - 1) + K$$

Ma già solo considerando il primo termine sappiamo che vale

$$\frac{n}{2}(\log_2 n - 1) + K = \Theta(n \log_2 n)$$

Quindi possiamo concludere che il tempo che HeapSort richiede è $\Theta(n \log_2 n)$.

Conclusioni finali

HeapSort dimostra che:

- Scegliere una buona rappresentazione per i dati spesso facilita la progettazione di buoni algoritmi
- E' importante pensare a quale può essere una buona rappresentazione dei dati prima di implementare una soluzione.
- Algoritmo di ordinamento sul posto per confronto che impiega tempo $O(n \log n)$.
- Sfrutta le proprietà della struttura dati astratta Heap.

Quick Sort

AUTORE: Denny Acciaro
[\[acciarogennaro@gmail.com\]](mailto:acciarogennaro@gmail.com)

Fonti

- <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/5-quicksort.pdf>
 - <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/6-quicksort-analisi.pdf>
 - <https://www.geeksforgeeks.org/quick-sort/>
 - <https://youtu.be/URoFMjosM80?t=2831> (Lezione 12)
 - <https://www.youtube.com/watch?v=lRnc0L16n04> (Lezione 13)
 - <https://www.youtube.com/watch?v=rqYz-dskYxc> (Lezione 14)
 - https://www.youtube.com/watch?v=kQU-WzlB3_0 (Lezione 15)
 - <https://www.youtube.com/watch?v=dV-JgquCwHo>
-

Introduzione	2
Codice e descrizione algoritmi	3
QuickSort()	3
Garanzia di terminazione dell'algoritmo	4
CASO 1 e 3 ($p \leq q \leq r$) oppure ($p < q \leq r$)	4
CASO 2 e 4 ($p \leq q < r$) oppure ($p < q < r$)	4
Requisiti di Partiziona	5
Partiziona()	5
$j < p$	7
$j = r$	8
Analisi	9
Tempo computazionale di partiziona()	10
Tempo computazionale di quicksort()	13
CASO PEGGIORE [$q = 1$ oppure $q=n-1$]	13
CASO MIGLIORE [$q = n/2$]	15
Conclusione dei due casi	16
Dimostrazione che il numero totale di nodi = $2n-1$	16
CASO MEDIO	18
Ipotesi	21
Tesi	21
Semplificazione	21
Il caso base	24
Passo induttivo	25
Conclusioni Finali	30

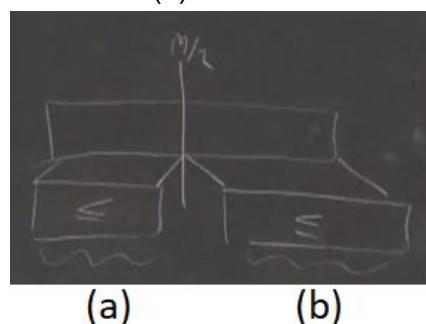
Introduzione

Il quicksort è un algoritmo di ordinamento basato sul concetto di Divide et Impera; La sua implementazione viene effettuata con due algoritmi, **quicksort** e **partiziona**.

Come HeapSort nasceva da un problema di SelectionSort, anche QuickSort risolve un problema di MergeSort in particolare: *MergeSort prende una sequenza, la divide esattamente nella sua metà e ordina le due sottosequenze in modo indipendente l'una dall'altra e poi fonde queste due sottosequenze ordinate per avere la sequenza presa all'inizio ordinata.*

Il problema è che non abbiamo alcuna assunzione sulla relazione di ordine tra gli elementi che si trovano nelle due sottosequenze divise precedentemente a metà.

In altre parole, la sottosequenza (a) può contenere elementi che sono sia minori che maggiori degli elementi che si trovano in (b).



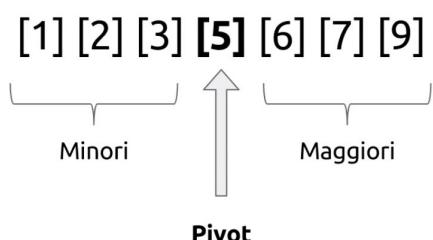
Questo rende la fusione tra le due sequenze complessa.

Potremmo ridurre la complessità della fusione se noi sapessimo a priori, che tutti gli elementi della prima sottosequenza sono minori degli elementi della seconda sottosequenza; la fusione sarebbe semplicemente la concatenazione di queste due sottosequenze.

Questo concetto è espresso della seguente proprietà:

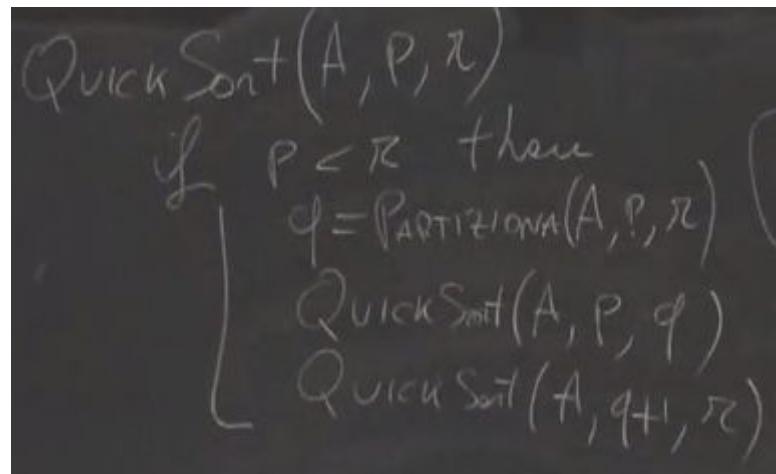
$$\forall 1 \leq i \leq q; \forall q + 1 \leq j \quad A[i] \leq A[j]$$

ossia,



Codice e descrizione algoritmi

QuickSort()

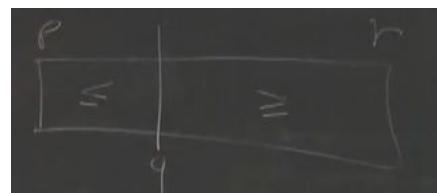


I parametri p ed r individuano la sottosequenza di cui quella chiamata si occupa.

Esattamente come MergeSort, **la prima cosa da fare è controllare se ci troviamo in un caso base**, ossia quando la sottosequenza ha lunghezza strettamente minore di 2.

Se non siamo in un caso base (ossia quando ho almeno due elementi nella sottosequenza), **dobbiamo ottenere indice di suddivisione (q) il quale verrà generato della funzione partiziona()** che a sua volta cercherà di mantenere la proprietà introdotta prima. **Una volta conclusa l'esecuzione di partiziona, avremo l'indice q che divide la sequenza** (che va da p ad r) **in due sottosequenze** (da p a q e da q ad r).

La prima sottosequenza contiene quindi tutti gli elementi che sono minori o uguali di tutti gli elementi della sottosequenza a destra; questo avviene perché partiziona() ha garantito la proprietà.



Quindi quello che ci rimane da fare è eseguire ricorsivamente quicksort su le due sottosequenze.

Garanzia di terminazione dell'algoritmo

Essendo un algoritmo ricorsivo è necessario garantire che prima o poi termini.

Per questa ragione dobbiamo assicuraci che il valore di q , calcolato da partiziona, sia in grado di garantire sempre il raggiungimento del caso base.

Quindi:

Sappiamo che q compreso tra p ed r , ma esistono 4 casi in cui ciò avviene:

$$\begin{aligned} p \leq q \leq r \\ p \leq q < r \\ p < q \leq r \\ p < q < r \end{aligned}$$

Ma tutte e 4 i casi vanno bene per la terminazione di QuickSort?

Quindi ci dobbiamo chiedere, per ogni caso, se l'unico vincolo su q è il caso che stiamo analizzando, sono sicuri che qualsiasi valore di q che rispetta il vincolo garantisce la terminazione di quicksort?

Procediamo empiricamente:

CASO 1 e 3 ($p \leq q \leq r$) oppure ($p < q \leq r$)

La condizione $q = r$ è un problema per la terminazione perché la prima chiamata effettua una chiamata su una sequenza che va da p ad r (se $q=r$), ma essa stessa viene chiamata su una sequenza che va da p ad r .

In questo modo, la prima chiamata continuerà ad eseguire chiamate su sequenze della stessa lunghezza, non arrivando mai al caso base.

NB. Non è un problema per la seconda chiamata (quella che va eseguita su $q+1, r$) perché se $q=r$, avremo che passeremo alla prossima esecuzione di quicksort i parametri: $r+1, r$.

Ma dato che $r+1$ è maggiore di r , significa che è una sequenza vuota ed è un caso base per QuickSort, quindi non fa nulla (perché non entra nel corpo dell'if).

Quindi i casi 1 e 3 non possono andar bene perché possono ipotizzare che q sia uguale ad r .

CASO 2 e 4 ($p \leq q < r$) oppure ($p < q < r$)

La condizione $q = p$ non è un problema perché la prima chiamata ricorsiva andrebbe da p a p , ossia chiamerebbe QuickSort su una sottosequenza di lunghezza 1 la quale è un caso base e quindi non verrebbe presa in considerazione dal corpo dell'if.

La seconda chiamata invece, sarebbe eseguita su una sottosequenza che va da $p+1$ ad r la quale è strettamente minore rispetto a quella della quale è stata eseguita (che andava da p ad r).

Quindi la condizione $q = p$ non è un problema e dato che il caso 2 contiene più valori del caso 4, **possiamo considerare $p \leq q < r$ come proprietà da rispettare.**

Requisiti di Partiziona

Da questo discorso concludiamo che i seguenti requisiti devono essere rispettate al termine dell'algoritmo partiziona()

$$\forall 1 \leq i \leq q \quad \forall q+1 \leq j \leq m \quad A[i] \leq A[j] \quad (1)$$

$$p \leq q < r \quad , \quad (2)$$

Il primo requisito serve per garantire il fatto di non dover eseguire il merge alla fine dell'algoritmo, mentre il secondo serve per garantire che l'algoritmo termini correttamente.

Partiziona()

Dato che partiziona() viene chiamato se e solo se p è minore di r , essa può assumere che la lunghezza della sottosequenza sia almeno 2.

Partendo dal requisito 1 sviluppiamo **l'idea per partizionare la sequenza** che inizia in p e finisce in r in due sottosequenze divise da un valore q **si basa sulla selezione di un elemento x .**

La selezione di questo elemento non è particolarmente influente per l'algoritmo quindi per semplicità prendiamo il primo elemento dell'array, ossia quello in posizione p .

Una volta selezionato x , dobbiamo impostare alla sua sinistra tutti e soli gli elementi che sono minori o uguali di x e contemporaneamente alla sua destra tutti e soli gli elementi che sono maggiori o uguali di x .

In matematiche:

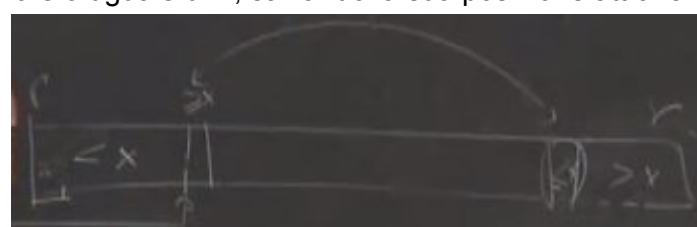
$$\forall p \leq i \leq q \quad A[i] \leq x \quad (A)$$

$$\forall q+1 \leq j \leq r \quad x \leq A[j] \quad (B)$$

Questo concetto può essere implementato così:

Parto da p ed inizio a scorrere l'array finché non trovo un elemento che è maggiore o uguale di x (perchè non rispetta la prima la condiziona (A)) e mi salvo la sua posizione attraverso un indice.

Faccio la stessa cosa, ma partendo da r e scorrendo al contrario l'array cercano un elemento che è minore o uguale di x , salvando la sua posizione attraverso un altro indice.



Adesso posso scambiare gli elementi puntati dai due indici se sono diversi fra loro.

Dopo lo scambio, continuo con lo stesso ragionamento sulla parte che era compresa tra i due indici (che non è stata ancora confrontata).

Ad un certo punto questi due indici si dovranno incrociare sarà il valore q che stiamo cercando.

Partiziona(A, p, r)

$x = A[p]$

$i = p - 1$

$j = r + 1$

repeat

repeat

$j = j - 1$

until ($A[j] \leq x$)

repeat

$i = i + 1$

until ($A[i] \geq x$)

if ($i < j$) then

swap(A, i, j)

until ($i \geq j$)

return j

Cerca il primo elemento da destra che sia minore o uguale al Pivot x

Cerca il primo elemento da sinistra che sia maggiore o uguale al Pivot x

Se l'array non è stato scandito completamente $i < j$ (i due non incrociano) allora gli elementi vengono scambiati

Se l'array è stato scandito completamente $i \geq j$ (i due indici si incrociano) allora termina il ciclo

Dato che abbiamo sviluppato partiziona a partire dal primo requisito sappiamo che esso è soddisfatto ma non abbiamo alcun indizio per capire se anche il secondo requisito lo sia.

Dimostrare che $p \leq q < r$ al termine dell'esecuzione è un po' più delicato a causa dei due repeat until interni perché non fanno alcun controllo sugli indici i e j , rispettivamente, ma terminano solo in base al confronto tra $A[j] / A[i]$ e x .

Al termine di partiziona $q = r$, quindi dobbiamo dimostrare che (al termine dell'esecuzione) non possano valere queste

- $j \geq r$
- $j < p$

perché se una di queste fosse vera avremo una violazione sul secondo requisito.

Il caso $j \geq r$ può essere semplificato in $j = r$ perché la variabile j è impostata inizialmente ad $r+1$ e viene sicuramente decrementata almeno una volta perché per come abbiamo strutturato l'algoritmo (ossia con l'uso dei repeat until) ciò avviene indipendentemente della condizione $A[j] \leq x$.

Quindi dato che parte da $r+1$, viene decrementato almeno una volta e non viene mai decrementato possiamo semplificare $j \geq r$ in $j = r$.

$j < p$

Il caso $j < p$ non si può verificare perché

- Se non avvengono scambi:
Inizialmente x è inizializzato nella posizione p ed esso fa da “barriera” per il repeat di j perché anche se j non trovasse nulla di più piccolo o uguale ad x per tutto l’array letto da r verso p , si fermerebbe nella posizione $p + 1$ perché incontra proprio x . Questa situazione avviene quando non ci sono scambi da effettuare e rappresenta una situazione dove non sussiste il problema di $j < p$.
- Se avvengono potenzialmente degli scambi:
Supponiamo però ci troviamo in una generica iterazione, non possiamo più assumere che x si trovi in posizione p perché potrebbero **esserci stati degli scambi**.
Quindi ci troviamo nella situazione in cui non sappiamo come garantire la terminazione del repeat di j se x si trova dopo j .



Quello che ci interessa di garantire è che nella porzione in blu che va da p a q ci sia almeno un valore che sia minore o uguale ad x .

- Se nella posizione p c’è x
- Se nella posizione p non c’è x :
Questo avviene però se e solo se c’è stato uno scambio tra posizione p ed un’altra posizione. Ma uno scambio metterebbe in posizione p un valore che è minore o uguale ad x (perché j si era fermato in quella posizione).

Quindi nella posizione p in ogni momento o è presente x oppure un valore che è minore o uguale ad x .

Quindi la posizione p fa da barriera per j , ossia è il valore minimo che j potrà avere.
Quindi il caso $j < p$ non si può verificare.

$j = r$

Il caso $j=r$ significa che i e j si trovano contemporaneamente nella posizione r e questo non deve accadere perché altrimenti il valore ritornato da partiziona sarebbe proprio r e questo causerebbe danni per la terminazione dell'algoritmo.

Dato che j parte da $r+1$ per fermarsi in r significa che deve aver eseguire un solo decremento ossia che stiamo facendo la prima iterazione del repeat esterno, perché ad ogni iterazione del repeat esterno sicuramente verrà decrementato j .

Una volta che j si è fermato in r , essendo la prima iterazione del repeat non sono stati ancora eseguiti scambi, quindi x si trova ancora nella prima posizione p e i si trova in posizione $p-1$.

Il repeat di i parte da $p-1$ ed esegue un incremento ma trova subito, in posizione p , proprio l'elemento x e si ferma.

Quindi ci troviamo nella situazione in cui $i=p$ e $j=r$ e stiamo alla fine del repeat di i .

Adesso viene eseguita la condizione $i < j$, la quale è necessariamente vera perché stiamo considerando l'assunzione che $p < r$, perché noi chiamiamo partiziona() se e solo se p è minore di r .

Quindi avviene lo scambio e dato che i è minore di j almeno un'altra iterazione del repeat esterno viene fatta.

Quindi j verrà decrementato almeno un'altra volta.

NB. Questo ragionamento vale solo se j si ferma in r , ma per fare ciò deve essere necessariamente la prima iterazione del repeat esterno perché j non viene mai incrementato. Quindi per tutte le altre iterazioni non ha senso continuare l'analisi.

[RIASSUNTO]

- 1) Se j si ferma in r , stiamo alla prima iterazione del repeat esterno
- 2) Se stiamo alla prima iterazione, x si trova in p
- 3) Se x si trova in p , i si ferma in p
- 4) $i < j$ per l'assunzione $p < r$
- 5) Avviene lo scambio, quindi c'è un'altra iterazione del repeat esterno
- 6) j viene decrementato almeno un'altra volta.

Analisi

L'analisi di quicksort si effettua in funzione del numero di elementi n della sequenza che deve ordinare, che sono proprio:

$$n = r - p + 1$$

Osservando l'algoritmo di quicksort, possiamo descrivere l'equazione di ricorrenza, notando che, ad ogni iterazione:

- Viene eseguito il corpo dell'if
- Viene eseguito solo la testa dell'if e la condizione è falsa

Se la condizione è falsa, dato che questo confronto impiega tempo costante perché è indipendente dalla lunghezza n avrà un costo computazionale di 1.

Se la condizione è vera dobbiamo sommare:

- 1) $\Theta(1)$ per l'esecuzione del confronto
- 2) Tempo_partiziona(n), che vedremo dopo
- 3) Tempo_quick_sort(parte_sinistra)
- 4) Tempo_quick_sort(parte_destra)

$$\begin{cases} 1 & \text{se } n \leq 1 \\ \Theta(1) + T_p(n) + T_{QS}(?) + T_{QS}(?) & \text{se } n > 1 \end{cases}$$

Le lunghezze di parte_sinistra e parte_destra dipendono dal valore di q .

Ma sicuramente sappiamo che:

$$\begin{aligned} \text{parte_sinistra} &= q - p + 1 \\ \text{parte_destra} &= r - (q+1) + 1 \end{aligned}$$

Dato che vale: $p \leq q < r$, possiamo dire che:

- 1) **Il valore minore della parte sinistra è 1** perché q può essere al minimo uguale a p e sostituendo avremo $p-p+1=1$.
- 2) **Il valore massimo della parte sinistra è $r - p$** perché q può essere al massimo $r-1$ e sostituendo avremo $(r - 1) + p + 1 = r - p$.
- 3) Ma il valore $r-p$ è minore strettamente di n (che vale $r-p+1$)
- 4) Quindi potremmo dire che la dimensione della chiamata di sinistra sarà:

$$1 \leq q - p + 1 \leq n-1$$

[ndr. se è strettamente minore di n può essere al massimo $n-1$]

- 5) Per quanto riguarda la porzione di destra, possiamo dire che anche la dimensione della chiamata di destra sarà compresa tra 1 e $n-1$

$$1 \leq r - (q + 1) + 1 \leq n-1$$

Questo è vero perché la parte sinistra e destra sono complementari fra loro (ossia la loro somma è pari ad n), quindi quando la parte sinistra varrà 1, la parte destra deve valere n , dualmente, se la parte di sinistra varrà $n-1$ la parte destra deve valere $n-1$.

Quindi entrambi gli argomenti delle chiamate ricorsive nell'equazione di ricorrenza sono compresi fra 1 ed $n-1$ e sappiamo che vale l'equazione di complementarietà:

$$\text{parte_destra} = n - \text{parte_sinistra}$$

Le variabili `parte_sinistra` e `parte_destra` sono state chiamate dal prof q ed n-q, da non confondersi con il valore q dato da `partiziona()`.

NB. dato che l'equazione è ricorsiva questi parametri non sono unici ma ne esistono tanti quante sono le chiamate ricorsive che vengono eseguite.

$$\begin{cases} 1 & \text{se } n \leq 1 \\ \Theta(1) + T_p(n) + T_{QS}(q) + T_{QS}(n-q) & \text{se } n > 1 \end{cases}$$

Dato che fissato n, è possibile che esistano parametri q dell'equazione di ricorrenza perché questo parametro non dipende solo da n ma anche dai valori da ordinare.

Il fatto che questo parametro q possa cambiare allora induce funzioni che sono asintoticamente diverse?

Per rispondere a questa domanda dovremmo distinguere i diversi casi tra peggiore e migliore, e se questi sono diversi fra loro, dovremmo considerare anche il caso medio.

Prima di far ciò, possiamo analizzare il tempo di `partiziona` per sostituirlo nell'equazione di ricorrenza.

Tempo computazionale di `partiziona()`

Dato che le condizioni dei cicli del repeat dipendono dai valori della sequenza e non della sua lunghezza è difficile sapere a priori quanto `partiziona()` costi.

Quello che possiamo notare è che il ciclo esterno verrà al massimo a costare $n/2$ perché dato che se non faccio uno scambio il ciclo finisce e il numero degli scambi può essere al massimo $n/2$ (perché scambio il primo elemento con l'ultimo, il secondo col penultimo ecc..).

Adesso possiamo sfruttare la proprietà che ci dice che **al termine del repeat esterno i può valere solo j oppure j+1** perché:

- **Se i due indici si incrociano senza effettuare scambi**

Se non vengono effettuati scambi vuol dire che dopo il repeat di j e il repeat di i, i è maggiore di j.

Ma dato che non vengono effettuato scambi, ciò può avvenire solo alla prima iterazione del ciclo esterno (perché lo ripetiamo solo se avvengono scambi), e dato che il repeat di i viene eseguito solo una volta (sempre se non avvengono scambi) perché la sua condizione è vera, infatti `A[i]` ed `x` saranno proprio lo stesso valore.

Quindi l'unico caso che può sussistere è che i sia uguale a j ed entrambi siano uguali a p.

- **Se i due indici si incrociano avendo effettuato degli scambi**

Se è stato eseguito almeno uno scambio, gli indici non partiranno più dall'esterno ma si troveranno all'interno della sequenza, inoltre, tutti gli elementi che si trovano tra p ed i sono minori o uguali del pivot e tutti gli elementi che si trovano tra j ed r sono maggiori o uguali del pivot:



A questo punto, analizziamo l'iterazione del repeat esterno quando i e j si incrociano. Per come è scritto il codice, verrà prima eseguito il repeat di j , poi quello di i (quindi ragioneremo prima su j e poi su i)

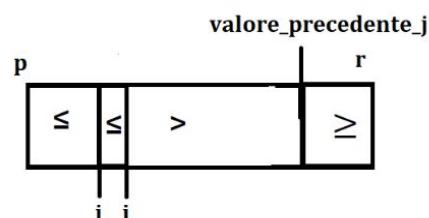
Qui nascono due casistiche:

- j si ferma ad i

In questo caso j "visita" la posizione i e ci trova un valore x che minore o uguale di X e si ferma.

Ma se non si è fermato prima di i è perché TUTTI i valori tra j e i sono maggiori strettamente (se fosse stato un valore uguale si sarebbe fermato).

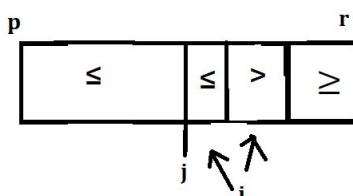
Se j arriva fino ad i (quindi $j=i$), l'unica cosa che i potrà fare è un solo incremento perché si troverà già un valore più grande. Quindi avremo che i è uguale a $j+1$.



- j si ferma prima di i , ed è i ad arrivare da j

Allo stesso modo di prima j arriverà alla prima posizione che contiene un numero che è minore o uguale.

Ma i si può fermare solo nella posizione precedente ad j (nel caso sia verificata l'uguaglianza col pivot) oppure nella posizione $j+1$, perché contiene un valore maggiore del pivot.

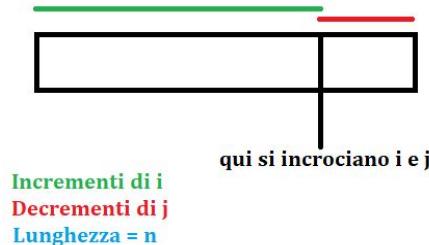


Quindi i potrà valere j oppure $j+1$.

[RIASSUNTO]

Se j raggiunge i , sicuramente i farà un incremento quindi sarà di uno più grande ma si ferma subito ($i=j+1$), mentre se i raggiunge j , i si può fermare a j se vale l'uguaglianza con il pivot ($i=j$) oppure se è strettamente minore, quello dopo dev'essere strettamente maggiore e quindi si fermerà in $j+1$ ($i=j+1$).

Tutto questo ci serve per il calcolo del tempo di partiziona perché **se i sarà uguale a j** , indipendentemente in che punto in incrociano i e j , **la somma** degli incrementi di i più la somma dei decrementi di j è **pari a $n+1$** . [nda. non è stato spiegato perché $n+1$ e non n]



Se invece, **i è pari a $j+1$** **la somma** degli incrementi di i più la somma dei decrementi di j è **pari a $n+2$** .

Ciò significa che la somma complessiva quante volte vengono eseguiti i repeat di i e di j prima di arrivare alla terminazione dell'algoritmo di partiziona() ottengo $n+1$ oppure $n+2$, ossia un valore lineare.

Dato che il numero massimo di scambi è $n/2$, il valore $n+2$ è un limite superiore di tutto l'algoritmo perché è maggiore al numero di scambi.

$$T_{\text{partiziona}} = O(n+2) = \Theta(n)$$

e possiamo sostituirlo nell'equazione di ricorrenza precedente:

$$\begin{cases} 1 & \text{se } n \leq 1 \\ \Theta(n) + T_{QS}(q) + T_{QS}(n-q) & \text{se } n > 1 \end{cases}$$

Il valore del parametro dell'equazione di ricorrenza q è compreso tra 1 e $n-1$ (vedi punto 4 a pagina 9).

Tempo computazionale di quicksort()

CASO PEGGIORE [$q = 1$ oppure $q=n-1$]

Se consideriamo questi valori estremi, 1 ed $n-1$, avremo la stessa equazione di ricorrenza. Infatti, provando ad assumere che ad ogni livello di ricorsione il valore di q sia sempre 1, ossia quando, indipendentemente dall'input n , la sequenza di sinistra sarà lunga 1 e tutti gli altri elementi si trovano nella chiamata destra, avremo:

$$\begin{cases} 1 & \text{se } n \leq 1 \\ \Theta(n) + T_{QS}(1) + T_{QS}(n-1) & \text{se } n > 1 \end{cases}$$

Invece, assumendo che il valore di q sia sempre $n-1$ avremo che la lunghezza della sequenza di destra sarà sempre lunga 1 e tutti gli altri elementi ($n-1$) andranno nella sequenza di sinistra.

Quindi le due equazioni sono esattamente uguali.

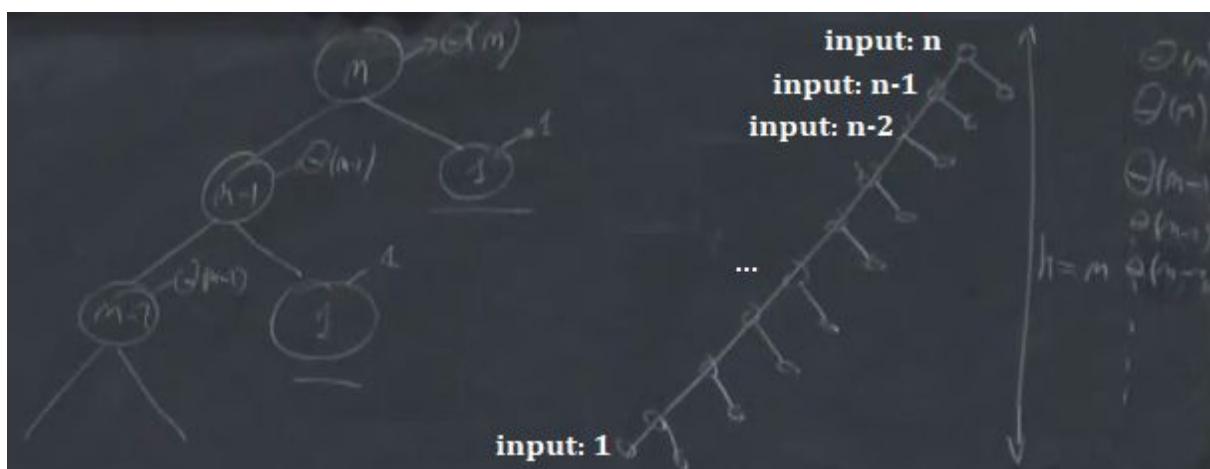
Per risolvere questa particolare equazione di ricorrenza usiamo la tecnica degli alberi di ricorrenza, partendo dalla chiamata principale che ha input n e ci da un costo locale pari a $\Theta(n)$. Come "costo locale" si intende tutto quello che fa parte della chiamata di ricorrenza che non è una chiamata ricorsiva.

L'equazione ha due chiamate ricorsive, quindi ogni nodo avrà due figli.

Un figlio della radice riceverà in input il valore $n-1$ e avrà un costo locale di $\Theta(n-1)$.

In questo caso però l'altro dei uno dei due figli riceverà in input il valore 1, il quale è un caso base che non effettuerà chiamate ricorsive.

Quindi l'albero sarà:



Dato che è praticamente un albero degenere in cui ogni nodo ha un figlio che ha un input pari ad 1, l'altezza di questo albero è pari a n

$$h = n$$

Ogni livello da un contributo pari alla somma $\Theta(\text{input}) + 1$, dove l'1 è per il figlio destro.

Quindi al livello i -esimo è il contributo è di

$$\Theta(n-i+1) + 1$$

ad esclusione del livello 0 che ha un contributo di $\Theta(n)$.

Quindi in questo caso, il tempo di quicksort è pari alla sommatoria di tutti questi contributi.

$$T_{QS}(n) = \Theta(n) + \sum_{i=1}^{n-1} \Theta(n-i)$$

La sommatoria quando $i = 1$ vale $\Theta(n)$, mentre quando $i=n-1$ vale $\Theta(2)$ (che sono le ultime due foglie del nodo che ha come input 1).

Questa è una sommatoria banale e il suo risultato è pari a :

$$\frac{n \cdot (n+1)}{2} - 1$$

[\[la dimostrazione\]](#)

Dato che troviamo una moltiplicazione $n \cdot n$, il risultato di questa equazione di ricorrenza è quadratica.

$$T_{QS} = \Theta(n^2)$$

[RIASSUNTO]

Se esistesse una sequenza in input tale che ogni sua sottosequenza, su cui quicksort lavora, viene divisa in due parti (sinistra e destra) tali che una delle due ha un solo elemento mentre l'altra contiene tutti gli altri elementi causerebbe un tempo computazionale di quicksort quadratico..

Questo accade quando la sequenza è composta da elementi tutti strettamente crescenti.

CASO MIGLIORE [$q = n/2$]

Se invece di considerare i valori estremi di q , considerassimo un q SEMPRE uguale a $n/2$, cioè se esistesse una sequenza in input che viene sempre divisa in due e tutte le sottosequenze da lei generate vengono sempre divise in due avremo un'equazione di ricorrenza pari a :

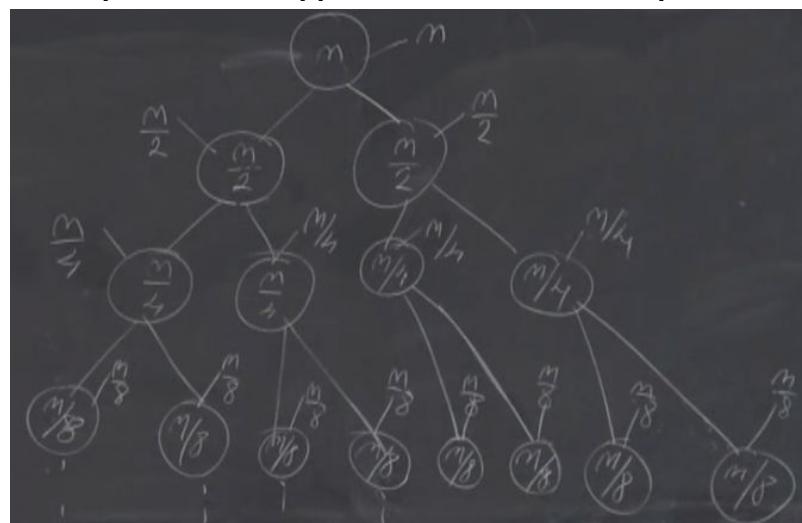
$$\begin{cases} 1 & \text{se } n \leq 1 \\ \Theta(n) + T_{QS}(n/2) + T_{QS}(n/2) & \text{se } n > 1 \end{cases}$$

Ma questa equazione è la stessa del MergeSort e la sua forma chiusa è

$$T_{QS} = \Theta(n \log_2 n)$$

Una sequenza che soddisfa ciò deve chiedere sempre a partiziona() di eseguire il massimo numero di scambi (che al massimo sono $n/2$). Un caso banale di ciò avviene se la sequenza contiene tutti gli stessi elementi perché gli indici si fermano ad ogni incremento/decremento e si fermano nel mezzo, effettuando $n/2$ scambi. Dato che la sequenza contiene elementi tutti uguali, anche le sue sottosequenze avranno ancora elementi tutti uguali e partiziona si fermerà sempre a $n/2$.

L'albero di ricorrenza di questa equazione sarà un albero pieno se l'input ha una lunghezza che è una potenza di 2 oppure sarà un albero completo se non lo è.



Ogni livello contribuisce per $\Theta(n)$ e l'altezza dell'albero è $\log_2 n$ perché ogni volta divido per due l'input.

Conclusione dei due casi

Nel primo esempio abbiamo un $\Theta(n^2)$ ed abbiamo un albero di altezza n in cui ogni nodo da un contributo di $\Theta(n)$. Nel secondo esempio abbiamo un $\Theta(n \log_2 n)$ ed abbiamo un albero di altezza $\log_2 n$ in cui ogni nodo da un contributo di $\Theta(n)$.

Quindi possiamo dire che:

$$T_{QS} = \Theta(\text{contributo} * \text{altezza_albero})$$

Questi due esempi dimostrano che il tempo di quicksort dipende non solo della lunghezza n ma anche da come queste sequenze sono fatte.

Dato che esistono almeno due classi di istanze che asintoticamente determinano tempi di esecuzione diversi, è ragionevole considerare il caso peggiore, il caso migliore ed il caso medio. **Fissando un numero di nodi n , l'albero binario in cui ogni nodo interno ha grado due con altezza maggiore è quella del caso peggiore mentre l'albero con altezza minore è quello del caso migliore.**

Proviamo adesso a generalizzare la forma degli alberi.

Quello che sappiamo è che:

- 1) **Ogni nodo interno ha grado due**, ossia ha due figli; non può esistere un nodo che ha un solo figlio perché quando viene effettuato una chiamata ricorsiva (destra o sinistra) sicuramente viene eseguita anche l'altra.
Questo vincolo è molto "largo" perché contiene sia alberi pieni sia albero "a spina" come quelli dell'esempio 1.
- 2) **Il numero delle foglie è pari ad n** perché una foglia corrisponde ad un caso base ed ogni caso base ha lunghezza 1 e quindi corrisponde ad un elemento della sequenza.
- 3) *Dimostrazione che il numero totale di nodi = $2n - 1$*

Combinando le informazioni del punto 1 e 2, deduciamo che **il numero totale di nodi è pari a $2n-1$** .

$$\forall n \geq 1 \quad NN(n) = 2n - 1$$

Dove $NN(n)$ rappresenta il numero di nodi di un albero binario **con tutti i nodi interni di grado 2**.

NB. incominciamo da 1 perché impostare $n=0$, significa considerare un albero vuoto.
Questo può essere dimostrato per induzione forte:

a) **CASO BASE [$n=1$]**

E' il caso in cui l'albero ha un solo nodo.

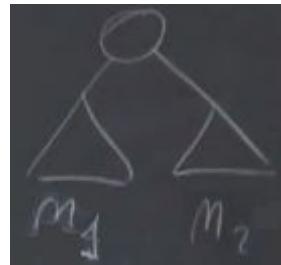
Sostituiamo il valore di n avendo: $NN(1) = 2*1-1 = 2-1 = 1$

Dato che l'unico nodo è anche una foglia, il caso base è vero banalmente.

b) **PASSO D'INDUZIONE [$n>1$]**

Sia fissato un $n > 1$ che rappresenta il numero delle foglie. Ciò significa che esiste almeno un nodo interno che ha due sottoalberi non vuoti, perché tutti i nodi interni hanno grado 2.

Questi due sono sottoalberi di un albero in cui tutti i nodi interni sono di grado 2 quindi anche a loro volta sono sottoalberi in cui tutti i nodi interni hanno grado 2 (ndr. è vero perché se così non fosse l'albero originale non avrebbe che TUTTI i nodi interni hanno grado 2).



Sul numero delle foglie di questi due sottoalberi possiamo dire che:

- 1) Sia n_1 e n_2 il numero delle foglie dei due sottoalberi, la loro somma è pari a n perché dato che tutte le n foglie si trovano contenute nei due sottoalberi perché il nodo interno "radice" (padre dei due sottoalberi) non è, ovviamente, una foglia.

$$n = n_1 + n_2$$

- 2) Sia n_1 e n_2 devono essere maggiori di 0 perché se n_1 fosse 0 il numero di nodi del sottoalbero sinistro sarebbe 0 e quindi non ci sarebbero nodi interni (questo perché se non esistono figli non possono esistere padri); ma se il numero di nodi interni è 0, il sottoalbero di sinistra non esiste e il nodo interno "radice" avrebbe grado 1. Lo stesso discorso si applica pari pari al sottoalbero destro.

$$n_1 > 0$$

$$n_2 > 0$$

Da queste tre proprietà deduciamo che:

$$n_1 < n$$

$$n_2 < n$$

Applico quindi l'ipotesi induttiva su n_1 e n_2 :

$$\text{NN}(n_1) = 2 * n_1 - 1$$

$$\text{NN}(n_2) = 2 * n_2 - 1$$

Ma il numero di nodi totali, ossia $\text{NN}(n)$, lo posso esprimere come il numero di nodi del sottoalbero destro + il numero di nodi del sottoalbero sinistro + 1 per la "radice"

$$\text{NN}(n) = \text{NN}(n_1) + \text{NN}(n_2) + 1$$

Sostituiamo:

$$\text{NN}(n) = 2 * n_1 - 1 + 2 * n_2 - 1 + 1$$

$$\text{NN}(n) = 2 (n_1 + n_2) - 1$$

Ma $n_1 + n_2 = n$:

$$\text{NN}(n) = 2 n - 1$$

CASO MEDIO

Sia nel caso migliore sia nel caso peggiore esistono algoritmi che sono decisamente più veloci di quicksort, ad esempio InsertionSort nel caso migliore è lineare e HeapSort e MergeSort nel caso peggiore impiegano $n(\log_2 n)$.

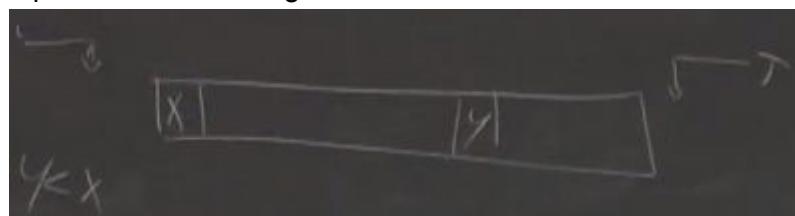
Analizziamo quindi il caso medio. Prima di tutto, in bocca al lupo.

Premesse:

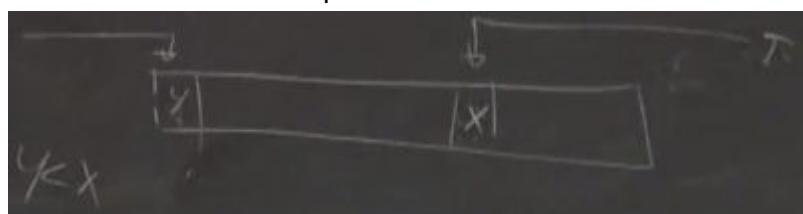
- 1) Considereremo solo elementi distinti in una sequenza.
- 2) **Definiamo come rango il numero delle sequenze che sono minori o uguali del pivot.**

Il rango varia da 1 a n e se ad ogni iterazione il rango fosse sempre 1 otterremo il caso peggiore. Il rango si lega strettamente alla dimensione della parte sinistra, perché se il rango (r) è di dimensione 1 allora la parte sinistra (q) sarà anch'essa di dimensione 1.

Dal rango 2 in poi avremo che la parte sinistra inizierà a valere la formula $q=r-1$ perché, ad esempio, sia x il pivot (quindi in prima posizione) e che ci sia un elemento y che è minore o uguale del pivot e tutti gli altri elementi sono strettamente maggiori di x , avendo quindi una situazione del genere:



Dato che j viene decrementato finché non trova un elemento minore o uguale del pivot, si fermerà nella posizione di y . Allo stesso modo, i viene incrementato finché non trova un elemento maggiore o uguale del pivot e si fermerà nella posizione di x . I due indici non si sono incrociati e quindi avviene lo scambio tra x e y .



Si riprende ad iterare il ciclo esterno perché è avvenuto uno scambio, e j verrà ancora decrementato finché non trova un elemento minore o uguale del pivot, ma tutti gli elementi compresi tra i e j sono tutti maggiori strettamente del pivot quindi j si fermerà nella prima posizione.

i invece partendo dalla prima posizione viene incrementato finché non trova un elemento maggiore o uguale del pivot ma questo si trova subito nella seconda posizione perché:

- Se nella seconda posizione si trova uno dei tanti elementi maggiori del pivot, i si ferma perché ha trovato un elemento maggiore.
- Se nella seconda posizione si trova proprio il pivot, i si ferma perché ha trovato un elemento uguale.

Quindi la parte sinistra è composta da un solo elemento (y) e il rango è pari a due perché esistono 2 elementi minori o uguali del pivot $\{x, y\}$.

Quindi, possiamo scrivere questa tabella tra il rango e la dimensione della parte sinistra:

rango	lung. sx
1	1
2	1
3	3
4	3
..	..
m	$m-1$

Adesso arriva il bello, inizio lezione 15:

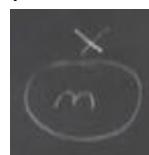
Se considero che un qualsiasi elemento della sequenza può diventare pivot (semplicemente invece di prendere il primo elemento, randomizziamo la scelta del pivot) abbiamo che partiziona() potrà restituire in modo casuale le parti sinistra e destra rispetto alla tabella qui sopra. **Quindi questi tipi di partizionamento avranno una possibilità di $1/n$ di essere selezionati.**

Proviamo adesso a definire la funzione di tempo di n la quale mi restituisce il tempo che impiega quicksort mediamente su una sequenza di lunghezza n .

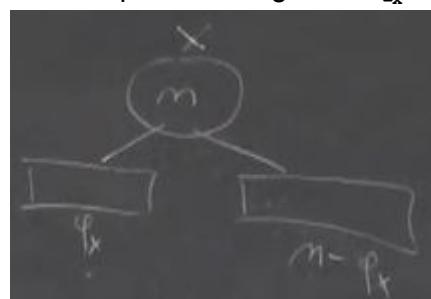
Ricordando che il caso induttivo della funzione di ricorrenza del quicksort è questo:

$$T_{QS}(n) = \Theta(n) + T_{QS}(q) + T_{QS}(n-q)$$

Prendiamo un'arbitraria chiamata ricorsiva che prende in input di lunghezza n ; questa chiamata effettua una scelta casuale del pivot che ritorna un pivot con un generico rango x .



Se x è il rango del pivot della chiamata, verranno generate due chiamate ricorsive che lavoreranno su due sottosequenze, la prima di lunghezza q_x e la seconda $n-q_x$.



Se io voglio calcolare il tempo medio di QuickSort **considerando che il rango del pivot selezionato da partiziona() all'inizio sia stato fissato ad x** , potrei sommare il tempo

impiegato da partiziona() nella radice + tempo impiegato per ordinare una sequenza q_x + tempo impiegato per ordinare una sequenza $n-q_x$.

Ciò semplicemente ho preso l'equazione del caso induttivo e ho sostituito il tempo $T(q)$ e $T(n-q)$ rispettivamente con il tempo $T_{\text{medio}}(q_x)$ e $T_{\text{medio}}(n-q_x)$.

$$TM(n) = TM(q_x) + TM(n - q_x) + \Theta(n)$$

Quindi dato che:

- 1) L'equazione appena trovata è "in funzione" della scelta casuale del pivot (quindi anche del suo rango).
- 2) Ogni rango ha la stessa possibilità di essere selezionato (ossia $1/n$).

per trovare il tempo medio su una qualsiasi sequenza di lunghezza n , indipendentemente del rango che partiziona() ci fornisce, possiamo sommare tutti i possibili tempi medi "in funzione" di tutti i ranghi e poi dividerli per il numero dei ranghi, ossia effettuare una media aritmetica.

In altre parole, dovrò sommare al variare del rango sui suoi possibili valori (ossia da 1 ad n) il tempo medio che quicksort impiega su una sequenza di lunghezza q_r (cioè il valore rispettivo di q determinato dal rango r [vedi la tabella]) + il tempo medio che quicksort impiega per ordinare l'altra parte della sequenza + il tempo di partiziona.

Così sto sommando n termini, uno per ogni possibile valore di rango per ottenere la media, divido tutto per n , avendo quindi l'equazione di ricorrenza del tempo medio:

$$TM(n) = \begin{cases} 1 & \text{se } n=1 \\ \frac{1}{n} \sum_{r=1}^n \left(TM(q_r) + TM(n-q_r) + \Theta(n) \right) & \text{se } n>1 \end{cases}$$

Per risolvere questa equazione di ricorrenza non andremo a trovare la forma chiusa, perché è un burdell, ma procederemo in modo indiretto per dimostrare che la soluzione è:

$$TM(n) = O(n * \log_2 n)$$

ci basta il limite superiore perché dato che il tempo migliore è $\Theta(n \log_2 n)$ non è possibile che il caso medio sia migliore del caso migliore.

Usando la definizione di limite superiore avremo:

$$TM(n) = O(n * \log_2 n) \Leftrightarrow \exists c, n_0 > 0 \quad \forall n > n_0 \quad TM(n) \leq c * \log_2 n$$

dato che lavoriamo sulle lunghezze n (ossia, interi positivi) e che dobbiamo dimostrare l'esistenza di c e n_0 per tutte le lunghezze da un certo punto in poi, possiamo procedere per induzione.

Dato che il valore di c e n_0 non lo sappiamo, assumeremo che esistano e questo non ci darà problemi finché il valore di c e n_0 che necessitiamo sia positivo; se ci troviamo davanti ad un c e n_0 negativo ci dovremmo fermare e la dimostrazione non sarà valida.

Procediamo.



Ipotesi

$$TM(m) = \begin{cases} \Theta(1) & \text{se } m \leq 1 \\ \frac{1}{m} \sum_{r=2}^m (TM(q_r) + TM(m-q_r) + \Theta(r)) & \text{se } m > 1 \end{cases}$$

Tesi

$$\exists c, n_0 > 0 \quad \forall n > n_0 \quad TM(n) \leq c * \log_2 n$$

[Semplificazione] passo 1. Semplificazione

Dato che se r è uguale a 1 o a 2 abbiamo la stessa equazioni di equivalenza perché $q_1 = q_2$ [vedi la tabella a pag. 19], possiamo “estrarre” il caso $r=1$ dalla sommatoria, avendo una situazione del genere:

$$TM(n) = 1/n [<\text{CASO } r=1> + <\text{SOMMATORIA da 2 a n}>]$$

Ma la parte di $<\text{SOMMATORIA da 2 a n}>$ è pari ad:

$$\sum_{r=2}^m (TM(q_r) + TM(m-q_r) + \Theta(r))$$

la quale possiamo trasformarla in una sommatoria dipendente da r ad una dipendente da q usando la tabella che mette in relazione r e q :

$$\sum_{q=1}^{m-1} (TM(q) + TM(m-q) + \Theta(m))$$

Quindi possiamo scrivere $TM(n)$ così:

$$\frac{1}{n} \left[\underbrace{\overbrace{TM(1) + TM(n-1) + O(n)}_{\text{caso } r=1}} + \sum_{q=1}^{n-1} \left(TM(q) + TM(n-q) + O(n) \right) \right] \quad \text{caso } r>2$$

[Semplificazione] passo 2. Ragioniamo sul caso $r=1$:

- 1) $TM(1) = \Theta(1)$ perché è rappresenta un caso base
- 2) Su $TM(n-1)$ non sappiamo precisamente quanto vale perché è proprio quello che stiamo cercando, ma sappiamo che al massimo varrà $\Theta(n^2)$ perché non può essere peggio del caso peggiore.
- 3) Il caso $r=1$ possiamo esprimere quindi così: $\Theta(1) + \Theta(n^2) + \Theta(n)$.
- 4) Dato che quella somma non può essere più grande di n^2 possiamo esprimere tutto come $\Theta(n^2)$.
- 5) Sostituiamo $\Theta(n^2)$ al caso $r=1$.

[Semplificazione] passo 3. Distribuiamo la divisione $1/n$ su entrambi i membri

$$TM(n) = \frac{1}{n} O(n^2) + \frac{1}{n} \sum_{q=1}^{n-1} \left(TM(q) + TM(n-q) + O(n) \right)$$

[Semplificazione] passo 4. Semplifichiamo $1/n (\Theta(n^2))$

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

[Semplificazione] passo 5. Distribuiamo la sommatoria sulle somme isolando $\Theta(n)$

Possiamo distribuire la sommatoria in due porzioni, la prima contenente i due tempi medi e l'altra che contiene $\Theta(n)$:

$$\frac{1}{n} \sum_{q=1}^{n-1} \left(TM(q) + TM(n-q) \right) + \frac{1}{n} \sum_{q=1}^{n-1} \Theta(n)$$

[Semplificazione] passo 6. Forma chiusa della sommatoria di $\Theta(n)$

Dato che sto sommando $n-1$ volte una funzione lineare otterrò una funzione quadratica ma che moltiplicato per $1/n$ ci restituirà una funzione lineare

$$\frac{1}{n} \left(\sum_{q=1}^{n-1} \Theta(n) \right) = \frac{1}{n} \Theta(n^2) = \Theta(n)$$

[Semplificazione] passo 7. Semplifichiamo $O(n)$ e $\Theta(n)$

Ora la nostra funzione del tempo medio è questa:

$$TM(n) = O(n) + \frac{1}{n} \left(\sum_{q=1}^{n-1} TM(q) + TM(n-q) \right) + \Theta(n)$$

ma dato che:

$$O(n) + \Theta(n) = \Theta(n)$$

perché $O(n)$ è al massimo lineare mentre $\Theta(n)$ è proprio lineare quindi $O(n)$ andrà a contribuire alla costante moltiplicativa di $\Theta(n)$ e quindi asintoticamente la somma tra questi due si può assimilare a $\Theta(n)$

Avremo quindi:

$$TM(n) = \frac{1}{n} \left(\sum_{q=1}^{n-1} TM(q) + TM(n-q) \right) + \Theta(n)$$

[Semplificazione] passo 8. Semplifichiamo la sommatoria

La sommatoria di $TM(q)$ al variare di q fra 1 e $n-1$ vale:

$$TM(1) + TM(2) + \dots + TM(n-1)$$

La sommatoria di $TM(n-q)$ al variare di q fra 1 e $n-1$ vale:

$$TM(n-1) + TM(n-2) + \dots + TM(1)$$

Le sommatorie quindi generano gli stessi valori in ordine inverso.

Dato che questi valori sono sommati tutti due volte possiamo togliere $TM(n-q)$ e sostituirlo con un altro $TM(q)$, avendo quindi:

$$TM(n) = \Theta(n) + \frac{1}{n} \left(\sum_{q=1}^{n-1} 2 * TM(q) \right)$$

Quel 2 è un fattore moltiplicativo costante indipendente dall'indice q , quindi può essere raccolto a fattor comune.

$$TM(n) = \Theta(n) + \frac{2}{n} \left(\sum_{q=1}^{n-1} TM(q) \right)$$

[Semplificazione] passo 9. Conclusione

Quindi, per concludere l'equazione di ricorrenza sulla quale faremo la dimostrazione sarà:

$$TM(n) = \begin{cases} \Theta(1) & \text{Se } n \leq 1 \\ \Theta(n) + \frac{2}{n} \sum_{q=1}^{n-1} TM(q) & \text{Se } n \geq 2 \end{cases}$$

nda. La dimostrazione inizia qui: https://youtu.be/kQU-Wzlb3_0?t=2528

Promemoria: Ipotesi

$$TM(m) = \begin{cases} 1 & \text{Se } m \leq 1 \\ \Theta(1) + \frac{2}{m} \sum_{q=1}^{m-1} TM(q) & \text{Se } m \geq 2 \end{cases}$$

Promemoria: Tesi

$$\exists c, n_0 > 0 \quad \forall n > n_0 \quad TM(n) \leq c * \log_2 n$$

[Dimostrazione] passo 1. $n_0 = 2$

La dimostrazione per induzione necessita di un caso base.

Dato che la nostra tesi necessita che la proprietà sia valida da un certo n_0 (e a noi basta la sua esistenza), ci selezioniamo un $n_0 = 2$ perché essendoci il log il caso $n_0 = 1$ sarebbe banale (perché $\log_2(1) = 0$).

[Dimostrazione] passo 2. Il caso base

Fissato n_0 possiamo sostituirlo nella nostra tesi per avere il caso base:

$$\exists c > 0 \forall n \geq 2 \quad TM(n) \leq c \cdot n \log_2 n$$

Quindi dobbiamo verificare che sia vero:

$$TM(2) \leq c * 2 * \log_2(2) = 2c$$

Ma $TM(2)$ lo possiamo sostituire con il suo caso dell'equazione di ricorrenza, ossia

$$TM(2) = \Theta(2) + \frac{2}{2} \left(\sum_{q=1}^1 TM(q) \right) = \Theta(2) + \Theta(1)$$

ndà. Il $\Theta(1)$ viene dedotto della sommatoria, perché $2/2=1$ e la sommatoria che va da 1 ad 1 di $TM(1)$ è semplicemente un singolo caso base che vale $\Theta(1)$.

Questi due theta $\Theta(2)$ e $\Theta(1)$ sono due costanti ma derivano da due parte diverse del codice: **$\Theta(2)$ deriva da partiziona mentre $\Theta(1)$ deriva dal confronto per capire se stiamo nel caso base.** Quindi sono costanti diverse che possiamo generalizzare con c_1 e c_2 entrambe positive.

Quindi dobbiamo verificare che:

$$c_1 + c_2 \leq 2c$$

Ma noi non conosciamo il valore di c però la proprietà che noi dobbiamo verificare ci dice che **deve esistere** una costante c positiva tale che vale $c_1 + c_2 \leq 2c$.

Ma indipendentemente dai valori di c_1 e c_2 sicuramente esisterà un valore c che moltiplicato per 2 sarà maggiore della loro somma.

[Dimostrazione] passo 3. Vincolo del caso base su c

Quindi possiamo scegliere un c a piacere a patto che rispetti questa proprietà:

$$c \geq \frac{c_1 + c_2}{2}$$

Questo vincolo è importante perché dovrà essere imposto anche sul passo induttivo perché il valore di c è deve essere fissato per tutti gli elementi su cui il passo induttivo lavora. Durante il passo induttivo dovremmo imporre un altro vincolo su c ma quello dovrà essere comunque compatibile con questo vincolo del caso base, in questo modo siamo sicuri che esista un'unica costante c condivisa tra il caso base e il passo induttivo.

[Dimostrazione] passo 4. Passo induttivo

Il passo induttivo deve valere da un certo valore in poi. Questo valore dev'essere strettamente maggiore di 2 (perché non è il caso base) e poi assumiamo, per il principio di induzione forte, che **tutte le proprietà siano valide dal caso base per tutti i valori strettamente più piccoli** del valore che stiamo considerando.

Questo valore verrà denotato con "k".

Dato che $k > 2$, nell'equazione di ricorrenza ci troveremo nel caso non base e quindi avremo:

$$TM(k) = \Theta(k) + \frac{2}{k} \left(\sum_{q=1}^{k-1} TM(q) \right)$$

[Dimostrazione] passo 5. Ipotesi induttiva su $TM(q)$

La sommatoria che contiene $TM(q)$ vale:

$$TM(1) + TM(2) + \dots + TM(k-1)$$

ossia, **sono tutti i valori dei tempi medi strettamente più piccoli del valore k** che stiamo considerando. Quindi, per ipotesi induttiva sappiamo che per tutti questi valori vale la tesi:

forall $1 \leq q \leq k-1$ $TM(q) \leq cq \log_2 q$
per tutti i valori di q vale la tesi

[Dimostrazione] passo 6. Usando l'ipotesi induttiva

Dato che:

$$\begin{aligned} 1) \quad TM(k) &= \Theta(k) + \frac{2}{k} \left(\sum_{q=1}^{k-1} TM(q) \right) \\ 2) \quad TM(q) &\leq c k \log_2 k \end{aligned}$$

posso dire che il tempo medio su k deve essere minore uguale di ciò:

$$TM(k) \leq \Theta(k) + \frac{2}{k} (c) \left(\sum_{q=1}^{k-1} q * \log_2(q) \right)$$

E la nostra tesi diventa quindi:

$$TM(k) \leq \Theta(k) + \frac{2}{k} (c) \sum_{q=1}^{k-1} (q \log_2 q) \leq c k \log_2 k$$

[Dimostrazione] passo 7. Forma chiusa della sommatoria

Ci serve adesso la forma chiusa della sommatoria

$$(c) \sum_{q=1}^{k-1} q * \log_2(q)$$

In realtà questa è una sommatoria molto complessa e quindi non la risolviamo ma ci limitiamo a trovare un maggiorante, ossia un valore che è sicuramente un po' più grande ma che è sufficiente per la nostra dimostrazione.

Dimostriamo che:

$$\sum_{q=1}^{k-1} q * \log_2(q) \leq \left(\frac{k^2}{2} \log_2 k - \frac{k^2}{8} \right)$$

Per eseguire questa dimostrazione dobbiamo tenere conto di un certo grado di approssimazione, infatti, se approssimiamo troppo non arriveremo alla soluzione. Quindi:

1) **Dividiamo in due la sommatoria** nel seguente modo:

Creo due sommatorie, la prima sommatoria da 1 alla metà degli elementi e la seconda sommatoria della metà degli elementi alla fine. Ovviamente la somma tra queste due sarà uguale all'unica sommatoria precedente.

$$\sum_{q=1}^{k-1} (q \log_2 q) = \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q \log_2 q) + \sum_{q=\lceil \frac{k}{2} \rceil}^{k-1} (q \log_2 q)$$

2) **Approssimiamo la sommatoria di destra** nel seguente modo:

- a) I vari $\log_2 q$ che si trovano nell'argomento della sommatoria possono essere approssimati per eccesso al valore di $\log_2 k$ (perché sono tutti minori di $\log_2 k$ perché al massimo arrivano al valore $\log_2 k - 1$):

$$\sum_{\lceil q=\frac{k}{2} \rceil}^{k-1} (q \log_2 q) \leq \sum_{\lceil q=\frac{k}{2} \rceil}^{k-1} (q \log_2 k)$$

- b) In questo modo adesso ho un valore costante indipendente dall'indice della sommatoria che posso raccogliere a fattor comune avendo:

$$\sum_{\lceil q=\frac{k}{2} \rceil}^{k-1} (q \log_2 q) \leq \sum_{\lceil q=\frac{k}{2} \rceil}^{k-1} (q \log_2 k) = \log_2 k \cdot \sum_{\lceil q=\frac{k}{2} \rceil}^{k-1} (q)$$

3) **Approssimiamo la sommatoria di sinistra** nel seguente modo:

- a) Allo stesso modo di prima vediamo che i vari $\log_2 q$ che si trovano nell'argomento della sommatoria possono essere approssimati per eccesso al valore di $\log_2(k/2)$ (perché sono tutti minori di $\log_2 k$ perché al massimo arrivano al valore $\log_2(k/2)$)
- b) Allo stesso modo di prima arriviamo a:

$$\sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q \log_2 q) \leq \log_2 \frac{k}{2} \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q)$$

- 4) **Applico la seguente proprietà dei logaritmi** sulla sommatoria di sinistra

Input:

Result:

$$\log_2\left(\frac{x}{2}\right) = \log_2(x) - 1 \quad \text{True}$$

avendo quindi:

$$\log_2\left(\frac{k}{2}\right) \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q) = \log_2(k) \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q) - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q)$$

quindi siamo arrivati al punto dove:

$$\sum_{q=1}^{k-1} (q \cdot \log_2 q) = \log_2(k) \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q) + \log_2(k) \sum_{q=\lceil \frac{k}{2} \rceil}^{k-1} (q) - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q)$$

Ipotesi	1° parte della sommatoria sinistra	sommatoria di destra	2° parte della sommatoria sinista
---------	------------------------------------	----------------------	-----------------------------------

- 5) Adesso posso **raccogliere a fattor comune il $\log_2(k)$** che si trova sia nella sommatoria chiamata “1° parte della sommatoria sinistra” sia nella sommatoria “di destra”. Quindi questo logaritmo dovrebbe moltiplicare la somma di queste due sommatorie, ma la somma di queste due sommatorie è proprio la sommatoria iniziale da 1 a k:

$$\sum_{q=1}^{k-1} (q \cdot \log_2 q) = \log_2(k) \sum_{q=1}^{k-1} (q) - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q)$$

- 6) **La forma chiusa della sommatoria che va da 1 a k di q** (quella di sinistra) è:

$$\sum_{q=1}^{k-1} (q \cdot \log_2 q) = \log_2(k) \frac{k^2 - k}{2} - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} (q)$$

- 7) Dato che il valore massimo della sommatoria è un (tetto di $k/2$) - 1 sicuramente è maggiore o uguale di $(k/2) - 1$ stesso, per definizione di tetto (che è parte intera superiore):

$$\left\lceil \frac{k}{2} \right\rceil - 1 \geq \frac{k}{2} - 1$$

Quindi se la sommatoria che arriva fino ad (tetto di $k/2$) - 1 sarà maggiore o uguale della sommatoria che arriva a $(k/2) - 1$:

$$\sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q \geq \sum_{q=1}^{\frac{k}{2} - 1} q$$

Se moltiplico ambi i membri per -1 la diseguaglianza cambia segno:

$$-\sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q \leq -\sum_{q=1}^{\frac{k}{2} - 1} q$$

Quindi adesso posso dire che parte di destra dell'equazione nel punto 6 è minore o uguale degli stessi componenti a patto di sostituire la sommatoria col (tetto di $k/2$) - 1 con $(k/2) - 1$:

$$\sum_{q=1}^{k-1} (q \log_2 q) = \log_2 k \left(\frac{k^2 - k}{2} \right) - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q \leq \log_2 k \left(\frac{k^2 - k}{2} \right) - \sum_{q=1}^{\frac{k}{2} - 1} q$$

Tesi di partenza Eq. del punto 6

- 8) Abbiamo fatto tutto ciò per poter trovare la forma chiusa della sommatoria che è pari ad:

$$\sum_{q=1}^{\frac{k}{2} - 1} q = \frac{\frac{k}{2} - 1 \cdot \frac{k}{2}}{2}$$

NB. è sempre la stessa formula: [qui trovi la dimostrazione](#)

- 9) Moltiplicando $\log_2 k$ per la frazione avremo:

$$\log_2 k \left(\frac{k^2 - k}{2} \right) = \frac{k^2}{2} \log_2 k - \frac{k \log_2 k}{2}$$

Avremo quindi:

$$\sum_{q=1}^{k-1} (q \log_2 q) \leq \frac{k^2}{2} \log_2 k - \frac{k \log_2 k}{2} - \frac{\frac{k}{2} - 1 \cdot \frac{k}{2}}{2}$$

Ma dato che:

$$\frac{\frac{k}{2} - 1 \cdot \frac{k}{2}}{2} = \frac{\left(\frac{k}{2}\right)\left(\frac{k}{2}\right) - \frac{k}{2}}{2} = \frac{\left(\frac{k^2}{4} - \frac{k}{2}\right) \cdot \frac{1}{2}}{2} = \frac{\frac{k^2}{8} - \frac{k}{4}}{2}$$

Possiamo sostituire:

$$\sum_{q=1}^{k-1} (q \log_2 q) \leq \frac{k^2}{2} \log_2 k - \frac{k \log_2 k}{2} - \left(\frac{k^2}{8} - \frac{k}{4} \right)$$

La parte sottolineata in verde è esattamente ciò che vogliamo, quindi concentriamoci sui termini in rosso.

- 10) Dato che il termine $k \log_2 k$ è più grande di k per ogni k maggiore o uguale di 2 possiamo dire che la loro somma è negativa:

$$-\frac{k \log_2 k}{2} + \frac{k}{4} \leq 0$$

Quindi l'espressione è pari ad, dove "qualcosa" è la somma negativa tra $(k \log_2 k)/2$ e $k/4$:

$$\sum_{q=1}^{k-1} (q \log_2 q) \leq \frac{k^2}{2} \log_2 k - \frac{k^2}{8} - qualcosa$$

Ma dato che abbiamo tolto "qualcosa" sicuramente è minore o uguale degli stessi termini a cui non ho tolto quel "qualcosa"

$$\sum_{q=1}^{k-1} (q \log_2 q) \leq \frac{k^2}{2} \log_2 k - \frac{k^2}{8} - \text{qualcosa} \leq \frac{k^2}{2} \log_2 k - \frac{k^2}{8}$$

11) Quindi abbiamo dimostrato che la tesi è più piccola (o uguale) del valore che dovevamo dimostrare possiamo concludere che la dimostrazione è vera e vale:

$$\sum_{q=1}^{k-1} q * \log_2(q) \leq \left(\frac{k^2}{2} \log_2 k - \frac{k^2}{8} \right)$$

[Dimostrazione] passo 8. Sostituiamo la forma chiusa

Sostituendo la forma chiusa appena trovata possiamo dire che:

$$\Theta(k) + \frac{2}{n}(c) \left(\sum_{q=1}^{n-1} q * \log_2(q) \right) \leq \Theta(k) + \frac{2}{k}(c) \left(\frac{k^2}{2} \log_2 k - \frac{k^2}{8} \right)$$

Dato che $TM(k)$ è proprio minore o uguale della parte sinistra [vedi punto 6] posso applicare la transitività:

$$TM(k) \leq \Theta(k) + \frac{2}{k}(c) \left(\frac{k^2}{2} \log_2 k - \frac{k^2}{8} \right)$$

Applicando delle semplici operazioni matematiche avremo che:

$$1) \quad \cancel{\frac{2}{k}}(c) \left(\frac{k}{2} \log_2 k \right) = \cancel{2c} \frac{k}{2} \log_2 k = c k \log_2 k$$

$$2) \quad \cancel{\frac{2}{k}}(c) \left(-\frac{k^2}{8} \right) = c \left(-\frac{k}{4} \right) = -\frac{ck}{4}$$

Quindi arriveremo a:

$$TM(k) \leq c k \log_2(k) - \frac{ck}{4} + \Theta(k)$$

N.B. nota che $TM(k) \leq c k \log_2(k)$ è la tesi che vogliamo dimostrare.

[Dimostrazione] passo 9. Vincolo del passo induttivo

Dato che questi due termini:

$$-\frac{ck}{4} + \Theta(k)$$

Sono entrambi lineari in k , in cui il primo è sottratto e il secondo è aggiunto.

$\Theta(k)$ è il costo del partizionamento e quindi è limitato inferiormente e superiormente della costante c_1 la stessa che si trova nel caso base

$$\Theta(k) = c_1 * k$$

[Dimostrazione] passo 10. Conclusione

Quindi potremmo dire che:

$$TM(k) \leq c k \log_2 k - c \frac{k}{4} + c_1 k$$

Ora se questo termine fosse negativo:

$$-c \frac{k}{4} + c_1 k$$

avremmo risolto perché il tempo medio di k sarebbe pari a

$$TM(k) \leq c * k * \log_2 k - "qualcosa"$$

cioè significa che $TM(k)$ è minore o uguale di una quantità un po' più piccola di $c * k * \log_2 k$. Se è più minore o uguale di una quantità più piccola di $c * k * \log_2 k$ sicuramente allora sarà minore o uguale di $c * k * \log_2 k + "qualcosa"$.

Ciò funziona ma dobbiamo essere sicuri che quel "qualcosa" sia negativo:

$$(-c \frac{k}{4} + c_1 k) \leq 0$$

In questa disequazione l'unica cosa che posso scegliere è c perché è la costante che stiamo cercando di dire che esiste, mentre, c_1 è la costante di partiziona.

Dato che dobbiamo solo dimostrare l'esistenza di c possiamo sceglierla come ci pare a patto che rispetti anche il vincolo del caso base.

Quindi, quale vincolo deve rispettare c affiché la disequazione sia vera?

$$1) \quad (-c \frac{k}{4} + c_1 k) \leq 0$$

$$2) \quad c_1 k \leq c \frac{k}{4} //\text{cancello } k$$

$$3) \quad c_1 \leq \frac{c}{4}$$

$$4) \quad c \geq 4c_1$$

Quindi esiste una costante $c \geq 4c_1$ che rende :

$$TM(k) \leq c k \log_2 k$$

Questa costante c è compatibile con quella del caso base perché

$$1) \quad \text{Vincolo del caso base} = c \geq \frac{c_1 + c_2}{2}$$

$$2) \quad \text{Vincolo del passo induttivo} = c \geq 4c_1$$

Entrambi i vincoli sono del "tipo maggiori o uguali", ossia mi basta selezionare il massimo tra il valore dei due vincoli per rispettarli entrambi.

Quindi il tempo medio è

$$TM(n) = \Theta(n \log_2 n)$$

C.V.D., Afammok

Conclusioni Finali

E quindi, al finale, è un buon algoritmo. Salut'm a sort.

Il limite minimo degli ordinamenti

AUTORE: Denny Acciaro
[acciariogennaro@gmail.com]

Fonti

- 1) <https://www.youtube.com/watch?v=J1ltJuxnkN4> (Lezione 16)
- 2) <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/7-limite-ordinamenti.pdf>
- 3) https://it.wikipedia.org/wiki/Approssimazione_di_Stirling

Proprietà degli alberi di decisione	3
Analisi del limite inferiore asintotico per il caso peggiore	4
Analisi del limite inferiore asintotico per il caso medio	6
Dimostrazione della forma dell'albero completo	7

Durante lo studio di un problema dobbiamo poter capire quando ha ancora senso continuare a cercare algoritmi per trovare quello migliore possibile.

NB: Non andremo a studiare gli algoritmi di ordinamento perché ne sono potenzialmente infiniti, andremo quindi a studiare il **problema** considerando quali tipi di strumenti computazionali la classe di algoritmi utilizza per risolvere il problema.

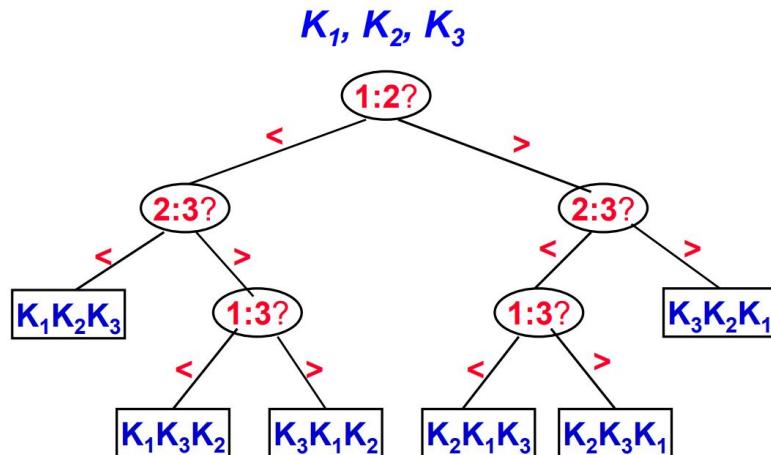
Nel nostro caso andremo ancora una volta ad analizzare gli algoritmi di ordinamento basati sui confronti; dove appunto il **confronto** è lo strumento computazionale elementare. Infatti in tutti gli algoritmi studiati finora gli scambi sono usati soltanto per tenere traccia dei confronti (*infatti, non ha senso scambiare due elementi senza confrontarli precedentemente, come faresti a dire che li stai scambiando nel modo corretto?*).

Introduciamo quindi l'**albero dei confronti (o in generale, di decisione)** come una struttura dati ad albero in cui i nodi sono i confronti e gli archi rappresentano i possibili risultati del confronto.

Nel nostro caso dato che dobbiamo confrontare solo se due numeri sono maggiori o minori fra loro (assumiamo, per semplicità, che tutti gli elementi siano distinti) esistono solo due possibili casi [il primo numero è minore del secondo e viceversa], quindi ogni nodo interno ha grado due (perché non è possibile che un confronto abbia un solo esito).

Usiamo l'albero dei confronti per ordinare una stringa di 3 elementi distinti:

Siano dati tre elementi arbitrari:



Il nodo “1:2?” indica semplicemente la relazione d’ordine tra l’elemento in posizione 1 e quello in posizione 2. Se K_1 è minore di K_2 va a sinistra altrimenti a destra; così per tutti i nodi.

Essendo 3 gli elementi distinti, esistono 3 fattoriale (6) possibili permutazioni della stringa, che sono tutte rappresentate nell’albero delle decisioni dalle foglie.

Deduciamo quindi che **gli esiti dei confronti di un percorso testimoniano la proprietà di ordinamento della permutazione**, ossia, ad esempio, la permutazione $\langle K_1, K_2, K_3 \rangle$ è stata generata del percorso (1 minore di 2, 2 minore di 3) e la sua validità è testimoniata appunto dai confronti che la compongono.

Possiamo notare che nel caso della sequenza $\langle K_1, K_3, K_2 \rangle$ è presente un confronto in più rispetto a $\langle K_1, K_2, K_3 \rangle$, questo perché avendo avuto un esito negativo nel secondo confronto tra K_2 e K_3 non sappiamo la relazione d’ordine che sussiste tra K_1 e K_3 e quindi dobbiamo eseguire un nuovo confronto.

Generalizzando: sia n la lunghezza della sequenza da ordinare, **il numero di confronti necessari per ordinare la sequenza è pari almeno al numero di coppie ordinate**.

Esempio: considerando la sequenza $\langle K_1, K_2, K_3, K_4 \rangle$ è possibile controllare se questa è ordinata effettuando almeno i confronti tra le coppie ordinate presenti, ossia $\langle K_1, K_2 \rangle$; $\langle K_2, K_3 \rangle$ e $\langle K_3, K_4 \rangle$. Se almeno una di questi 3 confronti sarà negativo il numero di confronti sicuramente sarà maggiore ma al minimo sono pari a 3.

Non è possibile avere la certezza che la permutazione sia ordinata a meno di questi confronti perché se ad esempio non confrontassimo K_2 e K_3 non potremmo dire quale tra $\langle K_1, K_2, K_3, K_4 \rangle$ e $\langle K_1, K_3, K_2, K_4 \rangle$ sia ordinata, ovviamente.

Un’altra cosa che possiamo notare è che **ogni confronto partiziona l’insieme costituito dalle partizioni in sottoinsiemi compatibili col risultato del confronto stesso**.

Ed esempio, il confronto “1:2?” dell’albero di prima divide l’albero in due sottoalberi: quello di sinistra avrà delle permutazioni in cui l’elemento nella 1° posizione viene prima di quello

nella 2° posizione, mentre il sottoalbero destro avrà tutte e sole le permutazioni in cui #2 viene prima di #1.

Tutto questo viene usato per discriminare fra loro le permutazioni.

[RIASSUMENDO] Proprietà degli alberi di decisione

Un albero di decisione di ordine n ha queste proprietà

- 1) Sono alberi binari
 - 2) Ogni nodo interno ha grado due
 - 3) Il numero di foglie è almeno $n!$ con n pari al lunghezza della sequenza
 - 4) Per un percorso $\langle K_{i_1}, K_{i_2}, \dots, K_{i_n} \rangle$ che va dalla radice alla foglia vale:
 $\forall z \in [1 \dots n-1] \exists$ nodo interno che confronta K_{i_z} e $K_{i_{z+1}}$
- ossia, nel percorso dalla radice alla permutazione sicuramente per ogni coppia di elementi adiacenti, in quella permutazione, deve essere stata confrontata nel percorso.

L'altezza di un albero di decisione non può essere minore di n perché per la proprietà 4, dovranno essere eseguiti $n-1$ confronti almeno a cui bisogna sommare 1 per la permutazione che si trova come foglia.

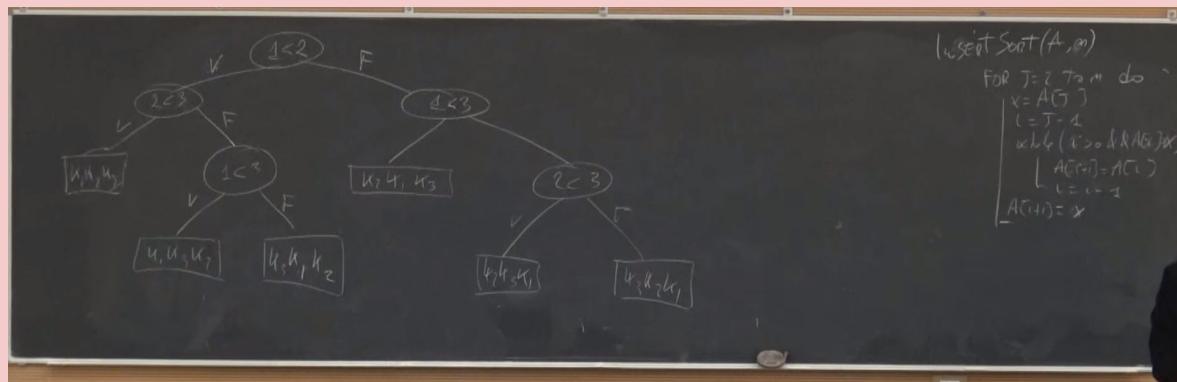
Questa altezza, la quale per definizione è il percorso più lungo, corrisponde al percorso che contiene più confronti, quindi sarà il massimo numero di confronti necessario che l'albero richiede per arrivare a quella permutazione, quindi è il "massimo lavoro possibile".

Ogni algoritmo di ordinamento, fissato un certo n , possiamo dedurre che esiste uno specifico albero di decisione. Ad esempio, Insertion Sort con un n pari a 3 avrà un albero di decisione che identifica proprio il comportamento che Insertion Sort ha eseguito per arrivare alla soluzione del problema. Se n sarà pari a 4, l'albero sarà diverso ma non esistono due alberi diversi per lo stesso n e per lo stesso algoritmo.

Ne consegue che fissato un certo n ed un certo algoritmo, l'altezza dell'albero di decisione corrisponde al numero di confronti che vengono eseguiti al massimo, quindi nel **caso peggiore**.

E' ragionevole chiedersi quale sia l'altezza minima per un albero di decisione, questa sarebbe creata dal caso peggiore del miglior algoritmo possibile.

Esempio di Insertion Sort su $n=3$



Spiegazione: <https://youtu.be/J1ItJuxnkN4?t=3381>

Analisi del limite inferiore asintotico per il caso peggiore

Fissato un certo n , il nostro scopo sarà quindi quello di capire l'altezza minima che può avere un albero di decisione per risolvere il problema. [Quindi fissato un algoritmo, dato che l'altezza rappresenta il caso peggiore stiamo cercando il miglior caso peggiore].

Quello che sappiamo è che:

- 1) Il numero di permutazioni, quindi il numero di foglie, è pari ad $n!$
- 2) Dato che per ogni generico albero binario di altezza h vale questo:

$$\forall \text{AB} \text{ d'altezza } h \quad N^{\circ} \text{ foglie} \leq 2^h$$

perché al massimo l'albero binario di altezza h è pieno ed in tal caso il numero dei figli è pari a proprio a 2^h .

Inoltre essendo gli alberi di decisione degli alberi binari (proprietà 1), possiamo sostituire il numero delle foglie con $n!$, avendo quindi che:

$$n! \leq 2^h$$

mettendo in evidenza h , avremo un vincolo che vale per OGNI albero di decisione:

$$h \geq \log(n!)$$

Da ciò capiamo che l'altezza di un albero di decisione dev'essere almeno $\log(\text{numero_foglie})$, è possibile che l'altezza sia molto più grande di ciò ma sicuro non può essere più piccolo di $\log(n!)$.

- 3) Dato che questo vincolo vale per OGNI albero, dovrà valere anche per l'albero che ha l'altezza minima, che fa il numero minimo di confronti nel caso peggiore.
- 4) Usando l'approssimazione di Stirling affermiamo che:

$$N! \approx \sqrt{2\pi N} \left(\frac{N}{e} \right)^N$$

la funzione fattoriale è sostanzialmente una funzione esponenziale della forma n^n perché la parte della radice quadrata è un fattore moltiplicativo polinomiale di n e rimane quindi $(n/e)^n$.

$$\left\{ \begin{array}{l} n! \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \Rightarrow \text{effettua il log sui membri:} \\ h \geq \log_2(n!) \end{array} \right.$$

$$\Rightarrow \log_2(n!) \geq \log_2\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) \quad // \text{PROP DEL LOGARITMO DEL PRODOTTO}$$

$$\Rightarrow \log_2(n!) \geq \log_2(\sqrt{2\pi n}) + \log_2\left(\left(\frac{n}{e}\right)^n\right)$$

$$\Rightarrow \log_2(n!) \geq \frac{1}{2} \log_2 n + \frac{1}{2} \log_2(2\pi) + n(\log_2 n - \log_2 e) \quad // \text{PROP DEL LOG DEL QUOTIENTE}$$

- $\frac{1}{2} \log_2 n \Rightarrow$ termine logaritmico

- $\frac{1}{2} \log_2(2\pi) \Rightarrow$ termine costante

- $n(\log_2 e) \Rightarrow$ termine lineare

- $n(\log_2 n) \Rightarrow$ termine " $n(\log_2 n)$ "

QUINDI $\log_2(n!) \geq \text{ALMENO } n(\log_2 n)$

QUINDI

$h \geq \log_2(n!) \geq \text{QUANTITÀ ASINTOTICAMENTE EQUIVALENTE AD } n(\log_2 n)$

QUINDI

$$h = \Omega(n(\log_2 n))$$

5)

- 6) Questi calcoli significano che h non può valere meno di $n(\log n)$, ossia, **non può esistere un algoritmo che nel caso peggiore faccia meno confronti di $n(\log n)$** , che è il miglior caso peggiore.

Gli algoritmi ottimali per il caso peggiore sono MergeSort e HeapSort

Analisi del limite inferiore asintotico per il caso medio

Per l'analisi del caso medio dovremmo capire come misurare il tempo medio di un algoritmo usando il suo albero di decisione.

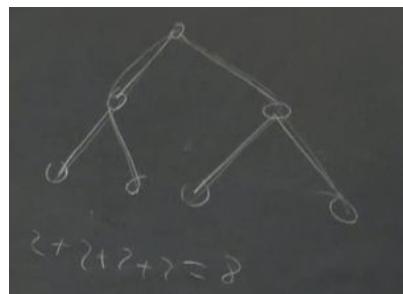
Dato che il nostro strumento computazionale elementare è il confronto, il tempo medio di un algoritmo rappresenta il numero medio di confronti per arrivare dalla radice alla foglia dell'albero di decisione.

Ma il numero di confronti dalla radice fino ad una foglia rappresenta un percorso quindi se riuscissi a capire mediamente quanto un percorso è lungo ciò ci darebbe informazioni su numero medio di confronti.

Quindi in sostanza dovremmo calcolare la lunghezza media dei percorsi.

In generale in un albero con k foglie, ci sono k percorsi che partono dalla radice ed arrivano ad una foglia, nel nostro caso le foglie sono $n!$ e ci sono $n!$ percorsi.

Definiamo come **lunghezza del percorso esterno** come la somma delle lunghezze di tutti i percorsi dalla radice ad una foglia.



Esempio: ogni foglia ha un percorso di lunghezza 2.

Quindi il tempo medio (ossia, il numero di confronti medio) di un albero di decisione è pari alla lunghezza del percorso esterno diviso $n!$ (ossia, è semplicemente la media aritmetica tra le lunghezze dei percorsi):

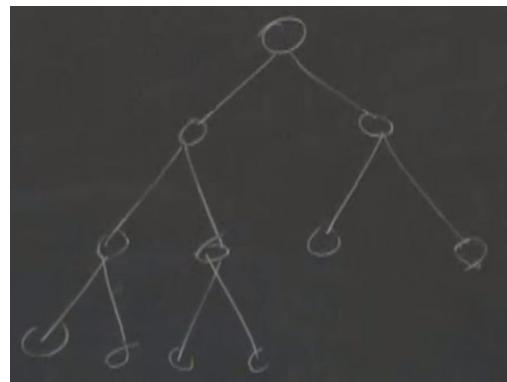
$$T_m(n) = \frac{PE(n)}{n!}$$

Dato che in questa analisi non abbiamo un albero specifico ma **vogliamo sapere qual è il numero medio di confronti minimo possibile**. Inoltre dato che $n!$ è il numero delle permutazioni e non è possibile modificare questo fattore perché altrimenti non si terrebbe conto di qualche permutazione, quindi l'unico modo di ridurre il tempo medio è quello di minimizzare il numeratore, perché ovviamente più piccolo è il numeratore più è piccolo il tempo medio.

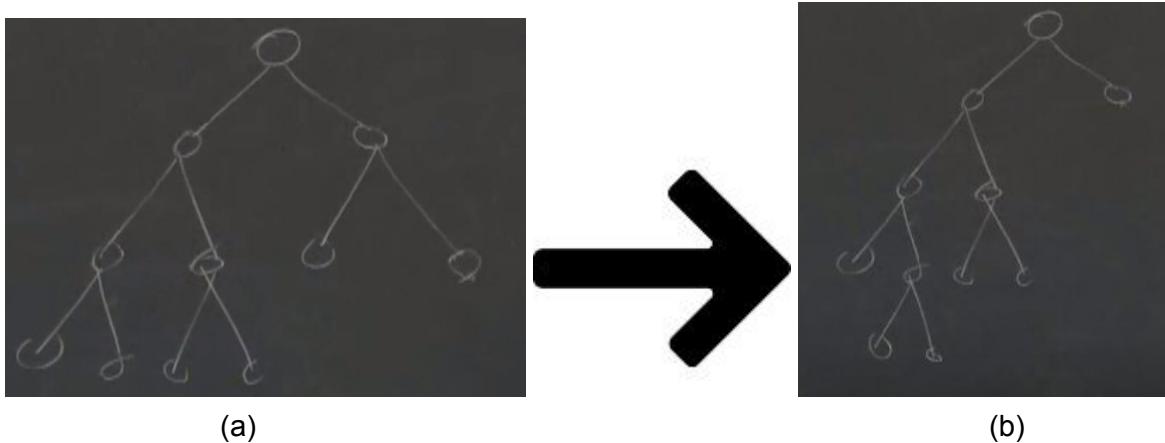
Dimostrazione della forma dell'albero completo

Un albero di decisione NON può essere un albero pieno perché l'albero pieno ha un numero di foglie pari ad 2^h ed esiste un unico caso in cui $n!$ è una potenza di 2, ossia quando $n=2$, ma per n maggiore di 2^n ! conterrà almeno un prodotto con un numero dispari quindi non potrà mai essere una potenza di 2, quindi un albero di decisione potrà al massimo essere un albero completo e non un albero pieno.

Ma un albero di decisione che minimizza il percorso esterno, quindi che ha il numero medio di confronti minore possibile, ha proprio la forma di un albero completo, ossia **è un albero in cui ogni permutazione si trova nella profondità pari ad h oppure ad $h-1$.**

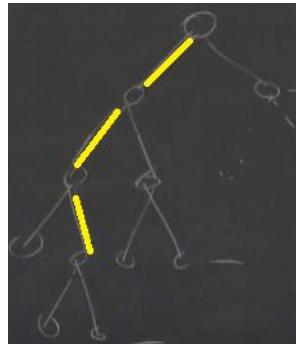


Questo è vero perché partendo dall'albero completo se prendiamo una permutazione e la spostiamo, modificandone la profondità: dato che non è possibile eliminare le foglie, posso soltanto “abbassare” la loro profondità per poter violare la proprietà di essere di un albero completo; infatti se “alzo” le foglie la condizione di albero completo non varia.

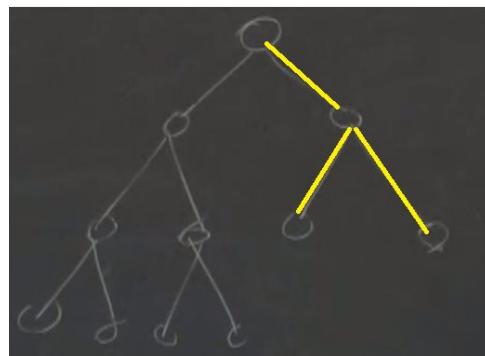


Definiamo PE come il percorso esterno dell'albero completo (a) e PE' come il percorso esterno dell'albero non completo (b).

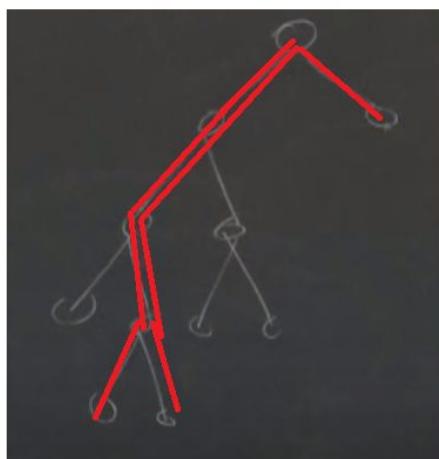
Dato che (b) è stato creato da (a) il suo percorso esterno parte dal percorso esterno di (a) a cui bisogna togliere il percorso che in (a) andava dalla radice alla foglia dove sono stati collegati i due nodi in (b), ossia:



inoltre bisogna togliere i percorsi dei nodi che sono stati tolti della loro posizione in (a):



e dobbiamo aggiungere i seguenti percorsi, che adesso si trovano in (b):



Il primo percorso che è stato tolto ha altezza h , i secondi percorsi tolti hanno altezza $h-1$; il percorso che è stato aggiunto a destra ha altezza $h-2$ e i due percorsi più lunghi che sono stati aggiunti a sinistra hanno entrambi altezza pari a $h+1$

Quindi:

$$\begin{aligned}
 PE' &= PE - h - 2(h-1) + (h-2) + 2(h+1) \\
 &= PE - h - 2h + 2 + h - 2 + 2h + 2 // (-2h \text{ e } +2h), (-h \text{ e } +h) \text{ e } (-2 \text{ e } +2) \text{ si annullano} \\
 &= PE + 2
 \end{aligned}$$

Quindi il percorso esterno del nuovo albero è peggiore dell'albero completo, **quindi la forma dell'albero completo è quella che minimizza i confronti.**

Essendo l'albero completo esistono soltanto due possibili livelli dell'albero: h e $h-1$ (ossia le foglie si trovano soltanto solo al livello h e $h-1$).

Inoltre sappiamo che il numero di foglie all'altezza h (N_h) più il numero di foglia ad altezza $h-1$ (N_{h-1}) sono pari al numero complessivo di foglie ossia $n!$

$$N_h + N_{h-1} = n!$$

Osserviamo che se a tutte le foglie a profondità $h-1$ aggiungiamo due figli l'albero completo diventa un albero pieno, da cui posso ricavare il numero di foglie conoscendo l'altezza.

Il numero di foglie che dobbiamo aggiungere per creare un albero completo sono due volte il numero di foglie di altezza $h-1$ [sono due perché ad ogni nodo all'altezza h aggiungiamo due nodi all'altezza $h-1$].

Quindi se sommiamo N_h a due volte N_{h-1} abbiamo un albero pieno il cui numero complessivo di foglie è pari a 2^h .

$$N_h + 2(N_{h-1}) = 2^h$$

Mettendo a sistema queste due equazioni avremo:

$$\begin{cases} N_h + N_{h-1} = n! \\ N_h + 2(N_{h-1}) = 2^h \end{cases} \Rightarrow N_h = n! - N_{h-1}$$

SOSTITUISCO N_h

$$\begin{cases} n! - N_{h-1} + 2(N_{h-1}) = 2^h \end{cases} \Rightarrow N_{h-1} = 2^h - n!$$

SOSTITUISCO N_{h-1}

$$N_h = n! - (2^h - n!) = n! + n! - 2^h = \underline{\underline{2n! - 2^h}}$$

$$\Rightarrow \begin{cases} N_h = 2n! - 2^h \\ N_{h-1} = 2^h - n! \end{cases}$$

Perciò avremo che $N_h = 2n! - 2^h$

Ora possiamo calcolare la lunghezza del percorso esterno, ossia la somma delle lunghezze di tutti i percorsi dalla radice ad una foglia. Avendo dimostrato che l'albero di decisione è un albero completo, le foglie possono trovarsi soltanto ad altezza h oppure ad altezza $h-1$. Sommiamo quindi le lunghezze dei percorsi ad altezza h e altezza $h-1$:

La somma di tutti i percorsi che arrivano fino alle foglie ad altezza h sarà pari a $h \cdot N_h$, ossia l'altezza (h) di un percorso moltiplicata per tutti i nodi di altezza h (N_h).

La somma di tutti i percorsi che arrivano fino alle foglie ad altezza $h-1$ sarà pari a $(h-1) \cdot N_{h-1}$, per lo stesso motivo.

Quindi:

$$PE = h \cdot N_h + (h-1) \cdot N_{h-1}$$

Adesso che abbiamo la lunghezza del percorso esterno possiamo calcolare il numero medio di confronti che abbiamo detto essere pari ad:

$$T(N) = \frac{PE(N)}{N!}$$

Calcoliamo PE in base ai valori di N_h e N_{h-1} trovati in precedenza:

$$\begin{cases} PE = h \cdot N_h + (h-1) N_{h-1} \\ N_h = 2N! - 2^h \end{cases} \Rightarrow PE = h \cdot (2N! - 2^h) + (h-1) N_{h-1} = h \cdot 2N! - h \cdot 2^h + (h-1) N_{h-1}$$

$$\begin{cases} PE = h \cdot 2N! - h \cdot 2^h + (h-1) N_{h-1} \\ N_{h-1} = 2^h - N! \end{cases} \Rightarrow PE = h \cdot 2N! - h \cdot 2^h + (h-1)(2^h - N!)$$

QUINDI:

$$\begin{aligned} PE &= h \cdot 2N! - h \cdot 2^h + (h-1)(2^h - N!) \\ &= \cancel{h \cdot 2N!} - \cancel{h \cdot 2^h} + \cancel{h \cdot 2^h} - \cancel{N!} \cdot h - 2^h + N! \\ &= h \cdot N! - 2^h + N! \end{aligned}$$

Quindi il nostro tempo medio sarà pari ad:

$$\frac{hN! - 2^h + N!}{N!} =$$

Distribuendo la divisione per i fattori al denominatore avremo:

$$\frac{hN! - 2^h + N!}{N!} = \frac{hN!}{N!} - \frac{2^h}{N!} + \frac{N!}{N!} = h - \frac{2^h}{N!} + 1$$

Ma dato che sappiamo che l'albero è un albero completo ed ogni albero completo con k foglie ha altezza pari a tetto di $\log_2(k)$

NB. la base è 2 perché ogni nodo ha sempre e solo due figli.

*Alb. Completo con k foglie
ha altezza $\lceil \log_2 k \rceil$.*

Perciò dato che il numero di foglie è $n!$ posso sostituire h con tetto di $\log_2(n!)$

$$h = \lceil \log_2 N! \rceil$$

avendo così:

$$= h + 1 - \frac{2^h}{N!} = \lceil \log_2 N! \rceil + 1 - \frac{\lceil \log_2 N! \rceil}{N!}$$

Dato che

$$\lceil \log_2 N! \rceil = \log_2 N! + \varepsilon$$

Ma nella nostra analisi questo ε è ininfluente al livello asintotico, quindi per semplicità possiamo toglierlo, togliendo il tetto al passaggio precedente:

$$= h + 1 - \frac{2^h}{N!} = \log_2 N! + 1 - \frac{\log_2 N!}{N!}$$

Otteniamo quindi che la divisione a destra è pari ad 1, avendo quindi:

$$\log_2(n!) + 1 - 1 \text{ ossia } \log_2(n!)$$

Ma dato che possiamo usare l'approssimazione di Stirling possiamo affermare che la funzione fattoriale è asintoticamente nella forma $(n \log n)$ abbiamo ottenuto che il minor numero di confronti nel miglior caso possibile è limitato inferiormente da $(n \log n)$, quindi anche in questo caso non si può fare meglio di $n \log n$:

$$\Omega(n \log n)$$

Gli algoritmi ottimali per il caso medio sono MergeSort, HeapSort e QuickSort

Alberi AVL

AUTORE: Denny Acciaro
[acciariogennaro@gmail.com]

Fonti

- <https://youtu.be/F66yTlf7hJq?t=2399> (**Lezione 24**)
 - <https://www.youtube.com/watch?v=82eeKPWAo58> (**Lezione 25**)
 - <https://www.youtube.com/watch?v=x62ZCTkeY8I> (**Lezione 26**)
 - <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/15-Alberi-AVL.pdf>
-

Introduzione e definizioni	2
Classe degli alberi bilanciati	2
Alberi perfettamente bilanciati	2
Alberi AVL	2
Alberi AVL minimi	3
Numero di nodi di un AVL minimo	4
Limite superiore per gli AVLM	5
Dimostrazione che il taglio non induce alberi non-AVL	5
L'altezza logaritmica di un AVL Minimo e non	7
Implementazione	9
Rotazioni	9
Rotazione a sinistra	9
Rotazione a destra	10
Doppia rotazione	10
Altezza	10
Inserimento	11
BilanciaSx	11
Cancellazione	13

Introduzione e definizioni

Agli ABR sono efficienti ma le operazioni di inserimento e di cancellazione di un nodo possono rendere l'ABR lineare sul numero di nodi.

Per ovviare a questo problema è stato introdotto una macroclasse di ABR chiamati bilanciati:

Classe degli alberi bilanciati

Una classe A di alberi è detta **bilanciata** se e solo se, per ogni albero appartenente alla classe l'altezza dell'albero è al massimo logaritmica sul numero di elementi dell'albero.

$$\forall T \in A \quad h(T) = O(\log_2 n)$$

Alberi perfettamente bilanciati

Un albero T è detto **perfettamente bilanciato** se per ogni suo nodo il numero di nodi del sottoalbero sinistro di quel nodo differisce al massimo di 1 rispetto a il numero di nodi del sottoalbero destro.

$$T \in APB \Leftrightarrow \forall x \in T, \quad |T(x).sx| - |T(x).dx| \leq 1$$

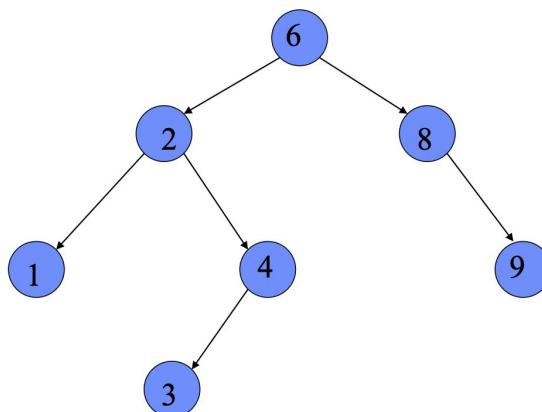
Gli alberi perfettamente bilanciati hanno prestazioni ottimali ($\log N$ garantito) ma inserimenti e cancellazioni complesse, perciò si usa un compromesso, ossia gli alberi AVL.

Alberi AVL

Un albero binario di ricerca è detto **AVL** se per ogni nodo x l'altezza del sottoalbero sinistro e l'altezza del sottoalbero destro differiscono al massimo di uno.

$$T \in AVL \Leftrightarrow \forall x \in T \quad |h(x.sx) - h(x.dx)| \leq 1$$

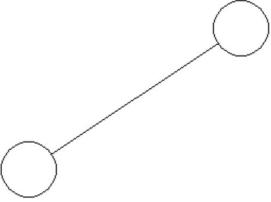
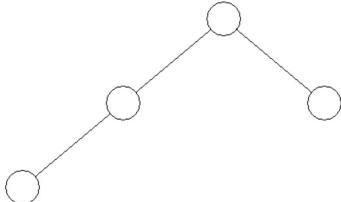
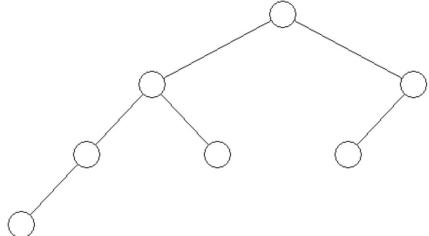
N.B. Tutti gli alberi perfettamente bilanciati sono AVL, il contrario non vale.



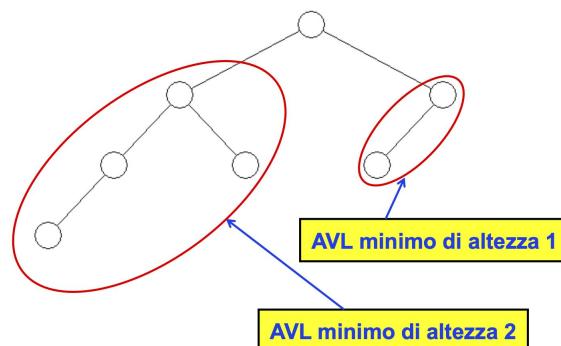
Alberi AVL minimi

Fissato un intero h , esiste un albero AVL il quale possiede il minor numero di nodi possibili; questo è definito come **AVL minimo** di altezza h .

Alcuni esempi:

AVL minimo con $h=1$	
AVL minimo con $h=2$	
AVL minimo con $h=3$	

Si può notare che l'AVL minimo di altezza 3 è composto a sinistra dall'AVL minimo di altezza 2 + l'AVL minimo di altezza 1 + la radice:



Questa è una proprietà generale degli AVL minimi di altezza h perché:

- Essendo di altezza h , l'albero dovrà contenere un sottoalbero di altezza $h-1$ (+ la radice), perché altrimenti banalmente non arriva all'altezza h .
- Questo sottoalbero con altezza pari ad $h-1$ dovrà essere per forza un AVL minimo. Questo si dimostra per assurdo:

Se parto da un AVLM di altezza h , esso conterrà albero di altezza $h-1$.

Se quest'albero non fosse un AVLM($h-1$), potrei sostituirlo con il vero AVLM($h-1$).

In questo modo sostituirei un sottoalbero AVL di h con un altro sottoalbero AVL di h che però ha meno nodi di quello precedente.

Ma questo implicherebbe il fatto che ho tolto qualche nodo dall'AVLM di altezza h e questo non è accettabile perché, per definizione di AVLM, non è possibile togliere nodi da un AVLM conservando ancora la proprietà di AVLM stesso.

Quindi questo smentisce l'ipotesi ed è un assurdo.

Quindi il sottoalbero di altezza $h-1$ dev'essere un AVLM.

- Per lo stesso ragionamento, l'altro sottoalbero di altezza h dev'essere necessariamente un AVL di altezza $h-2$.

Numero di nodi di un AVL minimo

Dato che se fissiamo h , esistono diversi alberi AVL Minimi con il minor numero di foglie e dato che questi nodi non possiamo né aggiungerli né toglierli senza violare la definizione di AVLM, possiamo dedurre che il numero di nodi di un AVLM è una funzione che dipende solo dall'altezza h .

Indaghiamo per scoprire questa funzione, partendo da un generico albero binario.

Il numero di nodi di un albero binario è composto della radice + il numero di nodi del sottoalbero sinistro della radice + sottoalbero destro della radice.

$$\forall T \in AB \quad N(T) = 1 + N(T.sx) + N(T.dx)$$

Applicando questa formula agli AVLM, sapendo che il numero dei nodi degli AVLM dipende dall'altezza avremmo:

$$N_M(h) = 1 + N_M(h-1) + N_M(h-2)$$

A questa dovremmo aggiungere i casi base, per avere l'equazione di ricorrenza:

$$N_M(h) = \begin{cases} 1 + N_M(h-1) + N_M(h-2) & \text{se } h > 1 \\ 2 & \text{se } h = 1 \\ 1 & \text{se } h = 0 \end{cases}$$

N.B. Questa proprietà non vale per gli AVL generici, ma solo per quelli minimi!

Limite superiore per gli AVLM

checkpoint: <https://youtu.be/F66yTlf7hJg?t=6134>

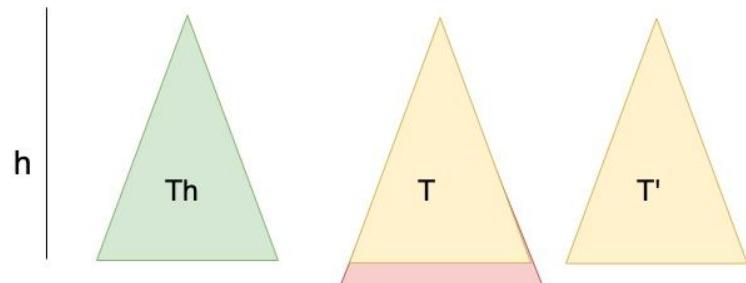
E' necessario convincerci che la classe degli AVLM di altezza h sia una buona classe per limitare superiormente tutti gli AVL di altezza h . Questo si trasforma in matematiche in ciò:

$$\begin{aligned} \forall h \geq 0, T_h &\in AVLM(h) \\ \forall T \in AVL \quad |T| = |T_h| &\Rightarrow h(T) \leq h(T_h) = h \end{aligned}$$

La dimostrazione di ciò si fa per assurdo.

Fissiamo h e definiamo l'AVLM di altezza h (T_h), poi prendiamo un AVL la cui altezza sia maggiore di T_h ma con lo stesso numero di nodi di T_h e chiamiamolo T .

A questo punto prendiamo l'albero T e tagliamo tutti i nodi che si trovano ad altezze superiori di h , generando l'albero T' .



Dato che l'altezza originale di T era maggiore di h ed ho tagliato tutti i nodi maggiori dell'altezza h , significa che ho tagliato almeno 1 nodo.

Quindi la cardinalità di T' è minore della cardinalità di T ed ha la stessa altezza di T_h , ma T_h ha più nodi di T' (perché T_h aveva lo stesso numero di nodi di T).

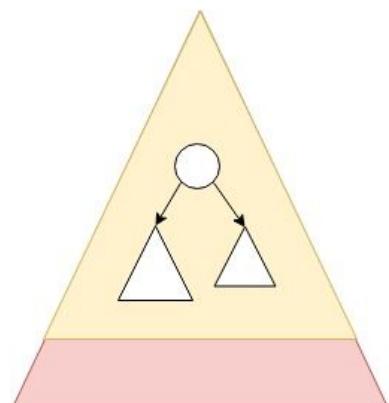
Se T' fosse ancora un AVL avremmo una contraddizione perché avremmo trovato un AVL che ha la stessa altezza h di T_h ma che ha meno nodi dell'AVLM(h).

Dimostrazione che il taglio non induce alberi non-AVL

Per dire che T' sia ancora un AVL sfruttiamo la proprietà che ci dice che se effettuiamo una operazione di taglio di quel tipo non possiamo trasformare un albero AVL in un non-AVL perché, ancora una volta per assurdo:

Se partiamo da un albero AVL e tagliamo i nodi dopo una certa altezza l'albero che ne rimane potrebbe essere un non-AVL solo se viola le proprietà degli AVL ossia che esiste almeno un nodo nel quale la differenza di altezza dei suoi due sottoalberi sia almeno 2.

Supponendo che esista questo nodo che viola la proprietà, la profondità delle foglie del sottoalbero minore (tra i due sottoalberi) dev'essere strettamente minore rispetto al taglio che abbiamo eseguito.



Ciò significa che il taglio non ha tolto alcun nodo al sottoalbero minore e che quindi conserva ancora la stessa altezza dall'albero AVL di partenza.

Invece l'altro sottoalbero ha due possibilità, o ha la stessa altezza o ha un'altezza minore rispetto all'altezza precedente; in ogni caso il taglio non ha aggiunto nodi a questo sottoalbero.

Quindi, se il taglio non ha aggiunto nodi al sottoalbero maggiore e non ha modificato l'altezza del sottoalbero minore e questi due sottoalberi hanno una differenza di altezze maggiore o uguale a 2, significa che già c'era prima e che quindi la tesi che l'albero iniziale fosse un AVL è sbagliata e quindi è un assurdo.

Questo ci permette di concludere la dimostrazione precedente dato che T' è un AVL di altezza h con meno nodi del AVLM di altezza h , e quindi significa che l'albero T non può esistere e quindi ciò ci dimostra che se riusciamo a trovare un limite superiore per gli AVLM di altezza h possiamo estenderlo anche per gli AVL non minimi della stessa altezza.

L'altezza logaritmica di un AVL Minimo e non

L'equazione di ricorrenza del numero di nodi di un albero AVL minimo è molto simile alla funzione all'equazione di ricorrenza dell'equazione di Fibonacci:

$N_M(h) = \begin{cases} 1 + N_M(h-1) + N_M(h-2) & \text{se } h > 1 \\ 2 & \text{se } h = 1 \\ 1 & \text{se } h = 0 \end{cases}$	$FIB(h) = \begin{cases} FIB(h-1) + FIB(h-2) & \text{se } h > 1 \\ 1 & \text{se } h = 1 \\ 0 & \text{se } h = 0 \end{cases}$
---	---

E' possibile dimostrare per induzione che vale questa uguaglianza

$$N_M(h) = FIB(h+3) - 1$$

[Caso Base]

Dato che ci sono due casi base in entrambe le equazioni di ricorrenza, dovremmo gestire due casi base:

1) $h=0$

- $N_M(0) = 1$
- $FIB(3) - 1 = FIB(2) + FIB(1) - 1 = 1 + 1 - 1 = 1$
 - Quindi: $N_M(0) = FIB(3) - 1 = 1$

Dove $FIB(3)$ sarebbe il 3° numero di Fibonacci, ossia 2.

2) $h=1$

- $N_M(1) = 2$
- $FIB(4) - 1 = 3 - 1 = 2$
 - Quindi: $FIB(4) - 1 = N_M(1) = 2$

[Passo induttivo]

Partiamo da un $h \geq 2$ e assumiamo che fino ad allora la veridicità dell'equazione sia stata verificata, avremo che:

- $\forall h \geq 2 \quad N_M(h) = 1 + N_M(h-1) + N_M(h-2)$

dove però vale ciò:

$$\begin{cases} N_M(h-1) = FIB(h-1+3) - 1 = FIB(h+2) - 1 \\ N_M(h-2) = FIB(h-2+3) - 1 = FIB(h+1) - 1 \end{cases}$$

Possiamo quindi sostituire i valori, avendo:

$$\begin{aligned} \forall h \geq 2 \quad N_M(h) &= 1 + FIB(h+2) - 1 + FIB(h+1) - 1 \\ &= FIB(h+2) + FIB(h+1) - 1 \\ &= FIB(h+3) - 1 \end{aligned}$$

C.V.D.

Ora che sappiamo che vale l'uguaglianza tra il numero di nodi in funzione di h e la funzione di Fibonacci, possiamo sfruttare la forma chiusa nota di quest'ultima per dedurre l'altezza h dal numero di nodi.

In particolare sappiamo che:

$$1) \quad N_M(h) = FIB(h+3) - 1$$

$$2) \quad FIB(h) \geq \frac{1}{\sqrt{5}} \cdot \frac{1+\sqrt{5}}{2}^h$$

La dimostrazione di questa disequazione la puoi trovare qui:

http://www.dis.uniroma1.it/~fiji/materiale_damore/2010-11/fibonacci.pdf

3) Mettendo insieme (1) e (2) avremo che:

$$N_M(h) \geq \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1$$

4) Dato che:

$N_M(h)$ = numero di nodi di un AVLM di altezza h

possiamo descrivere questo valore con una variabile n , potendo fare i calcoli per ricavare h

$$a) \quad \sqrt{5} \cdot (n+1) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$$

$$b) \quad \log_2 \left(\sqrt{5}(n+1) \right) \geq h+3$$

Dato che la parte a sinistra può essere ridotta asintoticamente a un $\log_2(n)$ avremo che:

$$h = O(\log_2(n))$$

Nota: siccome la base del logaritmo è maggiore di 0, posso applicare la formula del cambio di base e ricondurmi a $\log_2(n)$.

Da ciò deduciamo che l'altezza di un AVLM è limitata superiormente da una quantità logaritmica in dipendenza del numero di nodi.

Generalizzando, possiamo vedere la relazione come una funzione esponenziale tra il numero di nodi di un AVLM e la sua altezza.

$$N_M(h) \geq \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1$$

Della quale abbiamo ricavato la forma logaritmica dell'altezza.

Se aumentiamo il numero di nodi, la funzione conserva la sua forma esponenziale e quindi di conseguenza conserva la forma logaritmica dell'altezza.

Per questo motivo possiamo generalizzare il concetto di altezza logaritmica a tutti gli AVL (perché sono AVLM a cui possiamo aggiungere nodi) e quindi la classe degli AVL è una classe di alberi bilanciati.

Implementazione

L'implementazione di un AVL avviene tramite un oggetto del genere:

```
public class Nodo {  
    public int k;  
    public int h;  
    public Nodo sx;  
    public Nodo dx;  
}
```

In `h` si tiene traccia dell'altezza per verificare la violazione delle condizioni dell'AVL a tempo costante.

Rotazioni

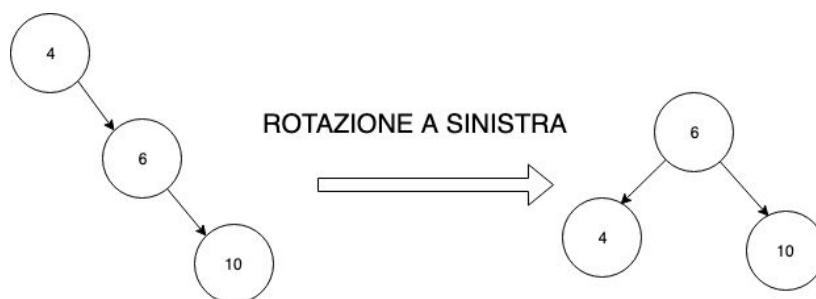
checkpoint: <https://youtu.be/82eeKPWAo58?t=2034>

Le operazioni di inserimento e cancellazione in un AVL, potrebbero facilmente violare le condizioni che la classe AVL ci impone. Per questa ragione introduciamo le operazioni di rotazione le quali modificano l'albero affinché le condizioni non vengano violate.

L'operazione di rotazione parte da un albero sbilanciato e modifica la radice facendo diventare uno dei suoi figli la nuova radice e posizionando la vecchia radice in modo opportuno. In questo modo l'altezza dell'albero si modifica e viene ridotta.

Si noti che la proprietà di ABR viene preservata.

Rotazione a sinistra



RotazioneSx(`T`)

```
| NewRoot = T.sx  
| T.sx = NewRoot.dx  
| NewRoot.dx = T  
| T.h = 1 + max(Altezza(T.sx), Altezza(T.dx))  
| return NewRoot
```

Rotazione a destra



```
RotazioneDx(T)
| NewRoot = T.dx
| T.dx = NewRoot.sx
| NewRoot.sx = T
| T.h = 1 + max(Altezza(T.sx), Altezza(T.dx))
| return NewRoot
```

Doppia rotazione

```
RotazioneDSx(T)
| T.sx = RotazioneDx(T.sx)
| return RotazioneSx(T)
```

Altezza

```
Altezza(T)
| if(T==Null)
| | return -1 //Per convenzione l'albero vuoto ha altezza -1
| return T.h
```

Inserimento

checkpoint: <https://youtu.be/82eeKPWAo58?t=4622>

Adesso possiamo sfruttare le rotazioni per fare un inserimento in un AVL.

Il nodo verrà aggiunto all'albero come se fosse un ABR, poi, partendo dal nodo inserito, ossia alla risalita dell'algoritmo ricorsivo, si effettueranno delle rotazioni se l'albero non è bilanciato secondo le regole AVL, questo lavoro viene effettuato dalle funzioni bilanciaSx() e bilanciaDx().

Al termine di InsertAVL ci sarà una sola rotazione.

```
InsertAVL(T,k)
| if( T != NULL ) then
| | if(t.k==k) then          //Se il valore k era già presente, non faccio nulla
| | | return T
| | | else if(T.k >k) then
| | | | T.sx = InsertAVL(T.sx,k)
| | | | T = bilanciaSX(T)    //N.B. bilanciaSX potrebbe cambiare la radice
| | | | else
| | | | | T.dx = InsertAVL(T.dx,k)
| | | | | T = bilanciaDX(T)
| | | | else
| | | | | T = AllocaNodo      //creazione del nodo
| | | | | T.h = 0
| | | | | T.k = k
| | | | return T
```

BilanciaSx

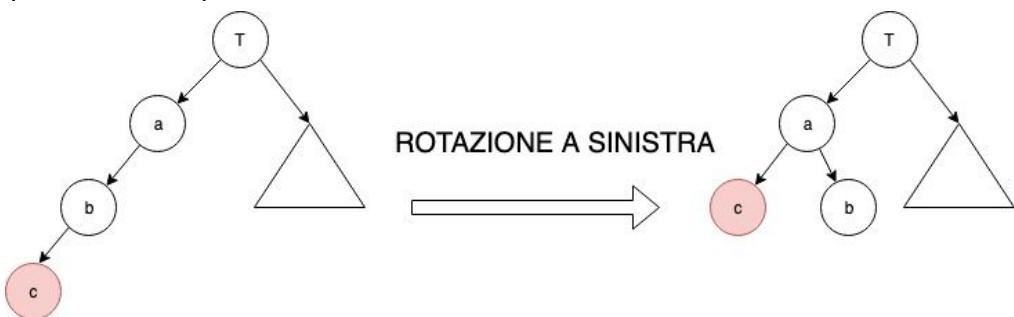
Controlla se c'è una violazione, ossia se la differenza delle altezze dei due sottoalberi di T è uguale a 2 (NB. l'unica violazione che può esserci è questa, perché se parto da un AVL e aggiungo un nodo o non c'è nessuna violazione oppure aggiungo un nodo al sottoalbero di altezza $h-1$ portandolo ad altezza h che quindi andrà in violazione perché differisce di 2 dall'altro sottoalbero che ha altezza $h-2$).

Inoltre dato che richiamo BilanciaSx() dopo l'inserimento a sinistra sono sicuro che il sottoalbero sinistro di T è quello più grande tra i due sottoalberi.

Dato che abbiamo aggiunto al sottoalbero sinistro di T, possiamo distinguere due casi:

- Ho inserito il nodo nel sottoalbero sinistro del sottoalbero sinistro di T
- Ho inserito il nodo nel sottoalbero destro del sottoalbero sinistro di T

Se ho verificato la violazione e mi trovo nel primo caso posso effettuare una rotazione sinistra per risolvere il problema.



Se ho verificato la violazione e mi trovo nel secondo caso posso effettuare una doppia rotazione per risolvere il problema perché viene effettuata prima la rotazione a destra la quale porta l'albero nelle condizioni del primo caso e che posso risolvere poi con una successiva rotazione a sx.



Infine se non ho verificato alcuna violazione è necessario aggiornare i valori delle altezze dei nodi.

```
BilanciaSx(T)
| if( altezza(T.sx) - altezza(T.dx) == 2) then
| | if(altezza(T.sx.sx) > altezza(T.sx.dx)) then
| | | T = rotazioneSx(T)
| | else
| | | T = rotazioneDSx(T)
| else
| | T.h = 1+ max(altezza(T.sx),altezza(T.dx))
return T
```

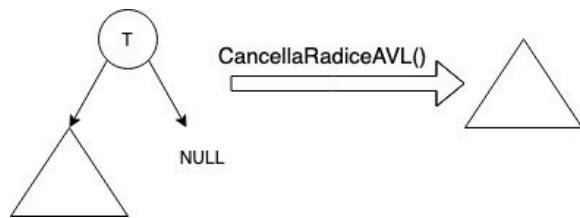
Cancellazione

Se cancello dal sottoalbero destro è possibile che il sottoalbero sinistro crei una violazione rispetto all'AVL, e viceversa.

```

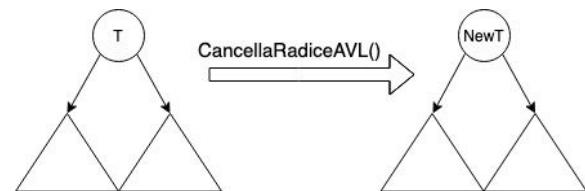
CancellaAVL(T,k)
| if(T!=null) then
| | if(T.k >k) then
| | | T.sx = CancellaAVL(T.sx,k)
| | | T = bilanciaDX(T)          //N.B. bilanciaDX potrebbe cambiare la radice
| | | else if(T.k<k) then
| | | | T.dx = CancellaAVL(T.dx,k)
| | | | T = bilanciaSX(T)
| | | | else
| | | | | T = CancellaRadiceAVL(T,k)
| | | | return T
    
```

Caso 1



Dato che restituisco un sottoalbero di un AVL che è per definizione anch'esso un AVL non ho necessità di effettuare un bilanciamento.

Caso 2



Dato che StaccaMinAVL() elimina un nodo dal sottoalbero destro potrebbe violare le condizioni di AVL

```

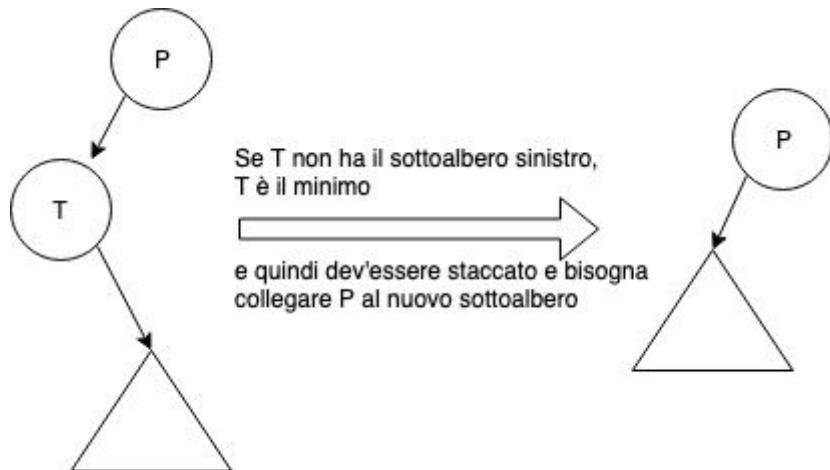
CancellaRadiceAVL(T,k)
| if(T != null) then
| | if(T.sx == null || T.dx == null) then //caso 1
| | | temp = T
| | | if(T.sx = null) then
| | | | T = T.dx
| | | | else
| | | | | T = T.sx
| | | | else                                //caso 2
| | | | | temp = staccaMinAVL(T.dx, T)
| | | | | T.k = temp.k
| | | | | T = bilanciaSX(T)
| | | dealloca temp
| | | return T
    
```

```

staccaMinAVL(T,p)      // 'p' sta per padre
| if(T != null) then
| | if(T.sx != null) then // se esiste il sottoalbero sinistro, T non può contenere il minimo
| | | ret = staccaMinAVL(T.sx,T) // quindi effettuo la chiamata sul suo sottoalbero sx
| | | newRoot = BilanciaDx(T)   // Dato che StaccaMinAVL elimina un nodo potrebbe violare le condizioni
| | else                   // se non esiste il sottoalbero sinistro, T contiene il minimo
| | | ret = T
| | | newRoot = T.dx        // la nuova radice sarà proprio tutto il sottoalbero destro.
| |
| | if(P.sx == T) then    // aggiorno i valori del padre
| | | P.sx = newRoot
| | else
| | | P.dx = newRoot
|
| return ret
return T

```

Nel caso base, in cui $T.sx=null$, avremo che:



Nel caso peggiore il numero di rotazioni è metà dell'altezza ossia è lineare sull'altezza.

Alberi Milanisti

AUTORE: Denny Acciaro
acciariogennaro@gmail.com

Fonti

- <https://youtu.be/x62ZCTkeY8I?t=3669> (Lezione 26)
 - https://www.youtube.com/watch?v=R9_m9ZwFJOA (Lezione 27)
 - <https://www.youtube.com/watch?v=RTsTQEgavDk> (Lezione 28)
 - <http://wpage.unina.it/benerece/ASD/Benerecetti/ASD-1/16-Alberi-Red-Black.pdf>
-

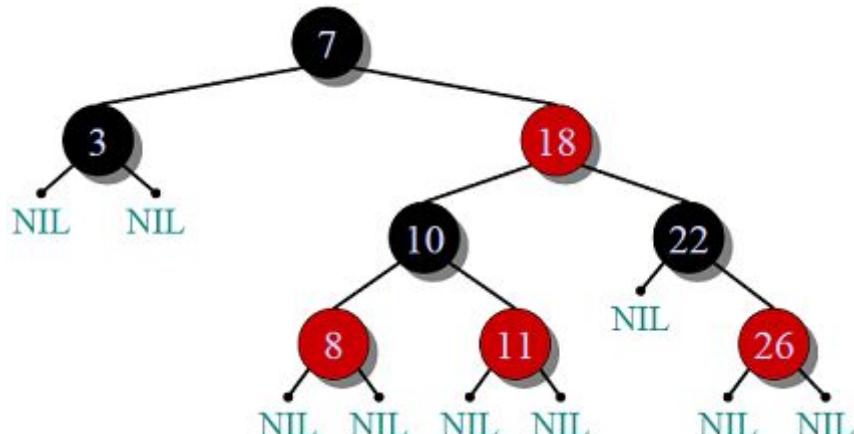
Introduzione	2
Altezza nera	2
Dimostrazione del numero di nodi interni	3
Base di induzione ($h = 0$)	3
Passo di induzione ($h > 0$)	3
Dimostrazione dell'altezza logaritmica	5
Inserimento in un albero RB	6
Rotazione a sinistra	6
Rotazione a destra	6
Inserimento	7
Caso 1: entrambi i figli di T sono rossi e c'è un doppio rosso a sx	8
Fix dei percorsi neri	9
Caso 2: T.dx è nero e c'è un doppio rosso in T.sx.dx	10
Fix dei percorsi neri	10
Caso 3: T.dx è nero e c'è un doppio rosso in T.dx.dx	11
Fix dei percorsi neri	12
Bilancia_Sx(T)	12
Tipo_violazione_sx(s,d)	13
Cancellazione in un albero RB	14
Cancella_radice_RB(T)	15
Stacca-min-RB()	16
Cancella_Bilancia_sx()	16
Caso 1: radice, C,E neri e D rosso	17
Caso 2: D, C, E neri	18
Fix dei percorsi neri	18
Caso 3: D, E neri e C rosso	19
Fix dei percorsi neri	19
Caso 4: D nero ed E rosso	20
Fix dei percorsi neri	20
Violazione-sx()	21
Implementazione dei casi	22

Introduzione

Un albero RB è un Albero binario di ricerca che soddisfa i seguenti vincoli:

- 1) Le chiavi sono presenti solo nei nodi interni.
- 2) Le foglie sono costituite da nodi NULL (detti *nodi sentinella*), in altre parole invece di avere un puntatore a null, gli ultimi nodi che contengono un'informazione hanno un puntatore che punta ad un nodo con valore null, in questo modo il primo vincolo viene rispettato.
- 3) Il colore di ogni nodo può essere o rosso o nero.
- 4) Se un nodo è rosso allora entrambi i figli sono neri.
- 5) Ogni nodo interno deve avere lo stesso numero di nodi neri per ogni percorso fino alle foglie.

Gli alberi rossi e neri sono un'estensione degli alberi pieni, infatti vengono allungati i percorsi usando i nodi neri.



La proprietà (4) impone che non possano esserci due nodi rossi adiacenti.

Possiamo vedere gli RB-Tree come un'estensione degli alberi pieni in quanto allungano i percorsi attraverso i nodi rossi.

Inoltre, un RB-Tree è meno limitato di un AVL e quindi ogni AVL è un RBTree ma non viceversa.

Altezza nera

Sia T un albero RB nel quale è presente un nodo interno x .

Il sottoalbero radicato in x appartenente a T ha una altezza che possiamo denotare con $h(x)$.

Definiamo l'altezza nera $bh(x)$ di un albero radicato in x come il numero di nodi neri lungo un qualsiasi percorso da x fino ad una foglia, **escludendo il colore di x** .

Infine indichiamo con $NI(x)$ il numero di nodi interni dell'albero radicato in x , $NI(x)$ è importante perché indica il numero di nodi che contengono informazioni.

Dimostrazione del numero di nodi interni

Vogliamo dimostrare che il numero di nodi interni appartenenti al sottoalbero radicato in un certo nodo x , è maggiore o uguale ad $2^{bh(x)} - 1$, in matematiche:

$$\forall h \geq 0 : NI(x) \geq 2^{bh(x)} - 1 \quad \text{con } h = h(x)$$

Per dimostrare ciò procediamo per induzione:

Base di induzione ($h = 0$)

L'unico albero Red 'n Black di altezza $h = 0$ è l'albero con un unico nodo con valore null.

Se l'altezza di tale albero è 0 allora l'altezza nera bh sarà uguale a 0.

Inoltre se l'altezza è uguale a 0, esiste un solo nodo e questo è una foglia quindi il numero di nodi interni $NI(x)$ è 0.

Quindi:

$$NI(x) = 0$$

$$bh(x) = 0$$

$$NI(x) \geq 2^{bh(x)} - 1 \Rightarrow 0 \geq 2^0 - 1 \Rightarrow 0 \geq 0 \quad (\text{sempre vero})$$

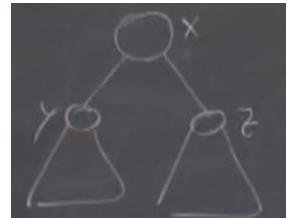
Passo di induzione ($h > 0$)

checkpoint: https://youtu.be/R9_m9ZwFJOA?t=1525

Dato che l'altezza è maggiore di zero, esiste sicuramente un nodo interno dal quale partono due sottoalberi sx e dx (che potrebbero anche essere null, non ci importa). Graficamente abbiamo una struttura del genere:

Il numero di nodi interni di questo albero con altezza > 0 è esprimibile con la somma dei nodi interni dei due sottoalberi + 1 per la radice, ossia:

$$NI(x) = 1 + NI(x.sx) + NI(x.dx)$$



dato che è una funzione ricorsiva, dobbiamo assicurarci che il valore che diamo in input nelle sottochiamate è strettamente minore rispetto a quello originario (altrimenti non terminerebbe mai).

Ciò è verificato perché a priori sappiamo sicuramente che vale ciò:

$$\begin{cases} h(z) < h(x) \\ h(y) < h(x) \end{cases}$$

Perciò posso effettuare la sostituzione, avendo: $NI(x) = 1 + NI(y) + NI(z)$

Inoltre, dato che le altezze di y e z sono minori di $h(x)$ possiamo dire che, per ipotesi induttiva, vale ciò:

$$NI(y) \geq 2^{bh(y)} - 1 \text{ e } NI(z) \geq 2^{bh(z)} - 1$$

Ragioniamo adesso sulle altezze nere:

Tra l'altezza nera di x e le altezze nere di z e y possono esserci solo due casi:

- 1) Sono uguali, se y/z è rosso
- 2) L'altezza nera di x è più grande di 1 rispetto alle altezza nera di y/z , se y/z è nero

Per questo motivo possiamo affermare che vale:

$$\begin{cases} bh(y) \geq bh(x) - 1 \\ bh(z) \geq bh(x) - 1 \end{cases}$$

Questo perché se l'altezza nera di y/z è uguale a quella di x , avremo che $bh(y) > bh(x) - 1$ mentre se vale che $bh(x) = bh(y/z) + 1$, ossia quando il nodo il nodo y/z è nero, avremo che $bh(y/z) = bh(x) - 1$.

Siccome la funzione esponenziale è una funzione monotona, posso elevare tutto il sistema alla 2:

$$\begin{cases} 2^{bh(y)} \geq 2^{bh(x)-1} \\ 2^{bh(z)} \geq 2^{bh(x)-1} \end{cases}$$

Sottraggo 1 a entrambi i membri:

$$\begin{cases} 2^{bh(y)} - 1 \geq 2^{bh(x)-1} - 1 \\ 2^{bh(z)} - 1 \geq 2^{bh(x)-1} - 1 \end{cases}$$

Posso adesso applicare la transitività dato che:

$$\begin{cases} NI(y) \geq 2^{bh(y)-1} \\ NI(z) \geq 2^{bh(z)-1} \\ 2^{bh(y)} - 1 \geq 2^{bh(x)-1} - 1 \\ 2^{bh(z)} - 1 \geq 2^{bh(x)-1} - 1 \end{cases} \Rightarrow \begin{cases} NI(y) \geq 2^{bh(x)-1} - 1 \\ NI(z) \geq 2^{bh(x)-1} - 1 \end{cases}$$

Applico la proprietà della somma tra disequazioni, che ci assicura che:

$$NI(y) + NI(z) \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1)$$

Sommo 1 ad ambo i membri:

$$1 + NI(y) + NI(z) \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$$

In questo modo a sinistra ottengo proprio il valore che abbiamo definito per $NI(x)$

$$NI(x) = 1 + NI(y) + NI(z) \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$$

semplifico i -1 e +1 finali

$$NI(x) = 1 + NI(y) + NI(z) \geq (2^{bh(x)-1} - 1) + 2^{bh(x)-1}$$

semplifico ancora

$$NI(x) = 1 + NI(y) + NI(z) \geq 2 \cdot 2^{bh(x)-1} - 1$$

ma dato che

$$2^1 \cdot 2^{bh(x)-1} = 2^{1+bh(x)-1} = 2^{bh(x)}$$

concludiamo che:

$$NI(x) \geq 2^{bh(x)} - 1$$

C.V.D.

Dimostrazione dell'altezza logaritmica

Vediamo adesso come dimostrare che per ogni RBTree l'altezza è limitata superiormente da un log in base 2 del numero dei nodi, ossia:

$$h = O(\log_2 n)$$

Sappiamo che l'unico modo che abbiamo per aumentare il numero di nodi in un percorso di un RBTree è quello di inserire nodi rossi.

Ciò significa che in un percorso di lunghezza h posso **solo** alternare nodi rossi e neri e questo ci permette di limitare inferiormente l'altezza nera:

$$\frac{h(x)}{2} \leq bh(x)$$

Sempre per la monotonicità della funzione esponenziale possiamo dire che:

$$2^{\frac{h(x)}{2}} \leq 2^{bh(x)}$$

Sottraggo 1 ambo i membri:

$$2^{\frac{h(x)}{2}} - 1 \leq 2^{bh(x)} - 1$$

Applico la disequazione che abbiamo dimostrato sul numero dei nodi interni

($NI(x) \geq 2^{bh(x)} - 1$) e vado di transitività:

$$\left\{ \begin{array}{l} NI(x) \geq 2^{bh(x)} - 1 \\ 2^{\frac{h(x)}{2}} - 1 \leq 2^{bh(x)} - 1 \end{array} \right. \Rightarrow NI(x) \geq 2^{\frac{h(x)}{2}} - 1$$

Questa proprietà vale per un qualsiasi nodo x di un RBTree ma se x è proprio la radice abbiamo che $h(x)$ è proprio l'altezza dell'albero e $NI(x)$ è proprio il numero di nodi interni complessivi del RBTree, ossia la taglia dell'albero, che possiamo denotare semplicemente con la variabile n , quindi:

- $n \geq 2^{\frac{h}{2}} - 1$
- $n + 1 \geq 2^{\frac{h}{2}}$
- $\log_2(n + 1) \geq \frac{h}{2}$
- $h \leq 2 \cdot \log_2(n + 1)$
- $h = O(\log_2 n)$

Perciò concludiamo che per ogni RBTree la sua altezza è limitata superiormente della taglia dell'albero stesso.

NB. Quel 2 nel penultimo passaggio è generato dal fatto che un RBTree può allungare un albero pieno al massimo del doppio della sua altezza, attraverso i nodi rossi.

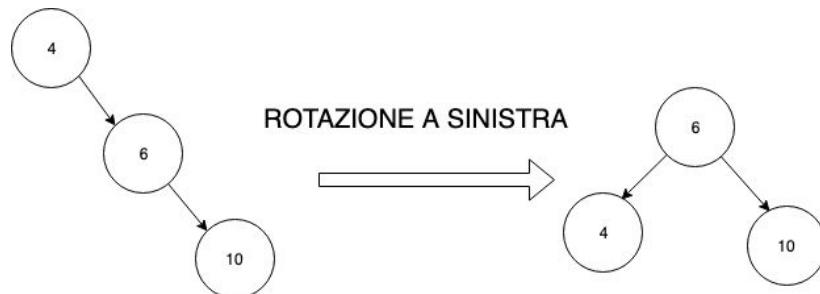
Inserimento in un albero RB

checkpoint: https://youtu.be/R9_m9ZwFJOA?t=2781

L'inserimento di un nuovo nodo in un RB-Tree potrebbe violare i vincoli che definiscono gli RB-Tree stessi. Per risolvere questo problema si utilizzano le rotazioni (che sono identiche a quelle degli alberi AVL):

Le rotazioni permettono di allungare o di accorciare i percorsi.

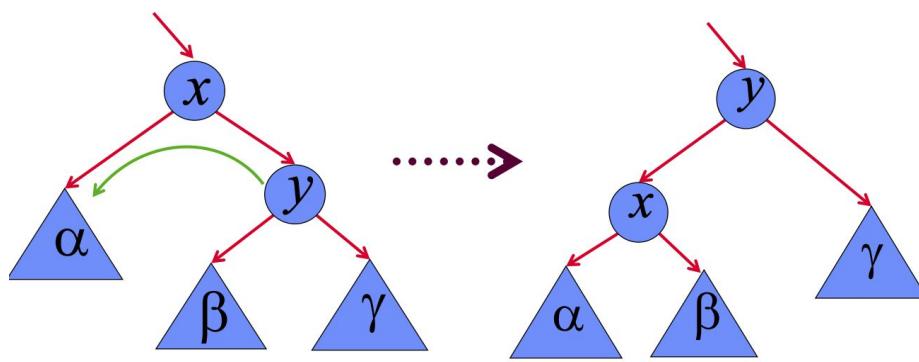
Rotazione a sinistra



RotazioneSx(T)

```
| fs = T.sx  
| T.sx = fs.dx  
| fs.dx = T  
| return fs
```

Rotazione a destra



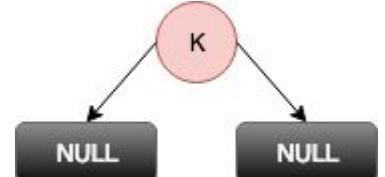
RotazioneDx(T)

```
| fd = T.dx  
| T.dx = fd.sx  
| fd.sx = T  
| return fd
```

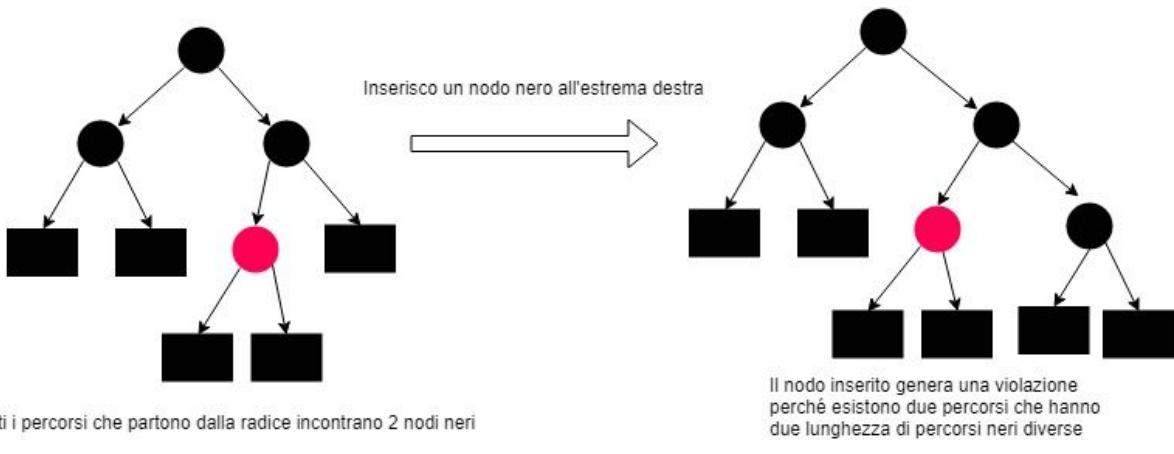
Inserimento

L'idea di un inserimento si basa sulla simulazione della ricerca del nodo, se lo troviamo non facciamo nulla, se non lo troviamo arriviamo ad un nodo null e quindi abbiamo trovato un possibile posto per inserire il nodo.

Per non violare il vincolo che ci dice che le foglie devono essere null sostituiamo il nodo null che abbiamo trovato con un nodo interno (a cui abbiamo assegnato il valore k da inserire) i cui due figli sx e dx sono entrambi null. Inoltre siccome le foglie devono essere sempre nere, questi due figli null saranno nere.



Per quanto riguarda il colore del nodo interno contenente il valore da inserire ci conviene colorarlo di rosso perché se lo colorassimo di nero avremo che il numero di nodi neri dei percorsi fino a quel nodo cambierebbe causando una violazione sicura dell'ultimo vincolo.



Se invece lo coloriamo di rosso il numero di nodi neri dei percorsi fino alla vecchia foglia non cambia e quindi quel vincolo viene preservato.

Quello che però può dare fastidio è il vincolo che dice che due nodi rossi non possono essere adiacenti, e questo dipende da come era formato l'albero prima.

Questo caso sarà gestito della risalita che cambierà i colori per evitare che si rompa tutto.

```
Inserimento(k,T)
| if( ! isNil(T) ) then      //isNil() ritorna true se T è un nodo "sentinella" null.
| | if( k < T.k) then
| | | T.sx = inserimento (k, T.sx)
| | | T = bilancia_sx(T);
| | else
| | | T.dx = inserimento (k, T.dx)
| | | T = bilancia_dx(T);
| | else          //Se il nodo non è presente lo creo assegnandoci il colore rosso.
| | T = allocaNodo(); T.k = K;
| | T.dx = null; T.sx = null;
| | T.color = RED;
return T
```

al termine dell'inserimento l'unica violazione da correggere è quindi quella dei due nodi rossi adiacenti perché è l'unica che può essere generata e questa violazione sarà risolta dagli algoritmi "bilancia".

Inoltre è da specificare due cose:

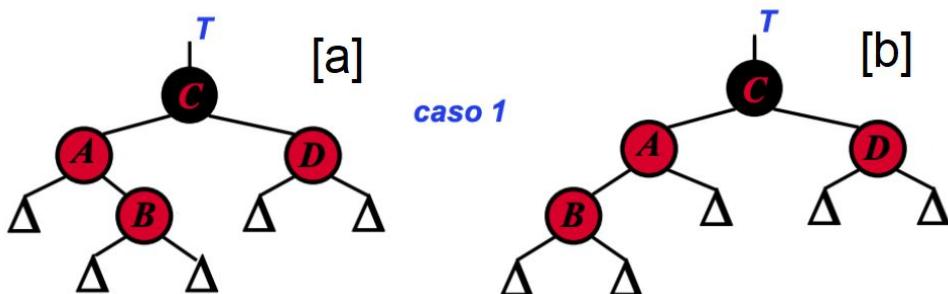
- 1) La radice è sempre e solo nera, infatti non ha senso considerarla rossa perché imporrebbe subito un vincolo sui nodi al primo livello e perché non viene considerata nel calcolo delle altezze nere.
- 2) Gli alberi o i sottoalberi di altezza minore strettamente di 3 non possono verificare violazioni.

La violazione di un doppio nodo rosso è sempre trovata da un nodo nero perché un nodo rosso non può avere altri due figli consecutivi rossi perché in tal caso ci sarebbero due violazioni e noi stiamo assumendo che l'algoritmo di inserimento possa al massimo generare una ed una sola violazione.

Quindi tutti i possibili casi hanno un T nero ed almeno un figlio rosso che ha a sua volta un figlio rosso.

Possiamo distinguere 3 casi:

Caso 1: entrambi i figli di T sono rossi e c'è un doppio rosso a sx

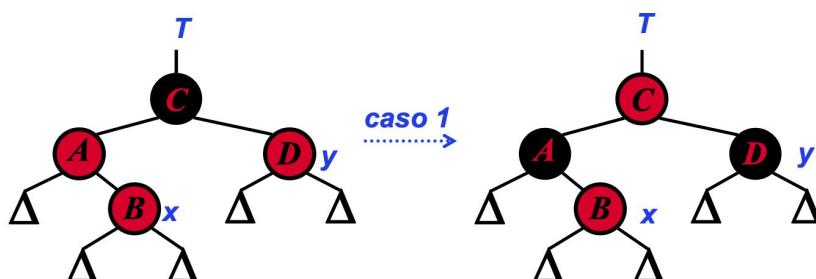


In entrambi i casi il figlio destro di T è rosso e vengono gestiti entrambi alla stessa maniera, ragion per cui li assimiliamo allo stesso caso. Nel caso 1 la violazione viene spostata verso l'alto senza effettuare rotazioni, semplicemente cambiando i colori.

Questo risolve il problema perché spostando la violazione verso l'alto arrivo prima o poi alla radice e dato che posso colorare la radice di nero, è facile ripristinare i vincoli di RBTree.

La risoluzione di questo caso consiste nel far diventare C rosso, e i suoi due figli neri.

La violazione quindi, dopo questa operazione, può avvenire tra C e suo padre, in questo modo la violazione è salita di due livelli.

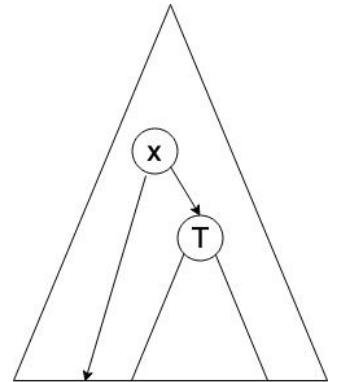


Fix dei percorsi neri

Inoltre, siamo sicuri che tutto ciò non crei problemi con il vincolo della lunghezza dei percorsi neri?

Sicuramente i percorsi che sono “disgiunti” dal sottoalbero radicato in T non ci creano problemi perché non sono stati modificati (e prima erano corretti); [ndr. vedi percorso che va da x alle foglie senza passare per T.]

Se però prendo un nodo x che si trova prima di C nell'albero il quale ha due percorsi uno “disgiunto” dal sottoalbero radicato in T ed uno che finisce proprio nel sottoalbero radicato in T; nel percorso disgiunto il numero di nodi neri non è cambiato e deve essere uguale al numero di nodi neri presente per ogni percorso del sottoalbero radicato in T.



Se questi due valori fossero diversi, avremmo due percorsi che partono dallo stesso vertice con due numeri di nodi neri diversi, ossia una violazione dell'ultimo vincolo.

In altre parole, mi devo preoccupare che ogni percorso che passa per T deve vedere lo stesso numero di nodi neri che vedeva prima.

Dato che nell'operazione tutti nodi diversi da A, C e D non vengono modificati sto sicuro che non danno problemi.

Ragion per cui, gli unici nodi che potrebbero generare problemi con questo vincolo sono proprio A, C e D; però dato che il colore nero scende di un livello tutti i nodi che si trovano al di sotto di A e di D continueranno a vedere lo stesso numero di nodi neri.

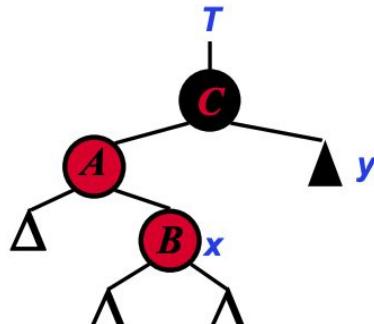
Così concludo che non è possibile violare il vincolo della lunghezza dei percorsi neri perché ho modificato solo il sottoalbero radicato in T e tutti i percorsi che passano in questo sottoalbero vedono lo stesso numero di nodi neri.

```
albero-RB Bilancia_sx_1(T)
T->color = red;
T->dx->color = black;
T->sx->color = black;
return T;
```

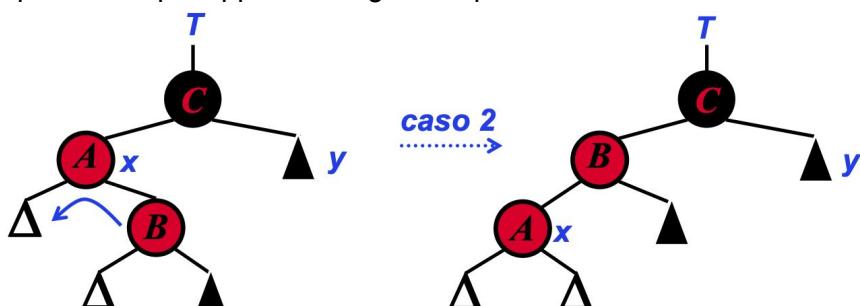
Caso 2: T.dx è nero e c'è un doppio rosso in T.sx.dx

Il caso due avviene quando:

- Il figlio destro di T è nero
- Il doppio rosso si trova tra il figlio sinistro di T e il suo relativo figlio destro, nipote di T.



La risoluzione consiste nell'effettuare una rotazione a destra di B. Questo ci porta a trasformare il nostro albero in un nuovo albero che non viola più il caso 2 ma viola il caso 3. A quel punto potremmo poi applicare l'algoritmo per risolvere il caso 3.



Fix dei percorsi neri

Come prima ci dobbiamo chiedere se siamo sicuri che la rotazione non crei problemi con il vincolo della lunghezza dei percorsi neri.

Dato che la rotazione va a cambiare praticamente solo il sottoalbero che prima era figlio sinistro di B per farlo diventare figlio destro di A possiamo facilmente accorgerci che il numero di nodi neri non viene modificato perché quel sottoalbero prima vedeva due nodi rossi (in ordine dall'alto verso il basso: A e B) e dopo continua a vedere due nodi rossi (in ordine dall'alto verso il basso: B e A).

Ragion per cui, a meno che prima non violasse quella proprietà non è possibile che dopo la rotazione si crei la violazione ma dato che noi assumiamo che durante l'inserimento l'unica violazione che può esistere è quella del doppio rosso siamo sicuri che non ci sono violazioni sul numero di nodi neri.

albero-RB Bilancia_sx_2(T)

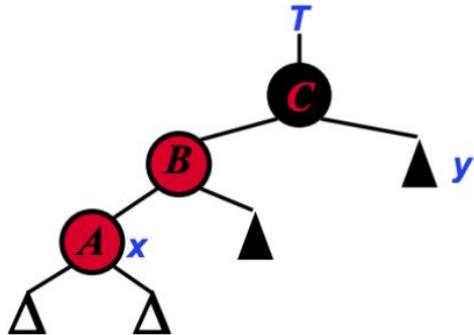
$T \rightarrow \text{sx} = \text{Ruota_dx}(T \rightarrow \text{sx});$

return T;

Caso 3: T.dx è nero e c'è un doppio rosso in T.dx.dx

Il caso due avviene quando:

- Il figlio destro di T è nero
- Il doppio rosso si trova tra il figlio sinistro di T e il suo relativo figlio sinistro, nipote di T.



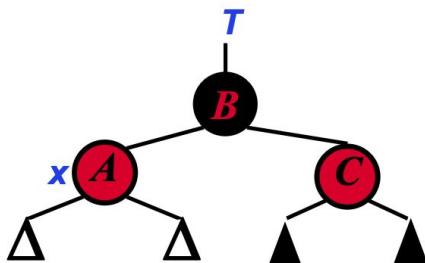
La risoluzione consiste nell'effettuare una rotazione a sinistra + un cambio di colore, vediamola man mano.

La rotazione avviene in questo modo:



Questa rotazione di per sé non preserva il numero di nodi neri perché ad esempio il sottoalbero sinistro di A prima vedeva un nodo nero (C) e dopo non ne vede nessuno.

Per risolvere questo problema dobbiamo scambiare i colori di B e C arrivando a questo albero:



NB. Avremmo potuto anche colorare il nodo A di nero, invece di scambiare i colori, per ristabilire il numero di nodi neri. Però ciò lascia B rosso e avrebbe potuto creare casini nel caso in cui il padre di B fosse rosso.

Fix dei percorsi neri

Ancora una volta ci dobbiamo chiedere se siamo sicuri che l'operazione non crei problemi con il vincolo della lunghezza dei percorsi neri.

Dato che tutti i nodi diversi da A,B e C appartenenti al sottoalbero radicato in C sia prima l'operazione che dopo vedono un unico nodo nero il problema non si pone perché il numero di nodi neri non è cambiato.

Infine verifichiamo che non sono stati create violazione del vincolo del doppio rosso dopo l'operazione:

L'unico nodo che potrebbe creare problemi è quello alla destra di C, perché il colore di C cambia. Dato che questo nodo per ipotesi aveva radice nera, non ci sono problemi se suo padre (ossia C) è rosso.

Perciò sappiamo che non ci sono state nuove violazioni per quanto riguarda il vincolo del doppio rosso.

Il caso 3 risolve il problema immediatamente, a differenza del caso 1.

```
albero-RB Bilancia_sx_3(T)
T = Ruota_sx(T)
T->color = black;
T->dx->color = red;
return T;
```

Bilancia_Sx(T)

L'algoritmo bilancia_sx(T) semplicemente identifica la tipologia di violazione in base ad un'altra funzione, ossia, tipo_violazione_sx(), ed in base a quale violazione è stata trovata applica i ragionamenti visti finora per risolverli.

```
albero-RB Bilancia_sx(T)
IF ha_figlio(T->sx)
  v = Tipo_violazione_sx(T->sx, T->dx)
  /* v = 0 nessuna violazione */
CASE v OF
  1: T = Bilancia_sx_1(T);
  2: T = Bilancia_sx_2(T);
  T = Bilancia_sx_3(T);
  3: T = Bilancia_sx_3(T);
return T;
```

Tipo_violazione_sx(s,d)

L'algoritmo che si occupa di capire in quale caso ci troviamo applica semplicemente il ragionamento sul colore dei figli di T che abbiamo già visto. Gli input di questo algoritmo sono i nodi figli di T.

```
tipo_violazione_sx(s,d)
| violazione = 0
| if(s.color = red && d.color = red) then
| | if(s.sx.color = red || S.dx.color = red) then
| | | violazione = 1;
| else
| | if(s.color = red) then
| | | if(s.dx.color = red) then
| | | | violazione = 2;
| | | else
| | | | if(s.sx.color = red) then
| | | | | violazione = 3;
| return violazione
```

Cancellazione in un albero RB

checkpoint: <https://www.youtube.com/watch?v=RTsTQEgavDk>

La cancellazione, come l'inserimento, si basa sullo stesso principio degli ABR.

Se il nodo che vogliamo cancellare è rosso non è possibile che generi violazioni mentre se il nodo da cancellare è nero sicuramente avremo una violazione sul vincolo dei percorsi neri.

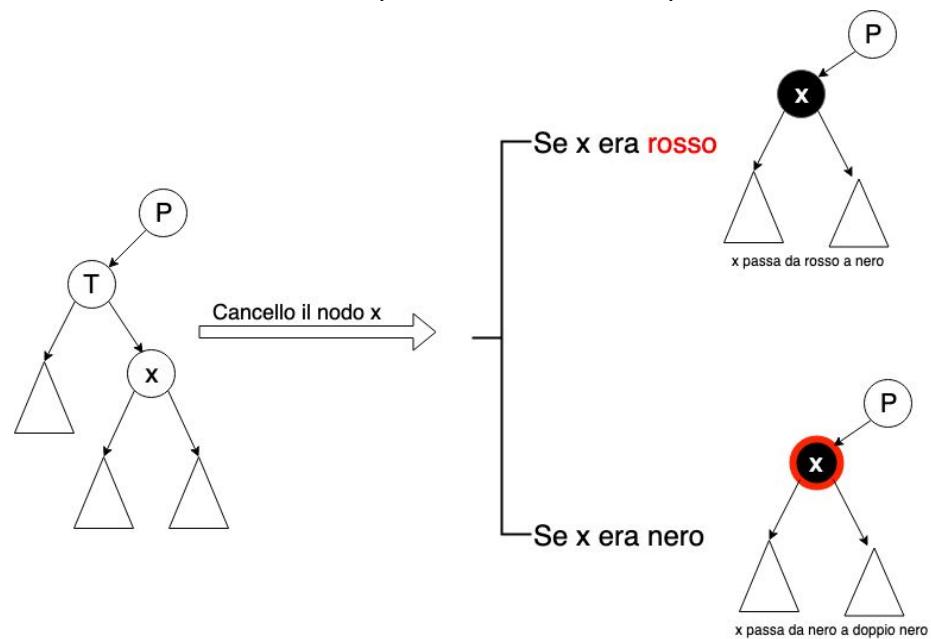
Per questa ragione useremo un escamotage: invece di ammettere una violazione sul vincolo dei percorsi neri creiamo un nuovo colore, **il doppio nero**, che conta il doppio per il calcolo del numero dei nodi neri.

L'algoritmo di cancellazione quindi si occuperà di ridistribuire il nero "di troppo" da qualche parte dell'RBTree con il nodo cancellato.

Dato che noi cancelliamo in base ad una chiave e le chiavi sono contenute solo nei nodi interni effettueremo una cancellazione solo sui nodi interni.

Quindi il nodo che cancelliamo ha sicuramente due figli.

Graficamente avremo quindi una situazione del genere e dovremo occuparci di gestire il colore doppio-nero. Nota che se il nodo che abbiamo inserito al posto di quello cancellato è rosso, non abbiamo nulla da fixare perché coloriamo semplicemente il nodo da rosso a nero.



Essendo derivato della cancellazione dell'ABR, l'algoritmo è molto simile a quello dell'inserimento.

```
Cancellazione(k,T)
| if( ! isNil(T) ) then          //isNil() ritorna true se T è un nodo "sentinella" null.
| | if( k < T.k) then
| | | T.sx = Cancellazione (k, T.sx)
| | | T = cancella_bilancia_sx(T);
| | else if (k > T.k) then
| | | T.dx = Cancellazione (k, T.dx)
| | | T = cancella_bilancia_dx(T);
| | else                      //Se il nodo è proprio contenuto in radice.
| | | T = cancella_radice_RB(T)
return T
```

Cancella_radice_RB(T)

L'algoritmo verifica se la radice ha due figli oppure no.

Nel caso in cui manca di figlio destro o sinistro, salva T in tmp e mette in T il figlio non-null e controlla il colore del vecchio T, ossia tmp.

Se il colore di tmp è nero controlla il valore del figlio non-null, il quale se è rosso diventa nero e se è nero diventa doppio nero.

Se invece la radice ha entrambi i figli, vado a recuperare il nodo più piccolo del sottoalbero destro il quale può avere figlio destro ma sicuramente non può avere un figlio sinistro, perché altrimenti sarebbe lui il nodo più piccolo del sottoalbero destro.

Salvo il nodo più piccolo del sottoalbero destro in tmp, sovrascrivo il valore della radice, chiamo la funzione di bilanciamento destro per verificare se nel nuovo albero c'è una violazione.

Al termine si dealloca tmp, che sia la radice originaria o il nodo preso da stacca-min() e si ritorna il puntatore della nuova radice.

```
Canc-Radice-RB (T)
IF IS-NIL (T->sx) || IS-NIL (T->dx) THEN
    tmp = T
    IF IS-NIL (T->sx) THEN
        T = tmp->dx
    ELSE IF IS-NIL (T->dx) THEN
        T = tmp->sx
    IF tmp->color = black THEN
        Propagate-Black (T)
    ELSE
        tmp = Stacca-Min-RB (T->dx ,T)
        T->Key = tmp->Key
        T = Canc-Bil-dx (T)
        dealloca tmp
RETURN T
```

```
Propagate-Black (T)
IF T->color = red THEN
    T->color = black
ELSE
    T->color = d-black
```

Stacca-min-RB()

Il concetto che applica è molto simile allo StaccaMin() per gli alberi ABR.

Finché può andare a sinistra, va a sinistra ricorsivamente; quando termina la chiamata ricorsiva tmp conterrà il nodo da staccare.

E' possibile che il T corrente sia in stato di violazione ragion per cui controlliamo se T è un figlio sx o dx di suo padre e chiamiamo gli algoritmi di bilanciamento in base a ciò.

Se non può andare a sinistra, abbiamo trovato il nodo da staccare e facciamo le operazioni per collegare il padre al figlio destro del nodo che vogliamo staccare.

Infine dato che il nodo T viene staccato, se il suo colore è nero, dev'essere propagato (attraverso la stessa funzione di prima) al suo figlio destro.

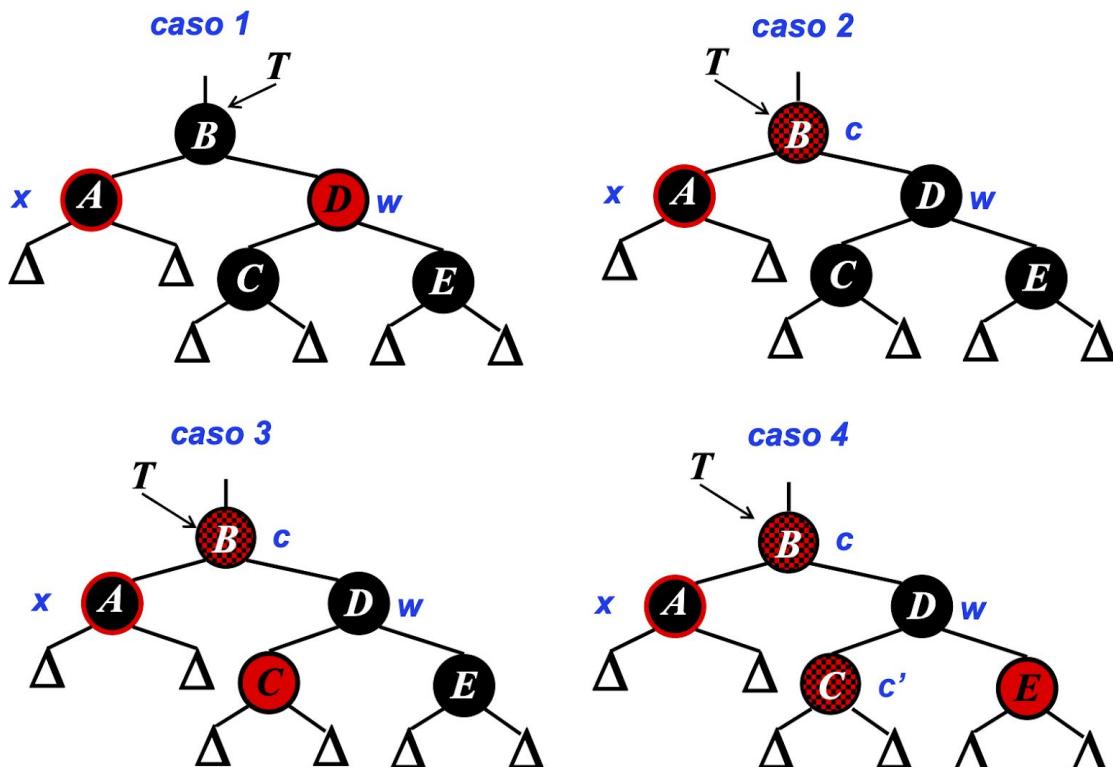
```

Stacca-Min-RB(T, P)
  IF NOT IS-NIL(T) THEN
    IF NOT IS-NIL(T->sx) THEN
      tmp = Stacca-Min-RB(T->sx, T)
      IF T = P->sx THEN
        P->sx = Canc-Bil-sx(T)
      ELSE
        P->dx = Canc-Bil-sx(T)
      T = tmp
    ELSE
      IF T = P->sx THEN
        P->sx = T->dx
      ELSE
        P->dx = T->dx
      IF T->color = black
        Propagate-Black(T->dx)
  return T

```

Cancella_Bilancia_sx()

L'algoritmo di bilanciamento deve distinguere 4 casi:

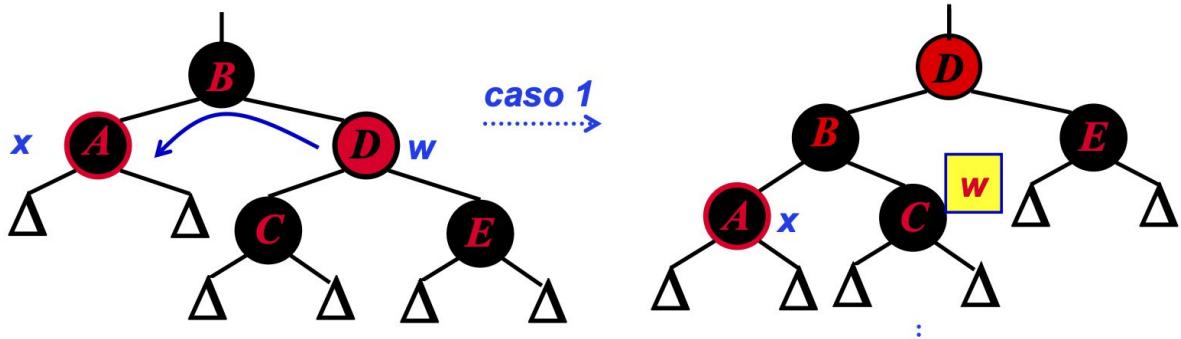


Legenda:

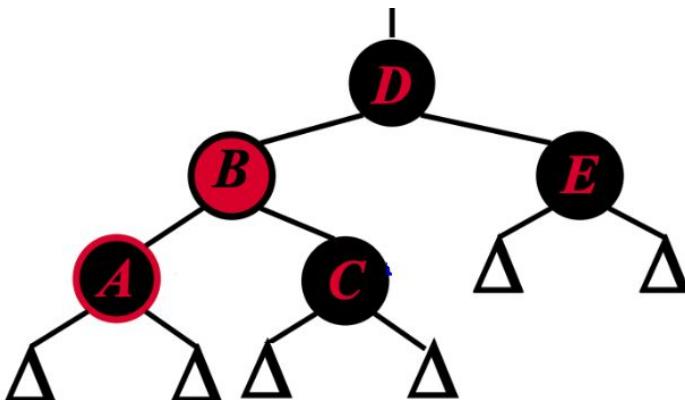
- 1) il rosso cerchiato indica il doppio-nero
- 2) il colore a scacchi indica che il colore non è influente, può essere sia rosso che nero.

Caso 1: radice, C,E neri e D rosso

Questo caso viene risolto attraverso una rotazione a destra di D, portandosi nel seguente albero:



Questo è problematico perché i sottoalberi radicati in E dopo la rotazione vedono un nero in meno. Perciò possiamo scambiare i colori di B e di D portandoci in questa situazione:



Così facendo la violazione è scesa di un livello rispetto al livello originario ed è ancora insolita. Quello che possiamo fare è considerare adesso l'albero radicato in B. Esso ha ancora una violazione sul suo figlio sinistro, ma il suo colore adesso è rosso e dato che C è nero significa che ci troviamo esattamente nel caso 2,3 o 4 (in dipendenza dei colori dei figli di C).

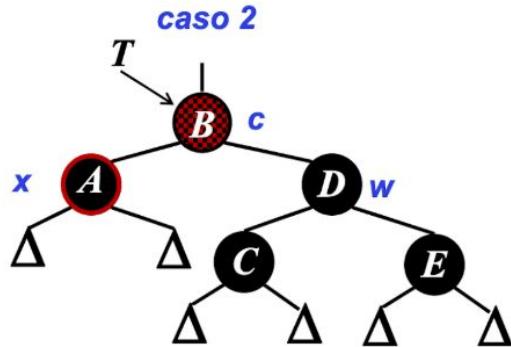
Ciò significa che non ci troviamo, ovviamente, di nuovo nel caso 1 perché l'albero che contiene la violazione ha la radice rossa (B). Quindi questo ci preclude da errori di loop di casi 1 innestati fra loro, in quanto il caso 1 può verificarsi una ed una sola volta.

Bisogna specificare che se il nuovo caso è un caso 2, ci troviamo nella condizione in cui B è rosso perché solo se B è rosso il caso 2 è risolutivo, se così non fosse il caso 2 porterebbe la violazione ad un livello sopra e questo genererebbe loop con il caso 1 che la porta sotto. Però ciò non accade proprio perché B è rosso.

Inoltre se il nuovo caso è un caso 4 è risolutivo, mentre se è un caso 3 esso si porta in un caso 4 ed è risolutivo (per dettagli vedi paragrafi dedicati).

Quindi spostandoci dal caso 1 al caso 2/3/4 siamo sicuri che la violazione si risolve.

Caso 2: D, C, E neri



Questo caso è simile al caso 1 dell'inserimento e la violazione viene spostata verso l'alto senza effettuare rotazioni, semplicemente cambiando i colori, spostando il nero di troppo che si trova in A su B.

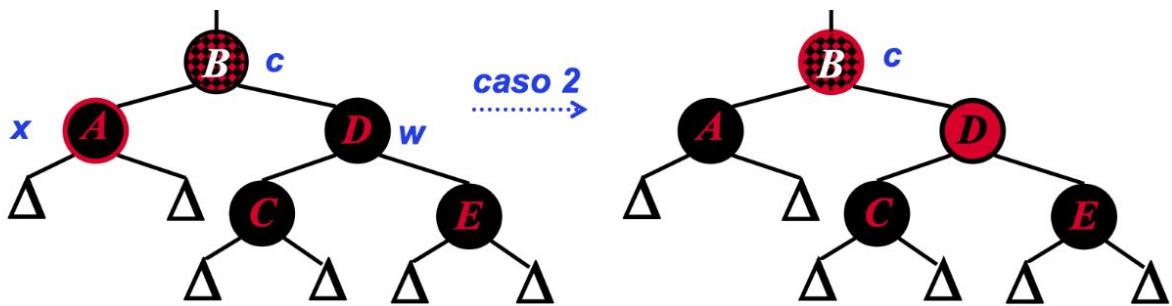
Adesso:

- 1) **Se B è rosso**, siamo fortunati e B viene colorato di nero e **la violazione è risolta**
- 2) **Se B è nero**, siamo sfortunati e B viene colorato di doppio nero e **la violazione resta** e verrà spostata di un livello sopra.

Fix dei percorsi neri

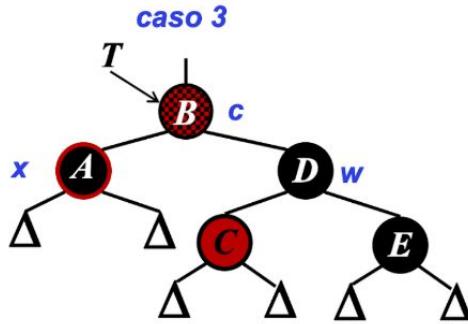
Per quanto riguarda il vincolo dei percorsi neri, il percorso che viene da A non subisce modifiche sulla quantità di neri che incontra ma i percorsi che si trovano nel sottoalbero destro di B adesso vedono un nero in più.

Per risolvere questo problema coloriamo D di rosso, ciò è possibile perché C e E sono già neri e quindi il vincolo che impone che i figli di un rosso devono essere necessariamente entrambi neri non ci crea fastidio.



Caso 3: D, E neri e C rosso

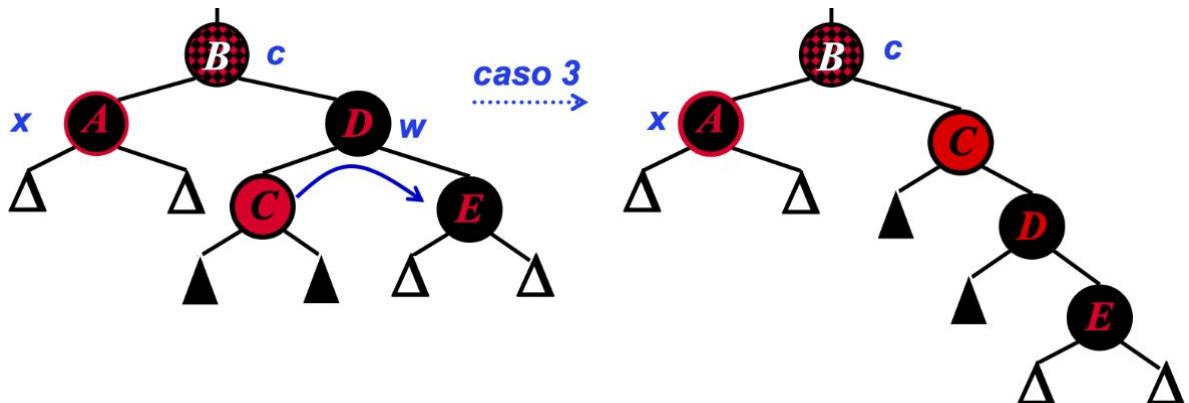
checkpoint:
<https://youtu.be/RTsTQEgavDk?t=2259>



Il caso 3 si riconosce dai nodi D ed E neri e C rosso; il colore di B non è importante mentre A ovviamente ha la violazione del doppio colore.

La risoluzione si basa, come nel caso 2 dell'inserimento, nel trasformare questo caso nel caso 4 (distinto dal fatto che ha D nero ed E rosso) il quale è subito risolutivo.

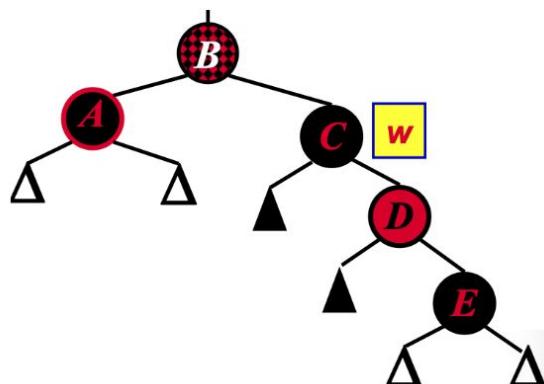
Per fare ciò applica una rotazione a sinistra da C, nel seguente modo:



Fix dei percorsi neri

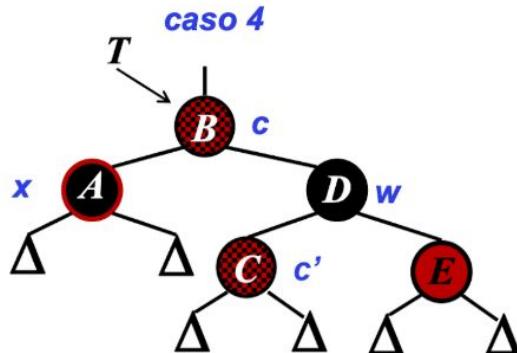
Come al solito, effettuando la rotazione i nodi discendenti a sinistra di C vedono un nero in meno nel loro percorso, violando il vincolo dell'arbitro.

Per questo motivo scambiamo i colori di C e di D arrivando a quest'albero:



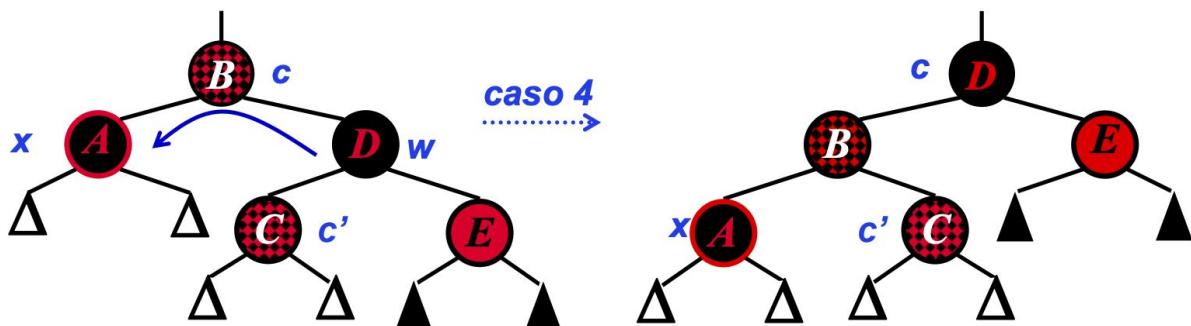
La violazione si trova esattamente allo stesso posto ma ora il sottoalbero radicato in B si trova nel caso 4.

Caso 4: D nero ed E rosso



Per risolvere il caso 4 effettuo una rotazione a destra; dopo ciò notiamo due cose:

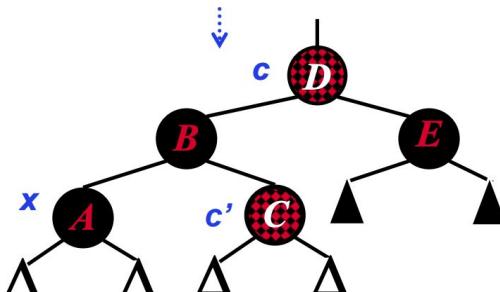
- 1) Il doppio nero esiste ancora
- 2) Certi sottoalberi vedono più neri rispetto a quelli che vedevano prima, come a esempio i due sottoalberi di A che passano da 2 a 3 neri (2 di A + 1 di D)



Fix dei percorsi neri

Per risolvere l'ultimo caso, è sufficiente "ruotare" i colori da sinistra verso destra ossia

- spostare il colore nero di D su E
- spostare il colore "neutro" (ossia indifferentemente se nero o rosso) di B su D
- spostare il colore doppio nero di A viene diviso con B, facendolo diventare nero



In questo modo qualsiasi nodo che si trovava nei sottoalberi vede esattamente gli stessi neri che vedeva prima della rotazione.

Riassunto

Caso 1	Rotazione a destra + chiamata ricorsiva per portarsi ai casi 2/3/4
Caso 2	Propagazione del nero verso l'alto, se B è rosso risolve subito
Caso 3	Rotazione sinistra, genera il caso 4
Caso 4	Rotazione destra, scambio di colori e risolve.

Una volta stabilito cosa fare per tutti e 4 casi l'algoritmo di bilanciamento sx è in grado di richiamare le sotto funzioni in base al caso trovato da `violazione_sx()`.

```
Cancella_Bil_sx(T)
  IF ha_figli(T)
    v = Violazione_sx(T->sx, T->dx)
    /* nessuna violazione se v = 0 */
    CASE v OF
      1: T = Canc_bil_1_sx(T);
          T->sx = Cancella_Bil_sx(T->sx);
      2: T = Canc_bil_2_sx(T);
      3: T = Canc_bil_3_sx(T);
      4: T = Canc_bil_4_sx(T);
  return T;
```

Violazione-sx()

checkpoint:

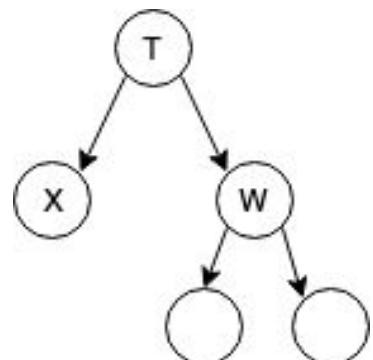
<https://youtu.be/RTsTQEgavDk?t=2961>

Violazione-sx() prende in input il figlio sinistro(X) e destro (W) del nodo che vogliamo controllare se ha una violazione a sinistra.

Per averla deve avere necessariamente il colore di X doppio nero, se ciò è vero controlla i colori di W. Se W è rosso, stiamo nel caso 1; se W è nero possiamo essere nei casi 2/3/4 i quali vengono discriminati dal colore dei suoi due figli:

Se sono tutte e due neri, ci troviamo nel caso 2, se il destro è nero e il sinistro è rosso stiamo nel caso 3, infine se il destro è rosso è il caso 4.

```
Violazione-sx(X,W)
viola = 0
IF X->color = d-black THEN
  IF W->color=red THEN
    viola = 1
  ELSE IF W->dx = black &
          W->sx = black THEN
    viola = 2
  ELSE IF W->dx = black THEN
    viola = 3
  ELSE /* W->dx = red */
    viola = 4
return viola
```



Implementazione dei casi

```
Canc-Bil-1-sx(T)
```

```
T = ruota-dx(T)
T->color = black
T->sx->color = red
return T
```

```
Canc-Bil-2-sx(T)
```

```
T->dx->color = red
T->sx->color = black
propagate-black(T)
return T
```

```
Canc-Bil-3-sx(T)
```

```
T->dx = ruota-sx(T->dx)
T->dx->color = black
T->dx->dx->color = red
T = Canc-Bil-4-sx(T)
return T
```

```
Canc-Bil-4-sx(T)
```

```
T = ruota-sx(T)
T->dx->color = T->color
T->color = T->sx->color
T->sx->color = black
T->sx->sx->color = black
return T
```

Grafi di Ciro a Mergellin

AUTORE: Denny Acciaro
[acciariogennaro@gmail.com]

Fonti

- <https://youtu.be/RTsTQEgavDk?t=3946> (Lezione 28) [Definizioni]
- <https://www.youtube.com/watch?v=dW6HNfl5VgM> (Lezione 29) [Definizioni]
- <https://www.youtube.com/watch?v=iY9DkVRMJJ8> (Lezione 30) [BFS]
- <https://www.youtube.com/watch?v=xs3XPSBkwkk> (Lezione 31) [BFS]
- <https://www.youtube.com/watch?v=VQqaycrT8vM> (Lezione 32) [DFS]
- <https://www.youtube.com/watch?v=bz1gcLoD-c> (Lezione 33) [Parentesi]
- <https://www.youtube.com/watch?v=JEf9ok2qhWE> (Lezione 34) [Aciclicità]
- <https://www.youtube.com/watch?v=pV2MDZdy7xQ> (Lezione 35) [Ord.Topologico]
- <https://www.youtube.com/watch?v=S8OQDsU0Ck> (Lezione 36) [CFC]
- <https://core.ac.uk/download/pdf/20366402.pdf> (Tesi di una tizia a caso)

Introduzione	3
Definizioni	3
Reach	5
Come rappresentare un grafo	8
Matrice di adiacenza	8
Operazioni sul grafo	8
Liste di adiacenza	9
Operazioni sul grafo	9
Algoritmi di raggiungibilità	10
Algoritmo BFS (Aampiezza)	10
Array dei colori	10
Array delle distanze (o delle stime)	11
Array dei predecessori	11
Codice di BFS	13
Usare BFS() per calcolare il percorso minimo tra due nodi	15
Complessità	15
Dimostrazione del funzionamento	16
Le tre “sotto proprietà”	16
Dimostrazione del calcolo corretto delle distanze e del predecessore	19
Algoritmo DFS (Profondità)	22
Codice	22
Array dei predecessori in DFS VS array dei predecessori in BFS	24

Teorema della struttura a parentesi dei tempi	25
Dimostrazione	27
Caso $d[v] < d[u] < f[u] < f[v]$	27
Caso $d[v] < f[v] < d[u] < f[u]$	28
Proprietà: $d[v] < d[u] < f[u] < f[v] \Leftrightarrow u \text{ è discendente di } v \text{ nella foresta}$	28
Teorema del percorso bianco	30
Dimostrazione	30
Archi di ritorno, archi in avanti e archi di attraversamento	32
Come capire se un grafo è aciclico	33
Dimostrazione della relazione di uguaglianza tra ciclo e arco di ritorno	33
Aciclico(G)	35
Ordinamento topologico	36
Come costruire ordinamento topologico	37
Se G è aciclico \rightarrow Esiste un vertice che non ha archi entranti	37
Se G è aciclico $\rightarrow G'$ sottografo di G è aciclico	37
Verifica della correttezza	37
GradoEntrante()	38
Init_Queue()	39
OrdinamentoTopologico()	39
OrdinamentoTopologicoV2()	41
Verifica della correttezza	42
Componenti Fortemente Connesse	43
Proprietà dei “percorsi inclusi in una CFC”	44
Proprietà del “due vertici della stessa CFC non possono essere contenuti in due alberi diversi”	45
Calcolo delle CFC	45
GrafoTransposto	46
[Extra] Riassunto degli algoritmi (utili per lo scritto)	49
[Extra] Algoritmi trovati online che potrebbero essere utili	53
Visita di tutti i percorsi da un vertice u ad un altro vertice v usando la DFS	53

Introduzione

Un **grafo** è una coppia di insiemi $\langle V, E \rangle$ in cui $E \subseteq V \times V$.

In teoria dei grafi:

- **V** sono detti **vertici o nodi**
- **E** sono detti **archi**

Una coppia di elementi (a, b) si dice **ordinata** se e solo se “l’ordine in cui compaiono gli elementi è significativo”, ossia se $(a, b) \neq (b, a)$.

I grafi, perciò, si dividono in:

- 1) **Grafi orientati** in cui le coppie di E sono ordinate.
- 2) **Grafi non orientati** in cui le coppie di E non sono ordinate.

Se la relazione sui dati non è simmetrica è necessario utilizzare un grafo non orientato.

Un grafo viene utilizzato per risolvere problemi di raggiungibilità e si differenzia dagli ABR i quali risolvono problemi di ricerca.

Definizioni

- La **taglia** di un grafo è la somma delle cardinalità di V e di E .

$$|G| = |V| + |E|$$

in cui il numero di archi è compreso tra 0 (se non sono presenti) e V^2 (se ogni nodo è connesso ad ogni altro nodo e se stesso)

$$0 \leq |E| \leq |V^2|$$

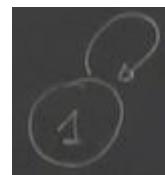
- Un nodo j è detto **adiacente** ad un nodo i se e solo se esiste un arco tra i e j .
- Un **percorso** di un grafo è una sequenza di k vertici del grafo $v_1 \dots v_k$ che soddisfa ciò:

$$\left\{ \begin{array}{ll} (a) & \forall 1 \leq i \leq k-1 \quad (v_i, v_{i+1}) \in E \\ (b) & \forall 1 \leq i \leq k \quad v_i \in V \end{array} \right.$$

La definizione (a) significa che ogni coppia di elementi adiacenti nella sequenza $v_1 \dots v_k$ sono nodi adiacenti, quindi, tutti i nodi presenti in un percorso sono connessi da degli archi a coppie di due, invece, la definizione (b) significa che tutti i vertici appartengono all’insieme V dello stesso grafo.

In un percorso di k elementi ci sono $k-1$ archi e questa rappresenta la lunghezza del percorso.

- Un grafo può permettere **percorsi infiniti**:



- **Percorso semplice**

Un percorso di G è detto semplice se e solo se non contiene vertici duplicati.

Per questo la sua lunghezza massima è $|V|$, perché al massimo contiene tutti e soli i vertici di quel grafo.

Dato che la sua lunghezza massima è un valore finito (e la lunghezza minima è finita $[0]$) l'insieme dei percorsi semplici è un insieme finito.

- **Ciclo**

Un ciclo è un percorso in cui ci sono almeno due vertici che coincidono.

- **Ciclo semplice**

Un ciclo si dice semplice se e solo se

- 1) v_1 e v_k coincidono
- 2) $v_2 \dots v_{k-1}$ è un percorso semplice

Ogni sequenza che contiene un numero maggiore di $|V|$ ha almeno un **ciclo semplice**

- Dimostrazione

Se il percorso p ha una lunghezza n che è maggiore di $|V|$ dovrà necessariamente avere almeno una ripetizione tra due nodi che chiameremo in modo generico v_i e v_j , quindi:

$$\begin{aligned} n &> |V| \\ p &= \langle v_1, v_2, \dots, v_i \dots v_j \dots v_n \rangle \\ v_i &= v_j \end{aligned}$$

In generale possono esistere molte ripetizioni in un percorso quindi considero il sottopercorso che parte da v_i e che arriva in v_j nel quale non sono presenti ripetizioni ad esclusione di v_i e v_j .

La lunghezza massima di questo sottopercorso è $|V|+1$ ed è un percorso semplice se escluso il primo v_i .

Quindi nel percorso p che non era un percorso semplice è contenuto un ciclo semplice e di conseguenza anche un percorso semplice.

- Quando un grafo non ha cicli semplici è detto **aciclico**

- **Sottografo**

Dato un grafo $G = \langle V, E \rangle$ diremo che un altro grafo $G' = \langle V', E' \rangle$ è sottografo di G se valgono

- 1) L'insiemi dei vertici di G' dev'essere completamente contenuto nell'insieme dei vertici di G , ossia tutti i vertici V' si devono trovare anche in V ($V' \subseteq V$)
- 2) L'insieme degli archi di G' dev'essere anch'esso completamente contenuto nell'insieme degli archi di G , ma dato che G' è un grafo sappiamo anche che E' dev'essere parte del prodotto cartesiano $V' \times V'$, quindi devono valere entrambe le condizioni e otteniamo che:

$$\left\{ \begin{array}{l} E' \subseteq E \\ E' \subseteq V' \times V' \rightarrow E' \subseteq E \cap (V' \times V') \end{array} \right.$$

Quindi E' è un sottoinsieme di E che contiene solo i vertici appartenenti a V' .

- **Sottografo massimale**

Un sottografo è detto massimale se contiene tutti i possibili archi che può contenere, ossia valgono queste proprietà:

$$\begin{cases} V' \subseteq V \\ E' = E \cap (V' \times V') \end{cases}$$

- **Reach**

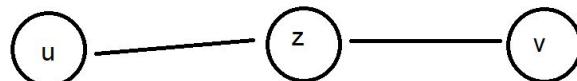
Dato un grafo $G = \langle V, E \rangle$ possiamo definire la relazione di raggiungibilità **Reach** come una relazione inclusa in $V \times V$ e ciò significa che è una relazione tra vertici. In particolare diremo che due vertici u, v sono in relazione reach, quindi sono raggiungibili, se e solo se esiste un percorso π nel grafo G in cui il primo elemento del percorso è u e l'ultimo elemento del percorso è v .

$$\begin{aligned} \forall u, v \in V \quad (u, v) \in \text{Reach} \Leftrightarrow \\ \exists \pi(\pi \text{ è percorso in } G \text{ AND } \text{first}(\pi) = u \text{ AND } \text{last}(\pi) = v) \end{aligned}$$

Questa relazione è una generalizzazione dell'insieme E stesso perché se tra due vertici passa un arco ne consegue che esiste almeno un percorso di lunghezza 1 che collega i due vertici.

Il contrario però non vale perché se la lunghezza del percorso è maggiore o uguale a due significa che può esserci un nodo "in mezzo" tra u e v nel percorso.

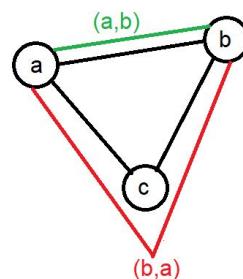
Ad esempio:



$$(u, v) \in \text{Reach} \text{ AND } (u, v) \notin E$$

Nota bene:

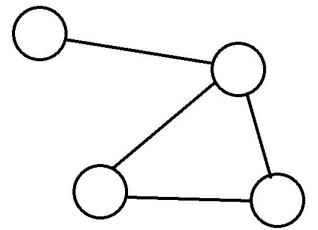
- 1) In Reach vale la proprietà riflessiva: $(v, v) \in \text{Reach}$
- 2) Non è necessario che una relazione sia in E per essere in Reach , ad esempio: $(v, v) \notin E \text{ AND } (v, v) \in \text{Reach}$
- 3) In Reach vale la proprietà transitiva
 $(v, u) \in \text{Reach} \text{ AND } (u, z) \in \text{Reach} \Rightarrow (v, z) \in \text{Reach}$
- 4) Se E è simmetrico anche Reach è simmetrica (e Reach è una relazione di equivalenza) ma se E non è simmetrico non implica che Reach non lo sia, ma potrebbe esserlo come in questo esempio:



- **Grafo connesso**

Un grafo si dice connesso se **non è orientato** e vale la seguente proprietà:

$$\forall (v, u) \in V \quad (u, v) \in \text{Reach}$$

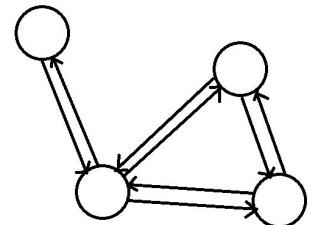


Questa proprietà ci dice che da ogni nodo è possibile raggiungere ogni nodo del grafo e siccome sono soddisfatte le condizioni per la relazione di equivalenza implica che tutti i nodi appartengono alla stessa grande classe di equivalenza.

- **Grafo fortemente connesso**

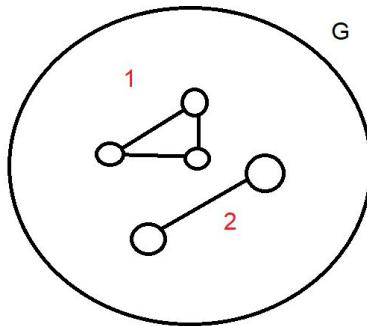
Invece, un grafo si dice fortemente connesso se **è orientato** e vale la seguente proprietà:

$$\forall (v, u) \in V \times V \quad (u, v) \text{ AND } (v, u) \in \text{Reach}$$



- **Componenti connesse**

Le componenti connesse di un grafo non orientato sono i sottografi di G massimali che sono anche grafi connessi, dove per "massimali" si intende che non possono essere estesi preservandone la proprietà di essere connessi, nell'esempio sotto ci sono 2 componenti connesse.



Le componenti connesse rappresentano anche tutte e sole le classi di equivalenza della raggiungibilità presenti in un grafo perché le classi di equivalenza sono massimali per definizione.

In un grafo orientato parleremo, equivalentemente, di componenti fortemente connesse.

- **Mutua Raggiungibilità**

La proprietà di Mutua Raggiungibilità è la seguente:

$$M\text{Reach} : \forall (v, u) \in M\text{Reach} \Leftrightarrow (u, v) \in \text{Reach} \text{ AND } (v, u) \in \text{Reach}$$

In generale $M\text{Reach}$ può contenere meno coppie di Reach e la differenza con quest'ultimo sta nel fatto che Reach contiene tutte le coppie che sono raggiungibili almeno in un verso, mentre $M\text{Reach}$ richiede che le sue coppie siano raggiungibili in tutti e due i versi.

In un grafo non orientato non c'è differenza tra $M\text{Reach}$ e Reach mentre in uno orientato vale: $M\text{Reach} \subseteq \text{Reach}$.

- **Equivalenza della raggiungibilità**

Se n nodi si trovano nella stessa classe di equivalenza posso rappresentare l'intera classe con un unico rappresentante perché (essendo appunto nella stessa relazione di equivalenza) se riesco a raggiungere un nodo di quella classe di equivalenza riesco ad accedere a tutti i nodi della sua classe.

Con questa logica è possibile “comprimere” graficamente i grafi che preserva le relazioni di raggiungibilità.

- **Distanza tra due vertici**

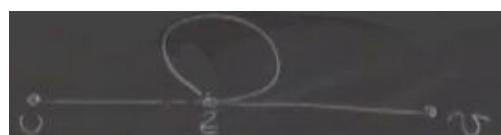
La distanza tra due vertici in un grafo è definita come il **numero minimo di archi** che bisogna attraversare per arrivare da un vertice all'altro. Il percorso con il numero minimo di archi non è per forza unico, ma se non è unico tutti questi percorsi minimi hanno necessariamente la stessa lunghezza.



Se due nodi non sono raggiungibili hanno distanza pari ad ∞ e una distanza tra un nodo e se stesso è pari a 0.

- **Eliminazione occorrenze multiple in un percorso non semplice**

Sia p un percorso non semplice, quindi contenente più occorrenze di uno stesso nodo:



Questa rappresentazione in realtà rappresenta infiniti percorsi del tipo: $\langle u, z, v \rangle$, $\langle u, z, z, v \rangle$, ..., $\langle u, z, \dots, z, v \rangle$ ossia posso “rimanere” nel loop su z da 0 ad infinite volte. Se il loop su z lo eseguo 0 volte esiste una sola occorrenza di del nodo z e quindi sono riuscito a creare un percorso semplice da uno non semplice.

In generale, quindi, per valutare la raggiungibilità tra due nodi ci basta limitarci **solo esclusivamente ai percorsi semplici**, se infatti non esistesse un percorso semplice che collega due nodi sicuramente non può esistere un percorso non semplice che lo faccia.

- Altre definizioni le trovi qui: https://it.wikipedia.org/wiki/Glossario_di_teoria_dei_grafi

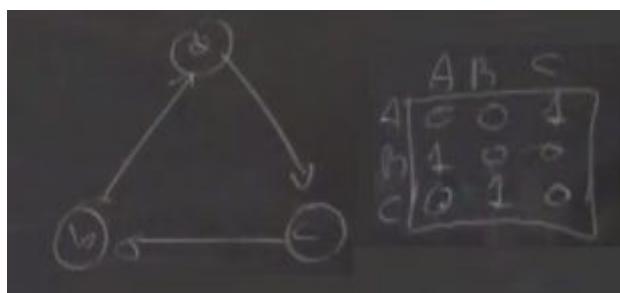
Come rappresentare un grafo

Matrice di adiacenza

Dato che E è per definizione un insieme di coppie, perché è parte del prodotto cartesiano di V con se stesso, potremmo pensare di rappresentare un grafo attraverso una matrice quadrata in cui le righe e le colonne rappresentano i vertici e ogni cella ha un valore booleano che rappresenta un arco: se è impostato a TRUE la connessione tra i due archi esiste, altrimenti no.

In questo modo la matrice implementa tutti i valori di E .

Ad esempio:



Dato che la matrice necessita di $|V|$ righe e $|V|$ colonne occuperà uno spazio di $|V|^2$, ossia è del tutto indipendente dal numero di archi.

In realtà può essere una rappresentazione utile se il numero degli archi è vicino al numero di vertici al quadrato:

$$|E| \cong |V^2|$$

Quindi, di solito, lo spazio occupato della matrice è molto più grande dell'effettiva taglia del grafo.

$$|V^2| >>> |V| + |E|$$

Il problema di ciò nasce dal fatto che conserva informazione sia per gli archi effettivi (quando la cella è true) sia per quelli che non esistono (quando la cella è false) e quindi conserva informazioni superflue.

Operazioni sul grafo

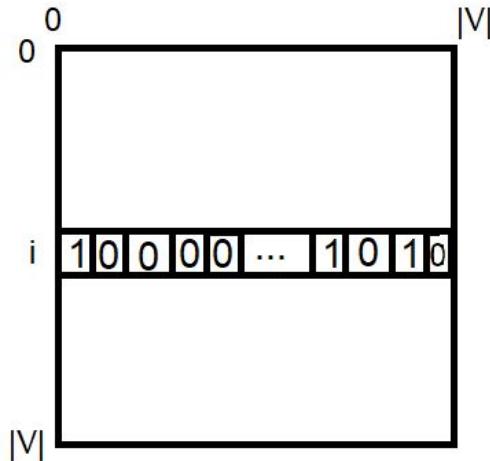
- **Archi**

Le operazioni di inserimento, cancellazione e check su un arco sono a **costo costante** perché è un semplice accesso in una matrice indicizzata.
Questo è un vantaggio per questa rappresentazione.

- **Vertici**

L'inserimento di un nuovo vertice è un po' complesso perché bisogna creare una nuova matrice con una dimensione pari a quella precedente + 1 e poi copiare TUTTA la vecchia matrice nella nuova che si effettua in un tempo quadratico.
Questo è un forte svantaggio per questa rappresentazione.

Inoltre per scorrere TUTTI gli archi uscenti è necessario un tempo di $\Theta(|V|)$ perché è necessario leggere tutta la riga rispettiva:



Liste di adiacenza

Le liste di adiacenza usano un array monodimensionale di lunghezza $|V|$ che contiene ogni vertice e per ogni cella dell'array è associata una linked list di nodi adiacenti a quel nodo.

Il costo per memorizzare un grafo attraverso una lista di adiacenza è pari ad:

$$a \cdot |V| + b \cdot |E|$$

dove:

- a è il costo della singola cella dell'array
- b è il costo di un nodo nella linked list

Quindi questo è assimilabile ad:

$$\Theta(|V| + |E|) \rightarrow \Theta(|G|)$$

quindi è lineare sulla taglia del grafo e quindi è un implementazione ottimale.

Operazioni sul grafo

1) Costo di controllo su un arco

Per vedere se un arco esiste tra due nodi si ha un tempo $\Theta(|V|)$ perché al massimo bisogna scorrere tutta la linked list del nodo di partenza.

2) Inserimento vertici

L'inserimento di un nuovo vertice su un array monodimensionale implica la creazione di un nuovo array di dimensioni uguali al precedente +1 e la successiva copia di tutti i dati dall'array precedente a quello nuovo con un conseguente tempo di $\Theta(|V|+1|)$.

3) Esplorazione degli archi uscenti

Qui c'è un'importante differenza tra le liste e le matrici.

Per esplorare gli archi uscenti nelle matrice bisogna effettuare un check su TUTTI gli elementi della riga, invece in una lista di adiacenza l'operazione richiede solo la visita sulla linked list collegata al nodo. Quindi gli archi che non esistono nel grafo non vengono proprio considerati, quindi l'operazione ha un costo di

$$\Theta(|archi\ uscenti|)$$

il che è un risultato ottimale.

Algoritmi di raggiungibilità

Gli algoritmi di raggiungibilità di un grafo sono simili agli algoritmi di visita degli alberi ma generalizzati, pertanto esiste sia la visita in ampiezza che per profondità.

Algoritmo BFS (Ampiezza)

La visita dei grafi ha un problema rispetto alla visita degli alberi, infatti, dato che in un grafo è possibile che ci sia un ciclo dobbiamo impedire agli algoritmi di visita di entrare in "loop" per garantire la terminazione dell'algoritmo.

Per visitare un grafo in ampiezza useremo una logica che descrive ogni nodo del grafo con 3 stati:

- 1) Nodo non visitato
- 2) Nodo in frontiera
- 3) Nodo visitato



L'idea è quella di "spingere" la frontiera verso i nodi non visitati, quindi lo stato di un vertice può seguire solo questo schema:

Non Visitato → Frontiera → Visitato

Array dei colori

Per raggiungere questo scopo definiamo la funzione c che associa ad ogni vertice uno dei possibili valori di stato:

$$c : V \mapsto \{\text{non-visitato}, \text{frontiera}, \text{visitato}\}$$

Questo valore di stato può essere codificato attraverso tre colori:

Valore di stato	Colore	Sigla colore
Non Visitato	Bianco	b
Frontiera	Grigio	g
Visitato	Nero	n

Quindi la funzione c sarà così definita:

$$c : V \mapsto \{b, g, n\}$$

Dato che associamo ad ogni nodo del grafo (che sono in numero finito) un valore, ossia il colore, possiamo descrivere questa funzione attraverso un array che chiameremo "array dei colori" e che terrà traccia della visita effettuata o meno per ogni nodo durante l'algoritmo di visita.

Per quanto riguarda la frontiera, essa verrà implementata attraverso una coda di tipo FIFO e tiene traccia dei nodi che sono attualmente in visita.

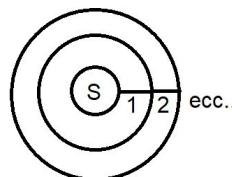
Quando un nodo entra in frontiera gli viene assegnato il colore grigio, riservato per i nodi che sono in coda, e quando esce della frontiera gli viene assegnato il colore nero perché significa che è stato visitato.

Array delle distanze (o delle stime)

Checkpoint: <https://youtu.be/xs3XPSBkwkk?t=1722>

L'algoritmo implementa la visita in ampiezza, quindi partendo da un vertice S inizierà ad esplorare tutti i nodi che si trovano a distanza uno da S (ossia, tutti i nodi in cui il percorso minimo da quel nodo ad S ha distanza 1), poi tutti quelli a distanza 2 da S ecc..

In questo modo tutti i nodi del grafo con distanza finita (ossia raggiungibili da S) alla fine verranno visitati:



Possiamo definire quindi la funzione d che associa, per ogni vertice del grafo, la sua distanza da S rappresentata da un numero naturale:

$$d : V \mapsto \mathbb{N}$$

Anche questa funzione è implementata tramite un array, allo stesso modo dei colori, e al termine dell'esecuzione dell'algoritmo $\text{BFS}()$ deve quindi valere la seguente proprietà:

$$\forall v \in V \quad d[v] = \delta(S, v)$$

dove $\delta(S, v)$ indica proprio la distanza tra il nodo S e il generico nodo v del grafo.

N.B. $\delta(S, v)$ dipende da come è fatto il grafo, $d[v]$ dipende da come la $\text{BFS}()$ calcola i valori.

Array dei predecessori

Checkpoint: <https://youtu.be/xs3XPSBkwkk?t=2812>

Sia π un percorso minimo da S ad u , esso sarà composto solo da sottopercorsi minimi.

Ossia se $\pi = \langle S, v_i \dots v_j, u \rangle$, dove v_i e v_j sono due nodi all'interno del percorso, sappiamo che il percorso che collega v_i e v_j è un percorso minimo.

Questo si può dimostrare per assurdo:

Sia π un percorso minimo e sia π' un percorso NON minimo appartenente a π , quindi che collega due nodi che fanno parte di π .

$$\pi' \in \pi$$

Ma se π' non è minimo significa che per forza deve esistere un altro percorso appartenente a π più piccolo, che chiameremo π'' , che collega gli stessi nodi che collegava π' .

Quindi se sostituisco π' con π'' posso costruire un percorso più piccolo.

Quindi abbiamo trovato una contraddizione sull'ipotesi e quindi non è possibile che π sia un percorso minimo.

Questa proprietà è importante perché ci dice che se il percorso minimo che va da S ad u passa per un altro nodo v, allora il sottopercorso che va da S a v è un sottopercorso minimo. Questo vuol dire che è sempre possibile costruire un percorso minimo da S ad un qualsiasi nodo u a patto che si conosca quale sia il predecessore di u su quel percorso minimo.

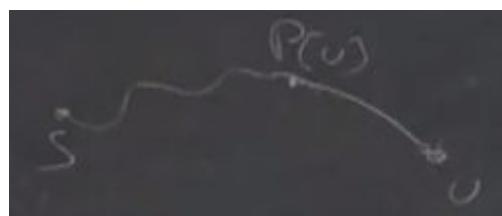
Per tenere traccia dei predecessori associamo ad ogni nodo un valore attraverso un altro array.

Definiamo quindi la funzione p che associa ad ogni vertice, un altro vertice:

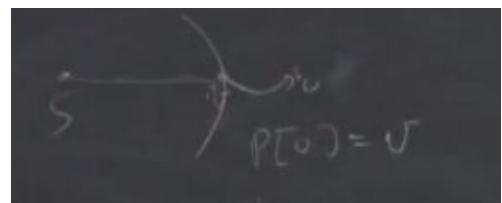
$$p : V \mapsto V \cup \{\text{null}\}$$

che rispetta questa proprietà:

$p[u] = v \Leftrightarrow \exists \text{ un percorso minimo da } S \text{ ad } u \text{ e } v \text{ è il penultimo vertice del percorso}$



Per implementare questa funzione quindi useremo un array ed associeremo ad ogni generico nodo u il suo predecessore (genericamente chiamato v) che sarà il nodo da cui abbiamo scoperto u per la prima volta attraverso un arco di dimensione 1.



Codice di BFS

Questo algoritmo quindi prende in input il grafo ed un nodo selezionato del grafo stesso e restituisce il valore di Reach del nodo selezionato, ossia restituisce i nodi del grafo che sono raggiungibili dal nodo selezionato.

RIGA	BFS(G, S)	$S \in V, G = \langle V, E \rangle$
1	FOR $v \in V$ DO	$S \in V, G = \langle V, E \rangle$
2	$C[v] = b$	
3	$d[v] = \infty$	
4	$P[v] = NIL$	
5	$Q = \{S\}$	INIT
6	$C[S] = g$	
7	$d[S] = 0$	
8	WHILE $Q \neq \emptyset$ DO	
9	$v = \text{TESTA}(Q)$	
10	FOR EACH $u \in \text{ADJ}(v)$ DO	
11	IF $C[u] = b$ THEN	
12	$Q = \text{ACCOLA}(Q, u)$	
13	$C[u] = g$	
14	$d[u] = d[v] + 1$	
15	$P[u] = v$	
16	$Q = \text{DECOSTA}(Q)$	
17	$C[S] = m$	

L'algoritmo prima di iniziare a visitare i nodi utilizza un for (che chiameremo init) che si occupa di associare ad ogni vertice il suo stato iniziale riempiendo i tre array.

- 1) L'array dei colori dev'essere inizializzato al colore bianco perché è quello che determina lo stato di non visitato.
- 2) L'array delle distanze dev'essere inizializzato ad infinito perché non sappiamo a priori se tutti i nodi di V sono raggiungibili, attraverso un percorso, da s .
Nel caso ci fosse un nodo non raggiungibile da s il valore della sua distanza dev'essere infinito, per definizione di distanza.
- 3) L'array dei predecessori dev'essere inizializzato a null perché non sappiamo a priori i predecessori dei vertici diversi da s (che andremo a calcolare strada facendo) e per s è corretto che sia uguale a null.

Una volta che tutto il grafo è inizializzato, inseriamo in coda il vertice s .

Dato che abbiamo inserito in coda un nodo e la coda rappresenta la nostra frontiera, è necessario cambiare lo stato del vertice s a grigio.

Inoltre dato che la distanza di s da se stesso è 0, lo inseriamo nell'array delle distanze.

L'algoritmo quindi alla riga 9 inizia ad iterare sugli elementi della coda finché la coda non risulta vuota.

A questo punto, si estraе il vertice che sta in cima alla coda e si esplorano i suoi archi uscenti e poi questi vengono aggiunti in coda. L'esplorazione degli archi uscenti è definita nella riga 10 dove $\text{adj}[v]$ rappresenta l'insieme dei nodi adiacenti a v .

Qui nasce un problema: tra i nodi adiacenti di un generico nodo v potrebbero esserci dei nodi che sono stati visitati in precedenza oppure che sono già in coda (e quindi verranno visitati sicuramente più tardi).

Per questo motivo effettuiamo il controllo sul suo stato prima di inserirlo in coda, infatti solo i nodi che non sono stati ancora visitati possono entrare in coda.

Se il confronto è verificato, mettiamo in coda il vertice c (il quale è adiacente a v) e coloriamo il suo u di grigio perché è entrato in coda.

Inoltre dobbiamo impostare la distanza di u alla distanza di $v+1$ perché u e v sono adiacenti e in quanto tali esiste un solo arco che li collega, quindi la distanza di u da s sarà quella di $v+1$.

Infine impostiamo il predecessore di u a v perché abbiamo scoperto u a partire da v .

Una volta che il ciclo for è finito possiamo eliminare l'elemento v della coda e colorarlo di nero perché è stata conclusa la sua visita.

E' interessante notare che dopo l'init nessun nodo è colorato di grigio, quindi, per via del confronto successivo, ogni nodo può entrare una sola volta in coda.

I vertici rimasti bianchi al termine di `BFS()` sono quelli non raggiunti dall'insieme `Reach` calcolato in `s`.

Qui puoi trovare un esempio di esecuzione di `BFS()` fatto alla lavagna:

<https://youtu.be/xs3XPSBkwkk?t=3575>

Usare BFS() per calcolare il percorso minimo tra due nodi

Dato che un percorso minimo è composto solo da sottopercorsi minimi, possiamo sfruttare l'array dei predecessori calcolato con BFS() possiamo creare un algoritmo che ricostruisce il percorso minimo tra due vertici del grafo.

```

PercorsoMin(G, S, v)
    BFS(G, S)
    IF P[v] ≠ NIL THEN
        COSTRUISCI PERCORSO(P, S, v)

COSTRUISCI PERCORSO(P, S, v)
    IF v = S THEN
        PRINT S
    ELSE
        COSTRUISCI PERCORSO(P, S, P[v])
        PRINT v
    
```

Questo algoritmo funziona effettuando iterazioni ricorsive sull'array dei predecessori.

Complessità

Checkpoint: <https://youtu.be/xs3XPSBkwkk?t=4498>

Il primo for per l'init è lineare sul numero dei vertici, quindi avrà una complessità di $\Theta(|V|)$.

Il for interno itera sul numero di archi uscenti dipendentemente dal vertice v che sta gestendo. Gli archi uscenti di un vertice vengono esplorati al massimo una sola volta (perché ogni vertice entra in coda una sola volta e non può più rientrare), quindi ciò implica che il for ha un costo che è al massimo pari al numero di archi $O(|E|)$.

Le ultime due istruzioni del while, ossia il decodamento di Q e il coloramento a nero del vertice v vengono eseguite sicuramente al massimo su tutti i vertici nel caso in cui la BFS raggiungesse tutti i vertici. Perciò avrà una complessità di $O(|V|)$.

Quindi avremo:

$$T_{BFS}(|G|) = \Theta(|V|) + O(|E|) + O(|V|) = O(|V| + |E|) = O(|G|)$$

Dimostrazione del funzionamento

Per dimostrare il corretto funzionamento della BFS dovremmo dimostrare 3 proprietà:

- Una per il calcolo delle distanze

$$\forall v \in V \quad d[v] = \delta(S, v)$$

- Una per il calcolo del predecessore

$\forall v \in V \quad p[v]$ è il predecessore di v nel percorso minimo tra S e v

- Una che mette in relazione il colore nero con la raggiungibilità da S

$$\forall v \in V \quad (S, v) \in \text{Reach} \Leftrightarrow c[v] = n$$

Le tre "sotto proprietà"

Per dimostrare queste tre proprietà sfruttiamo altre 3 proprietà più semplici che ci permetteranno di concludere la dimostrazione.

Sotto-proprietà 1 $\forall v \in V \quad (u, v) \in E \Rightarrow \delta(s, v) \leq \delta(s, u) + 1$

In altre parole, dato un arco da u ad v , la distanza di v da S non può essere più grande della distanza di u da S + 1.

Caso a) se v non è raggiungibile da S .

E' un caso banale perché la distanza tra v e S è infinito e dato che u non è raggiungibile da S (perché altrimenti lo sarebbe anche v) la distanza tra u e S è anch'essa infinito e quindi la proprietà vale perché $\infty \leq \infty + 1$.

Caso b) se v è raggiungibile da S .

Ragioniamo quindi per assurdo, ossia:

$$\delta(s, v) > \delta(s, u) + 1$$

Ciò significa che il percorso minimo che va da S a v è strettamente più grande del percorso minimo che va da S a $u + 1$.



Ciò implica che valga ciò: $|\pi| > |\pi'| + 1$, ma questa è una contraddizione perché π non può essere il percorso minimo tra v ed S perché avrei trovato un percorso più piccolo (π').

Quindi la tesi è sbagliata e vale la proprietà: $\delta(s, v) \leq \delta(s, u) + 1$

Sotto-proprietà 2) $\forall v \in V \quad d[v] \geq \delta(S, v)$

Per dimostrare ciò vediamo tutti i possibili casi:

Caso 1: v irraggiungibile da S

se v è irraggiungibile da S , sia $d[v]$ che δ valgono ∞ , quindi è vero banalmente

Caso 2: v raggiungibile da S

per verificare che questa proprietà vale **durante l'esecuzione** dobbiamo vedere ogni volta che viene modificato $d[]$ se questa proprietà vale ancora.

Se vale per ogni modifica di $d[]$ allora possiamo dire che vale durante l'esecuzione, quindi:

Caso 2.1: $d[S] = 0$

Dato che qui viene modificata solo la distanza di S e ci assegna il valore corretto, e tutte le altre distanze (che già erano valide perché impostate ad infinito) non sono state modificate, quindi la proprietà continua a valere.

Caso 2.2: $d[v] = d[u]+1$

Questa istruzione viene eseguita solo se u viene inserito in coda, quindi è possibile fare induzione sul numero di accodamenti, dimostrando che per ogni accodamento la proprietà continua a valere.

base) il caso base sarebbe quello con 0 operazioni di accodamento e abbiamo già visto che funzionava(vedi: v irraggiungibile da S)

passo) supponendo di aver effettuato k operazioni di accodamento e fino a quel punto la proprietà è stata verificata.

Consideriamo il $k+1$ accodamento su un generico nodo u .

Prima che esso venga eseguito la proprietà su u valeva.

Verifichiamo che dopo l'accodamento la proprietà su u continua a valere in questo modo:

Sia z il nodo che si trova nel top dello stack e questo nodo abbia un arco che lo collega a u il quale è ancora bianco, per questo motivo dovremmo accodare u . Per ipotesi induttiva sappiamo che la proprietà su z è valida e dopo l'esecuzione varrà $d[u] = d[z]+1$.

Sfruttando la proprietà 0 sappiamo inoltre che vale questo:

$$\delta(S, u) \leq \delta(S, z) + 1$$

Perciò abbiamo che:

$$\begin{cases} (a) & \delta(S, u) \leq \delta(S, z) + 1 \\ (b) & d[u] = d[z] + 1 \\ (c) & d[z] \geq \delta(S, z) \end{cases}$$

Da (c) possiamo dedurre che:

$$d[z] \geq \delta(S, z) \Rightarrow d[z] + 1 \geq \delta(S, z) + 1$$

ma dato che

$$d[u] = d[z] + 1$$

Quindi posso sostituire:

$$d[u] \geq \delta(S, z) + 1$$

ma per il punto (a) abbiamo che:

$$\delta(S, z) + 1 \geq \delta(S, u)$$

Quindi mettendo tutto insieme abbiamo che:

$$d[u] = d[z] + 1 \geq \delta(S, z) + 1 \geq \delta(S, u)$$

ossia,

$$d[u] \geq \delta(S, u)$$

C.V.D.

Questo ci assicura che le stime vengono calcolate sempre eccesso e mai per difetto.

Sotto-proprietà 3) **Durante l'esecuzione di BFS**, supponendo che ad un certo istante di tempo la coda Q sia composta da k elementi:

$$Q = \langle v_1, v_2, \dots, v_k \rangle$$

Allora valgono tra questi k elementi le seguenti proprietà:

$$2.1) \quad \forall 1 \leq i \leq k-1 \quad d[v_i] \leq d[v_{i+1}]$$

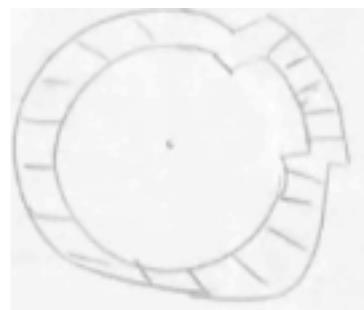
Questa proprietà ci dice che **la coda è formata da elementi le cui stime sono non decrescenti**. Ossia, andando avanti con l'algoritmo di BFS le stime possono solo crescere o rimanere uguali.

$$2.2) \quad d[v_k] \leq d[v_1] + 1$$

Questa proprietà ci dice che **la stima dell'ultimo elemento della coda è minore o uguale della stima del primo elemento della coda + 1**.

Ciò significa che ad un determinato istante, la coda ha tutti elementi che si trovano o alla stessa stima del primo elemento o alla stessa stima del primo elemento + 1.

Graficamente la coda assume questa forma:



Quindi ad esempio nella stessa coda non ci possono essere due elementi uno con stima 3 e l'altro con stima 5.

Dimostrazione del calcolo corretto delle distanze e del predecessore

Dimostriamo che, al termine della BFS valgono:

- 1) $\forall v \in V \quad d[v] = \delta(S, v)$
- 2) $\forall v \in V \quad p[v]$ è il precedente di v nel percorso minimo tra S e v , ossia $\forall v \in V \quad PM(S, v) = PM(S, p[v]) \cup v$.

Se v è raggiunto da S , sicuramente la sua distanza $d[v]$ sarà minore di ∞ .

I vertici raggiungibili da S possono essere partizionati in questo modo:

$$V_i = \{v \mid d(S, v) = i\}$$

ossia divido l'insieme dei raggiungibili da S in base alla loro distanza da S , quindi tutti quelli a distanza 1 si troveranno in V_1 , quelli a distanza 2 si troveranno in V_2 , ecc..

Ovviamente questo è un insieme finito e rispetta le 3 caratteristiche di partizione algebrica.
(ndr. che puoi trovare qui: [link](#)).

Per dimostrare il calcolo corretto delle distanze e dei predecessori facciamo induzione sull'indice i dimostrando ciò:

$\forall i \geq 0 : \forall v \in V_i \exists! \text{ istante in cui} :$

- 1) v viene inserito in coda
- 2) $c[v] = g$
- 3) $d[v] = i$
- 4) se $v \neq S, p[v] \in V_{i-1}$

Ossia, in altre parole, ciò ci dice che ogni vertice raggiungibile da S prima o poi viene messo in frontiera e nello stesso momento viene colorato di grigio, viene aggiornata la sua stima proprio alla sua distanza i ; inoltre se v non è la sorgente il suo predecessore si trova proprio a distanza $i-1$ della sorgente.

Il punto 3 è esattamente ciò che dobbiamo verificare per dimostrare la tesi (1) sul calcolo delle distanze e il punto 4 è esattamente ciò che dobbiamo verificare per dimostrare la tesi (2) sul calcolo dei predecessori perché se vale questo:

$$PM(S, V) = PM(S, p[v]) \cup v$$

esiste un arco tra $p[v]$ e v , e quindi il predecessore si dovrà trovare a una distanza minore di 1 da v e quindi $p[v]$ si troverà a una distanza di V_{i-1} .

Nel caso di non raggiungibilità

Dato che abbiamo già dimostrato l'invariante $\forall v \in V \quad d[v] \geq \delta(S, v)$ che è valida sempre, anche durante l'esecuzione, possiamo sfruttarla per dire che se S e v non sono raggiungibili $\delta(S, v)$ vale ∞ per definizione di distanza e $d[v]$ è inizializzata a ∞ e non viene mai modificata per come è stata definita la BFS (perché altrimenti sarebbe raggiungibile da S). Perciò il problema dell'ugualanza dell'array delle distanze e la funzione delta è banalmente vero nel caso di non raggiungibilità. Inoltre se v non è raggiungibile da S sicuramente $p[v]$ sarà sempre e solo NULL, quindi il problema della dimostrazione si trova solo nei nodi che sono raggiungibili da S .

Nel caso di raggiungibilità di v

Per dimostrare la tesi facciamo induzione su i .

Caso base) $i = 0$

Se $i = 0$ significa che stiamo cercando di verificare la proprietà su tutti i vertici a distanza 0 da S , e l'unico nodo che rispetta ciò è proprio S stesso.

Perciò banalmente possiamo vedere che quando S viene messo in coda, il suo colore è uguale a grigio e la sua distanza è impostata a 0 (che è lo stesso valore di i) quindi la tesi per il caso base è dimostrata.

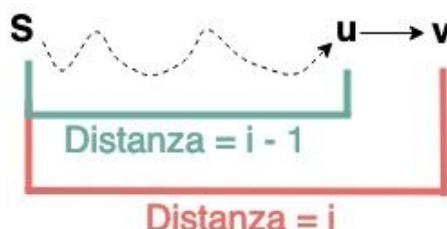
Passo) $i > 0$

Fissiamo un valore di i e lavoriamo sui nodi che appartengono a V_i .

Prima di tutto, siamo certi che tutti i nodi di V_i entrino in coda?

Sì, perché un qualsiasi nodo v in V_i è collegato a S dal percorso minimo di lunghezza i e dato il caso induttivo ($i > 0$), questo percorso ha almeno lunghezza 1 e quindi il percorso è non-vuoto.

Perciò sicuramente esisterà un nodo che si trova esattamente prima di v , che possiamo chiamare genericamente u che si trova a distanza $i-1$ da S e quindi u appartiene all'insieme V_{i-1} .



Sfruttiamo ora l'ipotesi induttiva che ci assicura che la proprietà sia valida fino a $i-1$.

Perciò sappiamo che il generico nodo u , predecessore di un generico nodo di V_i entrerà sicuramente in coda e quindi prima o poi si ritroverà nella testa della coda.

Quando si trova nella testa della coda, se v ancora non è stato visitato, siccome c'è un arco che collega u e v , visiterà v che entrerà in coda.

Però questo non ci assicura che proprio u , a distanza V_{i-1} a scoprire v , perché magari potrebbe esserci un nodo z la cui distanza da S è maggiore strettamente di $i-1$ adiacente a v che scopre v prima di u .

$$\delta(S, z) > i - 1$$

Ragioniamo per assurdo e vediamo che ciò non è possibile:

- 1) Se fosse z a scoprire v significherebbe che z entra in coda prima di u .
- 2) Sfruttiamo la sotto proprietà 2 abbiamo che vale ciò quando z entra in coda:

$$d[z] \geq \delta(S, z) > i - 1 \Rightarrow d[z] > i - 1$$

- 3) Sempre perché stiamo ragionando per assurdo, vediamo che u entra in coda dopo z .

- 4) Ma dato che la distanza tra u e s vale v_{i-1} , per ipotesi induttiva sappiamo per quel livello la tesi è verificata, quindi entrerà in coda e calcolerà correttamente distanza e predecessore.

$$d[u] = i - 1$$

- 5) Quindi vale ciò:

$$\begin{cases} d[u] = i - 1 \\ d[z] > i - 1 \end{cases} \Rightarrow d[z] > d[u]$$

- 6) Ma se $d[z]$ è maggiore di $d[u]$ significa che z si trova a una distanza maggiore da s rispetto a u .
- 7) Ma se z è più “lontano” da s rispetto ad u ed è entrato prima in coda abbiamo trovato un assurdo perché sfruttando la monotonicità della coda, ossia la sotto-proprietà 3 non è possibile che le stime di una coda decrescano ed è proprio quello che è successo in questo caso perché $d[u]$ è più piccolo di $d[z]$ ed entra dopo.

Per questo motivo non è possibile che v sia scoperto da nodi che si trovano a distanza maggiore rispetto a $i-1$ e quindi entrerà in coda con una stima pari ad i (e quindi la sua stima è impostata correttamente) e il predecessore verrà assegnato proprio al nodo u il quale appartiene a v_{i-1} (e quindi il suo predecessore è impostato correttamente).

Quindi anche in caso di raggiungibilità dei nodi da s la proprietà è verificata.

Algoritmo DFS (Profondità)

Checkpoint:<https://youtu.be/VQqaycrT8vM?t=3231>

La visita in profondità di un grafo generalizza la visita in profondità degli alberi, seguendo il suo stesso principio: ogni nodo viene visitato seguendo un percorso alla volta finché è possibile.

Per gli alberi il concetto di “seguire un percorso finché è possibile” è molto semplice ed è espresso dal fatto che ogni percorso di un albero sicuramente termina con una foglia, perché gli alberi sono aciclici.

Per i grafici lo stesso concetto non è banale come per gli alberi perché possono ammettere i cicli e quindi anche percorsi infiniti e quindi l’algoritmo per la visita in profondità potrebbe entrare in loop su un ciclo e non terminare mai.

Questi sono gli stessi problemi visti per la visita in ampiezza e pertanto useremo la stessa soluzione attraverso [l’array dei colori](#) che mi permette di tracciare lo stato di ogni vertice.

Codice

L’algoritmo per la ricerca in profondità è diviso in due funzioni, `DFS_Visit` e `DFS`.

Anche qui è presente una parte di inizializzazione degli array.

```
DFS_Visit ( G, S )
c[s] = g

for each u ∈ adj[S] do
| if ( c[u] = b ) then
| | p[u] = s
| | dfs_visit(G, u)

c[s] = n
```

- `DFS_Visit`

`DFS_Visit` prende in input il grafo ed il nodo sorgente il quale vogliamo esplorare.

Proprio perché vogliamo visitare il nodo `s` poniamo il suo colore a grigio e poi andiamo a considerare tutti i suoi nodi adiacenti.

Non sappiamo a priori lo stato di questi nodi adiacenti, proprio perché potrebbero esserci cicli nel grafo, perciò bisogna controllare se sono bianchi.

Se lo sono procediamo nella loro esplorazione tramite una chiamata ricorsiva di `DFS_Visit` su questo vertice bianco.

Quando questa chiamata termina assegniamo al valore dei predecessori del nodo adiacente `v`, il valore `s` perché è `s` che ha scoperto `v` avendolo trovato bianco.

Infine setto il colore del nodo `s` a nero perché è stato visitato.

```

DFS ( G )
  for v ∈ V do
    | c[v] = b
    | p[v] = null

  for each v ∈ V do
    | if (c[v] = b) then
      |   | dfs_visit(G, v)
      |

```

- DFS

L'algoritmo DFS prima di tutto inizializza il grafo in cui tutti i vertici vengono posti a bianco e tutti i predecessori vengono impostati a null.

Il secondo for seleziona sorgenti di visita in profondità perché il primo vertice del grafo sicuramente sarà bianco. Al termine di DFS_Visit su quel primo vertice ci possono essere due casi

- 1) Tutti i vertici sono neri
- 2) Esiste ancora qualche vertice bianco

(NB. Al termine di un DFS_Visit() non possono esserci ancora nodi grigi perché all'uscita tutti i nodi diventano neri).

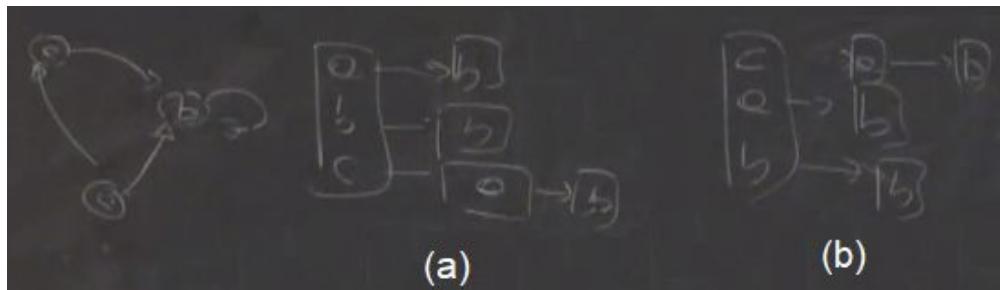
Quindi, se tutti i vertici sono neri, significa che tutto il grafo è stato visitato e il secondo for di DFS() deve concludersi, oppure, se ci sono ancora dei nodi bianchi questi sono altre sorgenti del grafo, perché non erano raggiungibili dalle sorgenti precedenti.

Array dei predecessori in DFS VS array dei predecessori in BFS

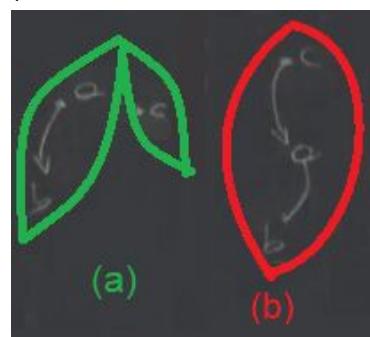
Nel caso della BFS, l'array dei predecessori contiene i percorsi minimi.

Invece al termine dell'algoritmo della DFS le cose sono diverse perché la DFS genera array dei predecessori diversi in base alla topologia del grafo.

Ad esempio, questo grafo può essere rappresentato tramite liste di adiacenza come (a) o (b) in modo equivalente.



Se si esegue l'algoritmo di DFS su (a) questo genererà nell'array dei predecessori sostanzialmente due alberi, mentre (b) genererà un unico percorso (quindi un albero) che è diverso da quello generato da (a).



Questa proprietà in realtà è una proprietà generale della DFS:

Sia G un grafo, l'array dei predecessori di G al termine dell'algoritmo DFS conterrà un insieme di archi dal quale è possibile generare un insieme di alberi (foresta) tutti fra loro sconnessi.

Le radici di questi alberi sono le sorgenti del grafo.

Dato che ogni vertice che si trova nel grafo si trova per forza anche nella foresta e dato che ogni arco che si trova nella foresta si trova necessariamente anche nel grafo, perché noi costruiamo la foresta durante la scoperta di un nuovo arco, possiamo dire che la foresta è un sottografo del grafo originario.

Quindi concludiamo che la DFS genera un'array dei predecessori che può essere anche molto diverso da quello generato dal BFS e quindi questo è un fattore di selezione tra BFS e DFS per la scelta di algoritmo da utilizzare per risolvere i problemi.

Infatti se vogliamo calcolare i percorsi minimi dovremmo usare la BFS mentre se non ci preoccupiamo di quali percorsi (anche non minimi) vengono seguiti e ci importa di visitare tutto il grafo dovremmo usare la DFS.

Teorema della struttura a parentesi dei tempi

Estendiamo il codice della DFS introducendo una variabile condivisa chiamata `tempo` la quale si occuperà di tenere traccia del momento in cui vengono scoperti i vertici e in cui la DFS termina la visita dei vertici.

A) DFS

```
DFS ( G )
  for v ∈ V do
    | c[v] = b
    | p[v] = null

  tempo = 0
  for each u ∈ V do
    | if (c[v] = b) then
    |   | dfs_visit(G, v)
    |
```

B) DFS Visit

```
DFS_Visit ( G, S )
  c[s] = g
  d[s] = tempo
  tempo = tempo + 1

  for each u ∈ adj[S] do
    | if ( c[u] = b ) then
    |   | p[u] = s
    |   | dfs_visit(G, u)

  c[s] = n
  f[s] = tempo
  tempo = tempo + 1
```

Dove l'array `f[]` rappresenta il tempo di fine visita di un nodo, ossia quando viene colorato di nero.

Su questi tempi valgono queste proprietà:

Proprietà 1)

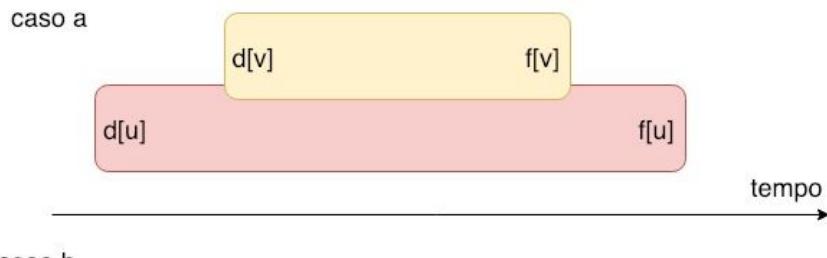
Per ogni coppia di vertici (u, v) , al termine della DFS potrà valere solo una di queste condizioni:

Proprietà a) $d[v] < d[u] < f[u] < f[v]$ OPPURE $d[u] < d[v] < f[v] < f[u]$

Proprietà b) $d[v] < f[v] < d[u] < f[u]$ OPPURE $d[u] < f[u] < d[v] < f[v]$

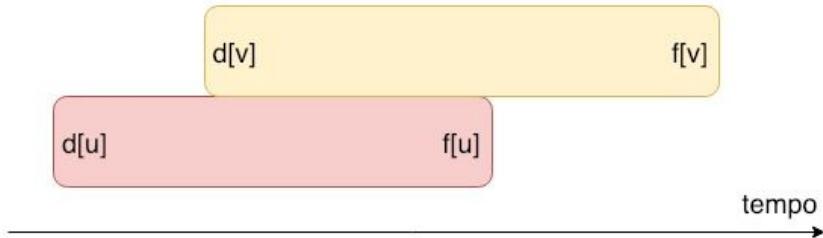
Ciò significa che i due intervalli sono completamente inclusi uno nell'altro (caso a) oppure sono completamente disgiunti uno dall'altro (caso b).

Graficamente avremo queste situazioni:



Da questo possiamo dedurre che è impossibile che accada una situazione di "accavallamento" di questo tipo:

IMPOSSIBILE



Proprietà 2) Della proprietà 1 sarà possibile dimostrare che se vale il caso (a), ossia se una visita ad un nodo u è inclusa nella visita di un nodo v allora u è un discendente di v nella foresta generata della DFS nell'array dei predecessori. Se invece vale il caso (b) tra i generici u e v non è presente alcuna relazione di discendenza.

Dimostrazione

```

DFS_Visit ( G, S )
c[s] = g
d[s] = tempo
tempo = tempo + 1

for each u ∈ adj[S] do
| if ( c[u] = b )
then
| | p[u] = s
| | dfs_visit(G,u)

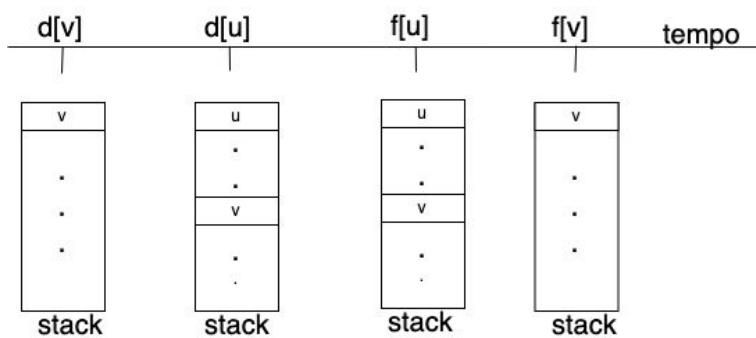
c[s] = n
f[s] = tempo
tempo = tempo + 1;

```

La prima cosa che possiamo notare è che una chiamata a `DFS_Visit` può solo modificare i tempi di inizio ($d[s]$) e di fine ($f[s]$) visita di S . L'unica operazione che viene eseguita su un vertice diverso è la scrittura dell'adiacenza nell'array dei predecessori ($p[u] = s$).

Essendo questo un algoritmo ricorsivo, significa che gli unici momenti in cui i tempi di inizio e fine di un vertice possono essere modificati sono quando in cima dello stack c'è un record associato ad una chiamata che ha ricevuto quel vertice in ingresso.

Ciò significa che ad esempio, al tempo $d[v]$ e al tempo $f[v]$ il nodo v si troverà proprio nel top dello stack.



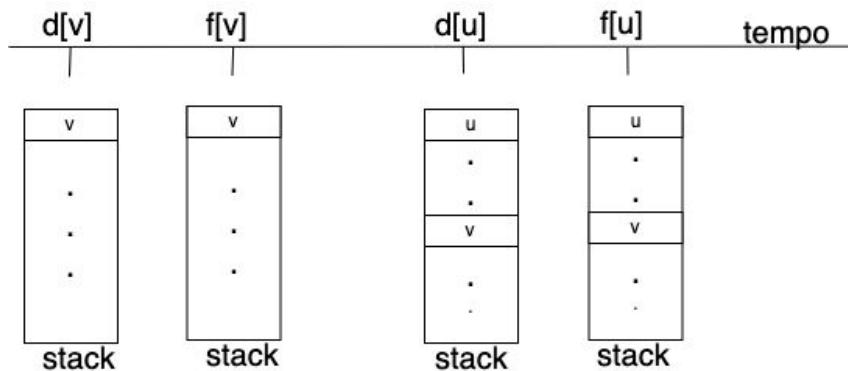
Caso $d[v] < d[u] < f[u] < f[v]$

Se un vertice v è stato inserito nello stack al tempo $d[v]$ potrà riemergere al top dello stack solo in due casi:

- 1) Cancello tutti i nodi dello stack fino a rimettere v nel top, ossia concludo tutte le chiamate DFS dei nodi che sono discendenti di v . Ciò è possibile perché l'algoritmo è stato strutturato per terminare solo quando tutti i nodi adiacenti sono stati visitati, ossia quando le chiamate ricorsive che esso effettua sono tutte concluse.
Quindi il caso $d[v] < d[u] < f[u] < f[v]$ è verificato.
- 2) Reinserisco un altro v in top senza guardare in faccia a nessuno:
Ciò non è possibile perché un nodo viene inserito nello stack di attivazione solo quando è stato scoperto, e dato che ogni nodo viene scoperto una ed una sola volta per via del colore bianco che viene subito cambiato in grigio non è possibile che l'algoritmo reinserisca un nodo già presente nello stack, nello stack stesso.

Caso $d[v] < f[v] < d[u] < f[u]$

Questo caso è ovvio perché non c'è nulla che impedisca una situazione del genere:



Ossia se tra l'inizio e la fine della chiamate su v non inserisco mai u nello stack e quindi non effettuo la visita su u , questa visita potrà essere effettuata dopo senza alcun problema.

Questo può succedere in tante situazioni e sicuramente può succedere se u non è raggiungibile da v .

Proprietà: $d[v] < d[u] < f[u] < f[v] \Leftrightarrow u$ è discendente di v nella foresta

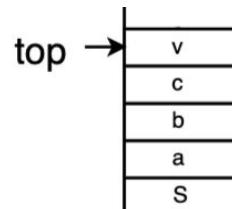
“ u è discendente di v nella foresta” significa che nella foresta di alberi creati della DFS, u e v appartengono allo stesso albero ed esiste un percorso che va da v ad u .

- 1) Caso $\Rightarrow: d[v] < d[u] < f[u] < f[v] \Rightarrow u$ è discendente di v nella foresta

Per dimostrare ciò pensiamo ad un generico stack di attivazione, come da esempio sul lato.

In questo caso sappiamo che, sempre perché viene effettuata una chiamata ricorsiva se e solo se viene scoperto un arco che porta ad un vertice bianco, esiste nel grafo un arco che collega s ad a .

$$S \mapsto a \in E$$



E sempre perché è S a scoprire il vertice a sappiamo che è il suo predecessore:

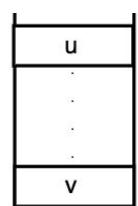
$$p[a] = S$$

Lo stesso identico ragionamento può essere eseguito per tutti i vertici che si trovano nello stack quindi, ad esempio avremo che:

$$b \mapsto a \in E, \quad p[b] = a \quad (p[a] = b^*)$$

E questo possiamo farlo fino a che non arriviamo al top dello stack.

Ricordando che la foresta viene gestita nell'array dei predecessori della DFS, possiamo dire che tutti gli archi che abbiamo trovato saranno contenuti nella foresta e quindi potremmo dire che v è discendente di s .



Per questo motivo se si verifica che $d[v] < d[u]$ avremo che v si trova prima nello stack rispetto a u e quindi u è discendente di v nella foresta generata della DFS.

- 2) Caso $\leftarrow u$ è discendente di v nella foresta \Rightarrow
 $d[v] < d[u] < f[u] < f[v]$

Questo si dimostra per induzione sulla lunghezza del percorso che connette v ad un altro vertice e vedremo che questo percorso è contenuto completamente nello stack.

Base) per $|\pi| = 1$, ossia se il percorso ha lunghezza 1.

Al tempo $d[v]$ la DFS inizierà ad esplorare i discendenti bianchi di v a distanza 1. Dato che, per ipotesi, u è discendente di v ed il percorso ha lunghezza uno esiste un unico arco in tutto il percorso che porta da v ad u .

Inoltre data l'ipotesi non è possibile che u sia già stato visitato da un'altra sorgente e quindi sicuramente al tempo $d[v]$ il colore di u sarà bianco.

Dato ciò sicuramente avremo che vale: $d[v] < d[u]$

Ora occupiamoci di $f[v]$: il tempo di fine visita di v può essere minore del tempo di fine visita di u ?

No, perché implicherebbe che (al tempo $d[v]$) il ciclo for di DFS_Visit eseguito su v ha trovato il nodo u non-bianco ma ciò non è possibile perché al tempo $d[v]$ il colore di u sarà bianco. Perciò vale: $d[u] < f[v]$

Infine, per il teorema della struttura a parentesi non è possibile che valga $f[u] > f[v]$ perché abbiamo già visto che vale ciò: $d[v] < d[u] < f[v]$. Quindi l'unica combinazione dei tempi di visita tra u e v se u è discendente di v nella foresta ed il percorso che li collega ha lunghezza 1 è proprio:
 $d[v] < d[u] < f[u] < f[v]$.

Passo) per $|\pi| > 1$ ossia se il percorso ha lunghezza maggiore di 1.

Essendo una dimostrazione per induzione assumiamo che la proprietà valga per il nodo che precede u nel percorso che lo porta da v . Il precedente di u lo chiameremo z , e varrà $d[v] < d[z] < f[z] < f[v]$.

La lunghezza del percorso da v a z è $|\pi| - 1$ ed esiste un arco tra (quindi un percorso di lunghezza 1) tra z e u .

Come prima ragioniamo sui tempi di inizio e fine di u :

1) è possibile che valga $d[v] < d[u] < d[z]$? Nope, perché non è z il vertice che scopre u , e perciò varrà questo: $d[v] < d[z] < d[u]$.

2) quand'è che avviene $f[u]$? Sempre per il teorema della struttura a parentesi u non può finire dopo il suo precedente z quindi deve per forza valere $d[v] < d[z] < d[u] < f[u] < f[z] < f[v]$ e quindi vale la tesi:

$$d[v] < d[u] < f[u] < f[v]$$

C.V.D.

Quindi se vale che l'intervallo di un vertice u è contenuto in quello di un altro vertice v sappiamo sicuramente che u è discendente di v , e viceversa.

Teorema del percorso bianco

Checkpoint:<https://youtu.be/bz1qcL0oD-c?t=3997>

$\forall u, v \in V \ u \text{ e' discendente di } v \text{ nella foresta} \Leftrightarrow$
 $\text{al tempo } d[v] \exists \text{ un percorso } \pi \text{ in } G \text{ che contiene solo vertici bianchi}$

Dove il percorso π non è necessariamente compreso nella foresta.

Questo teorema è importante perché **specificata quando** l'esistenza di un percorso in un grafo garantisce la discendenza tra i nodi di quel percorso.

Infatti, la sola esistenza del percorso tra a e b non ci dice che b è discendente di a nella foresta perché dipende da come il grafo è stato implementato.

Guarda il grafo presente nell'esempio nel paragrafo [Array dei predecessori in DFS VS BFS](#), esiste un percorso che va dal vertice c al vertice b , ma nell'implementazione (a) non genera una foresta in cui b è discendente di c .

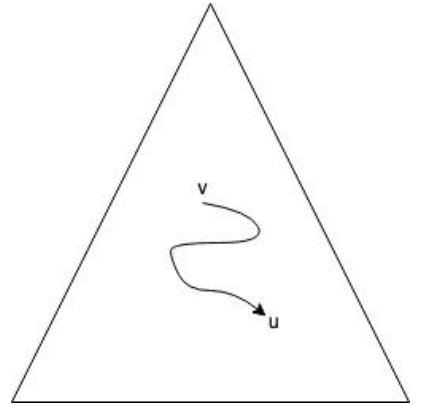
Dimostrazione

Checkpoint:<https://youtu.be/bz1qcL0oD-c?t=4260>

- 1) Caso \Rightarrow : se u è discendente di v nella foresta allora al tempo $d[v]$, ossia quando viene scoperto v , esiste il percorso con vertici tutti bianchi tra u e v .

Dato che u è discendente di v nella foresta e la foresta è un sottografo del grafo G , in cui ogni percorso che appartiene alla foresta sicuro appartiene al grafo.

Ciò significa che tutti gli archi che compongono il percorso che, nella foresta, vanno da u a v esisteranno tutti gli stessi archi anche nel grafo.



Il percorso bianco che stiamo cercando è proprio questo perché è proprio il percorso che la DFS ha seguito per arrivare da v ad u .

In soldoni, la DFS è partita da v e ha esplorato i suoi archi uscenti bianchi.

Uno di questi archi andrà in un nodo che fa parte del percorso che stiamo cercando.

Ricorsivamente, questo ragionamento continuerà finché non trovo u .

Quando trovo u , dato che la DFS è entrata solo in percorsi bianchi è ovvio che all'inizio (ossia quando v è stato scoperto, ossia $d[v]$) esiste il percorso tutto bianco che collega v ed u .

- 2) Caso \Leftarrow : se al tempo $d[v]$, quando viene scoperto v , esiste il percorso con vertici tutti bianchi tra u e v allora u è discendente di v nella foresta.

Ciò significa che partendo da v esiste almeno un percorso tutto bianco che collega v ad u .

Dato che potrebbero esserci più percorsi bianchi che soddisfano questa proprietà e non possiamo sapere a priori quale percorso prenderà la DFS, possiamo ragionare sul fatto che qualsiasi percorso bianco da v ad u la DFS possa prendere genererà un albero in cui u è discendente di v .

Per questo motivo possiamo prendere un percorso bianco a caso ed ignorare bellamente tutti gli altri, anche se questo non verrà selezionato della DFS per arrivare ad u da v .

A questo punto ragioniamo per assurdo: assumiamo che la tesi sia falsa, ossia che dato un percorso tutto bianco tra u e v allora u NON è discendente di v .

Definiamo un generico nodo t come il primo nodo che non diventa discendente di v nel percorso.

Ciò significa che tutti i nodi che vanno da v ad t sono tutti discendenti di v . Dato che t potrebbe essere u , ma non potrebbe mai essere v (perchè v è sempre discendente di se stesso, banalmente) possiamo dire che sicuramente esiste un nodo z , precedente a t nel percorso bianco che è discendente di v .

Sfruttiamo ora il teorema della struttura a parentesi che ci assicura che vale:

$$d[v] < d[z] < f[z] < f[v]$$

Ma mettiamolo un attimo da parte.

Pensiamo a QUANDO verrebbe scoperto t :

- 1) All'istante $d[v]$ è bianco per ipotesi perché appartiene comunque al percorso tutto bianco. Quindi t può essere scoperto solo dopo v .
- 2) Dato che abbiamo assunto che t non è discendente di v non è possibile scoprirla tra $d[v]$ e $f[v]$ perché, sempre per il teorema della struttura a parentesi, avremmo che l'assurdo che t sarebbe discendente di v .

Per queste ragioni concludiamo che t , per non essere discendente di v dovrebbe essere scoperto dopo $f[v]$.

Ma ciò non è possibile perché esiste l'arco $z \rightarrow t$. infatti se t fosse scoperto dopo la fine di v significherebbe che fino a quel momento il colore di t è rimasto bianco ma dato che esiste z , che è discendente di v e adiacente di t , avremo che la DFS quando arriva a z esplora i suoi adiacenti bianchi (tra cui t) e renderebbe tutti questi discendenti di v .

Ma questa è una contraddizione perché abbiamo assunto che t non è un discendente di v .

Quindi la tesi è dimostrata.

C.V.D.

Archi di ritorno, archi in avanti e archi di attraversamento

Checkpoint: <https://youtu.be/bz1qclOoD-c?t=5561>

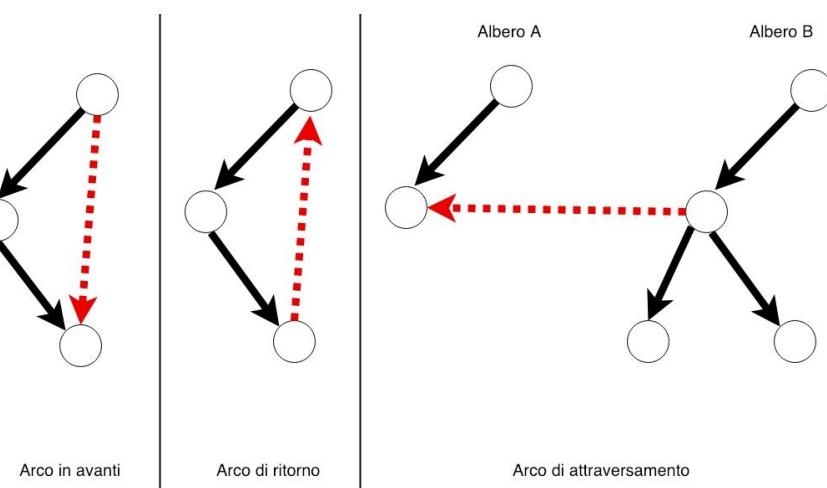
Se eseguiamo una DFS su un grafo G , possiamo ricavare una foresta nell'array dei predecessori; questa foresta conterrà alcuni archi di G e tutti gli stessi vertici di G , quindi è un sottografo di G

$$G = \langle V, E \rangle; F = \langle V, E' \rangle \text{ con } E' \subseteq E$$

Gli archi inseriti nella foresta sono tutti e soli gli archi che la DFS ha usato per scoprire nuovi vertici, ossia quando il colore del vertice di arrivo è bianco e quindi vale ($c[v] == b$).

Nell'insieme E' sicuramente non ci andranno le seguenti tipologie di archi:

- 1) Un arco che connette un antenato con un suo discendente (che non è il figlio) viene detto **arco in avanti**.
La DFS() si accorge di un arco in avanti attraverso la condizione ($c[v] == n \text{ AND } d[u] < d[v]$), perché (per il teorema delle parentesi) se v è discendente di u avremo che $d[u] < d[v] < f[v] < f[u]$ e quindi la relazione $d[u] < d[v]$ ci dice che v è stato scoperto dopo u e sappiamo che è stato terminato perché ha colore nero.
- 2) Un arco che connette un discendente con un suo antenato (che non è il padre) viene detto **arco di ritorno**. La DFS() si accorge di un arco di ritorno attraverso la condizione ($c[v] == g$), ossia se il vertice di arrivo è grigio perché è già stato visitato, ma non ha ancora concluso la chiamata e quindi è un antenato.
La DFS() si accorge di un arco di attraversamento attraverso la condizione ($c[v] == n \text{ AND } d[v] < d[u]$) perché, dato che v non è discendente di u non vale il teorema delle parentesi e quindi viene scoperto e terminato (perché il suo colore è nero) prima della scoperta di u , quindi $d[v] < d[u]$
- 3) Un arco che connette due nodi che non sono nella relazione di discendenza l'uno con l'altro viene detto **arco di attraversamento**. È possibile anche che la relazione di discendenza possa venire a mancare anche tra due nodi dello stesso albero.
La DFS() si accorge di un arco di attraversamento attraverso la condizione ($c[v] == n \text{ AND } d[v] < d[u]$) perché, dato che v non è discendente di u non vale il teorema delle parentesi e quindi viene scoperto e terminato (perché il suo colore è nero) prima della scoperta di u , quindi $d[v] < d[u]$



Come capire se un grafo è aciclico

Un grafo è detto aciclico se non contiene cicli semplici.

La DFS è in grado di riconoscere efficientemente se un grafo contiene cicli o meno, mentre la BFS non è in grado di farlo.

La DFS può risolvere il problema trovando gli archi di ritorno perché vale questa proprietà:

$$\exists \text{ ciclo} \Leftrightarrow \exists \text{ arco di ritorno}$$

Dimostrazione della relazione di uguaglianza tra ciclo e arco di ritorno

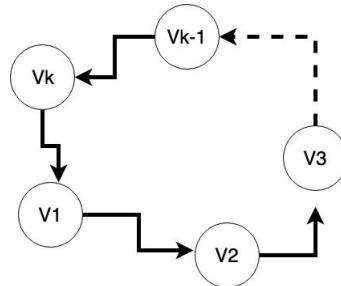
Checkpoint: <https://youtu.be/JEf9ok2qhWE?t=1079>

- 1) Caso \Rightarrow Se esiste un arco di ritorno \Rightarrow Esiste un ciclo nel grafo

Questo è un caso banale perché se esiste un arco di ritorno, questo andrà a finire in un vertice grigio quindi già visitato, perciò ho individuato l'esistenza del ciclo.

- 2) Caso \Leftarrow Se esiste un ciclo nel grafo \Rightarrow Esiste un arco di ritorno

Assumiamo adesso che esista un ciclo nel grafo, perciò avremo quindi una situazione del genere:

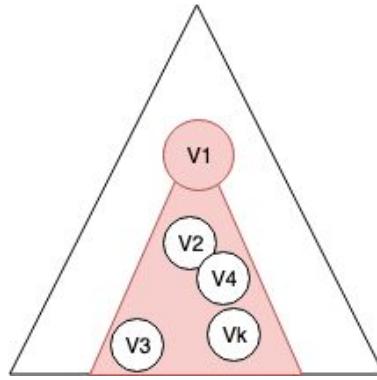


Dato che la DFS prima o poi visiterà tutti questi vertici, possiamo supporre che parta da uno qualsiasi e quindi scegliamo V1.

Per il teorema del percorso bianco sappiamo che al tempo $d[V1]$ tutti gli altri vertici del ciclo avranno colore bianco.

Quindi tra V1 ed ogni vertice del ciclo esiste un percorso bianco al tempo $d[V1]$, ciò implica che (sempre per il teorema del p.b.) tutti i vertici contenuti nel ciclo saranno discendenti di V1 nella foresta generata della DFS.

Quindi avremo una situazione del genere:



Nel momento in cui la DFS visiterà v_k esplorerà tutti i suoi adiacenti e dato che v_1 è adiacente di v_k , avremo che quando la DFS esplora v_1 da v_k lo troverà di colore grigio perché dato che v_k è discendente di v_1 possiamo applicare il teorema della struttura a parentesi che ci assicura che vale:

$$d[v_1] < d[v_k] < f[v_k] < f[v_1]$$

Dato che nel momento in cui noi stiamo esplorando gli archi di v_k ci troviamo (strettamente) tra $d[v_k]$ e $f[v_k]$.

Quindi dato che $f[v_1]$ non è ancora arrivato, il colore di v_1 è per forza grigio.

Quindi abbiamo trovato un arco che connette un discendente con un suo antenato nella foresta generata della DFS, ossia un arco di ritorno, e quindi possiamo calcolare l'aciclicità di un grafo.

Corollario

Quindi verificare l'aciclicità di un grafo è equivalente a controllare la presenza di un arco di ritorno.

Aciclico(G)

L'algoritmo per il controllo dell'aciclicità del grafo estende la DFS per controllare se è presente un arco di ritorno.

```
Aciclico (G)           DFS_Visit_A( G, S )
for each u ∈ V do //init   c[S] = g
|   c[v] = b
for each u ∈ V do          for each u ∈ adj[S] do
|   if (c[v] = b) then     |   if ( c[u] = b ) then
|   |   ret = DFS_Visit_A(G,v)   |   |   ret =
|   if (ret = false) then   |   |   if ret = false then
|   |   return false         |   |   |   return false
|   |                           |   else if ( c[u] = g ) then
|   |                           |   |   return false
|   |
return true                  return true
```

La chiamata sul generico nodo S di **DFS_Visit_A** esplorare la parte di grafo raggiungibile da S e se trova un arco di ritorno, ossia se trova un nodo adiacente ad S che era già stato visitato (quindi col colore grigio) oppure lo trova al ritorno delle chiamate ai sottografi raggiungibili da S , fa tornare falso.

L'algoritmo **Aciclico** esattamente come succedeva per **DFS** esplorare tutte le sorgenti del grafo.

La complessità di questi due algoritmi è la stessa di una visita in profondità del grafo perché l'unica cosa che è stata aggiunta è il controllo sul colore grigio, che asintoticamente non comporta alcuna variazione alla complessità perché il controllo aggiunge un costo costante, quindi la complessità di **Aciclico** è lineare sulla dimensione del grafo.

ndra. questo algoritmo è utile per fare gli esercizi per lo scritto.

Ordinamento topologico

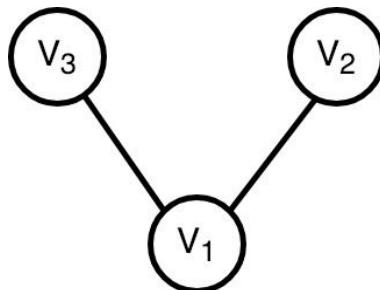
Checkpoint:<https://youtu.be/JEf9ok2ghWE?t=2290>

Una relazione di ordine si dice parziale se esiste almeno una coppia di elementi in questa relazione che non sono confrontabili fra loro.

Una permutazione può essere compatibile ad una relazione di ordine parziale se, intuitivamente, ogni elemento precedente nella permutazione è minore o uguale (rispetto alla relazione di ordine) degli elementi successivi nella permutazione.

Questo concetto è molto più chiaro con un esempio:

Se prendiamo questo grafo



e consideriamo v_1 minore di v_2 e di v_3 possiamo descrivere le permutazioni compatibili quelle che hanno v_1 prima di v_2 e di v_3 ; quindi avremo che:

- $\langle v_1, v_2, v_3 \rangle$ è compatibile
- $\langle v_1, v_3, v_2 \rangle$ è compatibile
- $\langle v_2, v_1, v_3 \rangle$ NON è compatibile
- $\langle v_3, v_2, v_1 \rangle$ NON è compatibile

Da questo esempio è facile considerare che se l'ordinamento è parziale esistono più modi di ordinare, a differenza di un ordinamento totale dove esiste uno ed un solo modo per effettuare un ordinamento.

Si dice relazione d'ordine parziale stretta una relazione che è:

- 1) Antiriflessiva
- 2) Antisimmetrica
- 3) Transitiva

Un grafo aciclico è sempre indotto da una relazione d'ordine parziale stretta.

Tutto ciò è utile per descrivere l'ordinamento topologico.

Un ordinamento topologico è una permutazione dei vertici del grafo che non viola i vincoli imposti dall'insieme degli archi del grafo.

In matematiche:

Sia π una permutazione di V .

π è un ordinamento topologico di $V \Leftrightarrow \forall (u, v) \in V, (u, v) \in E \mid u$ precede v in π

Ossia, preso un qualsiasi arco del grafo π deve contenere prima il vertice di inizio di questo arco e successivamente il vertice di fine dell'arco.

Come costruire ordinamento topologico

Se G è aciclico \rightarrow Esiste un vertice che non ha archi entranti

L'asserzione "un nodo v che non ha archi entranti" può essere tradotta algebricamente così:

$$\exists v \in V | \forall u \in V \quad (u, v) \notin E$$

Ricordando che G è finito, ossia che non consideriamo il caso in cui ci sono infiniti vertici, ragioniamo per assurdo: G aciclico e tutti i vertici hanno archi entranti; algebricamente:

$$\forall v \in V | \exists u \in V \quad (u, v) \in E$$

Prendiamo un generico vertice v_1 e sappiamo quindi che esiste un nodo v_2 che ha un arco entrante in v_1 ; Ciò genera due casi:

- 1) Se $v_2 == v_1$ abbiamo un cappio, quindi un ciclo, quindi una contraddizione.
- 2) Se $v_2 != v_1$ possiamo iterare lo stesso ragionamento per un qualsiasi v_3 (diverso da v_1 e v_2) che ha un arco entrante in v_2 .

Iteriamo questo ragionamento per tutti i nodi (finiti) del grafo, arrivando a v_n il quale deve soddisfare anch'esso la proprietà (perché vale $\forall v$!). Per soddisfare ciò v_n necessita di un v_{n+1} che dev'essere diverso da tutti gli n nodi del grafo ed ovviamente non può esistere. Quindi arriviamo a un assurdo.

Quindi vale che se G è aciclico, esiste un vertice che non ha archi entranti.

Se G è aciclico $\rightarrow G'$ sottografo di G è aciclico

Ragioniamo per assurdo: G aciclico e G' ciclico.

se G' è ciclico implica che contenga almeno un ciclo semplice, ossia un percorso con un nodo ripetuto.

Data la definizione di sottografo, ogni arco che si trova in G' si trova anche in G ; quindi ogni percorso di G' si trova in G .

Ma anche il percorso che determina il ciclo si dovrebbe trovare in G perché è parte di G' ma G è aciclico per ipotesi e quindi è una contraddizione.

Verifica della correttezza

Questo concetto insieme al concetto di prima, ci mostra come prendere i nodi dell'ordinamento topologico perché per costruire l'ordinamento topologico π posso iniziare a prendere il primo nodo che non ha archi entranti, che chiameremo v_1 .

Per questo motivo posso inserirlo in π .

$$\pi = v_1$$

Adesso quindi posso costruire questo sottografo:

$$G' = < V \setminus v_1, E' > \text{ con } E' = E \cap ((V \setminus v_1) \times (V \setminus v_1))$$

Ossia un sottografo con tutti i vertici di V tranne v_1 e tutti gli archi di E tranne quelli che entravano e/o uscivano da v_1 ; in altre parole, cancello v_1 dal grafo.

Per le proprietà sopra, G' è aciclico, quindi posso applicare la prima proprietà e dedurre che in G' esista un v_2 con nessun arco entrante.

$$\pi = v_1, v_2$$

v_2 si può trovare senza alcun arco entrante per due ragioni:

- Non aveva alcun arco entrante già prima della creazione di G' e quindi posso inserirlo dove voglio in π .
- Aveva un unico arco entrante che veniva da v_1 e quindi una volta cancellando v_1 , v_2 si trova senza alcun arco entrante e quindi deve venire dopo v_1 in π .

Posso iterare questo ragionamento per ogni vertice di G , finché non elimino tutti i vertici attraverso i sottografi, per arrivare a calcolare l'ordinamento topologico π .

GradoEntrante()

Il grado entrante di un vertice v è il numero di vertici che hanno v come adiacente, quindi che “entrano” in v .

Definiamo quindi una funzione che associa ad ogni vertice il suo grado:

Dato che la funzione è così definita $ge : V \mapsto \mathbb{N}$ e dato che il numero di vertici V è finito possiamo implementarla come un array.

```
GradoEntrante (G, ge)
  for each v ∈ V do //init
    |   ge[v] = 0

  for each v ∈ V do
    |   for each u ∈ adj[v] do
    |     |   ge[u] = ge[u]+1
```

La complessità è molto semplice da calcolare perché il primo ciclo for è $\Theta(|V|)$, il secondo è $\Theta(|E|)$ quindi tutta la funzione è lineare sulla dimensione del grafo, ossia $\Theta(|V| + |E|)$.

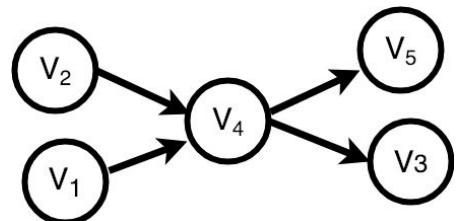
Init_Queue()

L'algoritmo per calcolare l'ordinamento topologico necessita di una struttura dati per conservare i vertici che hanno grado entrante uguale a zero, perché se un vertice ha grado entrante uguale a zero possiamo inserirlo nella posizione in cui è stato scoperto perché non viola il vincolo di precedenza.

Per chiarire questo concetto guarda questo grafo:

I vertici v_1 e v_2 hanno g.e. uguale a zero.

Fra tutti gli ordinamenti topologici possiamo selezionare:



- a) $\langle v_1, v_2, v_4, v_5, v_3 \rangle$
- b) $\langle v_2, v_1, v_4, v_5, v_3 \rangle$

Entrambi sono compatibili con la relazione di ordinamento topologico, quindi entrambi vanno bene come output per un eventuale algoritmo che calcola l'ordinamento topologico.

Il fattore interessante è che nel caso (a) stampiamo v_1 nella posizione 1 e v_2 nella posizione 2 e questo non viola il vincolo di precedenza.

Perciò per calcolare l'ordinamento topologico i nodi con g.e. = 0 sono importanti e li possiamo stampare nella stessa posizione in cui li abbiamo scoperti.

Per questo motivo si utilizza una coda la quale viene inizializzata e popolata dei nodi con grado entrante uguale a zero in questa funzione.

La complessità è lineare sul numero di vertici $\Theta(|V|)$.

```
Init_Queue (G, ge)
|   Q = ∅
|   for each v ∈ V do
|       |   if ge[v] = 0 then
|           |       |   Q = enqueue(Q, v)
|
|   return Q
```

OrdinamentoTopologico()

Adesso che abbiamo calcolato i vari valori dei gradi entranti per ogni vertice e che abbiamo inserito quelli con g.e. = 0 in una coda possiamo finalmente stampare l'ordinamento topologico.

L'idea per farlo è questa:

- 1) Estraiamo un nodo della coda (che ha g.e. = 0)
- 2) Lo stampiamo nella posizione corrente
- 3) Tutti gli archi uscenti di questo nodo dovranno essere stampati sicuramente dopo al nodo corrente.
- 4) Dato che i vertici raggiunti dal nodo corrente potrebbero avere altri nodi che lo raggiungono decrementiamo il grado uscente del vertice raggiunto di 1.
- 5) Se dopo il decremento, il vertice considerato ha grado entrante = 0 significa che tutti gli altri vertici che hanno un arco entrante nel vertice considerato sono stati già stampati, quindi posso stampare il vertice considerato.
- 6) Per stampare quel vertice, lo inserisco in coda e al prossimo giro del while verrà stampato.

```

Ord_topologico (G)
|   GradoEntrante (G, ge)
|   Q = Init_Queue(G, ge)
|
|   while( Q != Ø ) do
|       |   v = testa(Q)
|       |   stampa (v)
|
|       |   for each u ∈ adj[v] do
|       |       |   ge[u] = ge[u] - 1
|       |       |   if ge[u] = 0 then
|       |           |   Q = enqueue(Q, u)
|
|       |   Q = dequeue(Q)

```

La complessità di questo algoritmo eredita prima di tutto le complessità delle due funzioni che chiama, quindi consideriamo che:

- 1) GradoEntrante() ha una complessità di $\Theta(|V| + |E|)$
- 2) Init_Queue ha una complessità di $\Theta(|V|)$
- 3) Il corpo del for ha una complessità costante , quindi $\Theta(1)$
- 4) La testa del for viene eseguita su tutti gli archi uscenti, quindi tutto il for complessivamente è $\Theta(|E|)$
- 5) Infine l'operazione di estrazione della testa e di stampa lo devo fare per tutti i vertici del grafo, quindi avrà complessità di $\Theta(|V|)$
- 6) Quindi infine, dai punti (4) e (5) capiamo che tutto il while ha complessità $\Theta(|V| + |E|)$.
- 7) Infine: $\Theta(|V| + |E|) + \Theta(|V|) + \Theta(|V| + |E|) = \Theta(|V| + |E|) = \Theta(G)$, quindi è ottimale rispetto alla dimensione del grafo.

OrdinamentoTopologicoV2()

Checkpoint:<https://youtu.be/pV2MDZdy7xQ?t=2125>

Un altro modo per fare l'ordinamento topologico è diametralmente opposto a quello precedente: prima andavamo a cercare i nodi che vincolano altri nodi e mettevamo prima i nodi che vincolano e poi quelli che vengono vincolati.

Possiamo ragionare esattamente al contrario per raggiungere lo stesso obiettivo; ossia capiamo quali nodi non vincolano altri nodi e li poniamo alla fine dell'ordinamento topologico perché non è necessario che ci siano altri nodi dopo di lui.

Per capire quali sono i nodi che non vincolano nessuno pensiamo al caso di prima: un nodo x vincola un altro nodo y nella posizione dell'ordinamento topologico solo se esiste un **arco entrante** che collega y da x . Se l'arco entrante esiste l'ordinamento richiederà che x venga prima di y e quindi x vincola la posizione di y . Ne consegue che un nodo per non vincolare nessun'altro nodo non deve avere **archi uscenti**.

Stabilito ciò possiamo utilizzare una variante della DFS che mette in uno stack i nodi dopo averli visitati, ossia quando li colora di nero.

Usando questo stack possiamo costruire l'ordinamento topologico della fine, ossia quando anneriamo il nodo mettiamo il nodo dell'ultima posizione disponibile dell'ordinamento topologico, partendo quindi della fine e andando verso l'inizio.



```
OrdTopologico2(G)
  Init(G)  //colora tutti i nodi di bianco
  O = ∅
  for each v ∈ V do
    | if ( c[v] = b ) then
    |   | O = DFS_Visit'(G, v
  ,O)
    |
  return O
```

```
DFS_Visit' (G, v, Q)
  c[v] = g

  for each u ∈ adj[v] do
    | if ( c[u] = b ) then
    |   | Q = DFS_Visit'(G, u
  ,Q)
    |
  c[v] = n
  Q = push(Q, v)
  return Q
```

La complessità di questo algoritmo deriva strettamente dalla complessità della DFS in quanto le uniche istruzioni in più che vengono aggiunte sono:

- 1) $O = \emptyset$ che viene eseguita una volta sola
- 2) $Q = \text{push}(Q, v)$ che viene eseguita una volta per ogni nuovo vertice scoperto, quindi $\Theta(V)$ volte

Quindi la complessità di questo algoritmo è pari a $\Theta(V) + \Theta(|V| + |E|)$ che è uguale a $\Theta(|V| + |E|)$

Verifica della correttezza

Checkpoint: <https://youtu.be/pV2MDZdy7xQ?t=3441>

Premessa: indichiamo con il simbolo \prec_o la relazione di precedenza nell'ordinamento topologico, ossia se vale $x \prec_o y$ significa che x compare nell'ordinamento topologico prima di y .

L'output di questo algoritmo è una permutazione di tutti i vertici di G , e dobbiamo garantire che valga la proprietà di ordinamento topologico su questa permutazione; ossia deve valere:

$$\forall u, v \in V(u, v) \in E \Rightarrow u \prec_o v$$

(nda. è l'esatta definizione dell'ordinamento topologico, scritta in modo "abbreviato")

Ricordando la versione della DFS con i tempi, sappiamo che quando il nodo v viene colorato di nero ci troviamo alla fine della visita di v e quindi al tempo $f[v]$.

Possiamo mettere in relazione i tempi di fine visita con la relazione \prec_o nel seguente modo:

$$\forall u, v \in V, u \prec_o v \Leftrightarrow f[v] < f[u]$$

Questa relazione di equivalenza vale perché se u si trova prima di v dell'ordinamento topologico significa che ha arco che entra in v e quindi la DFS su u concluderà prima i suoi adiacenti, quindi v compreso, e poi u per il teorema della struttura a parentesi.

Quindi v finirà prima di u e quindi varrà $f[v] < f[u]$.

Essendo una relazione di equivalenza, posso scambiarla con la definizione di ordinamento topologico, avendo:

$$\forall u, v \in V(u, v) \in E \Rightarrow f[v] < f[u]$$

Se riusciamo a dimostrare questo, riusciamo a dimostrare che la permutazione in output è un ordinamento topologico.

In altre parole dobbiamo dimostrare che (alla fine della DFS), per qualsiasi arco del grafo, il tempo di fine visita del vertice di arrivo di un arco sia sempre inferiore del tempo di fine visita del vertice di partenza di quell'arco.

Per fare ciò consideriamo tutti i casi dei colori che una DFS_Visit può incontrare e due generici nodi u e v tali che u abbia un arco che lo collega a v .

- **Se u scopre v bianco**, non abbiamo problemi perché, per il teorema della struttura a parentesi sicuramente varrà questo:

$$d[u] < d[v] < f[v] < f[u]$$

- **Se u scopre v nero**, non abbiamo ancora problemi perché significa che l'esplorazione su v è già terminata o perché v viene scoperto da un'altro nodo z e quindi vale ciò:

$$d[u] < d[z] < d[v] < f[v] < f[z] < f[u] \rightarrow f[v] < f[u]$$



oppure, perché v fa parte di una sorgente diversa da u :

$$d[v] < f[v] < d[u] < f[u] \rightarrow f[v] < f[u]$$

- **Se u scopre v grigio**, significa che v è stato scoperto prima di u e non è ancora terminato, quindi u è un discendente di v e quindi deve terminare dopo la fine di u , ossia:

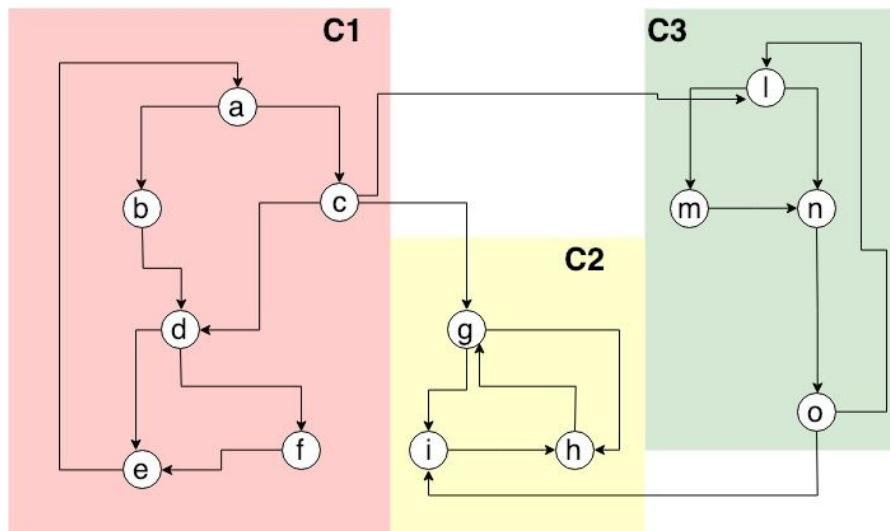
$$d[v] < d[u] < f[u] < f[v]$$

Ma questo accade solo se ci troviamo in un grafo ciclico e dato che l'ord.top. non è definito in un grafo ciclico questo caso non si presenta mai.

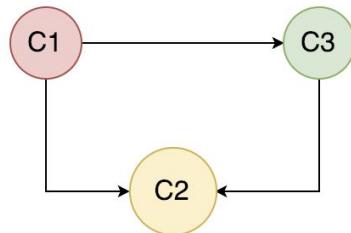
Questo dimostra che la permutazione costruita in quel modo è un ordinamento topologico.

Componenti Fortemente Connesse

Un grafo si dice fortemente connesso se da ogni nodo è possibile raggiungere ogni altro nodo del grafo. Un grafo, anche non fortemente connesso, può contenere al suo interno dei sottografi che sono fortemente connessi, come in questo esempio:



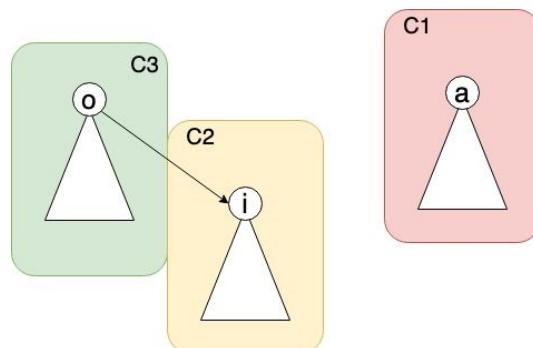
Ogni componente quindi è una classe di equivalenza rispetto alla relazione di raggiungibilità, perciò possiamo semplificare il grafo di esempio in un grafo compatto che viene chiamato **grafo delle componenti connesse**.



Il grafo delle componenti connesse è sempre **aciclico** anche se il grafo originario lo era, questo perché se ci fosse un ciclo, vorrebbe dire che ogni vertice che si trova in questo ciclo può raggiungere ogni altro elemento del ciclo e quindi genererebbe una nuova componente fortemente connessa.

I nodi che si trovano in una stessa componente connessa si trovano sicuramente nello stesso albero generato della DFS, ma il contrario non vale.

Ad esempio, se viene eseguita una DFS sul nodo "o" verrà generata una DFS del genere:



1) Proprietà dei “percorsi inclusi in una CFC”

Sia C una CFC e $u, v \in C$
 $\forall \pi | first(\pi) = u \text{ AND } last(\pi) = v \Rightarrow \pi \subseteq C$

Questa proprietà, in altre parole, dice che per ogni percorso che connette due vertici di una componente allora quello stesso percorso non può uscire da quella componente.

Dimostrazione

Dato che u e v appartengono alla stessa CFC esiste sicuramente un percorso che va da v a u ed uno che va da u a v . Questi percorsi potrebbero anche uscire della CFC, per adesso, il nostro scopo è dimostrare che ciò non accade mai.

Prendiamo quindi un generico nodo e chiamiamolo w di uno dei due percorsi e dimostriamo che si deve trovare per forza nella stessa CFC.

Ma dato che:

- 1) Per forza esiste un percorso da u a w perché ho scelto w proprio sul percorso che collega u a v .
- 2) Per forza esiste un percorso da w a u perché
 - a) w è collegato a v perché è sul percorso che collega u a v .
 - b) v è collegato a u per definizione di CFC

Quindi dato che u e w sono mutualmente raggiungibili, significa che u e w si troveranno nella stessa CFC.

Quindi dato che abbiamo fatto il discorso per un generico nodo w , possiamo concludere che tutto il percorso che collega u e w (ed il percorso che fa il contrario) si trova nella stessa CFC.

C.V.D.



- 2) Proprietà del “due vertici della stessa CFC non possono essere contenuti in due alberi diversi”

Dimostriamo che **al termine della DFS su G** vale ciò:

Sia C una CFC di G e $u, v \in C$, u e v ∈ stesso albero

Effettuando la DFS() su G sappiamo che prima o poi tutti i vertici saranno scoperti. Supponiamo che v_1 sia il primo vertice di C che viene scoperto.

Eseguendo la DFS su v_1 essa andrà a scoprire tutti i nodi che saranno raggiungibili da v_1 .

Per la proprietà di prima sappiamo che tutti i percorsi che verranno scoperti dopo $d[v_1]$ fanno parte della stessa CFC.

Per il teorema del percorso bianco, al tempo $d[v_1]$, tutti i nodi di C sono bianchi ed iniziano ad essere visitati dopo $d[v_1]$.

(nda. La proprietà di prima è fondamentale in questo passaggio perché è tramite quella che sappiamo che al tempo $d[v_1]$ tutti i nodi di C sono bianchi perché se c'è un percorso che parte da un nodo di C e ha almeno un nodo che non fa parte di C è possibile che quel nodo non sia bianco).

Quindi sempre per il teorema del percorso bianco sappiamo che la DFS genererà un albero, radicato in v_1 , che conterrà tutti i nodi di C.

Quindi non è possibile che due nodi della stessa CFC appartengano a due alberi diversi; algebricamente significa che **ogni CFC induce una relazione di equivalenza diversa**.

Calcolo delle CFC

Per le due proprietà descritte sappiamo che la DFS genera una foresta composta da uno o più alberi ed ogni albero contiene al suo interno una o più CFC.

Se volessimo “separare” queste CFC da ogni albero riuscirei a visitarle tutte.

Per capire il concetto di come fare questa separazione, ragioniamo nel caso in cui la DFS generi un solo albero.

Chiamiamo la radice di questo albero S e l'albero stesso lo chiamiamo T_s .

Sia v un nodo di T_s , dato che se un percorso si trova nella foresta sicuramente si trova nel grafo, sappiamo sicuramente che esiste un percorso da S a v, quindi:

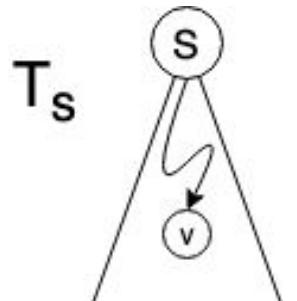
$$\text{se } v \in T_s \Rightarrow (S, v) \in \text{Reach}$$

Se riuscissimo a dimostrare che se v è un nodo T_s ed esiste (nel grafo) un percorso che parte da v ed arriva in S, ossia:

$$\text{se } v \in T_s \Rightarrow (v, S) \in \text{Reach}$$

avremmo dimostrato che S e v si trovano nella stessa CFC perché abbiamo trovato i due percorsi che assicurano la mutua raggiungibilità.

Per assicurarci di ciò sarà necessario utilizzare il grafo trasposto.



GrafoTrasposto

Sia G un grafo, chiamiamo G^T grafo trasposto di G se e solo se

- 1) contiene tutti e soli i vertici di G
- 2) ogni arco orientato di G esiste in G^T l'altro verso, formalmente:

$$\forall u, v \in V, \text{ se } (u, v) \in E \Rightarrow (v, u) \in E^T$$

In sostanza, si prende un grafo e si inverte la direzione di tutti gli archi.

La complessità di realizzazione di ciò è lineare sulla dimensione del grafo.

```

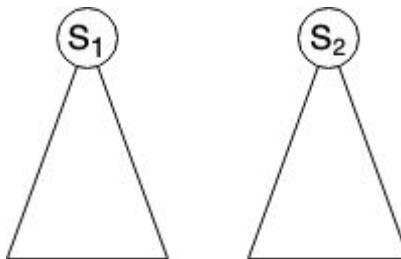
GrafoTrasposto (G)
VT = V
for each v ∈ VT do
| for each u ∈ adj[v] do
| | adj[u] = add( adjT[u], v)
return (< VT, ADJT > )

```

Ne consegue che se in G^T esiste un percorso da u a v esisterà un percorso da v a u nel grafo originale G .

Questo concetto ci è utile proprio perché, **dopo aver costruito l'albero T_s generato da una DFS, possiamo trasporre il grafo ed eseguire una nuova DFS sul grafo trasposto.** Confrontando i due alberi generati da queste due DFS potremmo dire quali nodi si trovano nella stessa CFC di S , perché la prima DFS verifica quali nodi sono raggiungibili da S e la seconda DFS verifica quali nodi raggiungono S , se un nodo soddisfa entrambe le raggiungibilità significa che è mutualmente raggiungibile da S e quindi appartiene alla sua CFC.

Dato che abbiamo fatto questo ragionamento su un singolo albero, è ancora valido se lo generalizziamo? Per semplicità, pensiamo al caso in cui esistano 2 alberi generati della DFS radicati in S_1 e S_2 :



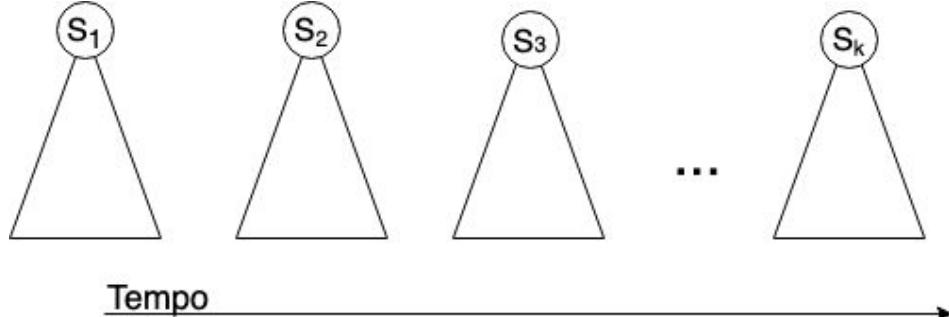
Se in uno dei due alberi ci fosse un arco di attraversamento che collega un nodo di Ts_1 ad un nodo di Ts_2 anche nella DFS del grafo trasposto troveremmo un arco di attraversamento da un nodo di Ts_2 ad un nodo di Ts_1 .

Questo è un problema perché per come abbiamo definito l'algoritmo per il calcolo di CFC avremmo trovato due nodi mutualmente raggiungibili che però fanno parte di due CFC differenti (perché si trovano in due alberi diversi) e questa è una contraddizione.

Ma analizziamo un po' di più questo problema: gli archi di attraversamento ci danno questo genere di problema solo se quando arrivano nell'altro albero trovano il nodo bianco.

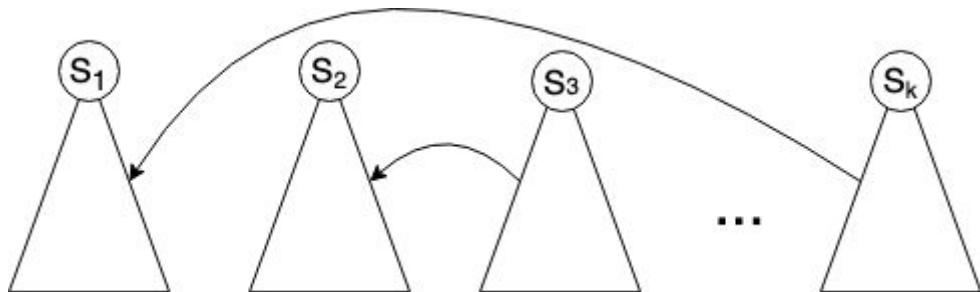
Se riusciamo a dimostrare che se esiste un arco di attraversamento esso arriva ad un nodo nero concludiamo che non ci crea problemi per il calcolo delle CFC.

Prendiamo k alberi e consideriamo i tempi di creazione di questi.



Quando costruiamo un albero tutti gli alberi che si trovano prima temporalmente sono già stati costruiti e quindi hanno tutti i nodi neri.

Gli archi di attraversamento possono solo andare nella direzione “contraria” al tempo per la definizione stessa di arco di attraversamento (nda. che è quando il nodo che visitiamo rispetta queste condizioni: $c[v] == n \text{ AND } d[v] < d[u]$).



Anche perché se da un albero A parte un arco che arriva in un albero A2 che si trova teoricamente dopo in senso temporale, il nodo dell'albero A2 è discendente dal nodo dell'albero A, quindi dovrebbe esso stesso trovarsi in A.

Inoltre consideriamo il k -esimo (l'ultimo) albero che la prima DFS ha generato; quando andiamo ad effettuare il grafo trasposto lo stesso k -esimo albero conterrà tutti e soli nodi che appartengono a .quell'albero, ossia non ci sono più eventuali archi di attraversamento (che per definizione vanno verso sinistra) perché se ci fossero nell'ultimo albero dopo la trasposizione significherebbe che ci stavano anche prima della trasposizione e ciò non è possibile per quello che abbiamo definito il paragrafo precedente.

Quindi se faccio partire la seconda DFS sull'ultimo albero eliminiamo il problema degli archi trasposti ma rimane l'ultimo problema: come selezionare le sorgenti in modo che la seconda DFS possa partire dall'ultima?

Quest'ordine di sorgenti può essere dedotto dalla prima DFS attraverso i tempi di fine visita dei nodi perché l'albero k -esimo è l'ultimo a terminare.

Per realizzare ciò la prima DFS utilizzerà una struttura dati d'appoggio che terrà memoria dei nodi in ordine decrescente di funzione del tempo di fine visita, ossia quando viene annerito un nodo esso viene inserito in cima; pertanto si utilizzerà uno stack.

Questo discorso l'abbiamo espresso per gli alberi generati della DFS ma è esattamente equivalente per i sottoalberi di ogni singola CFC che ogni albero può contenere.

```
CFC (G)
|   L = DFS_1(G)
|   GT = GrafoTrasposto(G)
|   DFS_2(GT, L)

DFS_2 ( G, L )
|   for v ∈ V do
|   |   c[v] = b
|   for each v ∈ L do
|   if (c[v] = b) then
|   |   dfs_visit(G, v)

DFS_1 (G)
|   c[v] = g
|
|   for each u ∈ adj[S] do
|   |   if ( c[u] = b ) then
|   |   |   Q = DFS_Visit' (G, u ,Q)
|   |
|   |   c[v] = n
|   |   L = push(Q,v)
|   return L
```

[Extra] Riassunto degli algoritmi (utili per lo scritto)

0) BFS

```
BFS(G,S)
  for v ∈ V do
    | c[v] = b
    | d[v] = ∞
    | p[v] = null

  Q = { S }
  c[S] = g
  d[S] = 0

  while Q ≠ ∅ do
    | v = testa(Q)
    | for each u ∈ adj[v] do
    |   | if (c[u]=b) then
    |   |   | Q = accoda ( Q , u )
    |   |   | c[u] = g
    |   |   | d[u] = d[v]+1
    |   |   | p[u] = v
    |
    |   Q = decoda (Q)
    |   c[v] = n
    |
```

Costruisci Percorso

```
CostruisciPercorso ( p, S, v)
  if ( v = S ) then
    | print S
  else
    |   CostruisciPercorso ( p, S, P[v] )
    |   print v
```

Percorso Minimo

```
Percorso Minimo (G, S, v)
  BFS(G,s)
  if ( p[v] ≠ null ) then
    CostruisciPercorso (p,S,v)
```

DFS

```
DFS ( G )
  for v ∈ V do
    | c[v] = b
    | p[v] = null

  tempo = 0
  for each v ∈ V do
    | if (c[v] = b) then
    |   | dfs_visit(G, v)
    |
```

DFS Visit

```
DFS_Visit ( G, S )
  c[s] = g
  d[s] = tempo
  tempo = tempo + 1

< VISITA S QUI - PRE ORDER >

  for each u ∈ adj[S] do
    | if ( c[u] = b ) then
    |   | p[u] = s
    |   | dfs_visit(G, u)

< VISITA S QUI - POST ORDER >

  c[s] = n
  f[s] = tempo
  tempo = tempo + 1;
```

DFS Visit A

```
DFS_Visit_A( G, S )
  c[s] = g

  for each u ∈ adj[S] do
    | if ( c[u] = b ) then
    |   | ret = DFS_Visit_A(G, u)
    |   | if ret = false then
    |     |   return false
    | if ( c[u] = g ) then
    |   | return false

  return true
```

DFS Visit ' (versione con pila)

DFSVisit' ritorna una pila che visitata dalla testa riporta i nodi in ordine di scoperta, ossia se v è più in alto di u nella pila significa che:

- 1) $d[v] < d[u]$
- 2) $f[v] > f[u]$

```
DFS_Visit' (G, v, Q)
c[v] = g

for each u ∈ adj[S] do
| if (c[u] = b) then
| | Q = DFS_Visit' (G, u, Q)
| |

c[v] = n
Q = push(Q, v)
return Q
```

Aciclico

```
Aciclico (G)
for each u ∈ V do
| c[v] = b

for each u ∈ V do
| if (c[v] = b) then
| | ret = DFS_Visit_A(G, v)
| if (ret = false) then
| | return false
| |

return true
```

Grafo Trasposto

```
GrafoTransposto (G)
VT = V
for each v ∈ VT do
| for each u ∈ adj[V] do
| | adj[u] = add( adjT[u], v)
return ( < VT, ADJT > )
```

Grado entrante

```
GradoEntrante (G, ge)
for each v ∈ V do //init
| ge[v] = 0
for each v ∈ V do
| for each u ∈ adj[v] do
| | ge[u] = ge[u]+1
```

Ordinamento topologico

```

Ord_topologico (G)
|   GradoEntrante (G, ge)
|   Q = Init_Queue(G, ge)
|
|   while( Q != Ø ) do
|       |   v = testa(Q)
|       |   stampa (v)
|       |
|       |   for each u ∈ adj[v] do
|       |       |   ge[u] = ge[u] - 1
|       |       |   if ge[u] = 0 then
|       |           |   Q = enqueue(Q,u)
|       |
|       |   Q = dequeue(Q)

```

CFC

```

CFC (G)
|   L = DFS_1(G)
|   GT = GrafoTransposto(G)
|   DFS_2(GT,L)

DFS_1 (G)
|   L = 0   |
|   for each v ∈ V do
|       |   if ( c[v] = b ) then
|       |       |   L = DFS1_Visit(G, u ,L)
|   return L

DFS1_Visit(G,v,L)
|   c[v] = g
|
|   for each u ∈ adj[v] do
|       |   if ( c[u] = b ) then
|       |       |   L = DFS1_Visit(G, u ,L)
|
|   c[v] = n
|   L = push(L,v)
|   return L

DFS_2 ( G, L )
|   for v ∈ V do
|       |   c[v] = b
|   for each v ∈ L do
|       |   if (c[v] = b) then
|           |       |   dfs_visit(G,v) //NORMALE DFS_VISIT

```

[Extra] Algoritmi trovati online che potrebbero essere utili

Visita di tutti i percorsi da un vertice u ad un altro vertice v usando la DFS

fonte: <https://www.geeksforgeeks.org/find-paths-given-source-destination/>

```
public void DFS_printAllPaths(Nodo sorgente, Nodo destinazione){  
    init_dfs();  
    List<Nodo> sorgenti = new ArrayList<>();  
    sorgenti.add(sorgente);  
    dfs_visit_printAllPaths(sorgente,destinazione,sorgenti);  
}  
  
private void dfs_visit_printAllPaths(Nodo sorgente, Nodo destinazione, List<Nodo> listaPathLocale ) {  
    setColoreAIlNodo(sorgente, Main.Colori.GRIGIO);  
  
    if(sorgente==destinazione){  
        System.out.println(listaPathLocale+"");  
        setColoreAIlNodo(sorgente, Main.Colori.BIANCO);  
        return;  
    }  
  
    for(Nodo nodo : sorgente.nodiAdiacenti){  
        if(getColoreDelNodo(nodo).equals(Main.Colori.BIANCO)){  
            listaPathLocale.add(nodo);  
            dfs_visit_printAllPaths(nodo, destinazione, listaPathLocale);  
            listaPathLocale.remove(nodo);  
        }  
    }  
    setColoreAIlNodo(sorgente, Main.Colori.BIANCO);  
}
```