



A.A. 2021-2022



Algoritmi e Strutture Dati

Università degli studi di Napoli

Federico II



Valentino Bocchetti



N86003405



vale.bocchetti@studenti.unina.it

Indice

1 Programma del corso	11
1.1 Materiale didattico	11
1.2 Prerequisiti	11
1.3 Sommatorie notevoli	12
2 Lezione 1	12
2.1 Algoritmo	12
2.1.1 Definizione di Algoritmo	12
2.1.2 Proprietà che un algoritmo deve possedere	13
2.2 Macchina di Turing	13
3 Lezione 2	14
3.1 Complessità degli algoritmi	14
3.2 Tempo di esecuzione e modello RAM	14
3.3 Un problema di conteggio	15
4 Lezione 3	17
4.1 Soluzione 2 al problema del conteggio	17
4.2 Soluzione 3	18
4.3 Soluzione 4	18
4.4 Conclusioni	19
4.5 Notazione asintotica	19
5 Lezione 4	20
5.1 Limite inferiore asintotico	20

5.2	Limite asintotico stretto	21
5.3	Interpretazione dei limiti asintotici	21
5.3.1	Esempi sulla notazione asintotica	21
5.4	Proprietà dei limiti asintotici	22
5.4.1	Transitività	22
5.4.2	Simmetria	23
5.5	Esempio d'uso della proprietà transitiva	23
5.6	Teorema sulla notazione asintotica	24
6	Lezione 5	24
6.1	Dimostrazione della terza implicazione	24
6.1.1	Caso $h(n)$ dall'alto	25
6.1.2	Caso $h(n)$ dal basso	26
6.1.3	Caso oscillatorio	26
6.2	Esempi di applicazione del teorema	26
6.3	Utilizzo della proprietà della monotonicità	27
6.4	Confutare una implicazione	28
7	Lezione 6	29
7.1	Tempo di esecuzione e ricorrenze	29
7.1.1	Somma massima di una sottosequenza contigua	29
8	Lezione 7	32
8.1	Soluzione 2	32
8.2	Soluzione 3	33

9 Lezione 8	35
9.1 Approccio incrementale e ricorsivo	35
9.1.1 Un primo albero di ricorrenza	35
9.1.2 Forma generale di equazioni di ricorrenza con funzioni a un solo parametro	36
9.1.3 Esempio 1 - Equazione di ricorrenza	37
9.1.4 Esempio 2 - Equazione di ricorrenza	40
9.1.5 Esempio 3 - Equazione di ricorrenza	41
9.1.6 Esempio 4 - Equazione di ricorrenza	44
9.2 Accenni sulle proprietà degli alberi binari	45
9.3 Algoritmi di ordinamento	46
9.3.1 Ordinamento di una sequenza	46
9.4 Insertion Sort	48
9.4.1 Analisi	49
9.4.2 Caso peggiore	50
9.4.3 Caso migliore	51
9.4.4 Caso medio	51
9.5 Merge Sort	52
9.5.1 Correttezza dell'algoritmo	52
9.5.2 Occupazione in memoria	53
9.5.3 Tempo di esecuzione	54
9.6 Selection sort	56
10 Lezione 13	57
10.1 Alberi binari completi	57

10.2 Alberi heap	59
10.3 Nuovo approccio al Selection Sort	60
10.3.1 Problema 1	60
10.3.2 Problema 3	62
11 Lezione 14	62
11.1 Ragionamenti sull'algoritmo di <code>Heapify</code>	62
11.2 Analisi di <code>HeapSort</code> e algoritmo completo	65
12 Lezione 15	66
12.1 Quick Sort	66
12.1.1 Partiziona	68
12.1.2 Analisi asintotica	70
13 Lezione 16	72
13.1 Caso medio	72
13.2 Caso migliore	73
13.3 Analisi	73
13.4 Calcolo del tempo medio	73
13.5 Tecnica per validare un'equazione di ricorrenza	76
13.5.1 Validazione per il caso base $n = 2$	77
13.6 Maggiorazione di una sommatoria	77
13.7 Conclusioni su QuickSort	78
14 Lezione 17	78
14.1 Problema generale sull'ordinamento	78

14.2 Alberi di decisione	79
14.3 Algoritmi di ordinamento e alberi di decisione	80
14.3.1 Proprietà degli alberi di decisione per gli algoritmi di ordinamento	80
14.4 Dimostrazione del teorema sull'ordinamento	81
14.4.1 Dimostrazione per il caso medio	82
15 Lezione 18	84
15.1 Strutture dati elementari	84
15.2 Operazioni su una struttura dati	84
15.3 Array (non) ordinato	85
15.4 Ricerca binaria	86
15.5 Liste	87
15.6 Definizione formale di lista e algoritmo di ricerca	88
16 Lezione 19	89
16.1 Operazioni su liste	89
16.2 Alberi Binari	92
16.3 Visite in profondità	92
17 Lezione 20	93
17.1 Visita in ampiezza (BFS)	93
17.2 Memoria aggiuntiva	94
17.3 Alberi binari di ricerca (ABR)	95
17.3.1 Definizione di ABR	95
17.3.2 Definizione ricorsiva	96
17.3.3 Operazione di ricerca	96

17.3.4 Operazioni di modifica	97
17.3.5 Inserimento	97
18 Lezione 21	98
18.1 Ricerca del minimo e del massimo	98
18.2 Ricerca del successore e predecessore	99
18.3 Cancellazione	102
19 Lezione 22	104
19.1 Alberi bilanciati di ricerca	104
19.2 Alberi perfettamente bilanciati	104
19.3 Alberi AVL	105
19.3.1 AVL minimi	106
19.4 Relazione tra altezza e numero di nodi	108
20 Lezione 23	110
20.1 Operazioni di modifica	110
21 Lezione 24	117
21.1 Operazione di cancellazione in AVL	117
21.2 Alberi Red-Black	120
22 Lezione 25	123
22.1 Dimostrazione che gli alberi Red-Black hanno altezza logaritmica sul numero di nodi	123
22.2 Inserimento in un albero Red Black	125
23 Lezione 26	130

23.1 Cancellazione negli alberi RB	130
24 Lezione 27	138
24.1 Conversione Algoritmo Ricorsivo in Iterativo	138
24.1.1 Introduzione	138
24.1.2 Memoria nella ricorsione	138
24.2 Algoritmo iterativo del meccanismo di ricorsione	139
24.3 Struttura dell'algoritmo	139
24.4 Esempi di traduzione	140
24.4.1 PrintTree	140
24.4.2 Altezza	141
25 Lezione 28	143
25.1 QuickSort	143
25.2 Algoritmo sadico	146
26 Lezione 29	148
26.1 Esercizio	148
26.2 Ricorsione in coda	150
27 Lezione 30	153
27.1 Grafi	153
27.1.1 Definizioni	153
27.1.2 Tipi di grafo	153
27.1.3 Grado di un vertice	154
27.1.4 Sottografo	154

27.1.5 Percorso	154
27.1.6 Raggiungibilità	155
27.1.7 Grafi ciclici e aciclici	155
28 Lezione 31	156
28.1 Grafi connessi e grafi completi	156
28.2 Altri concetti	156
28.3 Rappresentazioni concrete di grafi	156
28.3.1 Matrice di adiacenza	156
28.3.2 Liste di adiacenza	158
28.4 Visita in ampiezza	159
28.5 Algoritmo BFS	159
29 Lezione 32	160
29.1 Calcolo distanze e percorsi minimi	160
29.2 Correttezza della BFS	162
29.3 Algoritmo del percorso minimo	166
30 Lezione 33	167
30.1 Visita in profondità	167
30.2 Algoritmo DFS	168
30.3 Terminazione e complessità	168
30.4 Teorema della struttura a parentesi	170
31 Lezione 34	171
31.1 Foresta DF	171

31.2 Teorema del percorso bianco	173
31.3 Tipi di archi nella DFS	175
31.4 Verifica della ciclicità di un grafo	176
31.5 Ordinamento Topologico	178
32 Lezione 35	179
32.1 Ordinamento Topologico	179
32.2 Algoritmo del grafo entrante	180
32.3 Algoritmo con DFS	182
32.4 Componenti fortemente connesse	183
33 Lezione finale	185
33.1 Proprietà della CFC	185
33.2 Calcolo delle CFC	187

1 Programma del corso

- ▶ **Definizione e Analisi** di algoritmi
 - ◊ Correttezza;
 - ◊ Tempo di esecuzione
 - Notazione asintotica;
 - Tecniche di calcolo del tempo di esecuzione (es. Algoritmi di ordinamento);
- ▶ **Strutture** dati per rappresentazione di insiemi di dati
 - ◊ **Lista/Array** (non) ordinati;
 - ◊ **Alberi binari**;
 - ◊ **Alberi binari di ricerca**;
 - ◊ **Alberi bilanciati**;
- ▶ **Grafi**
 - ◊ Tipi di grafi;
 - ◊ Rappresentazione di grafi;
 - ◊ Algoritmi sui Grafi.

1.1 Materiale didattico

Su [docenti.unina](#) sono disponibili le lezioni frontali in *materiale didattico* nella cartella relativa ad **ASD1**.

Su [wpage](#) è possibile trovare informazioni sul corso, sugli esami e i lucidi delle lezioni.

1.2 Prerequisiti

1. **Analisi I** → Sommatorie, limiti, derivate e studio di funzioni;
2. **Algebra** → Funzione, relazione e loro proprietà, induzione (equivalente e ricorsione) e strutture algebriche;
3. **Programmazione 1** → **Programma/Computazione**, Strutture di controllo e **Iterazione/Ricorsione**.

1.3 Sommatorie notevoli

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

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

$$\sum_{i=1}^n i^3 = \left(\sum_{i=1}^n i \right)^2 = \left(\frac{n(n+1)}{2} \right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$$

$$\sum_{j=1}^n 1 = n - i + 1$$

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

Serie geometrica:

$$\sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1}$$

Ma se $0 < x < 1$ la serie converge e quindi

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

2 Lezione 1

2.1 Algoritmo

L'algoritmo è una descrizione non ambigua di una procedura. Sia A un algoritmo del problema P :

Per problema si intende l'insieme delle istanze $P = \{(I_1, O_1), \dots, (I_n, O_n)\}$

Ad esempio un problema potrebbe essere la differenza tra numeri interi ed una sua istanza la differenza tra 5 e 3.

Il nostro scopo sarà quello di definire una funzione:

$$T_A : I \rightarrow \mathbb{R}^T$$

ovvero che associa all'insieme delle istanze $I = (I_1, I_n)$ un valore numerico che dovrebbe rappresentare quanto tempo impiega l'algoritmo a risolvere il problema. Questo tempo chiaramente cresce con la complessità delle istanze.

Di nostro interesse è analizzare asintoticamente quando cresce la curva della funzione T_A al crescere della complessità delle istanze (quanto veloce arriva ad infinito).

2.1.1 Definizione di Algoritmo

Un **algoritmo** è una procedura ben definita per risolvere un problema:

Una sequenza di passi che, se eseguiti da un esecutore, portano alla **soluzione del problema**

La sequenza di passi che definisce un algoritmo deve essere **descritta in modo finito**, indipendente dal fatto che le computazioni (la sequenza di passi) possano essere infinite.

2.1.2 Proprietà che un algoritmo deve possedere

1. **Non Ambiguità** → tutti i passi che definiscono l'algoritmo devono essere non ambigui e chiaramente comprensibili all'esecutore;
2. **Generalità** → La sequenza di passi da eseguire dipende esclusivamente dal problema generale da risolvere, non dai dati che ne definiscono un'istanza specifica.
3. **Correttezza** → Un algoritmo è corretto se produce il risultato corretto a fronte di qualsiasi istanza del problema ricevuta in ingresso. Può essere stabilità, ad es., attraverso:
 - ▶ Dimostrazione formale (matematica);
 - ▶ Ispezione informale

La correttezza è la garanzia che l'algoritmo produca in output il valore corretto per ogni input.

4. **Efficienza** → Misura le risorse computazionali che esso impiega per risolvere un problema (preso un algoritmo e un modello computazionale si trova una relazione che data una misura ci permetta di dire quante risorse la macchina debba utilizzare per risolvere quella istanza). Alcuni esempi sono:
 - ▶ Tempo di esecuzione;
 - ▶ Memoria impiegata (limitata rispetto al tempo);
 - ▶ Altre risorse, come ad es. la banda di comunicazione.
5. **Semplicità** → L'algoritmo deve essere facile da capire, modificare e manutenere

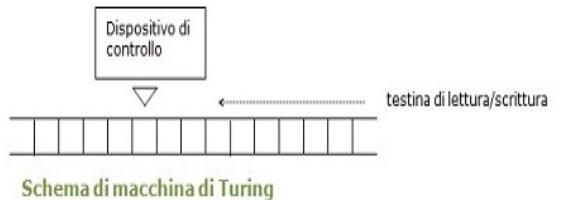
2.2 Macchina di Turing

Un noto modello computazionale è il modello della [Macchina di Turing](#).

È un modello astratto di un calcolatore che permette di eseguire delle operazioni su una struttura dati (nastro infinito) attraverso una testina (assimilabile ad un processore). Questo modello è dimostrato essere sufficientemente potente da poter calcolare tutto ciò che è calcolabile.

È composta da:

- ▶ Nastro di lunghezza infinita → In ogni cella può essere contenuta una quantità di informazione finita (un simbolo)
- ▶ Una testina + un processore + programma



Le possibili operazioni che può compiere in **una unità di tempo** sono:

- ▶ Leggere e Scrivere la cella di nastro corrente;
- ▶ Muoversi di una cella a sinistra o a destra;

- Restare ferma.

Nella macchina di Turing, l'algoritmo è praticamente l'insieme di regole, ovvero l'insieme di istruzioni che descrivono cosa la macchina deve fare in base all'input letto.

3 Lezione 2

3.1 Complessità degli algoritmi

La misura di complessità di una istanza è data dalla memoria utilizzata da quella istanza.

Ad es. per ordinare una sequenza di elementi la memoria è chiaramente proporzionale al numero di elementi della sequenza. Quindi, più elementi sono presenti e maggiore sarà il tempo impiegato dall'algoritmo per essere eseguito.

Come detto, il nostro interesse è analizzare asintoticamente quando velocemente cresce la curva della funzione $T_A : I \rightarrow \mathbb{R}^T$. Questa analisi delle prestazioni può avvenire in diversi modi:

- **Analisi sperimentale** → Si seleziona un calcolatore e si va ad eseguire il programma con il nostro algoritmo ed una batteria di test andando a misurare il tempo impiegato dall'algoritmo per ogni test. È ovviamente un metodo poco preciso, poiché dipendente dal calcolatore e il programma scelto.
- **Analisi asintotica** → Questo tipo di analisi trascende gli aspetti concreti (non misura il direttamente il tempo) e assume che il calcolatore sul quale viene effettuata la misurazione dell'algoritmo sia composto da operazioni elementari che impiegano la stessa unità di tempo (questa sarà infatti la nostra unità di tempo).

Lo scopo è quindi quello di dare delle misure ben precise per il comportamento di un algoritmo, anche se per alcuni algoritmi non è sufficiente la lunghezza dell'input, ma risulta rilevante anche come la sequenza è impostata → Per il problema di ordinamento infatti per alcuni algoritmi è rilevante anche lo stato della sequenza (se ordinata, poco ordinata o completamente disordinata); In questi casi va analizzato il caso migliore e il caso peggiore (da cui poi si potrà estrarre il caso medio).

3.2 Tempo di esecuzione e modello RAM

Il **tempo di esecuzione** di un programma può dipendere da vari fattori:

- HW su cui viene eseguito (velocità di calcolatore);
- **Compilatore/interprete** utilizzato → Compilatori differenti producono codice più o meno ottimizzato;
- Altri fattori come la **casualità**;
- **Tipo e dimensione dell'input** → Questa è anche l'unica dipendenza per noi ragionevole in quanto parte della definizione del problema.

Questo rende evidente che una misura di tempo (secondi) risulta inadatta alla misura di tempo di esecuzione. Ne consegue che

al fine di analizzare il **tempo intrinseco impiegato** da un algoritmo, serve un'analisi più astratta, impiegando un **modello computazionale**.

Un modello più vicino ad un calcolatore moderno è rappresentato dal modello computazionale **RAM**, che a differenza della Macchina di Turing (che è ad accesso sequenziale), ha accesso diretto.

Esso è composto da:

- ▶ **Memoria principale infinita**
 - ◊ Ogni cella di memoria può contenere una quantità di dati finita (posso scrivere un numero finito di simboli per ogni cella).
 - ◊ Impiega lo stesso tempo per accedere a ogni cella di memoria (e pertanto è più efficiente della Macchina di Turing)
- ▶ **Singolo processore + programma**
 - ◊ In 1 unità di tempo → Operazioni di lettura, passo di computazione elementare¹, scrittura;

3.3 Un problema di conteggio

Descrivere un algoritmo che accetta come input un intero $N \geq 1$ e produce in output il numero di coppie ordinate (i, y) tali che $i, j \in \mathbb{N}$ e $1 \leq i \leq j \leq N$

Es.

- ▶ Input $\rightarrow N = 4$;
 - ◊ $(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4)$
- ▶ Output $\rightarrow 10$.

Una possibile soluzione potrebbe essere la seguente:

```
1 int CONTACOPPIE(N)
2     RIS = 0          // Operazione di lettura: 1 operazione elementare
3     FOR i = 1 TO N DO    // Assegnamento di i e confronto con N: 2 operazioni elementari
4         FOR j = 1 TO N DO    // Assegnamento di j e confronto con N: 2 operazioni elementari
5             IF i <= j THEN    // due letture e un confronto: 3 operazioni elementari
6                 RIS = RIS + 1 // Una somma: 1 operazione elementare
7     RETURN RIS          // Scrittura: 1 operazione elementare
```

La correttezza di questo algoritmo è ovvia, però è anche abbastanza intuitivo che questo sia il modo peggiore di poter risolvere il problema precedente, poiché genera delle coppie inutili.

¹Per passo di computazione intendiamo operazioni come: addizione, moltiplicazione, assegnamento, confronto, accesso a puntatore, etc...

Risulta naturale notare delle imprecisioni nel conteggio delle operazioni elementari in alcune linee (ad esempio alla riga 7 oltre alla scrittura c'è anche una lettura), ma come sarà evidente, ciò non influisce sull'analisi (si potrebbe provare a rieseguire i seguenti calcoli con altri valori).

L'analisi precedente non basta a descrivere il comportamento dell'algoritmo (alcune linee sono ripetute più volte). Ne consegue che il calcolo totale del contributo di ogni linea sarà il prodotto tra il numero di *operazioni elementari* e il numero di ripetizioni della linea (ad esempio per la linea 2 e 7 il risultato sarà $1 \cdot 1 = 1$). Analizziamo meglio le altre linee di codice:

La riga n° 3 si ripete:

$$n + 1 \left(\sum_{i=1}^{n+1} 1 \right) \text{ volte}$$

NB: L'ultimo confronto è quello usato per uscire del ciclo, quindi il contributo totale della linea è $2(n + 1)$

La riga n° 4 invece $n + 1$ volte per ogni volta che si ripete il for precedente, quindi

$$\sum_{i=1}^n \left(\sum_{j=1}^{n+1} 1 \right)$$

Il totale quindi sarà $2(n + 1)n$

La riga n° 5 si ripete:

$$\sum_{i=1}^n \left(\sum_{j=1}^{n+1} 1 \right) = n^2, \text{ dunque } 3n^2$$

La riga n° 6 sicuramente si ripete un numero di volte compreso tra 0 e n^2 , quindi il valore della linea di codice è $0 \leq x \leq n^2$.

Ora non resta che analizzare la complessità totale:

$$\begin{aligned} T(N) &= 1 + 2(n + 1) + 2n(n + 1) + 3n^2 + x + 1 \\ &= 1 + 2n + 2 + 2n^2 + 2n + 3n^2 + x + 1 \\ &= 5n^2 + 4n + 4 + x \end{aligned}$$

È importante notare che il valore di x non influisce sul risultato dell'analisi poiché l'espressione sarà sempre del tipo $an^2 + bn + c$.

Dunque è una funzione parabolica.

Il precedente algoritmo avrà tempo di esecuzione **quadratico** sul valore dell'input.

4 Lezione 3

4.1 Soluzione 2 al problema del conteggio

Visto che una coppia vale come contributo al risultato solo quando il secondo elemento è maggiore del primo è evidente che possiamo migliorare il precedente algoritmo nel seguente modo:

```

1 int CONTACOPPIE(N)
2   RIS = 0
3   FOR i = 1 TO N DO
4     FOR j = 1 TO N DO
5       RIS = RIS + 1
6   RETURN RIS

```

Analizzando la complessità:

Linea	Costo unitario	Ripetizioni	Totale
2 e 6	1	1	1
3	2	$n + 1$	$2(n+1)$
4	2	$\sum_{i=1}^n \left(\sum_{j=1}^{n+1} 1 \right) = \sum_{i=1}^n (n - i + 2) =$ $\sum_{i=1}^n n - \sum_{i=1}^n 1 + \sum_{i=1}^n 2 = n^2 - \frac{n(n+1)}{2} + n =$ $\frac{n^2}{2} + \frac{3}{2}n$	$n^2 + 3n$
5	1	$\sum_{i=1}^n \left(\sum_{j=1}^{n+1} \right) 1 = \sum_{i=1}^n (n - i + 1) =$ $n^2 - \frac{n(n+1)}{2} + n = \frac{n^2}{2} + \frac{n}{2}$	$\frac{n(n+1)}{2}$

Avremo quindi:

$$T_2(N) = 1 + 2n + 2 + n^2 + 3n + \frac{n^2}{2} + \frac{n}{2} + 1 = \frac{3}{2}n^2 + \frac{11}{2}n + 4$$

Come il precedente algoritmo, anche questo algoritmo ha tempo di esecuzione **quadratico**, pertanto possiamo considerarli equivalenti dal punto di vista asintotico.

Ciò però non implica che impieghino lo stesso tempo, ma sul lungo periodo, il loro *peggioramento* segue la stessa andatura. Visto che dal punto di vista algoritmico non ci sono differenze, non abbiamo apportato nessun reale miglioramento.

4.2 Soluzione 3

Dalle precedenti analisi abbiamo notato che, fissato i , eseguiamo l'istruzione di incremento del risultato tante volte quanto il valore totale (che abbiamo calcolato) da aggiungere a **RIS**. Risulta pertanto logico aggiungere direttamente il risultato totale, scrivendo il seguente algoritmo:

```
1 int CONTACOPPIE(N)
2   RIS = 0 // (→ 1)
3   FOR i = 1 TO N DO // (→ 2n + 2)
4     RIS = RIS + (N - i + 1) // ( ∑i=1n 5 = 5n)
5   RETURN RIS // (→ 1)
```

Di conseguenza avremo che:

$$T_3(N) = 7n + 4$$

Questo tipo di funzione cresce linearmente e quindi è nettamente migliori ai precedenti algoritmi poiché, per la stessa crescita di \mathbb{N} , questa soluzione cresce più lentamente.

4.3 Soluzione 4

Il problema permette di ridurre ulteriormente il tempo di esecuzione.

Infatti sappiamo che per $i = 1$, il risultato è $\mathbb{N} - 1 + 1$.

Per $i = 1$, il risultato è $\mathbb{N} - 2 + 1$ e così via, fino ad arrivare ad \mathbb{N} a cui dobbiamo sommare 1 ai precedenti risultati. Pertanto:

$$RIS = \sum_{i=1}^N i = \frac{n(n+1)}{2}$$

Pertanto il nostro algoritmo si riduce a:

```
1 int CONTACOPPIE(N)
2   RIS = (N(N + 1)) / 2
3   RETURN RIS
```

Ne deduciamo che il problema di partenza è risolvibile a tempo costante, più precisamente in $T(\mathbb{N}) = 6$. Ciò significa che **qualsiasi istanza** è risolta con lo stesso tempo.

Era possibile arrivare a questa soluzione anche geometricamente. Infatti le coppie possono essere considerate come celle di una matrice quadrata²:

²Dove le righe rappresentano il primo valore della coppia e le colonne il secondo.

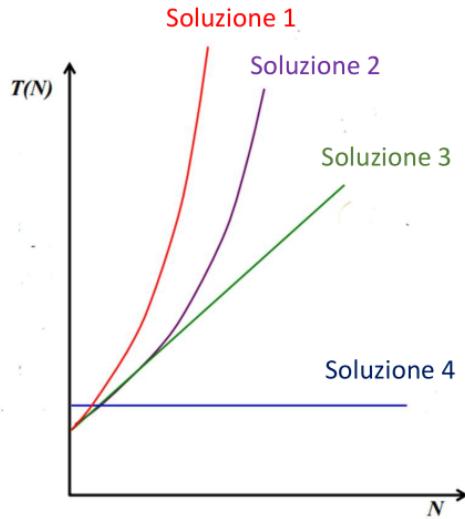
$$\begin{pmatrix} 1 & \dots & N \\ \vdots & \ddots & \vdots \\ N & \dots & N \end{pmatrix}$$

Le celle totali (ovvero tutte le coppie possibili) sono $N \times N$, ma quelle di nostro interesse sono tutte quelle sopra la **diagonale principale** (diagonale compresa).

Quindi essendo N elementi nella diagonale, sopra si trovano $\frac{N^2 - N}{2} + N = \frac{N(N+1)}{2}$.

4.4 Conclusioni

Riassunto dei tempi di esecuzione:



Ordine dei tempi di esecuzione:

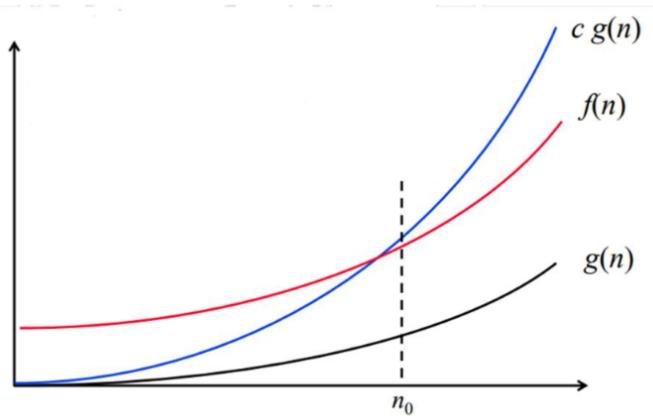
Supponiamo che 1 operazione atomica impieghi 10^{-9} secondi, la seguente tabella riporta i tempi impiegati dal calcolatore per risolvere algoritmi con tempo di esecuzione crescente al crescere dell'istanza

	1.000	10.000	100.000	1.000.000	10.000.000
N	1 μ s	10 μ s	100 μ s	1 ms	10 ms
$20N$	20 μ s	200 μ s	2 ms	20 ms	200 ms
$N \log N$	9.96 μ s	132 μ s	1.66 ms	19.9 ms	232 ms
$20N \log N$	199 μ s	2.7 ms	33 ms	398 ms	4.6 sec
N^2	1 ms	100 ms	10 sec	17 min	1.2 giorni
$20N^2$	20 ms	2 sec	3.3 min	5.6 ore	23 giorni
N^3	1 sec	17 min	12 giorni	32 anni	32 millenni

4.5 Notazione asintotica

Limite superiore asintotico

Diremo che una funzione $f(n)$ è in relazione *O grande* con una funzione $g(n)$ se $f(n)$ non cresce più velocemente di $g(n)$.



In simboli:

$$f(n) = \mathcal{O}(g(n))$$

Pertanto f cresce di meno o allo stesso modo di g (la si può considerare come la versione più permissiva dell' *o-piccolo*³ visto ad Analisi 1).

$$f(n) = \mathcal{O}(g(n)) \equiv \exists n_0 > 0, \exists c > 0, : \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

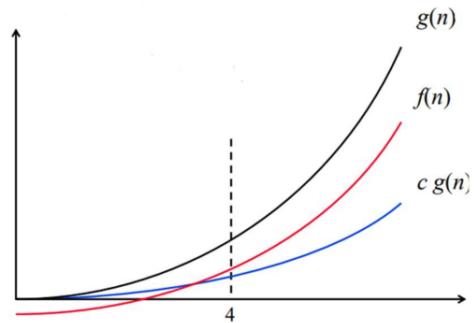
Di conseguenza più è piccolo n_0 e più grande dovrà essere la costante c , che nel caso del limite superiore asintotico è usualmente compreso in un intervallo tra 0 e 1.

5 Lezione 4

5.1 Limite inferiore asintotico

Limite inferiore asintotico

Diremo che una funzione $f(n)$ è in relazione *Omega grande* con una funzione $g(n)$ se $f(n)$ non cresce meno velocemente di $g(n)$.



In simboli:

$$f(n) = \Omega(g(n)) \equiv \exists n_0 > 0, \exists c > 0, : \forall n \geq n_0, f(n) \geq c \cdot g(n)$$

La costante c assume un valore maggiore di 1, poiché il suo scopo è quello di far superare alla funzione g la funzione f .

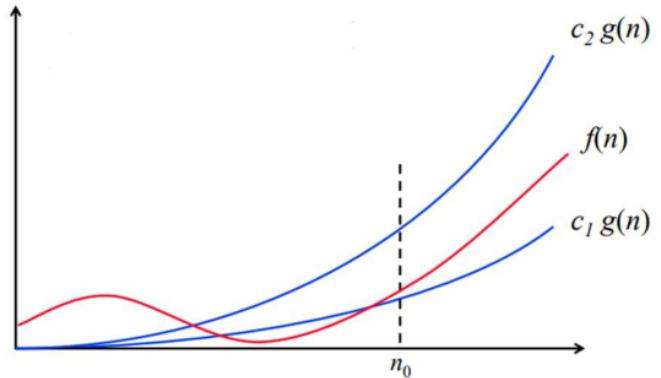
Risulta fondamentale che le costanti abbiano valore maggiore di 0 (in caso contrario il limite asintotico non sarebbe valido).

³Viene usato per individuare l'ordine di infinitesimo di una funzione rispetto ad una funzione campione, al tendere di x ad un determinato valore o all'infinito.

5.2 Limite asintotico stretto

Limite asintotico stretto

Diremo che una funzione $f(n)$ è *Theta* di una funzione $g(n)$ se $f(n) = \Theta(g(n))$ e $f(n) = \Omega(g(n))$



Diciamo che $g(n)$ è un limite asintotico stretto di $f(n)$

Quindi se valgono entrambe (le 2 funzioni crescono allo stesso modo), avremo la seguente relazione asintotica:

$$f(n) = \Theta(g(n)) \equiv f(n) = \Theta(g(n)) \wedge f(n) = \Omega(g(n)) \equiv \exists n_0 > 0, \exists c_1 > 0, \exists c_2 > 0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Si noti che n_0 è il valore maggiore tra gli n che soddisfano la relazione di *Omega* grande e *O* grande. È evidente che la condizione continua ad essere soddisfatta per un valore maggiore di n (infatti per definizione abbiamo $\forall n > n_0$).

Da n_0 in poi la funzione f segue l'andamento di g (è praticamente *intrappolata nel fascio creato dalle 2 costante con g*).

5.3 Interpretazione dei limiti asintotici

Perché ha senso confrontare gli algoritmi tramite le relazioni descritte in precedenza

Supponiamo che la funzione $f(n)$ sia un *Theta* di $g(n)$, ciò significa che esistono 2 costante che a partire da un n_0 continua a valere la precedente relazione.

Dal punto di vista computazionale significa che da un certo numero di istanza (n_0 , e da qui è evidente perché bisogna scegliere un valore positivo → non ha senso infatti un algoritmo con nessun input) se si eseguono 2 algoritmi (le 2 funzioni) su calcolatori con potenza di differente (la differenza di prestazioni dei calcolatori è data dalle costanti c_1 e c_2) essi terminano allo stesso momento

5.3.1 Esempi sulla notazione asintotica

1. Sia $f(n) = n$ e $g(n) = 2n$. Potremmo intuitivamente pensare che $f(n)$ cresca meno velocemente di $g(n)$, ma scegliendo $n_0 = 1$, $c_1 = \frac{1}{3}$ e $c_2 = 1$, possiamo far valere la seguente relazione:

$$\frac{1}{3}g(n) \leq f(n) \leq 1g(n) \iff \frac{2}{3}n \leq n \leq 2n$$

Ne consegue che $f(n) = \Theta(g(n))$ ⁴

2. Sia $f(n) = n^2$ e $g(n) = 2n$. È facile notare che le 2 funzioni non siano in relazione *Theta* tra loro, e quindi che non crescano allo stesso modo. Si dovrebbe avere $c_1 2n \leq n^2 \leq c_2 2n$.

- I. La prima relazione è banale da rendere vera e quindi risulta $f(n) = \Omega(g(n))$;
- II. Non banale è la seconda relazione. Supponendo che esista un c_2 che soddisfi la condizione $n^2 \leq c_2 2n \rightarrow$ Questo equivarrebbe a dire che $c_2 \geq \frac{n}{2}$ e ciò è assurdo.⁵

5.4 Proprietà dei limiti asintotici

5.4.1 Transitività

$$f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}h(n) \implies f(n) = \mathcal{O}(h(n))$$

1. Dimostrazione Per ipotesi abbiamo che:

$$\exists n_1 > 0, \exists c_1 > 0 : \forall n > n_1, f(n) \leq c_1 g(n)$$

e analogamente

$$\exists n_2 > 0, \exists c_2 > 0 : \forall n > n_2, g(n) \leq c_2 h(n)$$

Scegliendo $n_0 = \max\{n_1, n_2\}$ risulta che:

$$\exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n > n_0, f(n) \leq c_1 g(n) \wedge g(n) \leq c_2 h(n)$$

Da questo deduciamo che $c_1 g(n) \leq c_1 c_2 h(n)$.

La tesi si raggiunge scegliendo $c = c_1 c_2$ (ovviamente positivo). Abbiamo infatti:

$$\exists n_0 > 0, \exists c > 0 : \forall n > n_0, f(n) \leq c h(n)$$

⁴È importante notare che in questo caso si è fatto uso del buon senso per scegliere i valori delle costanti, ma come vedremo in seguito ci sono dei metodi ben precisi a tale scopo.

⁵È assurdo che esista una **costante** ≥ funzione crescente che tenda a ∞ ∀ n. (È possibile solo trovare una costante che per ogni punto è maggiore a tale funzione).

5.4.2 Simmetria

$$f(n) = \Theta(g(n)) \iff g(n) \leq \Theta(f(n))$$

1. Dimostrazione Per ipotesi abbiamo che:

$$\exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n > n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Da $c_1 g(n) \leq f(n)$ segue che $g(n) \leq \frac{1}{c_1} f(n)$.

In maniera analoga, da $f(n) \leq c_2 g(n)$ avremo che $\frac{1}{c_2} f(n) \leq g(n)$.

Ipotizziamo che $c_1^I = \frac{1}{c_2}$ e $c_2^I = \frac{1}{c_1}$ (entrambi ovviamente positivi), possiamo dire che:

$$\exists n_0 > 0, \exists c_1^I, c_2^I > 0 : \forall n > n_0, c_1^I f(n) \leq g(n) \leq c_2^I f(n)$$

come volevasi dimostrare⁶.

5.5 Esempio d'uso della proprietà transitiva

Questa proprietà risulta essere utile quando bisogna confrontare 2 funzioni molto *distanti* tra loro e quindi difficili da confrontare. Grazie alla transitività è possibile scegliere una funzione nel mezzo e confrontarla con entrambe (se risulta vera per una e falsa per l'altra allora non si arriva a nulla).

Consideriamo ad es. $f(n) = n^2 - 3n + 4$ e $h(n) = 2n^2 + 7n - 10$.

Lo scopo è prendere una funzione $g(n)$ affinché $f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}(h(n))$ così da avere per transitività $f(n) = \mathcal{O}(h(n))$.

Supponiamo che $g(n) = n^2$ e dimostriamo che $c_1 n^2 \leq n^2 - 3n + 4 \leq c_2 n^2$.

Scegliendo $c_2 = 1$ abbiamo:

$$n^2 - 3n + 4 \leq n^2, \text{ verificata per } n \geq \frac{4}{3}$$

mentre per $c_1 = \frac{1}{2}$ si ha $\frac{n^2}{2} \leq n^2 - 3n + 4 \iff 0 \leq \frac{n^2}{2} - 3n + 4 \iff 3n - 4 \leq \frac{n^2}{2}$ verificata $\forall n \geq 1$.

Analogamente si tratta il caso $g(n) = \mathcal{O}(h(n))$ e di conseguenza si ha anche che $f(n) = \mathcal{O}(h(n))$.

Questo esempio è banale, in quanto fin dall'inizio si vedeva che le 2 funzioni crescessero allo stesso modo, ma per funzioni esponenziali o logaritmiche potrebbe non esserlo, e quindi, risulta utile applicare questa proprietà transitiva come fatto in questo esempio.

⁶**NB:** Questa proprietà è valida **solo** per la relazione Theta.

5.6 Teorema sulla notazione asintotica

Siano $f(n)$ e $g(n)$ funzioni definite su un intervallo $[0, +\infty)$. Allora:

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

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

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0 \implies f(n) = \Theta(g(n))$$

Il precedente teorema è un utile metodo per dimostrare i limiti asintotici senza doversi preoccupare di calcolare le costanti.

Sfruttando la proprietà transitiva riusciamo anche a semplificare il limite. Ad es:

$$\lim_{n \rightarrow \infty} \frac{n^2 - 8n}{n^2 - 7} \text{ diventa } \lim_{n \rightarrow \infty} \frac{n^2 - 8n}{n^2} \wedge \lim_{n \rightarrow \infty} \frac{n^2}{n^2 - 7}$$

6 Lezione 5

6.1 Dimostrazione della terza implicazione

Se il $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$ con k costante, è intuitivo che nessuna funzione cresce più o meno dell'altra ma si mantengono ad una distanza proporzionale a k (se una cresce più velocemente dell'altra allora il limite del rapporto è 0 oppure ∞).

Partiamo dalla definizione di Θ

$$\Theta : \exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Supponiamo che $g(n)$ ad un certo punto sia sempre positivo (supposizione più realistica essendo una funzione di tempo), quindi è possibile dividere la relazione per $g(n)$.

Sia $h(n) = \frac{f(n)}{g(n)}$, dopo la divisione avremo che:

$$c_1 \leq h(n) \leq c_2$$

Ne segue che, essendo $h(n)$ sempre compreso tra 2 costanti positive:

$$\lim_{n \rightarrow \infty} h(n) = k, \text{ con } c_1 \leq k \leq c_2$$

(In modo del tutto analogo si trattano le altre due implicazioni).

Non resta che dimostrare che le due costanti c_1, c_2 che delimitano l'andamento di $h(n)$ esistano.

Supponiamo a tal scopo che $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$ e che quindi k sia asintoto orizzontale per il rapporto $h(n) = \frac{f(n)}{g(n)}$.

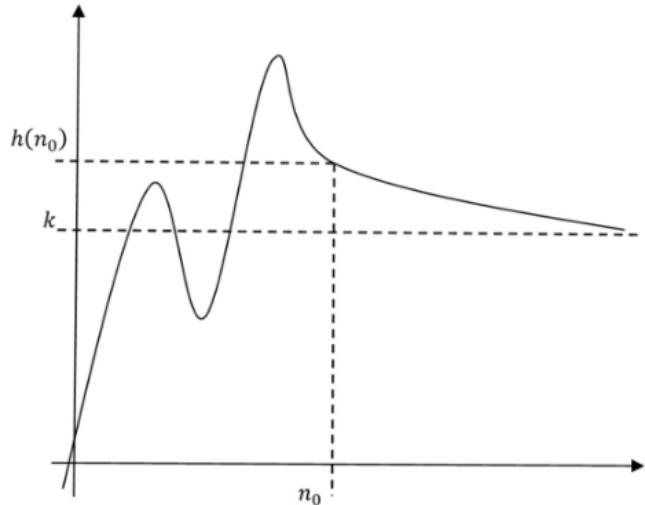
Per tale asintoto esistono solo 2 casi:⁷

- $h(n)$ tende a k dall'alto;
- $h(n)$ tende a k dal basso;

6.1.1 Caso $h(n)$ dall'alto

n_0 sarà il punto da cui la funzione $h(n)$ sarà sempre decrescente (è irrilevante il punto preciso di n_0 , l'importante è che esista e da quel punto in poi la funzione descresca).

Prendiamo ora la retta costante pari al valore di $h(n_0)$, ed ora possiamo osservare che da n_0 in poi il rapporto è tale che $\underbrace{k}_{c_1} \leq h(n) \leq \underbrace{h(n_0)}_{c_2}$



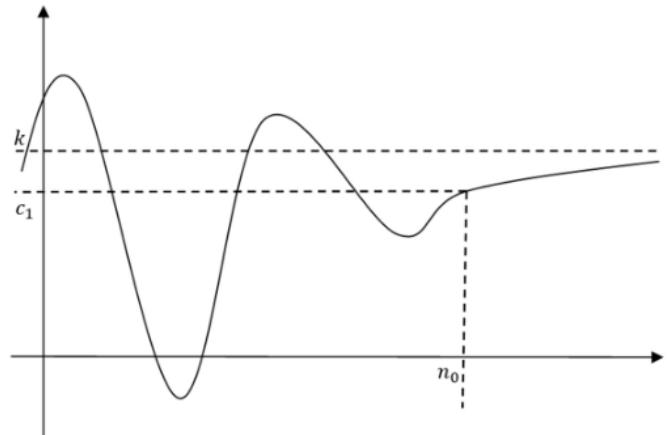
Le precedenti costanti sono state trovate avendo supposto che il rapporto $\frac{f(n)}{g(n)}$ esista e che la funzione (dopo lo studio della derivata prima di tale rapporto) sia decrescente per un intervallo $[n_0, +\infty)$.

⁷Esisterebbe anche un terzo caso, ma per esso k non è più un asintoto.

6.1.2 Caso $h(n)$ dal basso

Supposto che lo studio della derivata prima abbia portato alla conferma dell'esistenza di un intervallo per cui la funzione è sempre crescente abbiamo già trovato le nostre costanti

Infatti, per c_2 è ovvio che basti prendere l'asintoto k , mentre per c_1 bisogna scegliere un n_0 da $[n_0, +\infty)$ la funzione sia sempre crescente



Si noti che stavolta non basta solo che l'intervallo $[n_0, +\infty)$ sia crescente, ma deve anche essere sempre positiva (scegliere un punto in cui $\exists n \geq n_0 : h(n)$ sia negativa andrebbe contro la nostra definizione di Θ).

Una volta determinato n_0 , allora

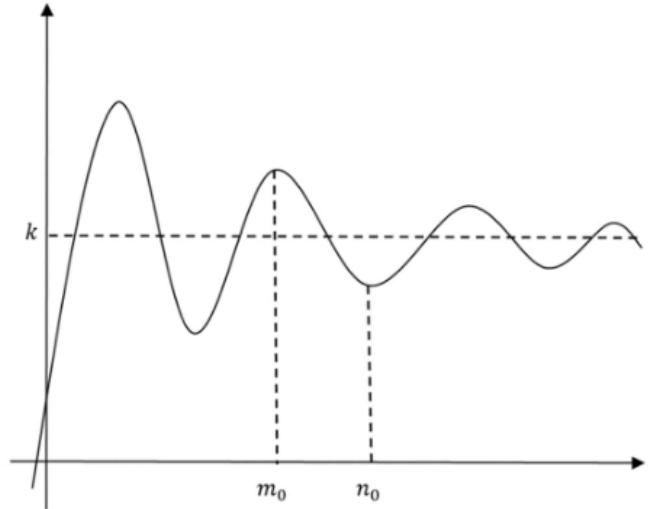
$$c_1 = h(n_0)$$

6.1.3 Caso oscillatorio

Se la situazione è quella rappresentazione nella figura a destra (l'ampiezza dell'oscillazione deve essere 0 ad un certo punto altrimenti $\lim_{n \rightarrow \infty} h(n) \neq k$) determinare le costanti è più complesso.

Bisogna scegliere un punto di minimo locale (n_0) e un massimo locale (m_0) in modo tale che i successivi minimi saranno sempre al di sopra del minimo locale scelto e i successivi massimi più bassi del massimo locale scelto.

n_0 sarà il punto maggiore (nel nostro caso è il minimo locale) tra i 2. Una volta scelto questi 2 punti le costanti saranno $c_1 = h(n_0)$ e $c_2 = h(m_0)$



Questo tipo di funzioni sono molto rare e pertanto nei nostri studi ci imbatteremo sempre e solo nei primi 2 casi

6.2 Esempi di applicazione del teorema

Importante è notare che

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \iff \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0 \text{ e } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0 \iff \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{k} > 0.$$

Questo dimostra che studiare il limite del rapporto o il limite del suo reciproco ci porta alla stessa conclusione.

1. Siano $f(n) = n$ e $g(n) = 4n - 10$.

$$\text{Avremo } \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} 4 - \frac{10}{n} = 4 \implies g(n) = \Theta(f(n)) \implies f(n) = \Theta(g(n))$$

Stabiliamo quindi le costanti per tale relazione. Sia $h(n) = 4 - \frac{10}{n}$ e facendone la derivata →

$$\frac{d}{dn} (4 - \frac{10}{n}) = \frac{d}{dn} 4 - \frac{d}{dn} \frac{10}{n} = -10 \frac{d}{dn} n^{-1} = \frac{10}{n^2}$$

Avremo quindi una funzione che è sempre crescente (essendo la derivata positiva).

Scegliamo ora un n_0 per cui la funzione sia sempre crescente e positiva (se ad es. scegliessimo 1 avremo $h(n) < 0$).

Sia $n_0 = 3$. Da questo ricaviamo che:

$$c_1 = h(3) = \frac{2}{3} \text{ e } c_2 = 4$$

Dunque $\forall n \geq 3, 2f(n) \leq g(n) \leq \frac{2}{3}f(n)^8$; ovvero $\frac{3}{2}g(n) \leq f(n) \leq \frac{1}{2}g(n)$

2. Se invece $f(n) = n$ e $g(n) = 4n + 10$ risulta $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 4$; quindi, le 2 funzioni sono sempre in relazione Θ tra di loro. In questo caso la funzione risulta decrescente essendo $h^I(n) = -\frac{10}{n^2}$

NB: In questi semplici esempi la funzione è sempre stata crescente o sempre decrescente in tutto l'intervallo $(-\infty, +\infty)$, ma nella maggior parte dei casi non è così → Bisogna sempre specificare l'intervallo.

6.3 Utilizzo della proprietà della monotonicità

$$f(n) = \Theta(g(n)) \implies \log(f(n)) = \Theta(\log(g(n)))$$

Per dimostrare la precedente implicazione utilizziamo la proprietà della monotonicità dei logaritmi, ovvero:

$$x \leq y \implies \log x \leq \log y$$

Da $f(n) = \Theta(g(n))$ per definizione $\exists n_0 > \exists c_1, c_2 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$, ma quindi $\log(c_1 g(n)) \leq \log(f(n)) \leq \log(c_2 g(n))$

A questo punto possiamo utilizzare la un'altra proprietà dei logaritmi:

⁸Ricordiamo che abbiamo studiato il rapporto $\frac{g(n)}{f(n)}$.

$$\log(x \cdot y) = \log(x) + \log(y) \rightarrow \log(c_1) + \log(g(n)) \leq \log(f(n)) \leq \log(c_2) + \log(g(n))$$

Esiste però la seguente proprietà:

$$\forall h(n), \log h(n) + k = \Theta(\log h(n))$$

Sfruttando la precedente proprietà:

$$\exists n_1 > 0, \exists c'_1, c'_2 > 0 : \forall n \geq n_1, c'_1 \log(g(n)) \leq \log c_1 + \log(g(n)) \leq c'_2 \log(g(n))$$

e

$$\exists n_2 > 0, \exists c''_1, c''_2 > 0 : \forall n \geq n_2, c''_1 \log(g(n)) \leq \log c_2 + \log(g(n)) \leq c''_2 \log(g(n))$$

Unendo le definizioni precedenti:

$$\exists n_3 = \max\{n_0, n_1, n_2\}, \exists c_1, c'_1, c_2, c''_2 > 0 : \forall n \geq n_3, c'_1 \log(g(n)) \leq \log c_1 + \log(g(n)) \leq \log(f(n)) \leq \log c_2 + \log(g(n)) \leq c''_2 \log(g(n))$$

Da cui per transitività del minore ed uguale otteniamo:

$$\exists n_3 > 0, \exists c'_1, c''_2 > 0 : \forall n \geq n_3, c'_1 \log(g(n)) \leq \log(f(n)) \leq c''_2 \log(g(n))$$

Quindi l'implicazione è dimostrata sotto la supposizione che $\forall h(n), \log(h(n)) + k = \Theta(\log(h(n)))$, ma ciò è evidente, essendo:

$$\lim_{n \rightarrow \infty} \frac{k + \log(h(n))}{\log(h(n))} = \lim_{n \rightarrow \infty} \frac{k}{\log(h(n))} + \lim_{n \rightarrow \infty} \frac{\log(h(n))}{\log(h(n))} = 1 > 0$$

6.4 Confutare una implicazione

Essendo il logaritmo una funzione, si potrebbe pensare di generalizzare l'implicazione precedente dicendo:

$$\forall h(n), f(n) = \Theta(g(n)) \implies h(f(n)) = \Theta(h(g(n)))$$

⁹Ciò ovviamente non basta. Inoltre se c_1 e/o c_2 sono compresi tra 0 e 1 il logaritmo sarà addirittura negativo.

¹⁰Ovviamente le costanti non sono le stesse per qualsiasi k , ma a noi interessa soltanto che queste esistano solotanto $\forall k \in \mathbb{Z}$.

Questo risulta però falso, e per confutarlo basta descrivere un esempio per cui la precedente implicazione non sia vera.

A tal proposito, siano $f(n) = n$ e $g(n) = 2n$ è evidente che siano in relazione Θ tra loro, infatti per $c_1 = \frac{1}{2}$ e $c_2 = 1$ abbiamo che:

$$\forall n \geq 1, \frac{1}{2}2n \leq n \leq 1 \cdot 2n$$

Sia la nostra funzione $h(n) = 2^n$; risulta quindi $2^{f(n)} = 2^n$ e $2^{g(n)} = 2^{2n}$, ma calcolando il limite di tale rapporto:

$$\lim_{n \rightarrow \infty} \frac{2^n}{2^{2n}} = \lim_{n \rightarrow \infty} \frac{2^n}{2^n \cdot 2^n} = \lim_{n \rightarrow \infty} \frac{1}{2^n} = 0$$

Essendo il limite una costante diversa da $k > 0$ concludiamo che le 2 funzioni non sono in relazione Θ tra di loro e dunque abbiamo dimostrato falsa l'implicazione (basta un controsenso per dire che la relazione è falsa, mentre per dire che è vera bisogna dimostrarla per ogni valido input).

7 Lezione 6

7.1 Tempo di esecuzione e ricorrenze

7.1.1 Somma massima di una sottosequenza contigua

Descriviamo un algoritmo che accetta in input una sequenza di numeri di una certa lunghezza $A = (a_1, a_2, \dots, a_n)$ dove $\forall i \in \mathbb{N}, a_i \in \mathbb{Z}$ ed un intero $N > 0$ che rappresenta la lunghezza della sequenza.

L'output di tale algoritmo è invece un numero $V \in \mathbb{N}$ che rappresenta il massimo valore tra le somme di tutte le sottosequenze contigue di A (esempio, una sottosequenza contigua di $a_1 a_2 a_3 a_4 a_5$ è $a_2 a_3$ oppure a_4 , mentre non è sottosequenza contigua $a_1 a_2 a_5$). Formalizziamo meglio il problema:

► Input →

- ◊ Una sequenza $A = (a_1, a_2, \dots, a_n)$ con $a_i \in \mathbb{Z}, \forall i \in \mathbb{N}$;
- ◊ Un intero $N \geq 1$;
- ◊ Esempio → $(2, -4, 8, 3, -5, 4, 6, -7, 2), N = 9$

► Output →

- ◊ Un intero V tale che $V = \sum_{k=1}^j a_k$ dove $1 \leq i \leq j \leq N$ e V è il più grande possibile
- ◊ Tutti gli elementi nella sommatoria devono essere contigui nella sequenza in input
- ◊ L'output del precedente esempio → $8 + 3 - 5 + 4 + 6 = 16$

1. Soluzione 1 Un primo approccio potrebbe essere quello di trovare la somma di tutte le sottosequenze e restituire la maggiore.

Sappiamo che il numero di sottosequenze è finito per sequenze contigue. Inoltre, tutte le sottosequenze contigue non vuote sono in corrispondenza biunivoca con la propria vista per il CONTACOPPIE, ne consegue che il numero di sottosequenze contigue, compresa quella vuota è pari a $\frac{N(N+1)}{2} + 1$

Abbiamo quindi decomposto il problema in:

- ▶ Generare tutte le sottosequenze;
- ▶ Sommare tutti gli elementi di una sottosequenza;

Si noti che poiché il valore di una sottosequenza vuota è 0, possiamo dire con certezza che $V \geq 0$ per qualsiasi sottosequenza.

Un possibile algoritmo è il seguente:

```

1 int MAXSUM(A,N)
2   V = 0
3   FOR i = 1 TO N DO
4     FOR j = 1 TO N DO
5       // Questo è l'unico blocco di codice che differisce dalla soluzione 2
6       // svolta per il CONTACOPPIE
7       SUM = 0
8       FOR k = i TO j DO
9         SUM = SUM + A[k]
10      IF SUM > V THEN
11        V = SUM
12
RETURN V

```

Ci si aspetta che questo algoritmo abbia almeno un $\Theta(n^2)$ vista la similitudine con la soluzione 2 di CONTACOPPIE¹¹.

Possiamo sfruttare pertanto il fatto che il blocco viene eseguito tante volte quanti i e j generati, ovvero $\frac{N(N+1)}{2}$ volte, per semplificare l'analisi dell'algoritmo (anche se andrebbe svolta nella sua totalità) andando a studiare le linee codice da 5 a 9.

La linea 5 ha un costo di 1 e viene eseguita un numero di volte pari a:

$$\underbrace{\sum_{i=1}^N \sum_{j=1}^N 1}_{\text{primo e secondo for}} = \sum_{i=1}^N (N - i + 1) = \frac{N(N+1)}{2}$$

In maniera analoga, le linee 8 e 9 hanno un contributo totale di circa $4 \frac{N(N-1)}{2}$

Per le linee 6 e 7 basta notare che una volta fissato i e j la differenza tra la testa del *for* ed il corpo è 1¹².

¹¹Più precisamente se il blocco in analisi risulta essere costante allora $T(N) = \Theta(n^2)$ altrimenti sarà ovviamente di più.

¹²In pratica il ciclo *for* viene eseguito una volta in più del corpo dovendo eseguire la condizione di uscita.

Tale differenza avviene ogni volta che genero una coppia $(i, j) \rightarrow$ la linea 6 viene eseguita, alla fine dell'algoritmo, $\frac{N(N-1)}{2}$ volte in più della linea 7.

La precedente intuizione ci permette di semplificare ulteriormente l'analisi, infatti, essendo già nell'ordine del quadratico tale differenza non comporta differenze dal punto di vista asintotico. Possiamo pertanto analizzare la linea 7 (che ha contributo singolo di 4) e viene eseguita il seguente numero di volte:

$$\begin{aligned} \sum_{i=1}^N \sum_{j=1}^N \sum_{k=1}^N 1 &= \sum_{i=1}^N \sum_{j=1}^N (j - i + 1) = \sum_{i=1}^N \left(\underbrace{\sum_{j=1}^N j}_{\sum_{j=i}^N j} - \underbrace{\sum_{j=1}^N 1}_{\sum_{j=i}^N i} + \underbrace{\sum_{j=1}^N 1}_{\sum_{j=i}^N 1} \right) \\ &\quad \sum_{j=i}^N j = \sum_{j=1}^N j - \sum_{j=1}^{i-1} j = \frac{N(N+1)}{2} - \frac{(i-1)i}{2} \\ &\quad \sum_{j=i}^N i = i \sum_{j=i}^N 1 = i(N - i + 1) \\ &\quad \sum_{j=i}^N 1 = N - i + 1 \end{aligned}$$

$$\sum_{i=1}^N \left(\frac{\sum_{j=1}^N j}{\frac{N(N+1)}{2} - \frac{(i-1)i}{2}} - \frac{\sum_{j=1}^N i}{i(N-i+1)} \frac{\sum_{j=1}^N 1}{N-i+1} \right) =$$

$$\begin{aligned} \sum_{i=1}^N \frac{N(N+1)}{2} - \sum_{i=1}^N \frac{i^2}{2} + \sum_{i=1}^N \frac{i}{2} - \sum_{i=1}^N i \textcolor{violet}{N} + \sum_{i=1}^N i^2 - \sum_{i=1}^N N i + \sum_{i=1}^N (N - i + 1) &= \\ \frac{N^2(N+1)}{2} - \frac{1}{2} \cdot \frac{N(N+1)(2N+1)}{6} + \frac{1}{2} \frac{N(N+1)}{2} - \frac{N^2(N+1)}{2} + \frac{N(N+1)(2N+1)}{6} - \frac{N(N+1)}{2} + \end{aligned}$$

$$\frac{N(N-1)}{2}$$

$$= \frac{1}{2} \left(\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} \right) - \frac{1}{2} \left(\frac{N^2}{2} + \frac{N}{2} \right) + \frac{N^2}{2} - \frac{N}{2} = \frac{1}{6} N^3 + \frac{1}{2} N^2 - \frac{2}{3} N$$

Ciò conclude la nostra analisi \rightarrow L'algoritmo descritto ha tempo di esecuzione $T(N) = \Theta(n^3)$

8 Lezione 7

8.1 Soluzione 2

Un notevole miglioramento al precedente algoritmo può essere fatto notando che molti calcoli vengono ripetuti (sprecando tempo). Il problema è legato al calcolo di una sottosequenza data (l'ultimo for della soluzione precedente). Infatti possiamo osservare che:

$$\sum_{k=1}^j A[k] = \sum_{k=1}^{j-1} j - 1 A[k] + A[j]$$

Ma $\sum_{k=i}^{j-1} A[k]$ è la quantità già presente in **SUM** ed è quindi già stata calcolata. Pertanto:

$$\sum_{k=i}^{j-1} SUM + A[k]$$

È evidente che una linea di questo tipo ha contributo costante (ne traiamo vantaggio delle iterazioni precedenti).

La suddetta osservazione non può essere utilizzata così come è → Bisogna comprendere quando azzerare **SUM** e sotto quali condizioni si può sfruttare il precedente algoritmo.

Fintanto che l'indice **i** resta sempre lo stesso si vuole aggiornare il precedente valore di **SUM**, mentre quando varia allora **SUM** va azzerato poiché ci si trova in una nuova sottosequenza.

Questa idea può essere formalizzata nel seguente algoritmo:

```
1 int MAXSUM(A,N)
2   V = 0 // (→ Θ(1))
3   FOR i = 1 TO N DO
4     SUM = 0 // La riga 3 e 4 hanno contributo (Θ(n))
5     FOR J = 1 TO N DO
6       SUM = SUM + A[i]
7       IF SUM > V THEN
8         V = SUM // Le righe 5-8 hanno contributo (Θ(n2))
9   RETURN V // (→ Θ(1))
```

Questo algoritmo impiega tempo $T(N) = \Theta(n^2)$ ed è quindi un miglioramento significativo dal punto di vista asintotico della soluzione 1

8.2 Soluzione 3

Del precedente algoritmo sicuramente si può dire che sia ottimale per il calcolo della somma di ogni sottosequenza contigua; infatti non è possibile crearne uno migliore se si vuole calcolare il valore di tutte le sottosequenze poiché, il numero di sottosequenze in una sequenza è di per sé quadratico.

Tuttavia il nostro algoritmo non richiede il calcolo di tutti i valori (è stata una nostra scelta questo approccio → l'algoritmo ottimale di un problema non è detto che sia ottimale anche su un altro problema). Si noti infatti che è inutile valutare tutte le sottosequenze e di conseguenza non è necessario esplorare del tutto la nostra istanza attraverso un algoritmo di brute force,¹³ che quasi mai sono esaustivi.

Attraverso l'analisi della nostra sequenza possiamo arrivare alla seguente intuizione:

Assumiamo che $\text{SUM}(i, j - 1) \geq 0$ e ciò significa che tutte le sottosequenze che vanno da i a $j - 1$ sono non negative → $\text{SUM}(i, k) \geq 0 \forall k : i \leq k \leq j$

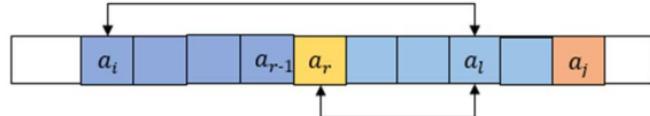


$$\text{NB: } \text{SUM}(i, k) \iff \sum_{k=i}^{j-1} a_k$$

A questo punto è evidente che sommando anche il valore presente nella cella j (a_j) si avrà o un risultato non ancora negativo oppure un risultato positivo:

Proprietà 1

$$\text{SUM}(i, j) \geq 0 \implies \text{SUM}(r, l) \leq \text{SUM}(i, l) \quad \forall i \leq r \leq l \leq j.$$



Ovvero, nessuna sottosequenza di una sequenza non negativa può essere migliore di quella di partenza. Infatti:

$$\sum_{\substack{k=1 \\ \geq 0}}^j a_k = \underbrace{\sum_{k=1}^{r-1} a_k}_{\geq 0} + \sum_{k=r}^j a_k \implies \sum_{k=r}^j a_k = \underbrace{\sum_{k=1}^j a_k}_{\geq 0} - \underbrace{\sum_{k=1}^{r-1} a_k}_{\geq 0} \implies \sum_{k=r}^j a_k \leq \sum_{k=i}^j a_k$$

Da questa proprietà capiamo che possiamo ignorare tutte le sottosequenze di una sequenza non negativa.

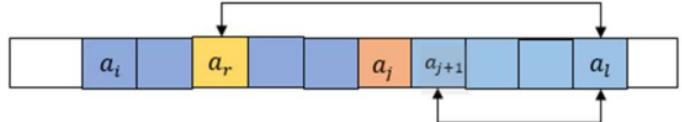
¹³Si intende un algoritmo che offre una soluzione esaustiva → Sonda tutti gli elementi possibili per quella istanza (nel nostro caso tutte le sottosequenze).

Proprietà 2

$$\text{SUM}(i, j) \leq 0 \implies \text{SUM}(r, l) < \text{SUM}(j+1, l) \quad \forall i \leq r \leq j \wedge \forall j+1 \leq l \leq n$$

Quindi se da i a $j - 1$ tutte le sequenze sono ≥ 0 ma la sequenza da i a j è negativa significa che qualunque sottosequenza che parte da $r \leq j$ ed arriva a $l > j$ è sicuramente minore della sequenza che va da j ad l

Infatti:



$$\sum_{k=j+1}^l a_k = \sum_{k=i}^l a_k - \underbrace{\sum_{k=i}^j a_k}_{< 0} \implies \sum_{k=j+1}^l a_k > \sum_{k=i}^l a_k$$

Quindi la proprietà 2 oltre a dirci di dover conservare la sequenza da i a $j - 1$ ci consente anche di ignorare altre sottosequenze.

Dopo queste analisi è abbastanza intuitivo che il codice seguente avrà complessità lineare poiché muoverò sempre in avanti:

1. Se $\text{SUM}(i, j) \geq 0$ allora è lo stesso i e incremento j
2. Se $\text{SUM}(i, j) < 0$ allora parto direttamente da $j + 1$ e vado avanti.

Dunque, i salti sono giustificati dalla proprietà 2 e il fatto di non dover esaminare le sequenze intermedie dalla proprietà 1. Ma allora l'algoritmo finale sarà il seguente:

```

1 int MAXSUM(A, N)
2   V = 0 // (→ i = 1)
3   j = 1 // (→ j = i)
4   SUM = 0
5   WHILE j <= N DO
6     SUM = SUM + A[j]
7     // (→ Caso 2 (V > SUM) :)
8     IF SUM <= 0 THEN
9       // (→ i = j + 1)
10      SUM = 0 // (→ Simula il salto)
11      // (→ Caso1)
12    ELSE IF SUM > V THEN
13      V = SUM
14      j = j + 1
15  RETURN V

```

Si noti che tutte le soluzioni di questo problema possono essere adattate affinché memorizzi anche gli indici per cui la sottosequenza è quella massima; Infatti, basta salvare gli indici di quando vado ad aggiornare \mathbf{v} .

Risulta banale che questo algoritmico abbia $T(N) = \Theta(n)$ è ottimale poiché supponendo di avere un input con valori positivi, è evidente che **MAXSUM** è la somma di tutti i positivi, devo scorrere la sequenza sia per controllare che siano positivi, sia per calcolarne la somma.

9 Lezione 8

9.1 Approccio incrementale e ricorsivo

Insertion Sort è una tecnica di soluzione incrementale (ottenuta pezzo per pezzo) → Parte da un array ordinato di dimensione 1 e ad ogni iterazione incrementa la porzione ordinata fino ad n .

In tutti gli algoritmi visti finora, siamo arrivati alla soluzione decomponendo il problema in sotto problemi più semplici da risolvere (*divide et impera*), i quali erano diversi tra loro (come ad es. generare coppie e sommare una sequenza).

Ad esempio, se ho una sequenza di lunghezza N e devo ordinarla, posso ridurre tale istanza in problemi di ordinamento di una sequenza di lunghezza minore di N . Questo tipo di approccio ricorsivo prende il nome di *divide et impera* (ne è un esempio il merge sort).

9.1.1 Un primo albero di ricorrenza

Prendiamo come esempio la funzione del fattoriale: $n = \begin{cases} 1 & \text{se } n = 1 \\ n \cdot (n - 1)! & \text{altrimenti} \end{cases}$

Che sarà in termini di equazioni di ricorrenza: $T_F(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T_F(n - 1) + \Theta(1) & \text{altrimenti} \end{cases}$

Per comprendere il numero di chiamate ricorsive che la suddetta funzione compie bisogna analizzare la struttura utilizzando l'albero di ricorrenza¹⁴.

¹⁴Un albero dove i nodi rappresentano le chiamate ricorsive e ad ogni nodo è associato il numero di ricorrenza e il contributo locale di tale nodo (abbiamo in questo modo anche il contributo di ogni livello dell'albero).

Nel nostro caso avremo il seguente albero di ricorrenza:

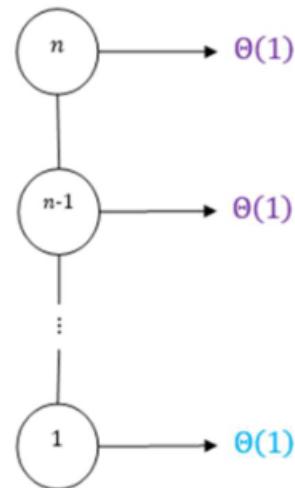
C'è una corrispondenza tra il livello dell'albero e l'output del programma. Avremo infatti al livello i un input $n - i$.

La foglia invece, per essere definita tale deve ricevere l'input del caso base (si noti infatti che il $\Theta(1)$ della foglia non è lo stesso $\Theta(1)$ degli altri livelli).

Il livello è foglia se $n - i = 1 \implies i = n - 1$.

A questo punto possiamo calcolare, partendo dal livello 0 il tempo di esecuzione della funzione fattoriale:

$$T_F(n) = \sum_{i=0}^{n-1} \Theta(1) + \Theta(1) = \Theta(n)$$



9.1.2 Forma generale di equazioni di ricorrenza con funzioni a un solo parametro

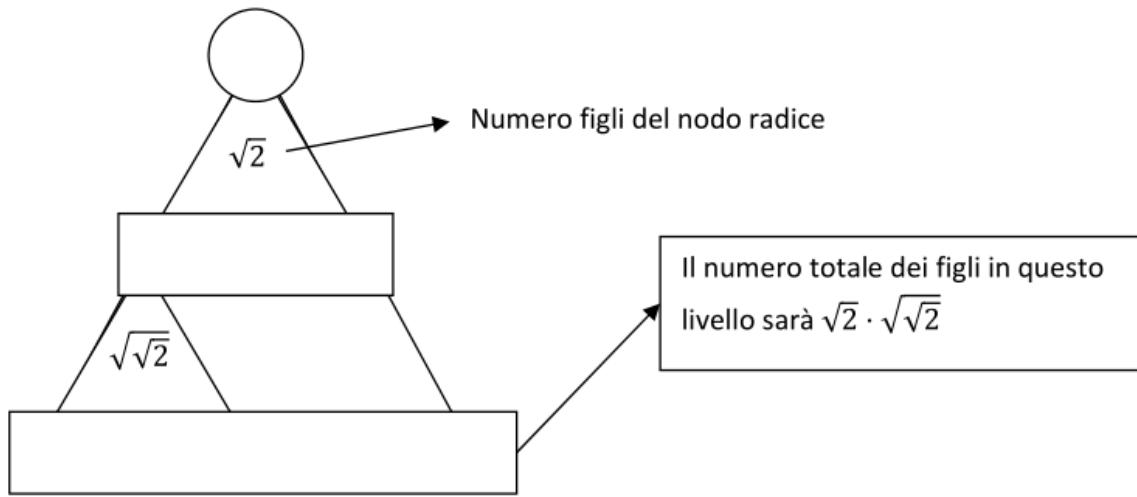
Forniamo una generalizzazione della struttura dell'equazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq k \\ \sum_{i=0}^{z(n)} T(f_i(n)) + g(n) & \text{se } n > k \end{cases}$$

- ▶ Il caso base, in linea di massima, è una costante → Potrebbero presentarsi dei casi in cui il caso base abbia un numero di operazioni lineare e quindi invece di avere un $\Theta(1)$ avremo un $\Theta(n)$
- ▶ $g(n)$ è il contributo delle operazioni nelle chiamate ricorsive;
- ▶ $z(n)$ è il numero di chiamate¹⁵;
- ▶ $f_i(n)$ è l'input che prende ogni chiamata ricorsiva (non è detto che sia la stessa per ogni chiamata), ed ovviamente, affinché risulti corretto l'algoritmo deve risultare $f_i(n) < n$

¹⁵Viene definita come una funzione e non come una costante in quanto il numero di chiamate potrebbe dipendere dal valore in input → Non è detto che una chiamata debba fare lo stesso numero di chiamate ricorsive di un'altra chiamata nello stesso algoritmo.

È importante notare che per calcolare il numero di nodi di un livello basta moltiplicare tra loro le espressioni dei livelli precedenti, ad es:



NB: La precedente rappresentazione è solo per questo specifico esempio → Non è questo il modo di disegnare un albero.

9.1.3 Esempio 1 - Equazione di ricorrenza

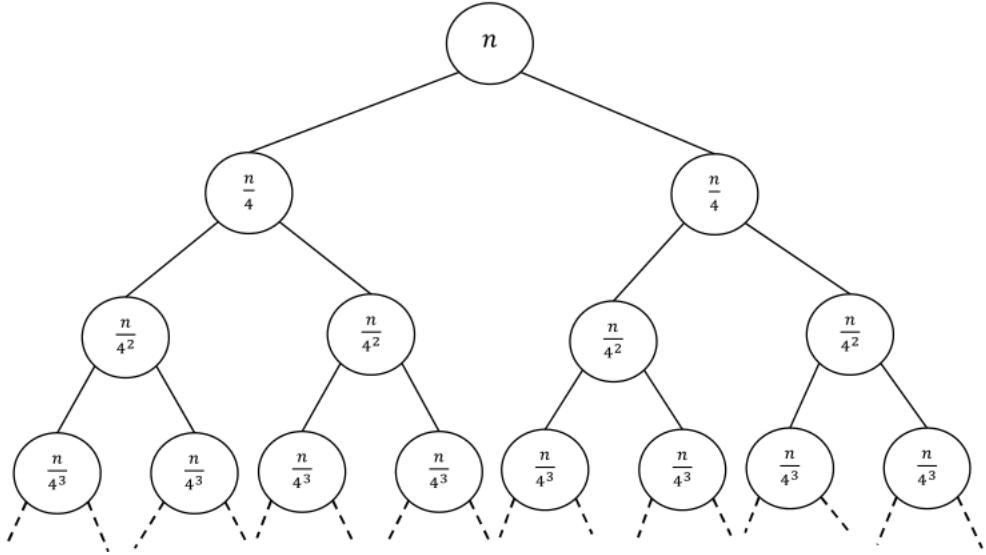
Sia $z(n) = 2$ (si hanno 2 chiamate ricorsive per ogni chiamata → questo si traduce in un albero binario) e siano:

- $f_i(n) = \frac{n}{4}$ (suddivisione input);
- $g(n) = n^2$ (tempo di esecuzione delle altre istruzioni).

Avremo la seguente equazione asintotica:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2T\left(\frac{n}{4}\right) + n^2 & \text{se } n > 1 \end{cases}$$

Che viene rappresentata dal seguente albero di ricorrenza:



NB: Abbiamo utilizzato la forma $\frac{n}{4^i}$ per rendere più semplice la relazione tra input e livello.

Dall'albero di ricorrenza capiamo che il termine generale di un input al livello i -esimo è $\frac{n}{4^i}$ (termine che ci permette di calcolare l'altezza dell'albero).

Si noti che vale la seguente proprietà:

$$f_i(n) = f_j(n), \forall 1 \leq i, j \leq z(n)^{16}$$

Quindi ogni nodo di ciascun livello ha lo stesso input →

- ▶ A livello 0 avremo contributo $g(n) = n^2$;
- ▶ A livello 1 avremo contributo $g\left(\frac{n}{4}\right) = \left(\frac{n}{4}\right)^2$;
- ▶ A livello 2 avremo contributo $g\left(\frac{n}{4^2}\right) = \left(\frac{n}{4^2}\right)^2$;
- ▶ E così via.

NB: È conveniente mantenere i risultati il più generale possibile (senza semplificare) in modo da non perdere le relazioni tra i livelli e poterne ricavare più semplicemente la somma.

Per il calcolo totale possiamo isolare i livelli (quindi calcolare il contributo di ogni livello) e in seguito effettuare un'unica somma in verticale (è necessario conoscere l'altezza dell'albero). Dunque:

¹⁶Ciò significa che i nodi in uno stesso livello ricevono lo stesso input → Ciò vale per ogni livello dell'albero.

Livello 0	n^2
Livello 1	$2 \left(\frac{n}{4}\right)^2$
Livello 2	$2 \cdot 2 \left(\frac{n}{4^2}\right)^2$
Livello 3	$2^3 \left(\frac{n}{4^3}\right)^2$
...	

Possiamo scrivere il livello 0 come $2^0 \left(\frac{n}{4^0}\right)^2$
È evidente che per il livello i -esimo avremo

$$2^i \left(\frac{n}{4^i}\right)^2 = n^2 \left(\frac{2^i}{4^{2i}}\right) = n^2 \left(\frac{2^i}{24^i}\right) = \frac{n^2}{8^i}$$

Per quanto riguarda l'altezza dell'albero bisogna ragionare sul contributo delle foglie, che sappiamo essere $1 \rightarrow$ Tale contributo può essere relazionato alla dimensione dell'input di un livello i (nel nostro caso $\frac{n}{4^i}$). Per calcolare l'altezza dell'albero:

$$\frac{n}{4^i} = 1 \implies n = 4^i \implies \log_4 n = i \log_4 4 \implies \frac{\log_a x}{\log_a a} = \frac{\log_n x}{\log_n a} \implies \frac{\log_2 n}{\log_2 4} = i \implies \frac{\log n}{2 \log 2} = i \implies i = \frac{\log n}{2}$$

Visto che l'altezza dell'albero è $h = \frac{1}{2} \log n$, il numero delle foglie sarà il numero dei figli elevato all'altezza dell'albero. Nel nostro caso:

$$n_f = 2^h = 2^{\frac{\log n}{2}} = (2^{\log n})^{\frac{1}{2}} = \sqrt{n}$$

Calcoliamo quindi il tempo di esecuzione della funzione:

$$T(n) = \underbrace{\text{caso base} \cdot n_f}_{\text{contributo}} + \sum_{i=0}^{h-1} \binom{\text{termine generale}}{generale} = \sqrt{n} + \sum_{i=0}^{\frac{\log n}{2}-1} \left(\frac{n^2}{8^i} \right) = \sqrt{n} + n^2 \sum_{i=0}^{\frac{\log n}{2}-1} \left(\frac{1}{8^i} \right)^i$$

Ma essendo $0 < \frac{1}{8} < 1$ si tratta di una serie geometrica convergente \rightarrow Questo ci semplifica lo studio della sommatoria grazie al seguente ragionamento:

$$\underbrace{\sum_{i=0}^0 x^i}_{x^0 = 1} \leq \underbrace{\sum_{i=0}^z x^1}_{\frac{1}{1-x}} \leq \underbrace{\sum_{i=0}^{\infty} x^i}_{\text{tende ad una costante}} \implies 1 \leq \underbrace{\sum_{i=0}^z x^i}_{\text{tende ad una costante}} \leq \frac{1}{1-x}$$

Visto che la nostra serie tende ad una costante k avremo che:

$$T(n) = \sqrt{n} + k n^2 = k n^2 + n^{\frac{1}{2}} = \Theta(n^2)$$

Volendo utilizzare la forma chiusa invece delle proprietà di convergenza avremo che:

$$\sum_{i=0}^{\frac{\log n}{2}-1} \left(\frac{1}{8} \right)^i = \frac{\left(\frac{1}{8} \right)^{\frac{\log n}{2}} - 1}{\frac{1}{8} - 1} = \frac{-\left(\frac{1}{8} \right)^{\frac{\log n}{2}}}{-\frac{1}{8} + 1} = \frac{1 - \left(\frac{1}{8} \right)^{\frac{\log n}{2}}}{\frac{7}{8}} = \frac{8}{7} \left(1 - \left(\frac{1}{8} \right)^{\frac{\log n}{2}} \right) = \frac{8}{7} \left(1 - \left(\frac{1}{2^3} \right)^{\frac{\log n}{2}} \right) =$$

$$\frac{8}{7} \left(1 - \left(\left(\frac{1}{2} \right)^{\log n} \right)^{\frac{3}{2}} \right) = \frac{8}{7} \left(1 - \frac{1}{(2^{\log n})^{\frac{3}{2}}} \right) = \frac{8}{7} \left(1 - \frac{1}{\sqrt{n^3}} \right)$$

La funzione $f(n) = 1 - \frac{1}{\sqrt{n^3}}$ tenderà a 1 per $n \rightarrow \infty$; pertanto $\frac{8}{7}$ sarà il limite superiore della nostra sommatoria.

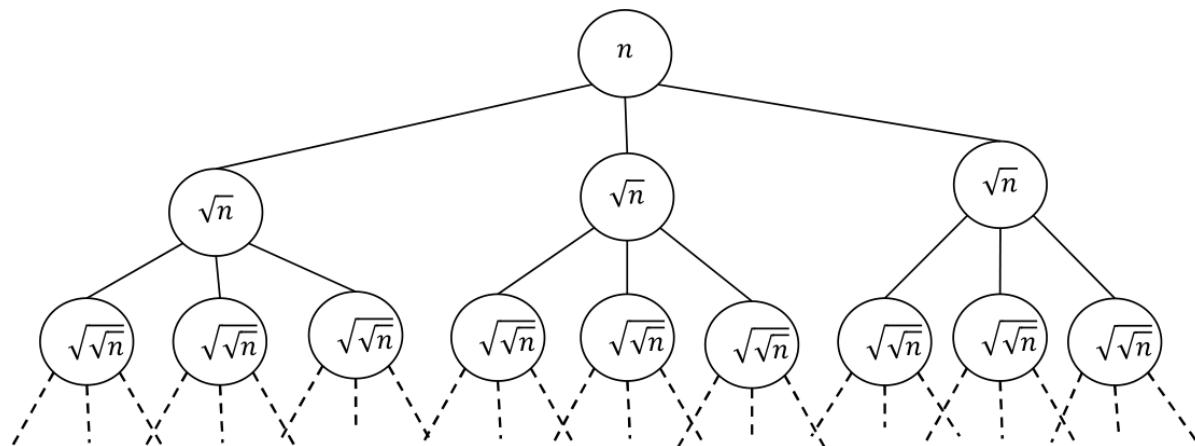
Per $n = 1$ risulta $f(1) = 0$, ma allora la nostra sommatoria tenderà ad una costante c tale che $0 \leq c \leq \frac{8}{7}$ dunque:

$$T(n) = \sqrt{n} + c n^2 \Theta(n^2)$$

9.1.4 Esempio 2 - Equazione di ricorrenza

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ 3T(\sqrt{n}) + 1 & \text{se } n > 2 \end{cases}$$

Da questa funzione, possiamo immaginare di avere un'altezza dell'albero minore rispetto all'esempio precedente (la funzione cresce più lentamente di $\frac{n}{4} \rightarrow$ il numero di figli non influisce sull'altezza dell'albero ma solo sulla sua ampiezza). Ciò significa che arriverà al caso base più velocemente.



Livello	Input per ogni nodo	Contributo per ogni nodo	Contributo del livello
0	n	1	1
1	$n^{\frac{1}{2}}$	1	3
2	$n^{\frac{1}{2^2}}$	1	3^2
3	$n^{\frac{1}{2^3}}$	1	3^3
		...	

Dunque, il contributo per il livello i è 3^i , mentre il suo input è $n^{\frac{1}{2^i}}$.

Andiamo quindi a calcolare l'altezza dell'albero confrontandolo con il massimo input per cui è verificato il caso base (quindi sarà il contributo del caso base per la dimensione del massimo input 1 + 2):

$$(n)^{\frac{1}{2^i}} = 2 \implies \log(n)^{\frac{1}{2^i}} \implies \frac{1}{2^i} \log n = 1 \implies \log n = 2^i \implies \log(\log n) = i$$

NB: $\log n > \log(\log n)$ → La supposizione fatta all'inizio è ora evidentemente vera, infatti $\log(\log n)$ è **esponenzialmente** minore di $\log n$.

Per quanto riguarda il numero di foglie avremo $3^h = 3^{\log(\log n)} = (\log n)^{\log 3}$ e dunque:

$$T(n) = (\log n)^{\log 3} + \sum_{i=0}^{\log(\log n)-1} 3^i = (\log n)^{\log 3} + \frac{3^{\log(\log n)} - 1}{2} = (\log n)^{\log 3} + \frac{1}{2} (\log n)^{\log 3} - \frac{1}{2} = \Theta((\log n)^{\log 3})$$

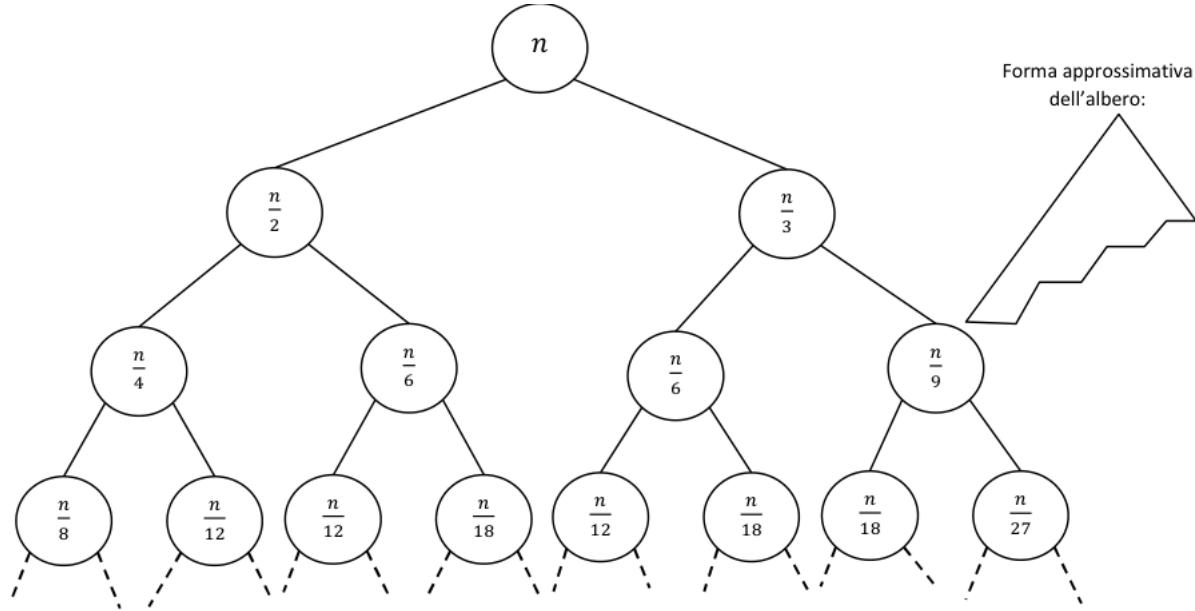
NB: $\log 3$ è una costante, quindi non va approssimata in nessun modo.

9.1.5 Esempio 3 - Equazione di ricorrenza

Prendiamo la seguente equazione di ricorrenza che ha $f_1(n) = \frac{n}{2}$ e $f_2(n) = \frac{n}{3}$:

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

Applichiamo quindi la tecnica dell'albero di ricorrenza:



Come si può notare anche dalla forma approssimativa dell'albero, è evidente che ci saranno dei rami (sequenze discendenti di nodi) di questo albero che arrivano prima alle foglie e percorsi che arrivano dopo. Nello specifico il ramo più a sinistra decresce più lentamente del ramo a più a destra → A livello 3 infatti avremo input di $\frac{n}{8}$ per il primo e $\frac{n}{27}$ per il secondo.

Andiamo ad associare il contributo totale di ogni livello semplicemente sommando il contributo di ogni nodo:

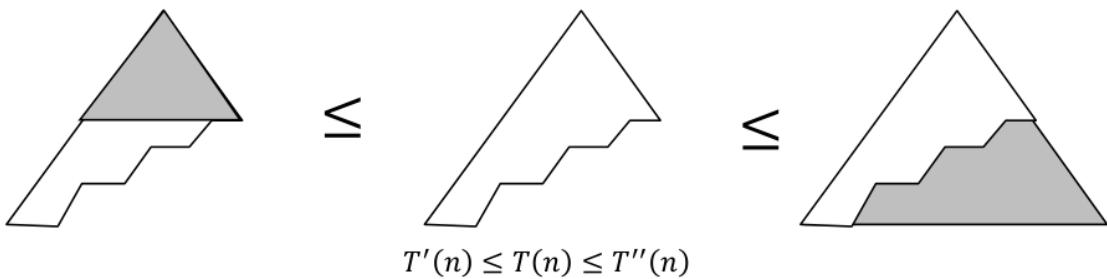
Livello 0	n
Livello 1	$\frac{n}{2} + \frac{n}{3} = \frac{5}{6} n$
Livello 2	$\frac{n}{4} + \frac{n}{6} + \frac{n}{6} + \frac{n}{9} = \frac{25}{36} n$
Livello 3	$\frac{n}{8} + \frac{3n}{12} + \frac{3n}{18} + \frac{n}{28} = \frac{5^3}{6^3} n$
...	

Da questo segue che per un livello i avremo $\left(\frac{5}{6}\right)^i n$ come termine generale

A questo punto non possiamo procedere come negli esempi precedenti → È importante notare che la relazione calcolata precedentemente vale solo per i livelli **pieni**, ovvero quei livelli che hanno il numero massimo di nodi possibile.

L'albero in questione non è pieno, tuttavia prendendo solo i livelli pieni, è evidente che è possibile ottenere un nuovo albero con un tempo di esecuzione minore rispetto a quello da calcolare → Sarà quindi un limite inferiore asintotico per $T(n)$

Analogamente, otterremo un limite notevole asintotico, se approssimando per eccesso, *fingiamo* che tutti i livelli del nostro albero siano pieni. Vale il termine generale:



I 2 diversi alberi avranno uno l'altezza del percorso più breve e l'altro quella del percorso più lungo.

Per il calcolo di questi valori si utilizza sempre lo stesso metodo visto finora:

- ▶ L'altezza del percorso più lungo, poiché si divide sempre per 2, si otterrà con la seguente relazione $\frac{n}{2^i} = 1 \implies \log n = 2^i = \log n$
- ▶ L'altezza del percorso breve invece sarà $\log n = 3^i \implies i = \log_3 n$

A questo punto è possibile calcolare i tempi di esecuzione delle 2 approssimazioni.

NB: Non useremo la forma completa poiché con buona approssimazione possiamo considerare il contributo dell'ultimo livello come se fosse un livello interno (andiamo ad approssimare per eccesso il contributo delle foglie):

$$T^i(n) = \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i = n \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i \stackrel{\text{serie geometrica con ragione } < 1}{\equiv} T^i(n) = \Theta(n)$$

$$T^{ii}(n) \sum_{i=0}^{\log n} \left(\frac{5}{6}\right)^i n = n \sum_{i=0}^{\log n} \left(\frac{5}{6}\right)^i = \Theta(n)$$

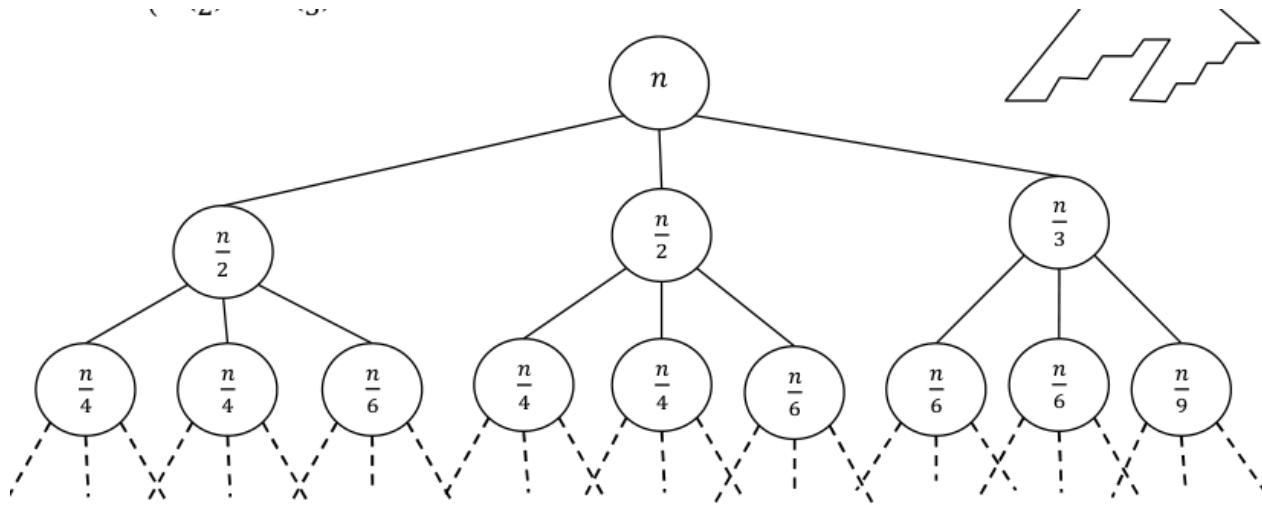
Entrambe sono comprese tra 1 e $\frac{1}{1 - \frac{5}{6}} = 6$

Ma allora $T(n) = \Theta(n)$ essendo limitata sia superiormente che inferiormente da funzioni lineari.

9.1.6 Esempio 4 - Equazione di ricorrenza

Limite superiore e inferiore che crescono in maniera diversa

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



Anche se questo albero di ricorrenza ha una diversa struttura rispetto l'esempio precedente, i problemi sono gli stessi, e quindi anche il modo per risolverli:

Livello	Contributo
0	n
1	$\frac{4}{3}n$
2	$\left(\frac{4}{3}\right)^2 n$
3	$\left(\frac{4}{3}\right)^3 n$
...	
i	$\left(\frac{4}{3}\right)^i n$
...	

L'altezza dell'albero pieno che farà da limite inferiore asintotico sarà quello con percorso più breve, dunque

$$n = 3^h \rightarrow h = \log_3 n$$

Invece l'albero che farà da limite superiore avrà altezza $\log n$. Dunque:

$$T^i(n) = \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i n \wedge T^{ii}(n) \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i n$$

A differenza dell'esempio precedente, la serie geometrica in questione non ha ragione compresa tra 0 e 1; pertanto bisognerà usare la forma chiusa della serie geometria:

$$\begin{aligned} T^l(n) &= n \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i = n \frac{\left(\frac{4}{3}\right)^{\log_3 n + 1} - 1}{\frac{4}{3} - 1} = 3n \left(\frac{4}{3} \cdot \left(\frac{4}{3}\right)^{\log_3 n} - 1\right) = 3n \left(\frac{4}{3} \cdot n^{\log_3 \frac{4}{3}} - 1\right) \\ &= 4n \cdot n^{\log_3 \frac{4}{3}} - 3n = 4 \underbrace{\left(n^{\log_3 \left(\frac{4}{3} + 1\right)}\right)}_{\text{cresce più velocemente di un } \Theta(n)} - 3n = \Theta\left(n^{\log_3 \left(\frac{4}{3} + 1\right)}\right) \\ T''(n) &= n \sum_{i=0}^{\log n} \left(\frac{4}{3}\right)^i = \Theta\left(n^{\log \left(\frac{4}{3} + 1\right)}\right) \end{aligned}$$

Per quanto siano molto vicine queste funzioni hanno **esponente diverso** → Pertanto della relazione

$$T^l(n) \leq T(n) \leq T''(n)$$

Possiamo soltanto dire che $T(n) = \Omega\left(n^{\log \left(\frac{4}{3} + 1\right)}\right)$ e $T(n) = O\left(n^{\log \left(\frac{4}{3} + 1\right)}\right)$ (nulla di più).

9.2 Accenni sulle proprietà degli alberi binari

Alberi pieni → Alberi più bassi possibili a parità di numero di nodi, ovvero:

- ▶ Tutte le foglie sono sullo stesso livello;
- ▶ Tutti i nodi interni hanno grado 2;
- ▶ Per ogni altezza h esiste uno e uno soltanto, albero binario pieno;
- ▶ Il numeri di nodi è esattamente $\sum_{i=0}^h 2^i = 2^{h+1} - 1$

In un albero pieno ho bisogno di un numero preciso di nodi, ovvero un valore pari ad uno in meno di potenza di 2 → Se ho un numero di elementi diverso da $2^x - 1$, non è possibile costruire un albero pieno.

Supponendo di avere un n che rispetti tale proprietà, allora l'altezza sarà risolta dalla seguente equazione:

$$n = 2^{h+1} - 1 \implies [\log n] = [\log(2^{h+1} - 1)]$$

Usiamo la base di tale algoritmo perché dovendo lavorare con interi è evidente che $\log(2^{h+1} - 1) \notin \mathbb{N}$. Essendo l'argomento diverso da una potenza della base.

Inoltre essendo $2^h \leq 2^{h+1} - 1 < 2^{h+1}$, risulta $\log 2^h \leq \log(2^{h+1} - 1) < \log 2^{h+1} \implies h \leq \log(2^{h+1} - 1) < h + 1$ e quindi $\lfloor \log(2^{h+1} - 1) \rfloor = h$. Pertanto

$$\lfloor \log n \rfloor = \lfloor \log(2^{h+1} - 1) \rfloor = \log 2^h = h \implies h = \lfloor \log n \rfloor$$

9.3 Algoritmi di ordinamento

9.3.1 Ordinamento di una sequenza

Il problema dell'ordinamento può essere formalizzato come segue:

- ▶ **Input** → Una sequenza $A = (a_1, a_2, \dots, a_n)$ con $a_i \in \mathbb{Z}$ dove $1 \leq i \leq n$ ¹⁷
- ▶ **Output** → A' una permutazione ordinata di A ¹⁸

Possiamo formalizzare meglio la definizione di permutazione ordinata nel seguente modo:

- ▶ Proprietà della **permutozione** → $\exists f : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ biettiva: $a_i = a'_{f(i)}$ con $1 \leq i \leq n$
 - ❖ Grazie alla biettività della funzione possiamo essere certi che ogni elemento in A sarà anche in A' e viceversa;
 - ❖ La condizione: $a_i = a'_{f(i)}$ ci dice che ogni elemento di A in posizione i avrà in A' $f(i)$
- ▶ Proprietà dell'ordinamento → $a'_i \leq a'_{i+1}$ con $i \leq i \leq n$

Prendendo l'esempio $A = (2, 6, 2)$ ci sono 2 funzioni che rispettano le seguenti proprietà:

$$f_1 : (1, 1), (2, 3), (3, 2) \text{ e } f_2 : (1, 2), (2, 3), (3, 1)$$

Possiamo dunque descrivere l'output anche come $A' = (a_{j_1}, a_{j_2}, \dots, a_{j_n})$ con $a_{j_k} \in A$ e $a_k \in A'$ con $1 \leq k \leq n$ e tale che $j_k = f^{-1}(k)$. Es.:

$$A = 42 \wedge A' = 24 \text{ allora } j_2 = f^{-1}(2) = 1$$

Dalle definizioni di **input/output** capiamo che il problema è, data una sequenza, una sua permutazione con determinate proprietà.

¹⁷ Sarebbe valido anche un qualsiasi insieme (non obbligatoriamente \mathbb{Z}), l'importante è che abbia una relazione d'ordine totale (altrimenti non sarebbe possibile ordinare gli elementi).

¹⁸ Ci potrebbero essere più permutazioni ordinate, come ad es $A = (2, 6, 2)$ abbiamo le seguenti permutazioni ordinate → (a_1, a_3, a_2) e (a_3, a_1, a_2) .

Approccio 1

Sappiamo che il numero di permutazione è finito per insiemi finiti, più precisamente se $|A| = n$ allora il numero di permutazioni è $\prod_{i=1}^n i = n!$

Essendo la relazione d'ordine in \mathbb{Z} totale è evidente che tra queste permutazioni ce ne sia almeno una ordinata. Quindi si potrebbero generare tutte le permutazioni ed andarne a prelevare una ordinata.

Questa soluzione è corretta, ma utilizzare un algoritmo di tipo brute force è inefficiente → dovremmo generare $n!$ permutazioni (e nel peggior caso la permutazione ordinata potrebbe essere l'ultima generale).

Si noti che $n \leq n! \leq n^n$

NB: Tra n e n^n ci sono infinite funzioni: $n \leq \dots \leq n^{1,2} \leq \dots \leq n^{1,5} \leq \dots \leq n^n$ e nessuna è un Θ di un'altra. Quindi scrivere $n^{2,20} = \Theta(n^{2,21})$ è un errore grave poiché crescono in maniera differente.

Da questo deduciamo che per qualsiasi algoritmo di **brute force** esiste almeno un input che costringe a esplorare tutti gli elementi dell'insieme prima di arrivare alla soluzione.

Anche supponendo per assurdo che si potrebbe generare una permutazione in tempo costante per verificare che la permutazione sia ordinata dobbiamo controllare tutti gli elementi (tempo lineare); Nel caso peggiore potrebbe generare $n! = \Omega(n^n)$ permutazioni.

Da questa analisi deduciamo che un algoritmo di questo tipo non è applicabile.

Approccio 2

Bisogna cercare un algoritmo che da una sequenza ordinata cerchi di passare ad una sequenza un minimo più ordinata fino ad arrivare alla sequenza ordinata, escludendo alcune permutazioni. Tutti gli algoritmi che vedremo saranno $T(n) = \Theta(n^2)$ proprio grazie a tale approccio.

Un modo per ordinare la sequenza è quello di prendere tutte le coppie e confrontarne a 2 a 2 gli elementi. Ad esempio per (a_1, a_2, a_3) arrivo alla sequenza ordinata tramite i seguenti confronti:

$$a_1 \leq a_2 \left\{ \begin{array}{l} a_1 \leq a_3 \left\{ \begin{array}{l} a_2 \leq a_3 \rightarrow (a_1, a_2, a_3) \\ a_3 < a_2 \rightarrow (a_1, a_3, a_2) \end{array} \right. \\ a_3 < a_1 \rightarrow (a_3, a_1, a_2) \end{array} \right.$$

Nel peggiore dei casi devo confrontare tutte le coppie delle sequenze con un tempo quadratico, o nel caso migliori i confronti impiegheranno $T(n) = \mathcal{O}(n)$

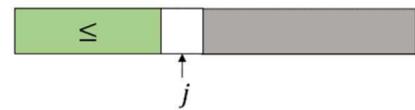
Ne consegue che in nostri algoritmi saranno $n \leq T(n) \leq n^2$ e si differenzieranno per le politiche di confronto utilizzate e di come tengono traccia dei confronti fatti.

9.4 Insertion Sort

L'idea di questo algoritmo è quella di dividere la sequenza in una parte ordinata e una non.

All'inizio possiamo supporre che la sequenza ordinata sia composta solo dal primo elemento, ed il resto degli elementi compongono la sequenza disordinata.

L'Insertion sort prende il primo elemento della parte disordinata (nel nostro caso j) e lo inserisce nella giusta posizione nella parte ordinata.



Quindi se $a_{j-1} \leq a_j$ allora mantiene la stessa posizione. In caso contrario a_{j-1} andrà in posizione j e dovrà confrontare a_j con a_{j-2} ; anche in questo caso se $a_{j-2} \leq a_j$ andrà nella posizione precedentemente liberata ($j-1$), altrimenti è a_{j-2} a spostarsi nella posizione $j-1$.

Ripetendo questo processo posso arrivare a confrontare a_j con a_1 :

- ▶ Se $a_1 \leq a_j$ posizionerò a_j nella posizione 2 (che sarà libera);
- ▶ In caso contrario andrà in posizione 1 ed a_1 in posizione 2 (questi spostamenti hanno ovviamente un loro costo).

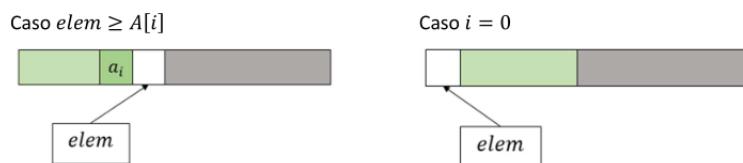
L'algoritmo sarà il seguente:

```

1  InsertSort(A, N)
2      // (La prima posizione gi ordinata)
3      FOR j = 2 TO N DO
4          i = j - 1 // (Indice della poszione ordinata)
5          elem = A[j]
6          WHILE (i >= 1 OR elem < A[i]) DO
7              A[i + 1] = A[i] // (nella vecchia cella libera posiziono A[i])
8              i = i - 1
9              A[i + 1] = elem

```

A prescindere dalla condizione di terminazione del ciclo `while` (riga 6), la riga 9 è sempre la stessa. Infatti:



9.4.1 Analisi

Studiamo solo il ciclo `while` in quanto unico a determinare il comportamento asintotico dell'algoritmo.

Questo ciclo può essere diverso per istanze con stessa dimensione a causa del confronto $elem \geq A[i]$ (questo è il motivo della preferenza al ciclo `for`).

Ne consegue che non possiamo associare un valore a $T(N)$ poiché non è detto che abbia lo stesso tempo ogni istanza di N elementi.

Ad es. per $(4, 3, 2, 1)$ non verrà mai eseguito il corpo del ciclo, mentre per $(3, 4, 2, 1)$ le istruzioni nel corpo saranno eseguite 6 volte.

Questa irregolarità costringe a fare i seguenti ragionamenti:

- ▶ Se l'algoritmo è sfortunato possiamo dire che $T(N) = \mathcal{O}(T_{max}(N))$ che rappresenta il tempo di esecuzione dell'algoritmo per i casi peggiori (quelli che fanno il massimo numero di operazioni);
- ▶ Se è molto fortunato, analogamente risulterà $T(N) = \Omega(T_{min}(N))$;
- ▶ Se risulterà $T_{max}(N) = \Theta(T_{min}(N))$ è evidente che $T(N)$ avrà anch'esso lo stesso ordine, altrimenti dovrà analizzare il caso medio.

Prima di analizzare in maniera esaustiva il ciclo `while`, vediamo la complessità delle altre linee:

```
1 InsertionSort(A, N)
2   FOR j = 2 TO N DO // 2 (N + 1 - 1)
3     i = j - 1 // 2 (N - 1)
4     elem = A[j] // 2 (N - 1)
5     // (Per il ciclo while possiamo descrivere per ora solo il costo delle singole istruzioni)
6     WHILE (i >= 1 OR elem < A[i]) DO // 4
7       A[i + 1] = A[i] // 3
8       i = i - 1 // 1
9       A[i + 1] = elem // 3 (N - 1)
```

Per quanto riguarda il ciclo `while` bisogna innanzitutto notare di come ci sono delle sequenze diverse che si comportano allo stesso modo; ovvero, tutte le sequenze che hanno la stessa relazione per ogni elemento in posizione i -esimo

$$(1, 3, 2, 4), (10, 25, 17, 60), (7, 29, 12, 30) \dots$$

Abbiamo quindi tante classi di equivalenza quante permutazioni di una sequenza → Ne consegue che per lo studio del ciclo `while` bisogna formalizzare la suddetta dipendenza.

È chiaro che l'input ha impatto solo sul numero di esecuzione del `while` (le linee analizzate precedentemente mantengono lo stesso costo).

Fissato j il while viene ripetuto un tot di volte partendo da 0 e, per $j \neq i$ il numero volte che viene ripetuto il blocco è diverso.

Possiamo introdurre una serie di parametri t_j , che indica il numero di volte che la testa del while viene eseguita all'iterazione j -esima del for;

Quest ultimo, variando da 2 a N avremo t_2, t_3, \dots, t_N parametri

Se t_j è il numero di volte che viene eseguita la testa del while, è evidente che per tutte le iterazioni la testa verrà eseguita $\sum_{j=2}^N t_j$

volte (tale sommatoria non ha una forma chiusa), mentre $\sum_{j=2}^N (t_j - 1)$ volte il corpo.

Dunque, possiamo rappresentare il tempo di esecuzione con la seguente espressione

$$\begin{aligned} T(N) &= 2N + 2(N-1) + 2(N-1) + 4 \sum_{j=2}^N t_j + 3 \sum_{j=2}^N (t_j - 1) + \sum_{j=2}^N (t_j - 1) + 3(N-1) \\ &= 4 \sum_{j=2}^N t_j + 4 \sum_{j=2}^N (t_j - 1) + 9N - 7 \end{aligned}$$

9.4.2 Caso peggiore

Se la seconda condizione del while è sempre verificata è chiaro che $i = 0$ (poiché all'inizio $i = j - 1$) si verifica dopo j iterazioni (uscendo quindi dal while);

Pertanto non è possibile eseguire più di j iterazioni (questo corrisponde al nostro caso peggiore).

Bisogna ora verificare se esista un input tale per cui $\forall j, t_j = j$. Questo input esiste ed è la **sequenza decrescente**:

$$\begin{aligned} T_{max}(N) &= 9N - 7 + 4 \sum_{j=2}^N (j-1) = 9N - 7 + 4 \left(\sum_{j=1}^N j - \cancel{1} \right) + 4 \sum_{j=1}^{N-1} (j-1) = \\ &= 9N - 7 + 4 \left(\cancel{N-1} \sum_{j=1}^{N-1} j - \cancel{1} + \cancel{N} \right) + 4 \sum_{j=1}^{N-1} (j-1) = 9N - 7 - 4 + 4N + 8 \sum_{j=1}^{N-1} j - 4 \sum_{j=1}^{N-1} 1 = \\ &= 13N - 11 + 8 \left(\frac{(N-1)(N-1+1)}{2} \right) - 4(N-1) = 9N - 7 + 4N(N-1) = \\ &= 4N^2 + 5N - 7 = \Theta(N^2) \end{aligned}$$

9.4.3 Caso migliore

Con sequenze ordinate accade che $\forall j : 2 \leq j \leq N$ risulta che $t_j = 1$.

Ad esempio per $(3,7,8,10)$ la seconda condizione del while risulterà falsa per ciascun $j \rightarrow$ ci saranno quindi 4 esecuzioni:

$$T_{min}(N) = 9N - 7 + 4 \sum_{j=2}^N 1 + 4 \sum_{j=2}^N 0 = 9N - 7 + 4(N-1) = 13N - 11 = \Theta(N)$$

9.4.4 Caso medio

Ora resta da capire con quanta frequenza avvenga il caso migliore e il caso peggiore, poiché non è in genere detto che il comportamento dell'algoritmo sia quadratico (per ora sappiamo solo che ci sono degli input che impiegano tempo quadrato e degli input che sono lineari).

Lo studio del caso medio in questo algoritmo è piuttosto semplice; infatti, per j fissato il valore atteso di t_j è, assumendo equiprobabilità, esattamente la media aritmetica, ovvero:

$$t_j = \frac{1+2+\dots+j}{j} = \frac{\sum_{k=1}^j k}{j} = \frac{1}{j} \cdot \frac{j(j+1)}{2} = \frac{j+1}{2}$$

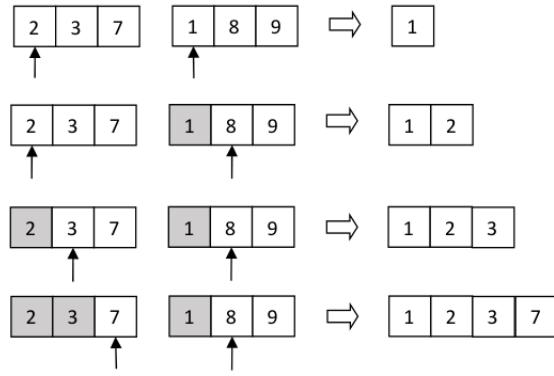
Ma allora:

$$\begin{aligned} T_{med}(N) &= 9N - 7 + 4 \sum_{j=2}^N \frac{j+1}{2} + 4 \sum_{j=2}^N \left(\frac{j+1}{2} - 1 \right) = 9N - 7 + 2 \sum_{j=2}^N (j+1) + 2 \sum_{j=2}^N (j-1) \\ &= 9N - 7 + 2 \sum_{j=2}^N j + 2 \sum_{j=2}^N 1 + 2 \sum_{j=2}^N j - \sum_{j=2}^N 1 = 9N - 7 + 4 \sum_{j=1}^{N-1} j - 4 + 4N = \\ &= 13N - 11 + 4 \frac{N(N-1)}{2} = 2N^2 - 2N - 11 = \Theta(N^2) \end{aligned}$$

Visto che il caso dominante è quello peggiore, possiamo assumere che l'algoritmo di `insertion sort` è un algoritmo che ha tempo di esecuzione quasi sempre quadratico.

9.5 Merge Sort

Algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola:



Terminata una sequenza non ci resta che mettere gli elementi di quella rimanente alla fine:



NB: La sequenza di un elemento sarà il nostro caso base per l'algoritmo ricorsivo.

Prendendo in considerazione una sequenza vuota come caso base otterremmo un loop con sequenze di un elemento (una sequenza di un elemento può essere divisa solo in una sequenza vuota ed un'altra sequenza da un elemento, quest'ultima dovrà nuovamente essere divisa...). Bisogna fare attenzione che l'algoritmo ricorsivo termini per il caso base scelto come nel nostro caso

Con caso base 1 risulta che $\forall N \geq 2$ si ha $1 \leq \left\lfloor \frac{N}{2} \right\rfloor \wedge 1 \leq N - \left\lfloor \frac{N}{2} \right\rfloor < N$; con $N = 1$ abbiamo una sequenza ordinata.

Definiamo pertanto in questo modo il nostro algoritmo:

```

1 MergeSort(A, p, r) // Dove p → indice di inizio e r → indice di fine
2   IF p < r THEN // Abbiamo almeno 2 elementi
3     q = (p + r) / 2 // q → base della dimensione
4     MergeSort(A, p, q)
5     MergeSort(A, q+1, r)
6     Merge(A, p, q, r) // Unione delle 2 sequenze

```

9.5.1 Correttezza dell'algoritmo

La sequenza iniziale ha $r - p + 1$ elementi. Affinché il nostro algoritmo funzioni, dobbiamo garantire che:

$$r - p + 1 > q - p + 1 \wedge r - p + 1 > \underbrace{r - q}_{r - (q + 1) + 1}$$

Dove $q = \left\lfloor \frac{p+r}{2} \right\rfloor$. Sappiamo che $\left| \frac{p+r}{2} \right| \leq \frac{p+r}{2}$, pertanto risulta:

$$r - p + 1 > \frac{p+r}{2} - p + 1 \implies 2r > p + r \implies r > p$$

Essendo la nostra ipotesi proprio $p < r$ (condizione dell'**if**) abbiamo dimostrato la prima implicazione. Analogamente si procede per la seconda:

$$r - p + 1 > r - \frac{p+r}{2} \implies -p > -\left(\frac{p+r}{2} + 1\right) \implies 2p < p + r + 2 \implies p < r + 2$$

Resta da dimostrare che, per qualsiasi input, l'algoritmo faccia un numero finito di chiamate ricorsive e che quindi, prima o poi, raggiungerà il caso baso; quando r è molto vicino a p (ovvero quando $r < p + 1$) è evidente che sarà $q = \left\lfloor \frac{p+r}{2} \right\rfloor = p$ e quindi entrambe le chiamate ricorsive non verranno effettuate (la condizione dell'if sarà alla prima chiamata $p < p$ e alla seconda $p + 1 < r$, condizioni entrambe false).

9.5.2 Occupazione in memoria

Di seguito riportiamo l'algoritmo di merge, la cui intuizione è stata descritta precedentemente:

```

1 Merge(A, p, q, r)
2   k = p // Indice che scorre la posizione di B
3   i = p
4   j = q + 1
5   WHILE i <= q AND j <= r DO
6     IF A[i] <= A[j] DO
7       B[k] = A[i]
8       i = i + 1
9     ELSE
10      B[k] = A[j]
11      j = j + 1
12    IF i < q THEN
13      j = i // Restano da copiare gli elementi della sequenza sinistra
14    // In caso contrario non modifico j visto che l'indice è già corretto
15    WHILE k <= r DO
16      B[k] = A[j]
17      j = j + 1
18      k = k + 1
19    // A questo punto andrebbero copiati gli elementi di B nuovamente in A

```

Il tempo di **merge** è lineare poiché vado ad effettuare almeno n scritture in memoria. È interessante notare che è stato necessario utilizzare un altro vettore → Questo significa che **MergeSort** ha bisogno di uno spazio aggiuntivo in memoria anch'esso lineare (un vettore pari all'input del programma oltre alle variabili locali) a causa dell'algoritmo **Merge**

Anche senza considerare **Merge** l'algoritmo di MergeSort ha bisogno di uno spazio aggiuntivo a causa delle chiamate ricorsive
 → Per ogni chiamata ricorsiva, prima di chiamare alla chiamata figlia, è necessario salvare i dati sullo stack di attivazione.

9.5.3 Tempo di esecuzione

```

1 MergeSort(A, p, r)
2   IF p < r THEN // La riga 2 e 3 contribuiscono con un Θ(1)
3     q = (p + r) / 2
4     MergeSort(A, p, q) // La riga 4 e 5 contribuiscono con un Θ(?)
5     MergeSort(A, q+1, r)
6     Merge(A, p, q, r) // Contributo di Θ(n)

```

Sarebbe assurdo dover calcolare il valore del tempo di esecuzione della funzione se per farlo devo conoscere il tempo di esecuzione della funzione stessa.

Ma, visto che la funzione è definita in modo induttivo, è possibili calcolarla su input più piccoli; dunque, è possibile sfruttare la definizione induttiva e definire in maniera induttiva anche la funzione tempo

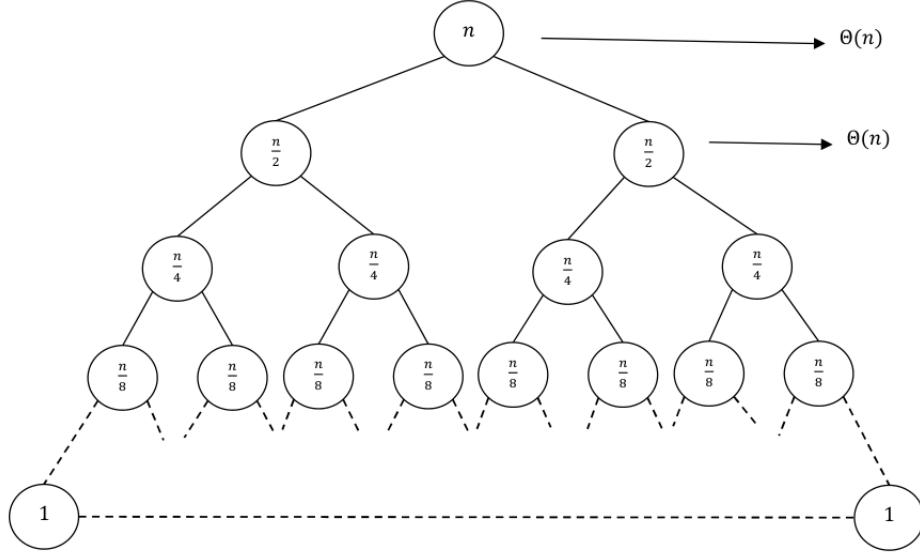
Con buona approssimazione avremo che

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

Ma $T_{Merge}(N) = \Theta(n)$ e $\Theta(1)$ è assimilato dal $\Theta(n)$, pertanto l'equazione di ricorrenza da risolvere sarà

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

Da cui ricaviamo l'albero di ricorrenza:



Quindi il nodo del livello 0 contribuisce per un $\Theta(n)$, ciascun nodo del livello 1 produce un $\Theta(n)$ poiché fa $\frac{n}{2}$ operazioni;¹⁹ analogamente si comportano tutti i livelli (fatta eccezione per l'ultimo) → In pratica tutti i livelli hanno contributo lineare eccetto quello della foglia.

Ogni nodo foglia contribuisce per un $\Theta(1)$, tuttavia per conoscere il contributo totale del livello devo prima conoscere il numero di foglie presenti, in modo da poter calcolare il tempo di esecuzione con la seguente equazione:

$$T_{MS}(n) = \Theta(1) \cdot \# \text{ foglie} + \sum_{i=0}^{\# \text{ livelli interni}} \Theta(n)$$

- ▶ **Numero livelli interni** → Dalla composizione dell'albero (è un albero binario completo) possiamo dire che ad un livello i ogni nodo prende in input $\frac{n}{2^i}$; ciò significa che alle foglie avrò $1 = \frac{n}{2^i} \implies 2^i = n \implies \log n = \log 2^i \implies i = \log n$ (Il numero di livelli interni è $\log n - 1$)
- ▶ **Numero foglie** → Sappiamo che le foglie sono a profondità $\log n$; sappiamo inoltre anche che un livello i ha 2^i nodi; pertanto il numero delle foglie è $2^{\log n} = n$

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

¹⁹Da non confondere con con il numero di elementi, infatti il nostro $\Theta(n)$ deriva dal contributo $2 T_{MS}\left(\frac{n}{2}\right) + \Theta(n)$.

9.6 Selection sort

Possiamo vedere questo algoritmo come il duale dell'Insertion sort → Seleziona una posizione e poi trova l'elemento da inserire all'interno.

Sia la posizione che l'elemento vengono ovviamente scelti con condizione di causa

Se iniziamo dall'ultima posizione, ovvero n , allora si dovrà trovare il massimo elemento della sequenza e poi scambiarlo con quello in posizione n (se invece si inizia dalla prima allora cercheremo il minimo).

A questo punto si prosegue con la posizione $n - 1$ scambiando l'elemento in quella posizione con la sequenza restante (da 1 a $n - 1$) e si procede in questo modo fino alla posizione 2 (ovviamente l'elemento in posizione 1 sarà già ordinato per la sequenza → è evidente che dopo gli $n - 1$ scambi in posizione 1 sia presente il minimo).

```

1 SelectionSort(A, n)
2   FOR i = n DOWNTO 2 DO
3     j = FindMax(A, i) // Ritorna la posizione del massimo
4     Swap(A, i, j)
5
6
7 // Corpo della funzione FindMax
8 int FindMax(A, i)
9   max = 1
10  FOR j = 2 TO n DO
11    IF A[max] < A[j] THEN
12      max = j
13  RETURN MAX

```

Visto che sappiamo che la scelta del massimo non può essere effettuata con un algoritmo meno che lineare (deve obbligatoriamente controllare tutti gli elementi) sembrerebbe che questa sia la soluzione migliore.

Tuttavia questo è un chiaro esempio di come usare soluzioni ottime per sottoproblemi non rende l'algoritmo così definito ottimale.

Questo non implica che l'idea sopra descritta sia pessima → Infatti, a seconda di come viene implementata quest'algoritmo avrà tempi di esecuzione molto diversi (Sarà possibile arrivare ad un algoritmo che abbia $T(n) = \Theta(n \log n)$)

Andiamo quindi a calcolare il tempo di esecuzione del precedente algoritmo:

$$\begin{aligned}
T_{SS}(n) &= \Theta(n) + \sum_{i=2}^n \underbrace{T_{FindMax(i)}}_{\Theta(i)} + \Theta(1) = \Theta(n) + \Theta\left(\sum_{i=2}^n i\right) = \Theta(n) + \Theta\left(\left(\sum_{i=1}^n i\right) - 1\right) \\
&= \Theta(n) + \Theta\left(\frac{n(n+1)}{2} - 1\right) = \Theta(n^2)
\end{aligned}$$

Si nota facilmente che il problema del nostro algoritmo è nella funzione `FindMax` → Non viene sfruttato il fatto che durante lo

scorrimento per la ricerca del massimo vengono eseguiti già dei confronti con alcuni elementi, ma **FindMax** non ne tiene traccia e li "dimentica" alla successiva iterazione.

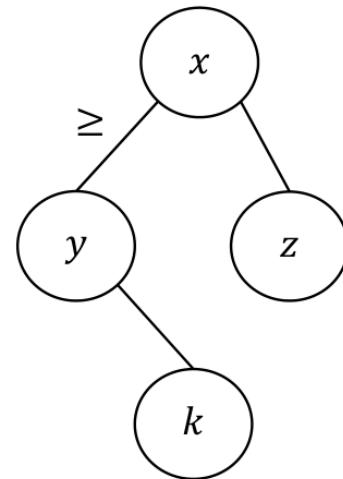
Per migliorare il nostro algoritmo si potrebbe pensare di salvare in una struttura dati l'informazione parziale dell'ordinamento degli elementi (durante una ricerca del massimo vengono fatti un numero lineare di confronti che ovviamente non bastano per l'ordinamento totale della sequenza)

Resta quindi da definire una struttura dati che ci permetta di mantenere queste informazioni parziali allo stesso modo di come una sequenza sia la giusta struttura per rappresentare una relazione d'ordine totale. Possiamo utilizzare un albero dove gli archi rappresentano la relazione tra gli elementi:

Quindi l'albero a sinistra, con l'arco che rappresenta la relazione di \geq , preserva le seguenti informazioni:

- ▶ $x \geq y$;
- ▶ $x \geq z$;
- ▶ $y \geq k$;
- ▶ $x \geq k$ (per transitività)

Quindi oltre alle informazioni dirette, abbiamo una relazione tra tutti gli elementi di uno stesso ramo



Dunque organizzando i valori in un albero dove l'arco rappresenterà la relazione di \geq possiamo migliorare il tempo di esecuzione dell'algoritmo **FindMax** e di conseguenza anche di **SelectionSort**

10 Lezione 13

10.1 Alberi binari completi

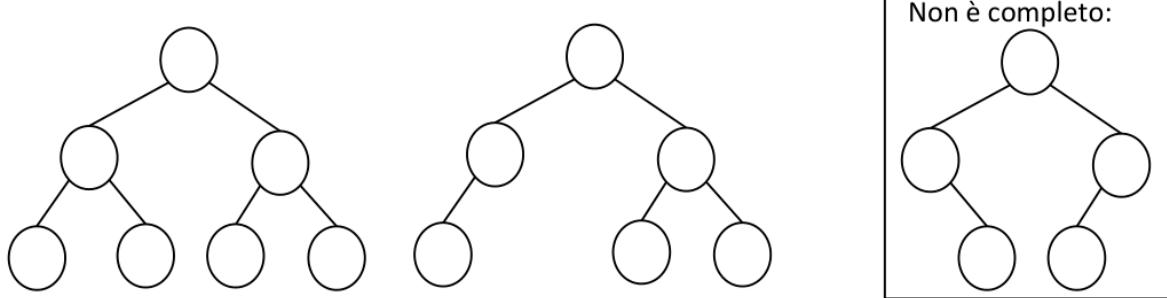
È necessario ai nostri scopi, rilassare alcuni vincoli dell'albero pieno → È evidente che non possiamo avere un algoritmo solo per determinati input. I vincoli da indebolire sono:

- ▶ Tutte le foglie sono sullo stesso livello;
- ▶ Tutti i nodi interni hanno grado 2.

Tale albero è conosciuto come albero binario completo, in cui a differenza dell'albero pieno:

- ▶ Tutte le foglie sono al livello h o al livello $h - 1$;
- ▶ Tutti i nodi interni hanno grado 2 tranne al più 1.

Esempi di alberi ammessi dalla precedente definizione:



NB: Ogni albero pieno è anche completo, ma non vale il contrario.

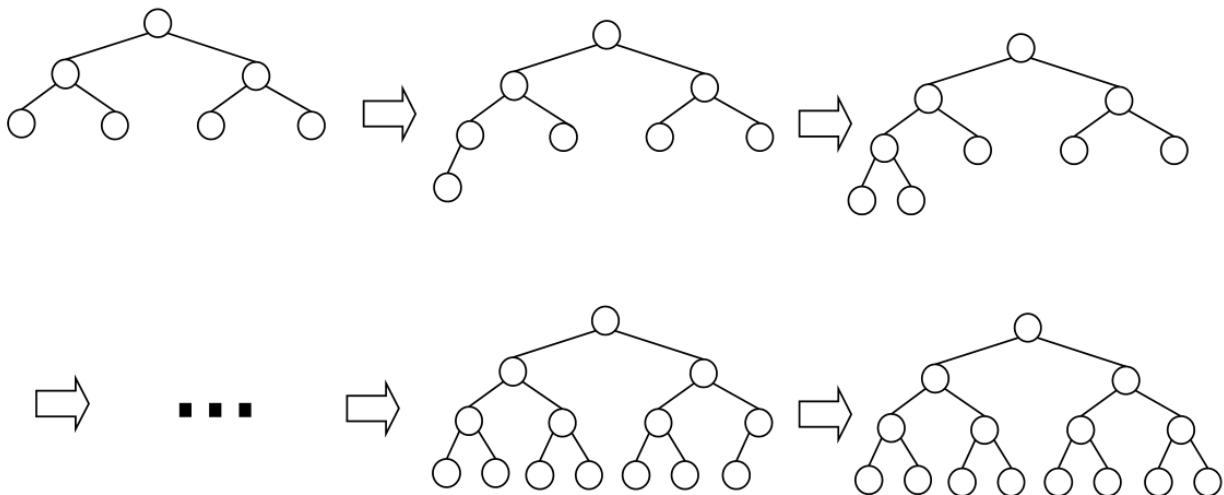
Questa definizione risolve il nostro problema della cardinalità della sequenza, poiché ora dato un qualsiasi n è sempre possibile definire un albero completo.

La dimostrazione del precedente enunciato avviene tramite induzione.

Abbiamo già dimostrato che esiste un albero pieno di $2^{h+1} - 1$ nodi (dunque esiste anche un albero pieno di $2^{h+2} - 1$), ma nessun albero pieno con numero di nodi compreso tra i precedenti 2.

Visto che h è arbitrario è evidente che se riesco a costruire un albero completo per ogni numero di nodi x per cui $2^{h+1} - 1 \leq x \leq 2^{h+2} - 1$; allora è possibile farlo $\forall x \in \mathbb{N}$ poiché abbiamo praticamente partizionato tutti i numeri naturali in intervalli di potenze di 2.

Ogni albero pieno è completo → Resta da dimostrare che è possibile costruire un albero completo per ogni nodo nell'intervallo. Di seguito riportiamo un esempio per $7 \leq x \leq 15$:



Risolto il problema della cardinalità, affinché sia utile al nostro algoritmo, bisogna dimostrare che anche per un albero completo vale la proprietà di essere il più corto con quel numero di nodi e che quindi è possibile scrivere l'altezza in corrispondenza del numero di nodi:

$$h = \lceil \log n \rceil$$

Non vale più che il numero di nodi $n = 2^{h+1} - 1$ poiché abbiamo già dimostrato che un albero completo può avere un numero di nodi qualsiasi; è vero che $2^h \leq n \leq 2^{h+1}$ e quindi il nostro numero di nodi o è una potenza di 2 o si trova tra 2 potenze di 2.

Dunque, sapendo che $\forall n \in \mathbb{N}, \exists h \geq 0$:

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

Essendo il logaritmo una funzione monotona conserva la relazione d'ordine:

$$\log(2^{h+1} - 1) \leq \log n \leq \log(2^{h+2} - 1) \implies \left| \underbrace{\log(2^{h+1} - 1)}_h \right| \leq |\log n| \leq \left| \underbrace{\log(2^{h+2} - 1)}_{h+1} \right|$$

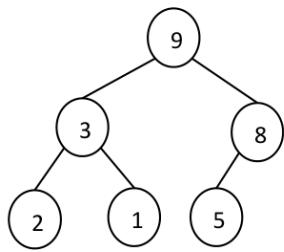
Ma quindi $h \leq |\log n| < h + 1$, ed essendo la base compresa tra h e $h + 1$ e, inoltre $|\log n| \neq h + 1$; È evidente che $|\log n| = h$ (come volevasi dimostrare).

10.2 Alberi heap

Per albero heap si intende una struttura dati con le seguenti proprietà:

- ▶ Un albero completo di n nodi
 - ◊ Proprietà strutturale che garantisce l'altezza più bassa per quel numero di nodi;
- ▶ Per ogni nodo interno x , il valore di x , è maggiore o uguale al dato associato ai nodi figli
 - ◊ Proprietà che garantisce la relazione d'ordine parziale tra gli elementi.

Ad esempio per la sequenza (3, 8, 5, 1, 9, 2) un possibile albero heap (non è detto che ci sia una singola rappresentazione per una determinata sequenza) è il seguente:



N.B.: se non c'è relazione di discendenza non abbiamo nessuna informazione nei nodi. Infatti, anche se il nodo con elemento 5 è ad un livello inferiore al nodo con elemento 3 non è vero che $3 \geq 5$

Per le 2 proprietà descritte in seguito possiamo essere certi che nella radice dell'albero è situato il massimo valore della sequenza, poiché dalla radice discendono tutti i nodi dell'albero. La proprietà 2 specifica chiaramente che un nodo non può essere più grande del suo antenato.

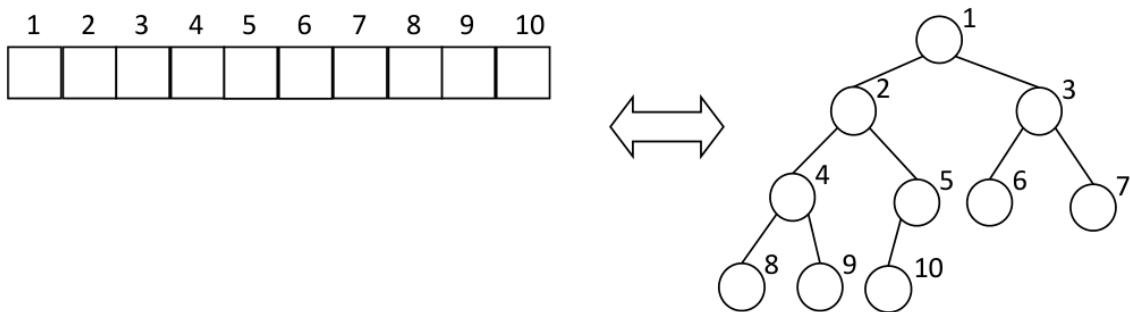
10.3 Nuovo approccio al Selection Sort

Dai precedenti ragionamenti abbiamo dedotto che la struttura più adatta a risolvere il nostro problema è l'albero heap. Ciò però ci porta a dover affrontare nuovi problemi:

1. Come rappresentare la struttura dell'albero con un vettore;
2. Costruire uno heap in maniera efficiente;
3. Mantenere la coerenza della struttura (e quindi le proprietà dello heap) anche dopo aver rimosso il massimo.

10.3.1 Problema 1

Ogni sequenza può essere rappresentata in un albero completo in un modo del tutto naturale ponendo come unica condizione che i nodi dell'albero devono essere contigui. Di seguito forniamo un esempio di tale corrispondenza:



Quindi gli indici dell'array vanno da sinistra a destra livello per livello.

In questo modo un qualsiasi elemento in posizione i nell'albero avrà figlio sinistro in posizione $2i$ e figlio destro in posizione $2i + 1$. (Devono essere indici esistenti, ovvero non superiore alla lunghezza dell'array).

Dunque il nostro array sarà un albero heap se $\forall i : 2i \leq n, A[i] \geq A[2i]$ e $\forall i : 2i + 1 \leq n, A[i] \geq A[2i + 1]$.

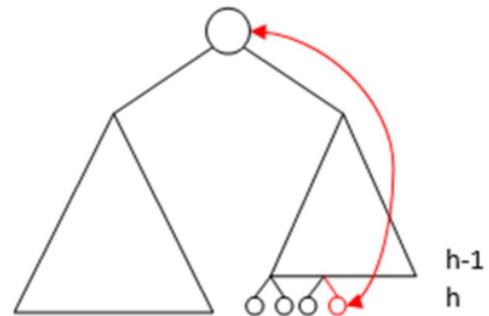
Da questa corrispondenza si ricava anche che le posizioni nell'array che contengono le foglie sono quelle per cui $2i > n$ ovvero che le foglie sono nella parte destra dell'array e conseguentemente i nodi interni sono situati nella parte sinistra con la radice in prima posizione.

Più precisamente i nodi interni vanno da $1 \leq i \leq \left[\frac{n}{2}\right] < i \leq n$.

Persino la stratificazione dei livelli può essere ottenuta facilmente (basta dividere per 2).

Prendiamo un generico albero heap, poiché le foglie sono contigue è evidente che se l'albero non è pieno esistono solo 2 possibilità:

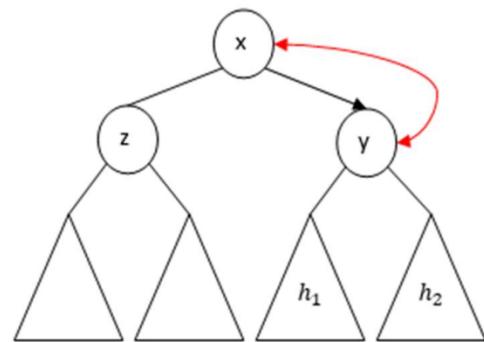
- ▶ Il sottoalbero sinistro è un albero pieno (come in figura) di altezza $h - 1$ (bisogna togliere il livello della radice);
- ▶ È il sottoalbero a destra a essere un albero pieno di altezza $h - 2$.



Per preservare l'albero il più possibile l'unico modo è scambiare i dati tra radice e foglia più a sinistra per poi cancellarla. Così facendo è evidente che i 2 sottoalberi mantengano la proprietà di essere alberi heap e l'unico nodo a poter introdurre una violazione sia la radice.

Tale proprietà può essere sfruttata; Essendo i 2 sottoalberi heap, il massimo del nuovo albero sarà o nella radice del sottoalbero destro o in quella del sinistro

Supponiamo di avere il caso in figura e quindi scambiamo x con $y \rightarrow$ Così facendo restiamo nella stessa condizione di prima (nella quale l'unica violazione si trova alla radice del sottoalbero destro con i sottoalberi h_1 e h_2 entrambi heap).



Procedendo ricorsivamente con questo approccio arriveremo alle foglie che saranno il nostro caso base.

Con questo algoritmo è evidente che avremo una chiamata ricorsiva per ogni chiamata e quindi l'algoritmo può fare un numero

di chiamate ricorsive al più pari all'altezza dell'albero, ovvero $\lceil \log n \rceil$.

Dunque possiamo affermare che questo algoritmo ha esecuzione pari a $O(\log n)$.

Non possiamo ancora dire di aver migliorato l'algoritmo → Se la costruzione dello heap richiedesse tempo quadratico allora la suddetta ottimizzazione non avrebbe senso.

Scriviamo l'algoritmo descritto supponendo che la sequenza in input sia *quasi heap*, ovvero abbia la situazione descritta in precedenza (di conseguenza l'unico nodo a poter violare la condizione è la radice → Se l'input iniziale non ha questa caratteristica l'algoritmo non funziona).

```
1 Heapify(A, i) // Dove i → indice del sottoalbero
2   // Definiamo il figli della radice
3   sx = 2i
4   dx = 2i + 1
5   // Supponiamo che heapsize sia una variabile globale che tenga traccia
6   // della dimensione corrente dell'albero
7   // (Nell'algoritmo principale abbiamo visto che dobbiamo cancellare i nodi)
8   IF 2i <= heapsize AND A[i] < A[sx] THEN
9     max = sx
10  ELSE
11    max = i
12  IF dx = heapsize AND A[max] < A[dx] THEN
13    max = dx
14  IF max != i THEN
15    Swap(A, i, max)
16  Heapify(A, max)
```

10.3.2 Problema 3

Cancellare un nodo mantenendo la proprietà strutturale è banale; Infatti cancellando il nodo foglia più a destra l'albero risultante sarà ancora un albero binario completo.

Si potrebbe pensare di sostituire il valore della radice (il nostro massimo) con quello di una foglia e cancellare quest ultimo (non è detto che la proprietà di ordinamento venga mantenuta e pertanto va ripristinata).

11 Lezione 14

11.1 Ragionamenti sull'algoritmo di `Heapify`

Il fatto che questo algoritmo non funzioni per sequenze che non siano quasi heap è dimostrabile facilmente.

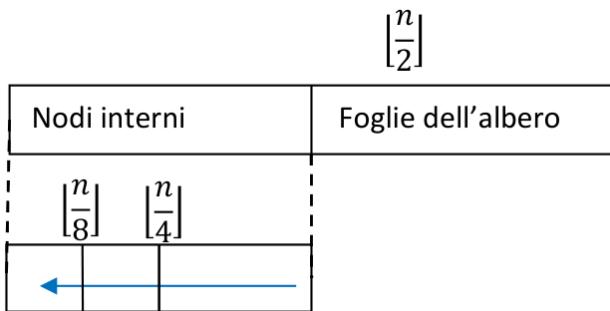
Infatti se funzionasse significherebbe che fosse possibile utilizzare `Heapify` così descritto per cercare il massimo di una sequenza (prenderemo la radice) → Sarebbe impossibile pensare che sia possibile cercare un massimo in un tempo minore di

$\Theta(n)$ (si ricordi che $T_H(n) = O(\log n)$).

Non potendo utilizzare Heapify direttamente sulla radice per sequenze arbitrarie, bisogna trovare un modo per trasformare una sequenza in heap. È banale che una foglia rispetti le proprietà di heap, allora se prendiamo i padri delle figlie possiamo applicare ad essi Heapify perché tutti i sottoalberi così definiti rispettano la proprietà necessaria al nostro algoritmo (di essere quasi heap).

Dopo aver applicato l'algoritmo a tutto il penultimo livello è evidente che tutti i sottoalberi di questo livello sono diventati heap rendendo ora possibile applicare Heapify ai padri dei padri delle foglie e così via fino a raggiungere la radice della sequenza.

L'unica cosa che resta da fare per l'algoritmo di costruzione dello heap è quello di come tradurre il precedente ragionamento in indici dell'array.



Questa rappresentazione evidenzia anche il giusto ordine da seguire per costruire l'heap dalla sequenza (ovvero da destra a sinistra).

Di seguito riportiamo l'algoritmo che costruisce l'heap

```

1 CostruisciHeap(A, n)
2   heapsize = n
3   FOR i = n/2 DOWNTON 1 DO
4     Heapify(A, i)

```

Possiamo sicuramente dire che ogni iterazione di Heapify è $\frac{n}{2} O(\log n)$ e quindi sia nel caso peggiore $O(n \log n)$.

Questo tipo di complessità andrebbe bene per i nostri scopi, ma andiamo a calcolare più precisamente il tempo di esecuzione poiché a seconda dell'approssimazione (se è eccessiva ad es) il tempo di esecuzione potrebbe cambiare.

Sia h l'altezza dell'albero, è evidente che la complessità di Heapify nel caso peggiore sarà $T_H(h) = O(h)$ dovendo alla peggio percorrere il percorso più lungo.

Sappiamo che $h = \lfloor \log n \rfloor$ e dunque è equivalente dire che $T_H(h) = T_H(n) = O(\log n)$, possiamo sfruttare questa equivalenza e studiare la complessità di `CostruisciHeap` usando l'altezza, poiché ragionare in termini di altezza semplifica il calcolo del tempo di esecuzione.

Per alberi di altezza 1 (nel nostro sono i sottoalberi rappresentati dai padri delle foglie e le foglie stesse) il contributo di Heapify sarà 1 e poiché il numero di nodi è $\lfloor \frac{n}{4} \rfloor$ allora ci saranno $\frac{n}{4}$ chiamate con un contributo totale pari a $\Theta(n)$ e non un

$O(n \log n)$.

Dobbiamo ragionare adesso sulle altre altezze.

Su tutti gli alberi di stessa altezza i , il contributo di una singola chiamata **Heapify** su ognuno di questi alberi sarà una certa costante moltiplicata all'altezza. In particolare:

Altezza sottoalbero	1	2	3	...	i	...
Costo Heapify	1	2	3		i	
Numero chiamate	$\frac{n}{4} = \frac{n}{2^2}$	$\frac{n}{8} = \frac{n}{2^3}$	$\frac{n}{2^4}$		$\frac{n}{2^{i+1}}$	

L'espressione ottenuta tenderà a ' per $i \rightarrow \infty$ ed è esattamente ciò che ci aspettiamo visto che il nostro scopo è salire dal penultimo livello (i padri delle foglie con altezza 1) fino alla radice dell'albero.

Per il tempo di esecuzione di **CostruisciHeap** resta solo da sommare tutti i contributi così definiti:

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

Tale sommatoria è assimilabile ad una serie geometrica con ragione $\frac{1}{2}$ ma sfortunatamente abbiamo un i come fattore moltiplicativo. Per la soluzione si possono sfruttare alcune proprietà delle derivate:

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

Da cui per la linearità delle derivate, segue:

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

Pertanto $\sum_{i=1}^n (i \cdot x^i)$ si riduce alla risoluzione di una derivata di una serie geometrica (in questo caso abbiamo una serie con ragione minore di 1, di conseguenza non abbiamo necessità di utilizzare la forma chiusa):

$$\begin{aligned} x \cdot 1 &\leq x \cdot \frac{d}{dx} \left(\sum_{i=1}^n x^i \right) \leq x \cdot \frac{d}{dx} \left(\frac{1}{1-x} \right) \\ \frac{d}{dx} \left(\frac{1}{1-x} \right) &= \frac{d}{dx} \left((1-x)^{-1} \right) = (-1)(1-x)^{-2}(-1) = \frac{1}{(1-x)} \end{aligned}$$

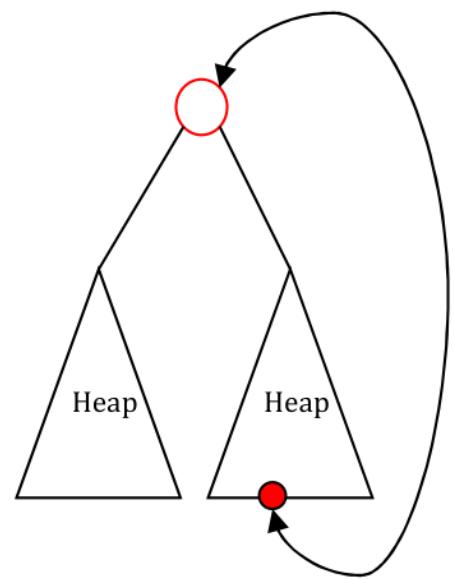
La nostra sommatoria cresce come una costante essendo limitata sia superiormente che inferiormente da una costante, più precisamente:

$$\frac{1}{2} \leq \sum_{i=1}^{\log n} \left(i \left(\frac{1}{2} \right)^i \right) \leq \frac{1}{\left(1 - \frac{1}{2} \right)^2} \implies \frac{1}{2} \leq \sum_{i=1}^{\log n} \left(i \left(\frac{1}{2} \right)^i \right) \leq 4 \implies T_{CH}(n) = O(n)$$

Sfruttando ancora il fatto che la complessità per la ricerca di un massimo non può essere meno di lineare risulta $T_{CH}(n) = \Theta(n)$

11.2 Analisi di HeapSort e algoritmo completo

Rappresentiamo graficamente l'idea ad alto livello dell'algoritmo:



Scambio l'elemento della radice (il massimo attuale della sequenza) con l'elemento della foglia più a destra.

Dopodichè rimuovo quest'ultima e mi trovo nella condizione ideale per applicare Heapify (si ricorda che alla cancellazione del nodo bisogna decrementare la variabile heapsize)

Il tempo di esecuzione può essere calcolato anche senza analizzare l'algoritmo, infatti:

$$T_{HS}(n) = T_{CH}(n) + T_H(n) = \Theta(n) + \sum_{i=2}^{n-1} O(\log n) = \Theta(n) + O(n \log n) = O(n \log n)$$

Anche se abbiamo approssimato la sommatoria, in questo caso si tratta di una approssimazione precisa (quindi non sarà necessario uno studio approfondito) poiché non è possibile in generale ordinare una sequenza di elementi arbitraria in meno di $n \log n$ (come vederemo in un teorema che dimostreremo in seguito) e quindi:

$$\Omega(n \log n) \leq T_{HS}(n) \leq O(n \log n) \implies T_{HS}(n) = \Theta(n \log n)$$

```

1 HeapSort(A, n)
2   CostruisceHeap(A, n)
3   FOR i = n DOWNTO 2 DO
4     // Scambio la radice con l'ultima foglia
5     SWAP(A, 1, i)
6     // Elimino l'ultima foglia in modo da ottenere un "quasi" heap
7     heapsize = heapsize - 1
8     Heapify(A, 1)

```

NB: Sono in una rappresentazione vettoriale, quindi decrementare la size del vettore è come cancellare l'ultima foglia nella rappresentazione ad albero.

Attualmente questo algoritmo di ordinamento è il migliore visto finora; anche se impiega $\Theta(n \log n)$ come Merge Sort, a differenza di questo non necessita memoria aggiuntiva.

12 Lezione 15

12.1 Quick Sort

L'idea è simile a quello del Merge Sort (sfrutta la tecnica del *Divide et Impera*), ma la decomposizione avviene a livello di istanza²⁰

Possiamo rappresentare questa tecnica come:

- ▶ Risolvi
 - ◊ Decomponi in sottoproblemi;
 - ◊ Risolvi i sottoproblemi;
 - ◊ Fondi soluzioni.

Mentre il Merge Sort ha una decomposizione in sottoproblemi semplici a discapito della fusione che è più complessa, l'algoritmo di Quick Sort decide di impiegare più tempo nella decomposizione così da semplificare di molto la fusione.

Infatti il quick sort decomporrà la sequenza in una maniera tale che alla fine si avranno le seguenti sequenze:



²⁰Non dividerà banalmente la sequenza a metà ma lo farà in base al valore degli elementi di tale sequenza.

Con $a_1 \leq a_{i+1}$, e quindi poiché entrambe le sottosequenze sono già ordinate e l'ultimo elemento della prima sequenza sia \leq al primo elemento della seconda, la fusione dei 2 è già ordinata.

Il metodo utilizzato da quick sort non riesce però a garantire che le 2 sottosequenze contengano più o meno lo stesso numero di elementi e ciò va ad impattare sul tempo di esecuzione.

Qualsiasi livello di ricorsione avrà da una parte una sequenza di q elementi e dall'altro $n - q$ con tutti gli elementi della sottosequenza sinistra minori o uguali a quelli dell'altra sottosequenza.

```

1 QuickSort(A, p, r)
2   IF p < r THEN // Siamo in un caso base
3
4     /*
5      Il punto in cui dividere la sequenza è più complicato da calcolare
6      Partiziona si occuperà di restituire l'indice che dividerà la sequenza e
7      farà in modo che tutti gli elementi della partizione da p a q siano
8      minori o uguali di quelli da q + 1 a r
9    */
10   q = Partiziona(A, p, r)
11   QuickSort(A, p, q)
12   QuickSort(A, q + 1, r)

```

Dopo le chiamate ricorsive non c'è bisogno di fare altro poiché questo algoritmo garantisce per transitività l'ordinamento totale della sequenza.

Se **Partiziona**, si comporta come descritto, **QuickSort** è corretto sotto le seguenti assunzioni:

- ▶ La partizione da p a q è strettamente minore in dimensione della sequenza da p a r ;
- ▶ La partizione da q a r è strettamente minore della sequenza da p a r ;
- ▶ Al termine di **Partiziona(A, p, r)** vale che

$$\forall i : p \leq i \leq q \wedge \forall j : q + 1 \leq j \leq r, A[i] \leq A[j]$$

(qualunque elemento prendo a sinistra, esso è minore o uguale di qualunque elemento prendo a destra)

Ed è chiaro che queste proprietà siano fondamentali sia per il ragionamento di ipotesi induttiva (le prime 2 proprietà) e sia per il fatto di non dover fare nessun merge alla fine (l'ultima).

Vediamo di dimostrare le prime 2 assunzioni, che possono essere rappresentate da un'unica equazione matematica:

$$p \leq q < r$$

Infatti, se è questo il caso è evidente che $r - p + 1 < q - p + 1 \wedge r - p + 1 < r - q + 1$.

La precedente proprietà se non fosse soddisfatta distruggerebbero l'algoritmo.

Supponiamo che $q = p - 1$, avremmo una chiamata ricorsiva (`QuickSort(A, p, p - 1)`) ovvero una istanza di 0 elementi.

Questa chiamata terminerebbe senza aver effettuato nulla (caso base), ma ciò comporta che la seconda chiamata, ovvero `QuickSort(A, p + 1 - 1, r)`, avrà tutti gli elementi della chiamata di partenza con la conseguenza che la suddetta chiamata si ripeterà in infinito e l'algoritmo non terminerà.

Caso analogo per $q = r \rightarrow QuickSort(A, p, r)$ sarà la chiamata che manderà in loop e l'algoritmo `QuickSort(A, r + 1, r)` la sequenza vuota.

Quindi **Partiziona** deve garantire 2 proprietà:

1. $p \leq q < r$ (per il motivo descritto precedentemente);
2. $\forall i : p \leq i \leq q \wedge \forall j : q + 1 \leq j \leq r, A[i] \leq A[j]$ (Scontato, in quanto non è si ha la certezza che la sequenza totale sia già ordinata).

12.1.1 Partiziona

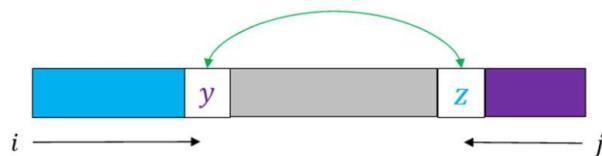
Quando viene chiamato **Partiziona** si ha la certezza che $p < r$, dunque possiamo scegliere un elemento $x \in A$, (detto anche pivot), metteremo nella parte sinistra tutti gli elementi minori od uguali di x , mentre a sinistra quelli maggiori o uguali²¹

In linea di principio x potrebbe essere scelto a piacimento, ma per come descriveremo l'algoritmo è necessario che venga spostato alla prima posizione

Dunque per semplicità prenderemo direttamente il primo elemento della sequenza.



Per scorrere la sequenza avrò 2 indici, i che partirà da sinistra e j che partirà da destra. Il primo indice i si sposterà a destra fin tanto che troverà un valore $y \geq x$ (i valori minori sono già nella posizione giusta), mentre j verrà decrementato fino a quando non sarà su un valore $z \leq x$ (quel valore dovrà andare nella sottosequenza di sinistra). Una volta che si sono fermati entrambi gli indici e $i < j$ scambierà i valori e ripeterà il processo fino a quando gli indici si incroceranno $i \geq j$.



²¹Le condizioni di uguaglianza sono necessarie sia a destra che a sinistra per garantire che nessuna delle 2 sottosequenze sia vuota (altrimenti andremo in controllo al loop precedentemente descritto). Se ad esempio avessi per la sequenza di sinistra la condizione di strettamente minore, potrebbe capitare di scegliere come **pivot** il minimo della sequenza e quindi nessun elemento verrebbe spostato a sinistra.

Nello scrivere l'algoritmo utilizzeremo il ciclo **REPEAT ... UNTIL** (condizione), questo ripeterà il blocco interno (fa almeno una iterazione) fino a quando la condizione non è verificata.

```

1 int Partiziona(A, p, r)
2     x = A[p]
3     j = r + 1
4     i = p - 1
5     REPEAT
6         REPEAT
7             j = j - 1
8             UNTIL A[j] <= x
9
10        REPEAT
11            i = i + 1
12            UNTIL A[i] >= x
13            IF i < j THEN
14                Swap(A, i, j)
15        UNTIL i >= j

```

Per confermare la correttezza di questo algoritmo si veda se all'istante in cui viene eseguita la linea 15 si ha $p \leq j < r$ poiché questo ci assicurerà la validità della prima proprietà necessaria per la correttezza di **QuickSort**, ovvero $p \leq j < r$.

A tal proposito, dobbiamo dimostrare che i casi $j < p$ e $j \geq r$ siano impossibili; per fare ciò basta controllare che non si verificano i casi $j = p - 1$ (essendo $i = p - 1$ è l'unico $j < p$ che potrebbe verificarsi vista la scrittura del nostro algoritmo) e $j = r$ (anche se $j = r + 1$ all'inizio, viene per forza decrementato almeno una volta).

- ▶ Caso $j = r \rightarrow$ Sappiamo che j viene decrementato almeno una volta e non riceve mai incrementi. Affinché sia $j = r$ significhi che siamo al **primo** ciclo di iterazione del **REPEAT** esterno; bisogna quindi dimostrare che i non arrivi mai a r , poiché è l'unico modo che ha per uscire dal ciclo con $j = r$. Visto che siamo alla prima iterazione allora $i = p - 1$ e al primo incremento di i (riga 10) non abbiamo ancora fatto scambi. Dunque x (il pivot) è ancora in prima posizione, ma allora al primo incremento di i esco subito dal ciclo essendo $i = p$ e $A[i] = x$. A questo punto $i < j$ e quindi eseguo la riga 13 (non si è ancora verificata la condizione di $i \geq j$). Ne consegue che devo fare almeno un'altra iterazione del blocco con la conseguenza che $j < r$;
- ▶ Caso $j = p - 1 \rightarrow$ Per dimostrare che questo caso non si verifichi basta garantire che il primo **REPEAT** interno (riga 6-8) non si ripeta all'infinito visto che $p < r$ (vero perché altrimenti non sarebbe eseguito lo stesso **Partiziona**) garantisce che j si trovi tra p ed r . Visto che non abbiamo modo di garantire di essere alla prima iterazione non è detto che x sia ancora nella prima posizione della sequenza poiché potrebbero esserci degli scambi, ma se questi sono avvenuti significa certamente che in $A[p]$ ci sia un valore minore o uguale a x . Ciò garantisce che j si fermerà sicuramente in p (ragionamento astratto, infatti se ci sono stati degli scambi allora $i > p$ e quindi j si fermerà sicuramente prima di arrivare a p essendo la condizione di uscita $i \geq j$; ovvero si esce dal ciclo quando gli indici si incrociano). Visto che la variabile i può essere solo incrementata è evidente che non potrà essere minore di p ma allora ciò significa che se j arriva a p in quella stessa iterazione la condizione $i \geq j$ sarà verificata e l'algoritmo terminerà con un valore $j \geq p$.

L'algoritmo **Partiziona** garantisce che alla sua terminazione, quando gli indici si incontrano, siano verificate le seguenti proprietà:

$$\forall z : p \leq z \leq j, A[z] \leq x \text{ e } \forall t : j + 1 \leq t \leq r, A[t] \geq x$$

Queste possono essere unite:

$$\forall z, t : p \leq z \leq j < t \leq r, A[z] \leq x \leq A[t]$$

Che rappresenta la seconda proprietà che volevamo garantire:

$$\forall i : p \leq i \leq q \wedge j : q + 1 \leq j \leq r, A[i] \leq A[j]$$

È facile osservare che se richiedessi il $<$ invece del \leq (analogamente $>$ al posto di \geq) non riuscirei più a garantire la prima proprietà che deve verificarsi al termine di **Partiziona** per la correttezza di **QuickSort**.

12.1.2 Analisi asintotica

Per il **QuickSort** non bastano le tecniche viste sinora poiché non è facile sapere quante volte le istruzioni di **Partiziona** vengano eseguite; in particolare per i REPEAT UNTIL interni posso solo dire che la somma delle loro iterazioni sia $n + 1$ o $n + 2$ (visto che l'algoritmo termina o con $i = j$ se l'input è dispari o $i = j + 1$ se l'input è pari) essendo l'unica condizione di uscita $i \geq j$ (quindi contributo lineare).

Il corpo dell'if al massimo viene eseguito per $\frac{n}{2}$ volte e dunque $T_P(n) = \Theta(n) + O(n) = \Theta(n)$ (Paragonandolo a Merge Sort abbiamo praticamente spostato il contributo lineare di merge all'inizio dell'algoritmo).

```

1  QuickSort(A, p, r)
2    IF p < r THEN
3      q = Partiziona(a, p, r) //  $T_{QS}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{QS}(q) + T_{QS}(n - q) + \Theta(n) & \text{se } n > 1 \end{cases}$ 
4      QuickSort(A, p, q)
5      QuickSort(A, q + 1, r)

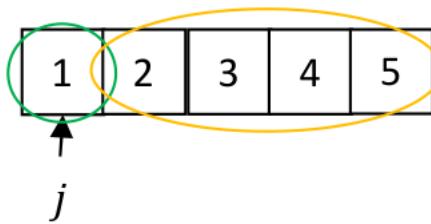
```

A causa di partiziona non sappiamo come viene diviso l'input tra le 2 chiamate poiché dipende da come viene scelto il pivot e dall'istanza di input. Posso fare i seguenti ragionamenti:

- ▶ Se tutti gli elementi sono uguali avrà una divisione della sequenza in 2 parti quasi uguali e questo varrà per ogni chiamata ricorsiva



- Se invece ho una sequenza ordinata, verrà suddivisa in una partizione da un solo elemento (quella di sinistra) e l'altra con i restanti elementi (quello di destra)



Questi 2 casi sono stati già studiati in precedenza. Se $q = \lfloor \frac{n}{2} \rfloor$, ma allora avremo $T_{QS}(n) = \Theta(n \log n)$. Essendo

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

Esattamente come merge sort, mentre per le sequenze ordinate si ha:

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

Quindi $T_{QS}(n) = \Theta(n^2)$ (simile all'esempio sul fattoriale). Più precisamente per le sequenze ordinate avremo un albero degenere dove un livello i -esimo (eccetto l'ultimo che è costante) ha un contributo di $n - 1$; dunque:

$$\sum_{i=0}^{n-1} (n-1) = n + (n-1) + (n-2) + \dots + \underbrace{(n-(n-1))}_1 = \sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$$

Stranamente quick sort si comporta nel modo peggiore per sequenze dove tecnicamente non ci sarebbe nessuna operazione da effettuare.

Si fa presente che a seconda del tipo di sequenza di dimensione n l'albero di ricorrenza potrebbe essere molto diverso. Il nostro scopo è capire che tipo di forma potrebbero avere gli alberi generati dalla equazione di ricorrenza:

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

Ovviamente $1 \leq q \leq n - 1$ altrimenti l'algoritmo non potrebbe terminare (i casi di $q = 0$ e $q = n$ sono già stati discussi); da questo ricaviamo che tutti i nodi interni hanno grado 2 poiché non trovandoci in un caso base l'algoritmo fa esattamente 2 chiamate ricorsive.

Visto che i casi base si hanno con sottosequenze di un elemento è evidente che ci saranno tante foglie quanti sono gli elementi della sequenza di input.

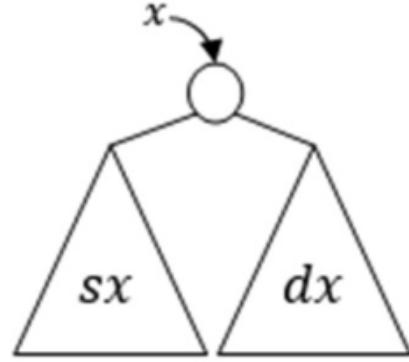
Questo ci permette di dimostrare che se il numero di foglie n_f è k allora tutti i nodi interni n_i sono $k - 1$.

Dimostriamo per induzione sull'altezza dell'albero che vale $n_f = n_i + 1$

- Per $h = 0$ avremo l'albero composto dalla sola radice e

quindi $\underbrace{1}_{n_f} = \underbrace{0}_{n_i} + 1$;

- Per $h = i > 0$, la radice avrà grado 2 e quindi sia il sottoalbero destro che sinistro saranno non vuoti e con altezza $h < i$ e quindi $n_{f_{sx}} = n_{i_{sx}} + 1$ e $n_{f_{dx}} = n_{i_{dx}} + 1$, mentre il numero totale delle foglie dell'albero sarà semplicemente $n_f = n_{f_{sx}} + n_{f_{dx}}$ in quanto una foglia non può appartenere ad entrambi i sottoalberi. Il numero di nodi interni nell'albero radicato in x sarà evidentemente $n_i = n_{i_{sx}} + n_{i_{dx}} + 1$ (la radice).



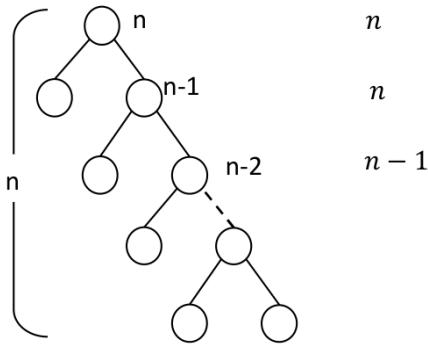
Mettendo insieme i precedenti ragionamenti, risulta:

$$n_f = \underbrace{n_{i_{sx}} + 1}_{n_{f_{sx}}} + \underbrace{n_{i_{dx}} + 1}_{n_{f_{dx}}} = \underbrace{n_{i_{sx}} + n_{i_{dx}} + 1}_{n_i} + 1$$

13 Lezione 16

13.1 Caso medio

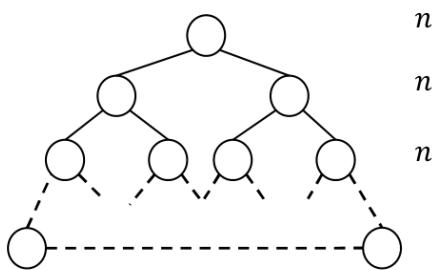
Prendiamo l'albero di ricorrenza del caso peggiore e andiamo a calcolare il contributo di ogni livello:



Da questa rappresentazione ricaviamo che il contributo di un livello $l \geq 1$ è $n - l + 1$. Di conseguenza

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

13.2 Caso migliore



Quindi $T(n) = \sum_{l=0}^h n$ con h altezza di un albero completo e quindi pari a $\log n$ ne consegue

$$T(n) = \sum_{l=0}^{\log n} n = \Theta(n \log n)$$

13.3 Analisi

È facile notare che per qualsiasi albero costruisca ogni livello avrà contributo lineare poiché sia quelli del caso migliore che del caso peggiore sono lineari e tutti gli altri alberi sono ovviamente nel mezzo.

Il tempo di esecuzione dipende quindi dall'altezza e non potendo sommare più di un numero lineare di livelli (essendo l'altezza dell'albero peggiore n) il caso peggiore è ovviamente $\Theta(n^2)$ mentre il migliore (parliamo dell'albero con altezza minima) è un $\Theta(n \log n)$.

Allo stato attuale non possiamo nemmeno affermare che questo algoritmo sia migliore di `insertion sort`, il quale aveva un caso migliore addirittura lineare. Bisogna dunque calcolare la media tra caso migliore e peggiore, ma non possiamo farlo semplicemente con una media aritmetica come in `insertion sort`.

Decomponiamo il problema sapendo che ad ogni livello di ricorsione l'algoritmo si occupa di scegliere, tramite `Partiziona`, su una certa istanza il valore q e quindi ogni nodo sceglie localmente un valore di q ; fissato quest ultimo verranno generati i 2 sottoproblemi con dimensione q e $n - q$.

13.4 Calcolo del tempo medio

Calcoliamo adesso il tempo medio di tutte le istanze in cui `Partiziona` fa la stessa scelta.

Dividiamo quindi il problema in classi di equivalenza → Ogni istanza di dimensione n che partiziona la sequenza nello stesso modo

(più precisamente la *prima* scelta di q deve essere la stessa, le altre non è detto che avvengano allo stesso modo) appartiene alla stessa classe di equivalenza.

Decomposto in classi calcoleremo la media di ogni classe facendone poi una media aritmetica.

Per effettuare il suddetto partizionamento, prendiamo istanze di dimensioni arbitraria in cui tutti gli elementi sono differenti
 → Tale assunzione è una approssimazione per eccesso visto che più elementi uguali ci sono e meno sproporzionata sarà la partizione (si faccia riferimento al caso migliore).

Secondo questa ipotesi la dimensione delle partizioni è univocamente determinata da quello che chiameremo il rango del pivot.

Data una sequenza e un pivot x , il *rango* $r(x)$ sarà il numero di elementi con valore \leq ad x nella sequenza. Essendo il pivot presente nella sequenza avremo $r(x) \geq 1$ perché almeno il pivot sarà \leq di sé stesso, più precisamente se come pivot scegliersi il minimo della sequenza avrei esattamente $r(x) = 1$, mentre con il massimo sarà $r(x) = n$.

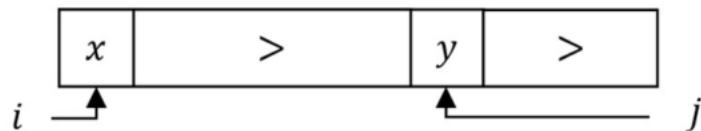
La scelta di q è proprio il pivot; se il rango è uno avremo il pivot a sinistra e tutto il resto a destra. Dunque

$$r(x) = 1 \implies q = 1$$

Anche per

$$r(x) = 2 \implies q = 1$$

perché con 2 valori minori o uguali al pivot avremo il seguente caso (sia $y \leq x$)



Dove avverrà l'unico scambio con la conseguenza di avere anche in questo caso una partizione di un elemento e l'altra con i restanti.

Seguendo questa logica per $r(x) = 3 \implies q = 2$, $r(x) = 4 \implies q = 3$ e così via (da 2 elementi in poi q crescerà sempre). Pertanto

$$r(x) \geq 2 \implies q = r(x) - 1$$

Scegliere il pivot equivale dunque a scegliere la dimensione della partizione q , ed il nostro algoritmo sceglie proprio il rango e dunque le classi di equivalenze saranno esattamente n :

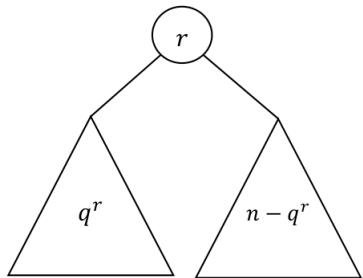
$r(x)$	1	2	\dots	n
	T_M^1	T_M^2		T_M^n

Intuitivamente $T_M^r(n)$ rappresenta la funzione che dato il rango r da il tempo medio delle sequenze di quella classe d'equivalenza

Il tempo medio può essere espresso con una media aritmetica di tutti i tempi medi delle classi d'equivalenza, ovvero:

$$T_M(n) = \frac{1}{n} \left(\sum_{r=1}^n T_M^r(n) \right)$$

Ma dato un certo rango r avremo il seguente albero di ricorrenza:



Intuitivamente

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

Dunque per l'albero a sinistra si ha

$$T_M^r(n) = \Theta(n) + T_M(q^r) + T_M(n - q^r)$$

Da cui segue:

$$T_M(n) = \frac{1}{n} \cdot \sum_{r=1}^n (T_M(q^r) + T_M(n - q^r) + \Theta(n))$$

Possiamo semplificare l'equivalenza $q^r = r - 1$. Scorporando il caso del rango $r = 1$ abbiamo una corrispondenza univoca per gli altri. Possiamo scrivere la precedente equazione come segue:

$$T_M(n) = \frac{1}{n} \left(\underbrace{(T_M(1) + T_M(n-1) + \Theta(n))}_{\substack{r(x)=1 \implies q=1}} + \sum_{r=2}^n (T_M(n-q^r) + \Theta(n)) \right)$$

Sappiamo che $T_M(1) = \Theta(1)$ e che $T_M(n-1) = O(n^2)$ perché non può essere peggiore del caso peggiore (questa approssimazione sarà ininfluente). Inoltre $r = q - 1$, ma allora:

$$T_M(n) = \frac{1}{n} \left(\underbrace{(\Theta(1) + \Theta(n^2) + \Theta(n))}_{O(n^2)} + \sum_{q=1}^{n-1} (T_M(q) + T_M(n-q) + \Theta(n)) \right)$$

Si noti che $\sum_{q=1}^{n-1} (T_M(q))$ e $\sum_{q=1}^{n-1} (T_M(n-q))$ sono esattamente gli stessi termini sommati in ordine inverso, quindi:

$$\begin{aligned}
T_M(n) &= \frac{1}{n} \left(O(n^2) + 2 \sum_{q=1}^{n-1} T_M(q) + \sum_{q=1}^{n-1} \Theta(n) \right) = \frac{1}{n} \left(\underbrace{O(n^2 + \Theta(n^2))}_{\Theta(n^2)} + 2 \sum_{q=1}^{n-1} T_M(q) \right) \\
&= \frac{\Theta(n^2)}{n} + \frac{2}{n} \sum_{q=1}^{n-1} T_M(q) = \Theta(n) + \frac{2}{n} \sum_{q=1}^{n-1} T_M(q)
\end{aligned}$$

Essendo questa equazione di ricorrenza complessa, introduciamo una tecnica di risoluzione.

13.5 Tecnica per validare un'equazione di ricorrenza

Dimostriamo che $T_M(n) = O(n \log n)$ così da confermare $T_M(n) = \Theta(n \log n)$; bisogno verificare (utilizzando l'induzione) che

$$\exists c, n_0 > 0 : \forall n \geq n_0, T_M(n) \leq c(n \log n)$$

Il caso induttivo sarà evidentemente valido per $n \geq 2$, essendo il caso $n = 1$ non verificato; visto che $\log 1 = 0$ risulta che $T_M \leq 0$ (ciò è assurdo), e di conseguenza il caso base è $n = 2$.

Grazie al fatto che $1 \leq q \leq n - 1$ possiamo scrivere $T_M(q) \leq c(q \log q)$ (questa sarà la nostra ipotesi induttiva); per transitività risulta:

$$T_M(n) = \Theta(n) + \frac{2}{n} \sum_{q=1}^{n-1} T_M(q) \leq \Theta(n) + \frac{2}{n} \sum_{q=1}^{n-1} c(q \log q)$$

Assumiamo per il momento che sia vera la seguente proprietà (che andremo a dimostrare successivamente):

$$\sum_{q=1}^{n-1} (q \log q) \leq \frac{n^2 \log n}{2} - \frac{n^2}{8}$$

Da ciò segue:

$$T_M(n) \leq \Theta(n) + \frac{2c}{n} \sum_{q=1}^{n-1} (q \log q) \leq \Theta(n) + \frac{2c}{n} \left(\frac{n^2 \log n}{2} - \frac{n^2}{8} \right) = \Theta(n) + c(n \log n) - \frac{cn}{4}$$

A questo punto se dimostriamo che $\Theta(n) - \frac{cn}{4} \leq 0$ allora risulterà (dopo la verifica del caso base) che $T_M(n) \leq c(n \log n)$

Sappiamo che $\Theta(n)$ è assimilabile ad un kn , allora risulta che $kn \leq \frac{cn}{4}$.

La costante k è fissata dalla relazione Theta, ma la costante c può essere scelta arbitrariamente. Basta scegliere pertanto $c \geq 4k$ per concludere che:

$$T_M(n) = O(n \log n) \text{ per } n \geq 2$$

13.5.1 Validazione per il caso base $n = 2$

$$T_M(2) = \Theta(1) + \frac{2}{2} \sum_{q=1}^{2-1} T_M q = \Theta(1) + T_M(1) = \underbrace{\Theta(1)}_{\substack{\text{costo di} \\ \text{partiziona}}} + \underbrace{\Theta(1)}_{\substack{\text{caso base} \\ \text{dell'equazione di ricorrenza}}} = k + a$$

Per $c \geq k + a$ anche il caso base è verificato.

Se scelgo un $c = \max\{k + a, 4k\}$ vale sia il caso base che quello induttivo e quindi risulta dimostrata la nostra tesi $T_M(n) = O(n \log n)$, da cui per il teorema secondo il quale un algoritmo di ordinamento non può essere meno di $n \log n$, segue:

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

13.6 Maggiorazione di una sommatoria

$$\sum_{q=1}^{n-1} (q \log q) \leq \frac{n^2 \log n}{2} - \frac{n^2}{8}$$

Dimostriamo la precedente assunzione, il modo più semplice per maggiorare $\sum_{q=1}^{n-1}$ è sfruttare il fatto che $q < n$ (ricavato da $1 \leq q \leq n - 1$) e quindi $\log n \implies q \log q \leq q \log n$. Dunque:

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

Tuttavia $\frac{n \log n}{2} \leq \frac{n^2}{8} \rightarrow$ Significa che la nostra maggioranza è stata eccessiva. Quello che possiamo fare è spezzare la sommatoria in 2 parti così da fare delle approssimazioni più precise:

$$\begin{aligned}
\sum_{q=1}^{n-1} &= \underbrace{\sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} (q \log q)}_{q \leq \lceil \frac{n}{2} \rceil} + \underbrace{\sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} (q \log q)}_{\text{approssimiamo come prima}} \leq \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} (q \log \frac{n}{2}) + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} (q \log q) = \\
&= \log \frac{n}{2} \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q = (\log n - 1) \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q = \\
&= \underbrace{\log n \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q}_{\text{uniamo le sommatorie}} + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q = \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q
\end{aligned}$$

Ora $\log n \sum_{q=1}^{n-1} (q)$ l'abbiamo già risolta; visto che stiamo maggiorando non c'è problema a sottrarre qualcosa di più piccolo e quindi sfruttiamo il fatto che $\lceil \frac{n}{2} \rceil \geq \frac{n}{2}$

$$\begin{aligned}
\sum_{q=1}^{n-1} &\leq \left(\frac{n^2 \log n}{2} - \frac{n \log n}{2} \right) - \sum_{q=1}^{\frac{n}{2}-1} q = \frac{n^2 \log n}{2} - \frac{n \log n}{2} - \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} = \\
&= \frac{n^2 \log n}{2} - \frac{n \log n}{2} - \frac{n^2}{8} + \frac{n}{4} \leq \frac{n^2 \log n}{2} - \frac{n^2}{8}
\end{aligned}$$

Poiché $\frac{n \log n}{2} \geq \frac{n}{4}$ e quindi $\frac{n}{4} - \frac{n \log n}{2} \leq 0$, togliendo un valore negativo la maggioranza resta valida.

13.7 Conclusioni su QuickSort

Abbiamo dimostrato che nel caso medio QuickSort ha un comportamento ottimo → Il tempo di esecuzione è quasi sempre $\Theta(n \log n)$ rendendolo nella pratica uno dei migliori algoritmi di ordinamento.

14 Lezione 17

14.1 Problema generale sull'ordinamento

Abbiamo visto di come non sia possibile ordinare una sequenza arbitraria di lunghezza n in meno di $n \log n$ ²². Ora dimostreremo questo enunciato

Se parliamo di sequenze arbitrarie è invece possibile dimostrare che il caso medio e peggiore (non vale per il caso migliore) richiedono un tempo minimo di $n \log n$.

²²È importante ricordare che esistono algoritmi in grado di risolvere l'ordinamento per una specifica sequenza in maniera lineare → Queste tecniche si basano sul fatto che l'input non sia arbitrario, ma abbia determinate proprietà (il counting sort ne è un esempio).

Fondamentale è definire lo spazio delle possibili operazioni per un generale algoritmo di ordinamento.

Una cosa certa è che il confronto degli elementi è un'operazione essenziale per l'ordinamento e quindi il problema può essere risolto solo attraverso i confronti.

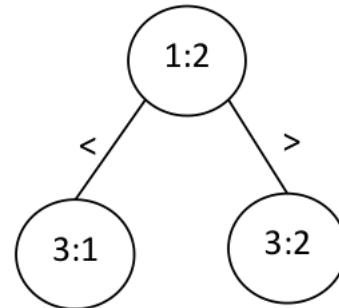
Lo scambio invece non è essenziale → Viene utilizzato solo per tener traccia dei confronti.

Da un punto di vista astratto possiamo quindi tenere traccia soltanto dei confronti ignorando gli scambi.

Il numero di confronti darà effettivamente la stima asintotica dell'algoritmo. Bisogna capire quanti confronti sono necessari per ordinare una sequenza nel caso peggiore e nel caso medio

14.2 Alberi di decisione

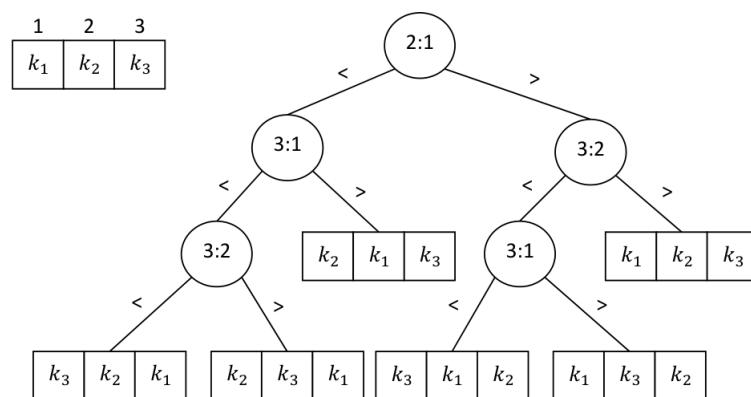
La natura di un confronto è semplicemente una funzione booleana → un confronto può essere visto come un nodo interno con all'interno 2 elementi che rappresentano gli indici dei valori da confrontare; l'arco rappresenterà l'esito del confronto:



Dopo aver confrontato l'elemento in posizione 1 con quello in posizione 2 non è detto che il successivo elemento da confrontare sia lo stesso a prescindere dall'esito

Potrei, ad es., confrontare 3 : 2 se $k_1 > k_2$ (mi sposto sul figlio destro) o 3 : 1 se $k_1 < k_2$

Di seguito riportiamo un esempio più concreto ordinando una sequenza di 3 elementi sfruttando gli esiti dell'albero di decisione²³ :



²³È interessante notare che l'albero precedentemente rappresentato ha come foglie tutte le permutazioni della sequenza (k_1, k_2, k_3) , che nel nostro caso rappresentano le possibili soluzioni di una sequenza ordinata. Tale albero è valido per qualsiasi sequenza di lunghezza 3 e quindi ha ordine 3.

In generale un albero di decisione è applicabile in un qualsiasi contesto in cui serve strutturare un insieme di decisioni:

- ▶ I nodi interni rappresentano le decisioni da prendere;
- ▶ Le foglie rappresentano le soluzioni dovute a tali scelte.

14.3 Algoritmi di ordinamento e alberi di decisione

Implicitamente ogni algoritmo di ordinamento che si basa su confronti crea un albero di decisione (un albero di decisione ci permette di ordinare una qualsiasi sequenza di dimensione fissata n).

Viceversa un algoritmo di ordinamento vale per qualsiasi $n \rightarrow$ È evidente che non c'è una corrispondenza univoca tra i 2.

Quello che possiamo dire è che per un algoritmo di ordinamento è possibile associare una classe di alberi di decisione, uno per ogni ordine:

- ▶ `AlgoritmoOrdinamento(1)` è associato ad un albero di decisione 1;
- ▶ `AlgoritmoOrdinamento(2)` è associato ad un albero di decisione 2;
- ▶ ...

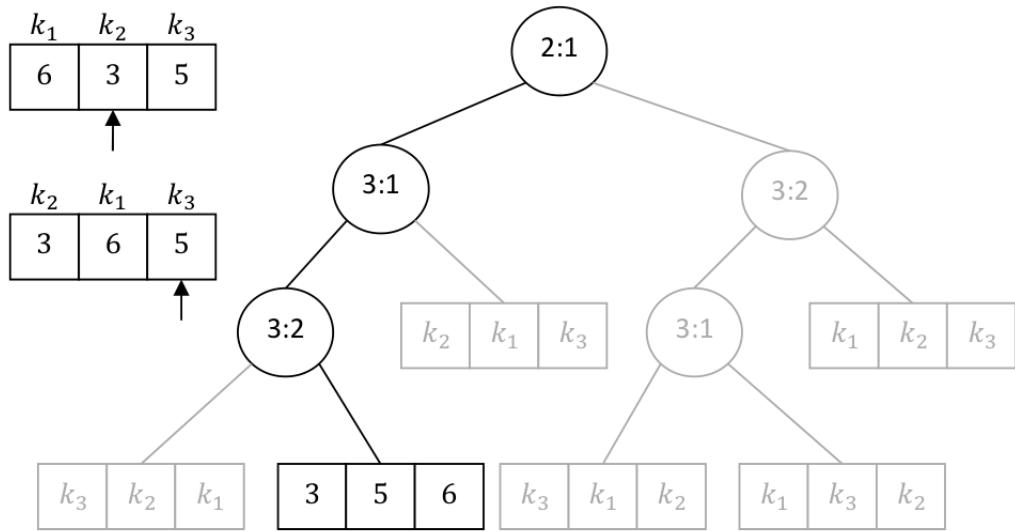
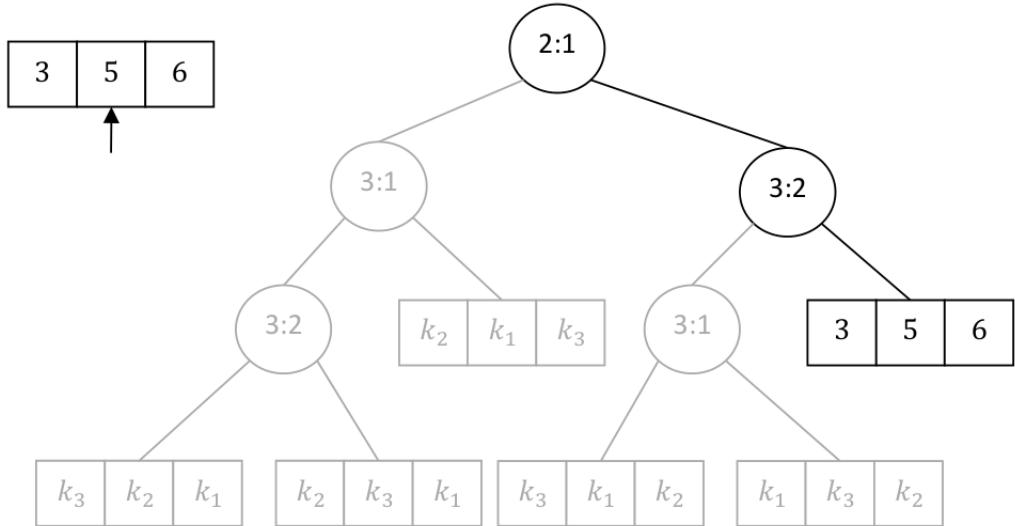
Ad algoritmi di ordinamento diversi possono essere associati 2 alberi di decisione diversi (basta pensare che non esiste un solo albero di decisione per una sequenza di ordine 3).

14.3.1 Proprietà degli alberi di decisione per gli algoritmi di ordinamento

- ▶ Alberi binari;
- ▶ Ogni nodo interno ha grado 2 (l'esito può essere < o >);
- ▶ Il numero di foglie è pari alle permutazioni della sequenza di input (quindi $n !$);
- ▶ Il percorso della radice a qualsiasi foglia confronta come minimo tutti gli elementi adiacenti
 - ◊ Per la sequenza (k_1, k_2, k_3) in qualsiasi percorso devo avere *almeno* i nodi $1 : 2$ (oppure $2 : 1$) e $2 : 3$ (oppure $3 : 2$);
- ▶ Non posso avere meno di $n - 1$ nodi interni, poiché il numero di coppie adiacenti in una sequenza è proprio $n - 1$ (conseguenza della precedente proprietà).

Negli alberi di decisione vengono rappresentati solo i confronti (non gli scambi, in quanto si suppone di avere memoria di tutti i confronti potendo di conseguenza ordinare la sequenza direttamente alla fine senza bisogni di passaggi intermedi).

Considerando tutto ciò presentiamo un esempio di albero di decisione per l'algoritmo `InsertionSort(A, 3)`:



Di conseguenza, fissato n , prendendo l'albero di decisione di ordine n ed una istanza, il numero di operazioni elementari che un algoritmo di ordinamento sarà asintoticamente equivalente al numero di decisioni che fa il corrispettivo albero di decisione (ovvero il numero di confronti prima di raggiungere la foglia).

Ciò implica che il numero di confronti che un algoritmo può fare è relazionato all'altezza dell'albero (nel caso di insertion sort sarà quadratico).

14.4 Dimostrazione del teorema sull'ordinamento

Fissato n ho un numero finito di alberi di decisione; quindi, il nostro scopo sarà vedere tra tutti gli alberi quello con altezza minima, in modo da trovare il numero minimo di confronti che un algoritmo fa nel caso peggiore (analizziamo il tempo di esecuzione del

miglior caso peggiore).

Qualunque sia l'albero T_n , esso avrà $n!$ foglie disposte su un altezza h , ma visto che sappiamo che in un albero binario il numero di foglie è al massimo 2^h posso scrivere $n! \leq 2^h \implies h \geq \log(n!)$.

Ne consegue che l'altezza di un albero di decisione non può essere meno di $\log(n!)$. Ciò si traduce in:

$$n! = \prod_{i=1}^n i \implies \log(n!) = \log\left(\prod_{i=1}^n i\right) \xrightarrow{\log(a \cdot b) = \log a + \log(b)} \log(n!) = \sum_{i=1}^n \log i \leq h$$

Essendo interessati ad una stima di questa sommatoria, proviamo a maggiorarla. Cerchiamo quindi qualcosa di più piccolo in modo da poter avere $h \geq \sum_{i=1}^n \log i \geq (\)$. Sappiamo che:

$$\sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \log n \implies \sum_{i=1}^n \log i = O(n \log n)$$

(non molto utile → il nostro scopo è garantire che h abbia un limite inferiore asintotico).

$$\sum_{i=1}^n \log i \geq \underbrace{\sum_{i=\lceil \frac{n}{2} \rceil}^n \log i}_{\text{sommo meno valori}} \geq \underbrace{\sum_{i=\lceil \frac{n}{2} \rceil}^n \log \frac{n}{2}}_{\text{sommo sempre il più piccolo}} = \log \frac{n}{2} \cdot \sum_{i=\lceil \frac{n}{2} \rceil}^n \log 1 = \frac{n}{2} \log \frac{n}{2} = \Theta(n \log n)$$

In questo modo abbiamo raggiunto l'obiettivo, infatti:

$$\frac{n}{2} \log \frac{n}{2} \leq \sum_{i=1}^n \log i \leq n \log n \implies h \geq \Theta(n \log n)$$

Abbiamo dimostrato quindi che per il caso peggiore un algoritmo di ordinamento, dovendo per forza fare tanti confronti quanto la lunghezza del percorso più lungo del suo albero di decisione, non può essere meglio di un $\Theta(n \log n)$

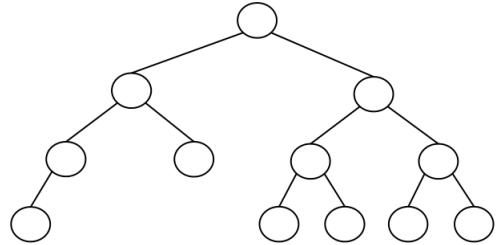
14.4.1 Dimostrazione per il caso medio

Il caso medio risulta più complesso → Per poter calcolare il tempo di esecuzione dovremo fare una media aritmetica tra la lunghezza del percorso esterno (somma dei percorsi dalla radice a ciascuna foglia) e il numero delle foglie dell'albero. Ad es:

$$T_M(n) = \frac{LPE}{\# \text{ foglie}}$$

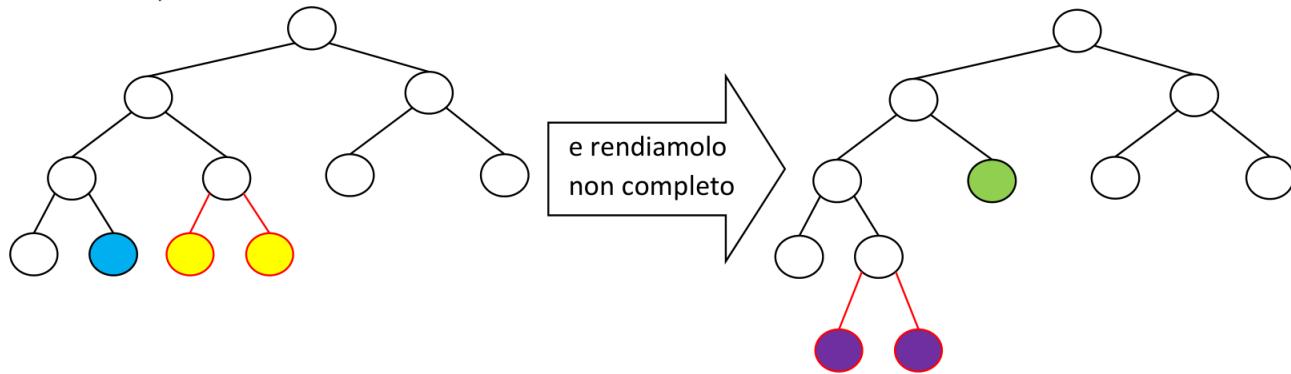
Calcoliamo la *LPE* da sinistra a destra:

$$T_M(3) = \frac{3 + 2 + 3 + 3 + 3 + 3}{3!} = \frac{17}{6}$$



Analogamente al caso peggiore, bisogna studiare il miglior caso medio e cercare tutti gli alberi che minimizzino il percorso esterno (per minimizzare quel rapporto o minimizziamo il numeratore o il denominatore; non possiamo modificare il numero di foglie).

Dimostriamo quindi che gli alberi binari che minimizzano la lunghezza del percorso esterno corrispondono all'albero completo



Andiamo a calcolare entrambi i percorsi esterni mettendoli in relazione.

Sia la lunghezza del percorso completo pari a k e andiamo a calcolare il percorso dell'albero non completo in funzione di k sapendo che h sia l'altezza dell'albero completo:

$$\text{LPE} = k - 2h + h - 1 - h + 2(h + 1) = k + 1$$

Il percorso esterno peggiora all'allontanarsi dalla completezza.

Ora, visto che si tratta di un albero completo sappiamo che le foglie sono ad altezza h o ad altezza $h - 1$. Sia N_h il numero di foglie ad altezza h e N_{h-1} il numero di foglie ad altezza $h - 1$. È evidente che il numero totale di foglie sarà $N_h + N_{h-1} = n!$, mentre

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

Sappiamo anche che per un albero binario pieno il numero di foglie sono 2^h , che in relazione alle foglie del nostro albero risulta $2N_{h-1} + N_h = 2^h$.²⁴ A questo punto avremo:

²⁴Ogni foglia ad altezza $h - 1$ avrà 2 figli.

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

Calcoliamo quindi il tempo medio:

$$\begin{aligned} T_M(n) &= \frac{LPE}{n!} = \frac{frach \cdot N_h + (h-1)N_{h-1}n!}{n!} = \frac{\frac{2h}{n}n - h2^h + h2^h - hn! - 2^h + n!}{n!} = \\ &= \frac{hn! - 2^h + n!}{n!} = h - \frac{2^h}{n!} + 1 \stackrel{h \log!}{\implies} T_M(n) = \log n! - \frac{2^{\log n!}}{n!} + 1 = \log n! \end{aligned}$$

Abbiamo dimostrato che $\log n! = \Theta(n \log n)$ e quindi abbiamo concluso che il tempo di esecuzione di un algoritmo d'ordinamento per una sequenza arbitraria di input n non possa essere, nel caso medio e peggiore, migliore di $\Theta(n \log n)$.

15 Lezione 18

15.1 Strutture dati elementari

Struttura dati concreta e struttura dati astratta

Una struttura dati concreta è un oggetto concreto nella quale codifico una struttura dati astratta, quest'ultima per un tipo di rappresentazione dei dati.

Una struttura dati astratta potrebbe essere una sequenza di numeri, ed una sua possibile struttura concreta potrebbe essere un vettore (la codifica più naturale) o una lista.

Già con selection sort abbiamo visto che rappresentare i dati in maniera opportuna può avere un enorme impatto sulla complessità dell'algoritmo (heap sort).

Come rappresentare un insieme di dati dinamico S^{25}

Consideriamo un elemento un oggetto atomico → Di conseguenza la sua struttura sarà irrilevante (il concetto di struttura dati è sempre lo stesso indipendentemente da come è fatto il dato); Si noti però che operazioni di confronto (o qualsiasi altra operazione che è dipendente dalla rappresentazione del dato) avrà una complessità diversa in base a come è strutturato il dato (confrontare un array è diverso da confrontare un intero).

15.2 Operazioni su una struttura dati

Consideriamo insiemi con una relazione d'ordine, e assumiamo che l'elemento **NIL** non sia mai appartenente all'insieme (S, \leq).

Per tale insieme possiamo definire le seguenti operazioni:

²⁵Per insieme dinamico si intende una collezione di elementi variabile nel tempo (è possibile aggiungere/rimuovere elementi).

- **Ricerca(S, k)** → Restituisce un elemento di S oppure **NIL** se $k \notin S$;
- **Inserimento(S, k)** → Restituisce un nuovo insieme $S^i = S \cup \{k\}$;
- **Cancellazione(S, k)** → Restituisce un nuovo insieme $S^i = S \setminus \{k\}$

Si noti che la presenza o meno dell'elemento k per le operazioni di inserimento e cancellazione (che sono simili tra loro) non cambia le loro definizioni; infatti se $k \in S$ allora l'inserimento restituirà semplicemente un insieme $S' = S$, mentre se $k \notin S$ allora sarà la cancellazione a restituire un insieme $S' = S$.

Gli algoritmi per le operazioni sugli insiemi sfruttano le caratteristiche della rappresentazione dell'insieme e questo significa che le operazioni di modifica (inserimento e cancellazione) dovranno mantenere intatte quelle caratteristiche (ad es. se devo aggiungere un elemento in una sequenza ordinata, devo aggiungerlo nella giusta posizione in modo da lasciare ordinata la sequenza).

Chiaramente la ricerca è un'operazione che non modifica la struttura dati e quindi preserva naturalmente le proprietà della struttura, mentre per le altre 2, più vincoli ho e più complesso sarà definire le operazioni.

L'operazione di ricerca non si limita solo alla ricerca dell'elemento k nell'insieme S , posso infatti ampliare tale operazione con le seguenti (e altre) operazioni di ricerca:

- **Successore(S, k)** → Restituisce l'elemento con la più piccola chiave $a > k$;
- **Predecessore(S, k)** → Restituisce l'elemento con la più grande chiave $a < k$;
 - ◊ Ad es. per $S = \{3, 6, 8\}$ avremo $\text{Successore}(S, 6) = 8$ e $\text{Predecessore}(S, 5) = 3$ e $\text{Successore}(S, 8) = \text{NIL}$ (non c'è nessun vincolo che imponga $k \in S$).

Altre operazioni di ricerca possono essere la ricerca del minimo e la ricerca del massimo.

15.3 Array (non) ordinato

Abbiamo implicitamente utilizzato gli array negli algoritmi di ordinamento. Adesso vediamo come si comporta un insieme S rappresentato con array.

Se volessimo fare **RicercaMax(S)** un algoritmo del genere richiederebbe tempo $\Theta(n)$, ma se lo rappresentassimo attraverso un **array ordinato** (un array tale che $\forall 1 \leq i \leq n, S[i] \leq S[i + 1]$) allora richiederebbe tempo $\Theta(1)$ (l'intero algoritmo di ricerca si ridurrebbe a restituire $S[n]$).

Un algoritmo definito per una classe è corretto anche nelle sue sottoclassi, il viceversa non sussiste.

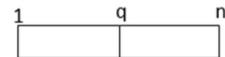
Se invece siamo nell'operazione di inserimento (o cancellazione), per un array non ordinato è un $\Theta(1)$ (basta aggiungere alla fine);

Scegliere i vincoli della struttura dati ha un impatto rilevante sulla complessità; Se ad es. ci troviamo di fronte ad insieme **poco dinamici** conviene usare un array ordinato; se invece abbiamo un insieme in cui bisogna fare poche ricerche e molti **inserimenti/cancellazioni** è chiaro che sia conveniente utilizzare un array non ordinato.

15.4 Ricerca binaria

Se un array non è ordinato per fare la ricerca di un elemento bisogna per forza scorrere al più tutti gli elementi; Se però l'array è ordinato la soluzione al problema cambia.

Ricerca(S, k) → Vado a dividere la sequenza in 2, quindi $q = \lfloor \frac{n}{2} \rfloor$



- ▶ Se $S[q] = k$, allora ho trovato l'elemento e restituisco l'indice;
- ▶ Se $S[q] < k$, allora significa che nel caso k fosse presente all'interno della struttura si troverebbe certamente a destra della sequenza, visto che per ogni elemento x della sequenza a sinistra vale la seguente proprietà $\rightarrow x \leq S[q] < k$. Il nuovo q sarà $\lfloor \frac{q+1+n}{2} \rfloor$ avendo già escluso q ;
- ▶ Se $S[q] > k$, allora cercherò nella sottosequenza di sinistra.

Mi fermerò in 2 casi:

- ▶ Ho trovato l'elemento;
- ▶ Mi trovo in una sequenza di un elemento (in questo caso se la cella non contiene l'elemento allora $k \not\in S$).

Ne consegue che la ricerca di un elemento in una sequenza si riduce alla ricerca dello stesso in una sottosequenza di grandezza dimezzata; più precisamente, dopo la prima operazione sarò in una sequenza di dimensione $\frac{n}{2}$, dopo la seconda in una dimensione $\frac{n}{2^2}$, dopo la terza in una di $\frac{n}{2^3}$, e così via.

Dopo i operazioni avrò una sequenza di grandezza $\frac{n}{2^i}$, raggiungerò quindi la soluzione quando dopo i operazioni avrò una sequenza di un elemento, quindi $\frac{n}{2^i} = 1 \implies i = \log n$

Avere una sequenza ordinata permette, come appena visto, di ottenere un algoritmo di ricerca in tempo logaritmico; più precisamente Ricerca(S, k) sarà un $O(\log n)$, mentre per una sequenza non ordinata $O(n)$.

Andiamo ad implementare l'algoritmo nel seguente modo:

- ▶ RicercaBinaria(S, p, r, k) che restituisce l'indice dove dovrebbe trovarsi k e implementerà il ragionamento visto precedentemente (p e r sono gli indici di inizio e fine sequenza);
- ▶ Ricerca(S, k), che restituisce l'indice giusto se $k \in S$ altrimenti un valore non valido.

```

1 int RicercaBinaria(S, p, r, k)
2   IF p < r THEN
3     q = (p + r) / 2
4     IF k = S[q] THEN
5       ret = q
6     ELSE IF k < S[q] THEN

```

```

7     ret = RicercaBinaria(S, p, q + 1, k)
8 ELSE IF
9     ret = RicercaBinaria(S, q + 1, r, k)
10 ELSE
11     ret = p
12 RETURN ret

```

1. Restituiamo p poiché r non è detto che appartenga all'array essendo che $q + 1$ potrebbe essere un indice al di fuori delle sequenze infatti $p \leq q < r \implies q + 1 \leq r$ (dopo i operazioni potrebbe esserci la chiamata `RicercaBinaria(S, n, n + 1, k)` che restituirebbe $n + 1$); quindi con p sono sicuri di non andare mai in segmentation fault.

```

1 int Ricerca(S, k)
2     i = RicercaBinaria(S, 1, n, k)
3 IF S[i] = k THEN
4     ret = i
5 ELSE
6     ret = -1
7 RETURN ret

```

Tutto quello di cui abbiamo discusso finora ci dice che il problema della ricerca è risolvibile in un tempo **esponenzialmente** minore di lineare (ovvero logaritmico). Ma array ordinati hanno anche argomenti a sfavore poiché operazioni di inserimenti e/o cancellazione sono lineari, anche se in realtà ad essere problematico non è il vincolo dell'ordinamento ma il fatto di avere una struttura dati sequenziale → Vedremo in seguito delle specifiche strutture dati di insiemi ordinato con operazioni di modifica molto meno dispersive.

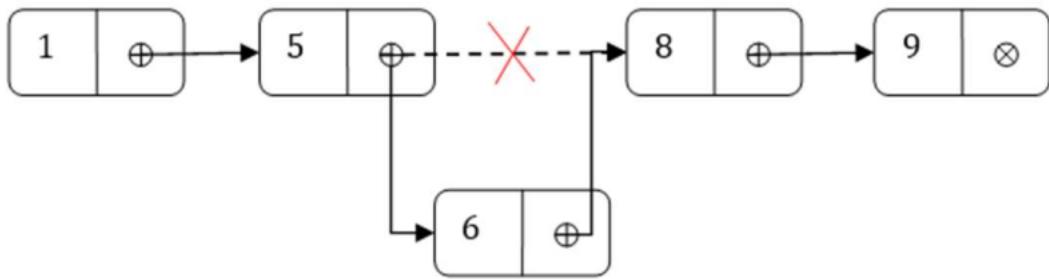
15.5 Liste

La lista rappresenta una struttura dati dinamica dove, a differenza dell'array (implementata come struttura statica, ed è quello che rende problematico le operazioni di inserimento e cancellazione), le operazioni di inserimento e cancellazione sono meno dispendiose.

La lista condivide con l'array la proprietà di linearità (o sequenzialità) ma è una struttura più flessibile poiché non richiede la contiguità in memoria come l'array.

La lista permette di avere elementi in una qualsiasi area di memoria rendendo le operazioni di inserimento e cancellazioni costanti
→ La sua struttura è composta da nodi, i quali hanno in sé un dato e una informazione su dove si trovi il nodo successivo.

Supponiamo di voler inserire l'elemento 6 nella seguente lista ordinata:



È evidente che questa operazione è costante poiché impiegherà un tempo pari alla somma del tempo delle operazioni elementari per creare un nuovo nodo e modificare i puntatori.

Il problema delle liste è la ricerca di un elemento → Per la sua struttura non è possibile direttamente accedere ad un qualsiasi elemento come con gli array ma devo obbligatoriamente partire dall'unico nodo di cui conosco l'indirizzo (tipicamente il primo) e scorrere la lista fino all'elemento desiderato.

Questo implica che accedere ad un elemento in mezzo alla lista richiede tempo lineare (dovrò fare $\frac{n}{2}$ letture in memoria).

Anche supponendo di avere una lista doppiamente puntata (quindi ogni nodo ha un riferimento al nodo precedente oltre che al successivo) e implementare la ricerca binaria, per raggiungere l'elemento della sottosequenza di grandezza $\frac{n}{2^i}$ devo partire dall'indice mediano della sequenza precedente: il costo complessivo quindi sarà:

$$\sum_{i=1}^{\log n} \left(\frac{n}{2^i}\right) = n \underbrace{\sum_{i=1}^{\log n} \left(\frac{1}{2}\right)^i}_{\text{tende a costante}} = \Theta(n)$$

15.6 Definizione formale di lista e algoritmo di ricerca

Una lista L è un oggetto con le seguenti proprietà:

1. \emptyset è un insieme vuoto di nodi → $L = \emptyset$;
2. Oppure contiene un nodo con un dato e un riferimento ad un oggetto L' dove L' è una lista (non c'è ambiguità poiché $|L'| < |L|$ per definizione).

Da questa definizione induttiva risulta naturale scrivere algoritmi ricorsivi per implementare le operazioni sulla lista; infatti, per l'operazione di ricerca possiamo implementare il seguente algoritmo:

```

1 int Ricerca(L, k)
2   IF L != NIL THEN
3     IF L -> key = k THEN
4       ret = L // Indirizzo del nodo che contiene il dato

```

```

5     ELSE
6         ret = L // Arrivati a questo punto significa che L = NIL
7     RETURN ret

```

16 Lezione 19

16.1 Operazioni su liste

Una **lista puntata** è un insieme dinamico in cui ogni elemento ha una chiave (*key*) ed un riferimento all'elemento successivo (*next*) dell'insieme.

Inoltre la lista è una struttura dati ad accesso strettamente sequenziale → Da ciò ne deriva un tempo lineare per tutti gli algoritmi che necessitano di una ricerca.

Definiamo ora altri tipi di liste:

- ▶ **Lista doppiamente puntata** → Insieme dinamico in cui ogni elemento ha
 - ◊ Una chiave;
 - ◊ Un riferimento (*next*) all'elemento successivo dell'insieme;
 - ◊ Un riferimento (*prev*) all'elemento precedente dell'insieme;
- ▶ **Lista circolare puntata** → insieme dinamico in cui ogni elemento ha
 - ◊ Una chiave (*key*) e un riferimento (*next*) all'elemento successivo dell'insieme;
 - ◊ L'ultimo elemento ha un riferimento alla testa della lista;
- ▶ **Lista circolare doppiamente puntata** → insieme dinamico che presenta
 - ◊ Una chiave (*key*), un riferimento (*next*) all'elemento successivo dell'insieme e un riferimento (*prev*) all'elemento precedente dell'insieme;
 - ◊ L'ultimo elemento ha un riferimento (*next*) alla testa della lista, il primo ha un riferimento (*prev*) alla coda della lista;

Poiché ogni elemento della lista è indipendente dagli altri e non sono disposti in maniera contigua in memoria, l'inserimento di un nuovo nodo in testa risulta banale:

```

1  /*
2   L'algoritmo è costante, ma se si volesse inserire il nodo in una determinata posizione (con una
3    conseguente ricerca) in una lista costante
4    l'algoritmo risulta comunque lineare anche se il nodo viene aggiunto e creato in tempo costante
5
6 */
7 Insert(L, k)
8     tmp = crea_nodo()
9     tmp -> key = k

```

```

9     tmp -> next = L
10    L = tmp
11    RETURN L
12
13 OrdInsiert(L, k)
14   IF L != NIL AND L -> key < k THEN
15     L -> next = OrdInsiert(L -> next, k)
16   ELSE // La chiave k è la più piccola in L
17     tmp = crea_nodo()
18     tmp -> key = k
19     tmp -> next = L
20     L = tmp
21
22 RETURN L

```

Una ricerca risulta necessaria anche nel caso in cui volessi avere una lista senza duplicati:

```

1 InsertUnica(L, k)
2   IF L = NIL THEN
3     tmp = crea_nodo()
4     tmp -> key = k
5     tmp -> next = NIL
6     L = tmp
7   ELSE IF L -> key != k THEN
8     L -> next = InsertUnica(L -> next, k)
9
10 RETURN L

```

Per la cancellazione di un elemento è necessaria la ricerca. Inoltre i casi ricorsivi da gestire sono 2 (si fa sempre affidamento alla definizione della lista):

- ▶ La lista è vuota;
- ▶ La lista contiene un nodo con una chiave ed un riferimento ad un'altra lista.

Da questo ne deriva il seguente algoritmo lineare:

```

1 // Si suppone che la lista non abbia duplicati
2 Cancella(L, k)
3   IF L != NIL THEN
4     IF L -> key = k THEN
5       L = L -> next
6       dealloca(tmp)
7     ELSE // Non abbiamo trovato k in L
8       L -> next = Cancella(L -> next, k)
9
10 RETURN L

```

Di seguito riportiamo altri esempi di algoritmi di cancellazione²⁶

```
1 // Cancella gli elementi pari
2 CancellaPari(L)
3     // Visto che la chiamata ricorsiva è fatta prima dell'eventuale cancellazione dell'elemento, l'
4         // algoritmo praticamente andrà a cancellare gli elementi partendo dall'ultimo
5 IF L != NIL THEN
6     L -> next = CancellaPari(L -> next)
7     IF (L -> key) % 2 = 0 THEN
8         tmp = L
9         L = L -> next
10        dealloca(L)
11
12 RETURN L
```

```
1 // Restituisce il numero degli elementi pari cancellati
2
3 // Visto che l'algoritmo non restituisce la lista ma un intero (che rappresenta il conteggio),
4     // abbiamo bisogno oltre al nodo da cancellare anche il suo precedente, in modo da poter
5         // ricostruire la lista.
6 int CancellaPariConta(L, Prev)
7     counter = 0
8     IF L != NIL THEN
9         counter = CancellaPariConta(L-> next, L)
10        counter = CancellaPariConta(L-> next, L)
11        IF (L -> next) % 2 = 0 THEN
12            Prev -> next = L -> next
13            dealloca(L)
14            counter += 1
15
16 RETURN counter
```

```
1 int CancellaPariContaTesta(L)
2     counter = CancellaPariConta(L -> next, L)
3     IF (L -> key) % 2 = 0 THEN
4         tmp = L
5         L = L -> next
6         dealloca(tmp)
7         counter += 1
8
9 RETURN counter
```

²⁶La funzione $x \% y$ sta per **x modulo y** (restituisce come risultato il resto della divisione euclidea del primo numero per il secondo).

16.2 Alberi Binari

Un albero è un insieme dinamico che può essere:

- ▶ Vuoto;
- ▶ Composto da $k + 1$ insiemi disgiunti di nodi:
 - ◊ Un insieme di cardinalità 1, detto **nodo radice**;
 - ◊ k alberi, ciascuno dei quali è detto **sottoalbero i-esimo** della radice (dove $1 \leq i \leq k$)

Un albero di questo tipo si dice di **grado k**, quando $k = 2$ si tratta di un albero binario.

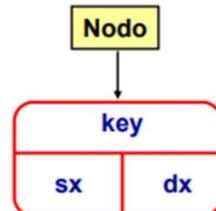
Gli alberi possono essere *visitati* (o *attraversati*) in diversi modi:

- ▶ *Visita in profondità* (verticale) → Si visitano tutti i nodi lungo un percorso, poi quelli lungo un altro e così via;
- ▶ *Visita in ampiezza* (verticale) → Si visita l'albero per livelli, visitando tutti i nodi a livello 0, poi a livello 1, fino ad arrivare al livello h ;

16.3 Visite in profondità

Gli alberi possono essere visitati in profondità in diversi modi:

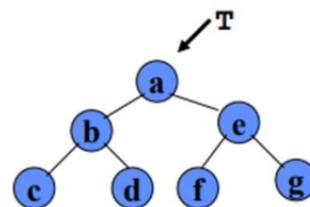
- ▶ *Visita in preordine* → Prima si visita il nodo e poi i suoi sottoalberi (quindi radice, sottoalbero sinistro e poi sottoalbero destro);
- ▶ *Visita in postordine* → Prima si visitano i sottoalberi e poi il nodo (quindi sottoalbero sinistro, sottoalbero destro e infine radice);
- ▶ *Visita in inordine* → Prima si visita il sottoalbero sinistro, poi il nodo e infine il sottoalbero destro;



Implementiamo di seguito gli algoritmi che stampano gli elementi di un albero binario nelle varie visite

```

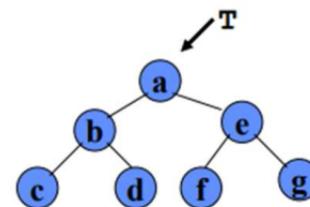
1 VisitaPreOrder(T)
2   IF T != NIL THEN
3     print(T -> key)
4     VisitaPreOrder(T -> sx)
5     VisitaPreOrder(T -> dx)
  
```



Output:
a b c d e f g

```

1 VisitaPostOrder(T)
2   IF T != NIL THEN
3     VisitaPostOrder(T -> sx)
4     VisitaPostOrder(T -> dx)
5     print(T -> key)
  
```

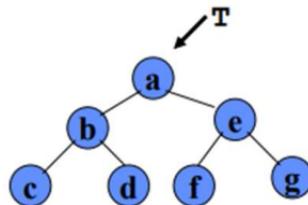


Output:
c d b f g e a

```

1 VisitaInOrder(T)
2   IF T != NIL THEN
3     VisitaInOrder(T -> sx)
4     print(T -> key)
5     VisitaInOrder(T -> dx)

```



Output:
c d b f g e a

I precedenti algoritmi fanno tutti parte della stessa classe di esplorazione di un albero; ovvero la visita in profondità.

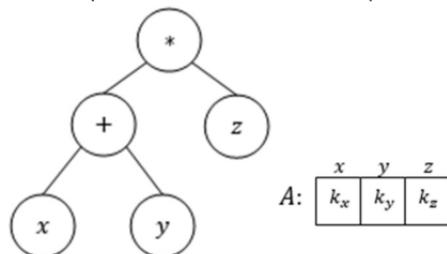
Dal punto di vista della ricerca è irrilevante quale tipo di visita applichiamo (si potrebbe avere una differenza solo dal punto asintotico), ma ci sono alcuni tipi di algoritmi che possono essere visitati solo con un determinato tipo di visita. Un esempio potrebbe essere un algoritmo che risolve espressione aritmetica:

```

1 Eval(T, A)
2   IF T -> key è una variabile THEN
3     RETURN A[T -> key]
4   ELSE // Visita in profondità
5     sx = Eval(T -> sx, A)
6     dx = Eval(T -> dx, A)
7     // Eseguo l'operazione
8     ris = Apply(T -> key, sx, dx)
9     RETURN ris

```

Esempio di traduzione di un'espressione



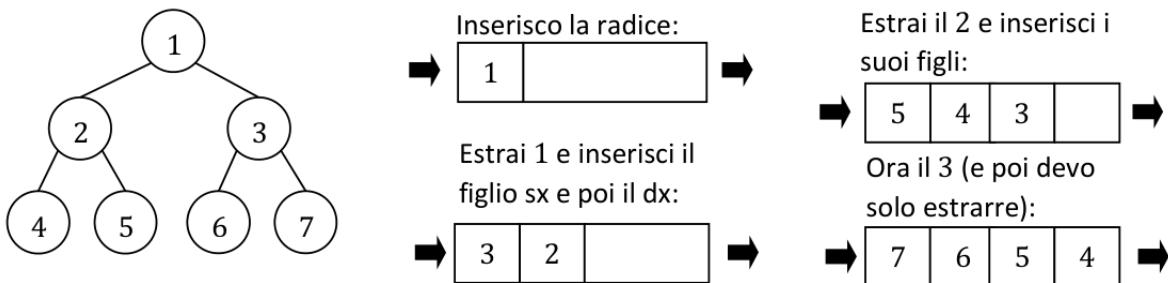
17 Lezione 20

17.1 Visita in ampiezza (BFS)

La difficoltà che si incontra in questo tipo di visita è dovuto dalla mancanza di collegamenti tra i nodi di uno stesso livello (abbiamo collegamenti solo per le discendenze dirette) → Di conseguenza non posso raggiungere tutti i nodi in uno stesso livello affidandomi solo sulle informazioni della struttura.

L'idea è quella di usare una struttura dati dove inserire le informazioni dei nodi da visitare nel giusto ordine. La struttura ideale per questo problema è la **coda** (inserisco gli elementi da un lato e li estraggo dall'altro).

In pratica inserisco in coda tutti i figli del nodo che sto visitando ed il prossimo nodo da visitare risulterà essere proprio il primo inserito nella coda



A questo punto l'implementazione dell'algoritmo conosciuto come BFS (**breadth-first-search**) risulta semplice:

```

1 BFS(T)
2   Q = { T }
3   WHILE Q != {} DO
4     x = Testa(Q)
5     visita(x)
6     Q = Accoda(Q, x -> sx)
7     Q = Accoda(Q, x -> dx)
8     Q = Decoda(Q)

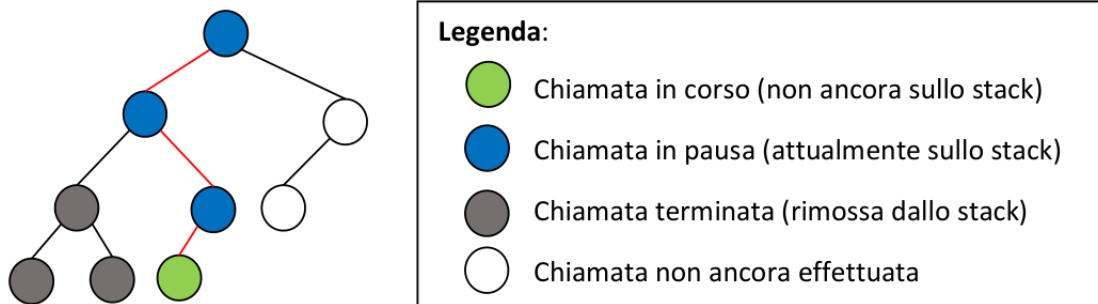
```

Il tempo di esecuzione sarà identico a quello delle visite in ampiezza. Supponendo infatti che l'algoritmo `visita` sia costante il tempo di esecuzione sarà pari al numero di nodi (quindi lineare).

La coda non incrementa il tempo asintotico (supposto che sia implementata come si deve, ad es utilizzando una lista con un puntatore in testa e uno in coda) perché le sue operazioni sono costanti.

17.2 Memoria aggiuntiva

Per una visita in profondità si richiede una quantità di memoria aggiuntiva pari alla lunghezza del percorso più lungo → L'algoritmo quando visita un nodo fa una chiamata ricorsiva al figlio (è proprio la chiamata ricorsiva che richiede memoria aggiuntiva per essere gestita); viene quindi salvato un record di attivazione sullo stack. La massima quantità di record è proprio la lunghezza del percorso più lungo poiché quando una chiamata termina essa viene rimossa dallo stack, ad es:



Di conseguenza la quantità di memoria è lineare e dipende dall'altezza dell'albero (se l'albero è costruito bene sarà logaritmica). In particolare la quantità di memoria aggiuntiva $M(n)$ per una visita in profondità sarà:

$$\underbrace{\log n}_{\substack{\text{albero} \\ \text{completo}}} \leq M(n) \leq \underbrace{n}_{\substack{\text{albero} \\ \text{degenero}}}$$

Questa analisi ci fa comprendere di come non abbia senso modificare la struttura dell'albero aggiungendo altri puntatori (come quello da figlio a padre) → Richiederebbe non solo uno spreco di memoria rispetto alla ricorsione, ma renderebbe anche più complessi da implementare gli algoritmi stessi

Anche per la visita in ampiezza la memoria richiesta è dipendente dalla forma dell'albero; in questo caso però visito un livello quando termina il precedente, e visto che ogni nodo accoda i suoi figli avremo nella coda tutti i nodi del livello $i + 1$ quando mi trovo all'ultimo nodo del livello i .

La massima memoria aggiuntiva possibile è all'ultimo livello di un albero pieno (tanta memoria quante sono le foglie). Ho pertanto uno spreco minore su un albero degenere (caso opposto alla visita in profondità):

$$\underbrace{1}_{\substack{\text{albero} \\ \text{degenero}}} \leq M(n) \leq \underbrace{\left\lceil \frac{n}{2} \right\rceil}_{\substack{\text{albero} \\ \text{completo}}}$$

Anche se il caso peggiore è lineare per entrambe le visite è evidente che la quantità di memoria necessaria per una visita in ampiezza è minore di quella necessaria per una visita in profondità.

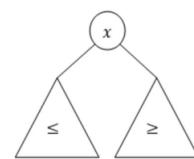
17.3 Alberi binari di ricerca (ABR)

17.3.1 Definizione di ABR

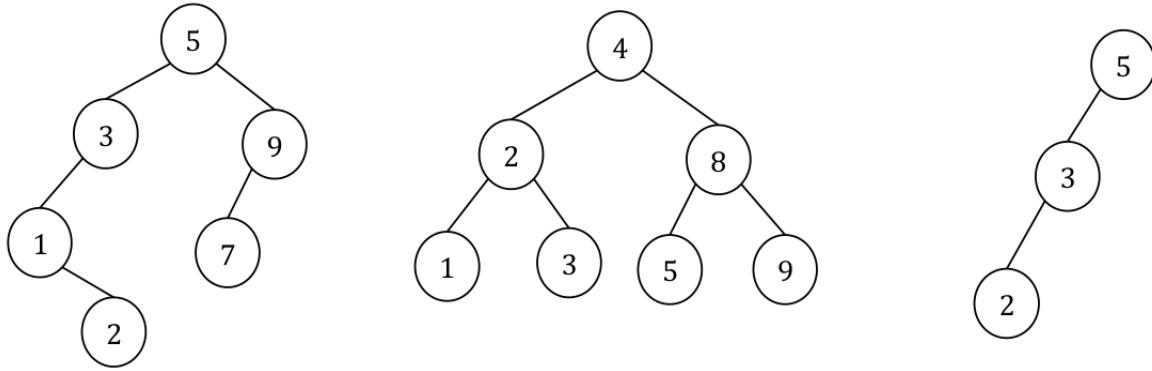
Gli alberi **binari di ricerca** (detti anche **alberi binari ordinati**), hanno un opportuno vincolo di ordinamento che darà dei progressi sul punto di vista asintotico.

Un albero binario di ricerca T :

- ▶ È un albero binario
 - ◊ Vuoto o composto da 3 insiemi disgiunti di nodi (un insieme di cardinalità 1, detto nodo radice e 2 sottoalberi, detti rispettivamente sottoalbero sinistro e destro);
- ▶ $\forall x \in T$
 - ◊ $\forall y \in x \rightarrow sx val(y) \leq val(x)$
 - ◊ $\forall y \in x \rightarrow dx val(x) \leq val(y)$



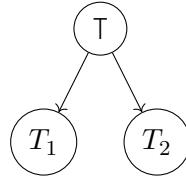
Si noti che a differenza degli alberi heap, negli alberi binari di ricerca si ha un ordinamento **totale** tra gli elementi. Di seguito riportiamo alcuni esempi di ABR:



17.3.2 Definizione ricorsiva

T è ABR se:

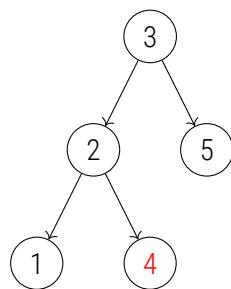
1. $T = \emptyset$
2. Dato T (rappresentato a lato)
 - T_1 è ABR e $\forall y \in T_1 \text{ } val(y) \leq val(r)$
 - T_2 è ABR e $\forall y \in T_2 \text{ } val(r) \leq val(y)$



NB Non bisogna confondere la definizione di ABR con quella di albero heap → Anche se un ABR rispetta sempre le regole di albero heap, il viceversa non è valido.

Il seguente albero, ad es, è uno heap ma non un ABR

È importante fare attenzione alla precisione della definizione → Se volessimo scrivere un algoritmo che verifichi se un determinato albero sia binario non basta verificare che ogni nodi abbia un figlio sinistro con dato minore ad esso e un figlio destro con dato maggiore.



17.3.3 Operazione di ricerca

Supponiamo di avere un ABR e di volere implementare un algoritmo che verifichi se un elemento $k \in T$.

L'idea è abbastanza semplice → Avendo accesso al nodo radice x basta confrontarlo con k :

- $x = k \rightarrow$ Elemento trovato;

- $k < x \rightarrow$ Il dato si trova nel sottoalbero sinistro;
- $k > x \rightarrow$ Il dato si trova nel sottoalbero destro;

Questo ragionamento è analogo a quello applicato alla ricerca binaria in un vettore. È chiaro che un ABR ci permetta di definire in modo naturale (non è più l'algoritmo a decidere la partizione, ma la struttura stessa dell'albero) l'algoritmo di ricerca:

```

1 Search(T, k)
2   IF T != NIL THEN
3     IF k < T->key THEN
4       RETURN Search(T->sx, k)
5     IF k > T->key THEN
6       RETURN Search(T->dx, k)
7   RETURN T

```

Si noti come il numero massimo di confronti nel caso peggiore sarà pari alla lunghezza del percorso più lungo → Il tempo di esecuzione è pari all'altezza dell'albero. Ne consegue che:

$$\underbrace{\log n}_{\substack{\text{albero} \\ \text{completo}}} \leq T(n) \leq \underbrace{n}_{\substack{\text{albero} \\ \text{degenero}}}$$

Pertanto se la forma dell'albero è buona abbiamo un significativo miglioramento rispetto ad un comune albero binario. Si potrebbe dimostrare (cosa che non faremo) anche che il tempo di questo algoritmo (considerando tutte le sue forme) è un $\Theta(\log n)$

17.3.4 Operazioni di modifica

Se ho un ABR e voglio modificarne la struttura devo ovviamente garantire che, dopo eventuali inserimenti o cancellazioni, l'albero risultante mantenga le proprietà di ABR

17.3.5 Inserimento

Supponiamo di volere inserire un elemento in un albero T ; ovvero di volere un albero $T^i = T \cup \{k\}$ (assumiamo che l'albero non abbia chiavi duplicate).

Ci sono solo 2 possibili casi:

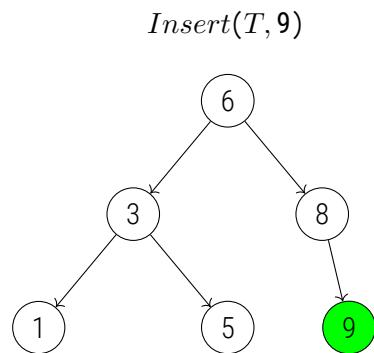
- L'albero iniziale $T = \emptyset \implies T^i = T \cup \{k\}$ (soluzione ideale);
- T non vuoto $\rightarrow k$ va inserito nel giusto sottoalbero vuoto
 - ❖ Infatti se $k /in T$ allora raggiungerò un nodo con un sottoalbero vuoto (nella peggiore delle ipotesi raggiungo una foglia che li ha entrambi vuoti) dove va inserito k .

È importante notare come tale algoritmo è una variazione della **Ricerca** (di conseguenza avrà la sua complessità):

```

1   Insert(T, k)
2     IF T -> NIL THEN
3       x = alloca_nodo()
4       x -> key = k
5       x -> sx = x -> dx = NIL
6       RETURN x
7
8     ELSE IF x < T -> key THEN
9       T -> sx = Insert(T -> sx, k)
10    ELSE IF x > T -> key THEN
11      T -> dx = Insert(T -> dx, k)
12    // Se nessuna condizione è verificata significa che k
13      appartiene a T
14    RETURN T

```

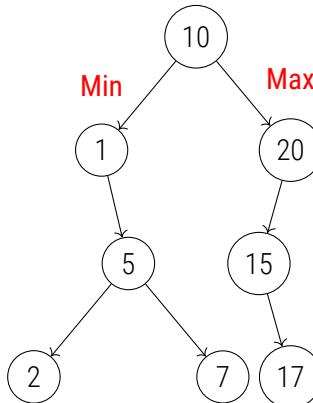


18 Lezione 21

18.1 Ricerca del minimo e del massimo

La ricerca del minimo e del massimo sono banali in un ABR (grazie al vincolo di ordinamento che questi offrono) → Si trovano al nodo più a sinistra e al nodo più a destra rispettivamente

Ne consegue che la ricerca sarà lineare sull'altezza poiché dobbiamo seguire un determinato percorso



Più precisamente, il minimo si trova seguendo sempre il ramo più a sinistra fino a trovare un nodo che non ha figlio sinistro (può avere figlio destro).

Analogamente, il massimo sarà il primo nodo del percorso estremo destro che non ha figlio destro.

Ragionando sul minimo per induzione, posso avere o un albero vuoto (restituirò **NIL**) o un nodo che è:

- ▶ O il minimo (se non ha figlio sinistro);
- ▶ O ci troviamo nel caso che il minimo si trova nel suo sottoalbero sinistro dove ripeterò il precedente ragionamento.

Avrò quindi il seguente algoritmo:

```

1 SearchMin(T)
2     ret = T
3     IF T != NIL THEN
4         x = SearchMin(T -> sx)
5         if x != NIL THEN
6             ret = x
7     RETURN T

```

Sarebbe possibile implementarlo anche in maniera iterativa (evitando così di dovere utilizzare memoria aggiuntiva):

```

1 SearchMinIter(T)
2     node = T // Conviene sempre non modificare l'input per non perdere il riferimento
3     IF T != NIL THEN
4         while node -> sx != NIL DO
5             node = node -> sx
6     RETURN node

```

Analogamente al minimo riportiamo gli algoritmi per la ricerca del massimo

```

1 SearchMax(T)
2     ret = T
3     IF T != NIL THEN
4         x = SearchMax(T -> dx)
5         if x != NIL THEN
6             ret = x
7     RETURN ret

```

```

1 SearchMaxIter(T)
2     node = T
3     IF T != NIL THEN
4         while node -> dx != NIL DO
5             node = node -> dx
6
7     RETURN node

```

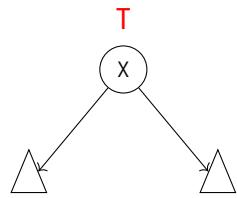
18.2 Ricerca del successore e predecessore

Prima di vedere l'algoritmo ricordiamo le definizioni di successore e predecessore:

- ▶ **Successore**(T, k) → Restituisce l'elemento con la più piccola chiave $a > k$;
- ▶ **Predecessore**(T, k) → Restituisce l'elemento con la più grande chiave $a < k$.

Se l'albero non è vuoto allora ho la seguente struttura con solo 3 possibili casi

- $x < k \rightarrow$ La radice contiene un valore minore della chiave
 - ◊ Se il successore esiste sarà a destra di x (ovvero il risultato ottenuto dalla chiamata $\text{Successore}(T, \rightarrow dx, k)$)
- $x > k \rightarrow$ Il successore sarà il risultato della chiamata $\text{Successore}(T, \rightarrow sx, k)$
 - ◊ Se $T \rightarrow sx = \emptyset$ allora il successore è x (a destra avrà solo valori maggiori di x e di conseguenza lo che il miglior candidato è x stesso)
- $x = k \rightarrow$ È evidente che collassa al caso $x < k$, perché se il successore esiste sarà nel sottoalbero destro

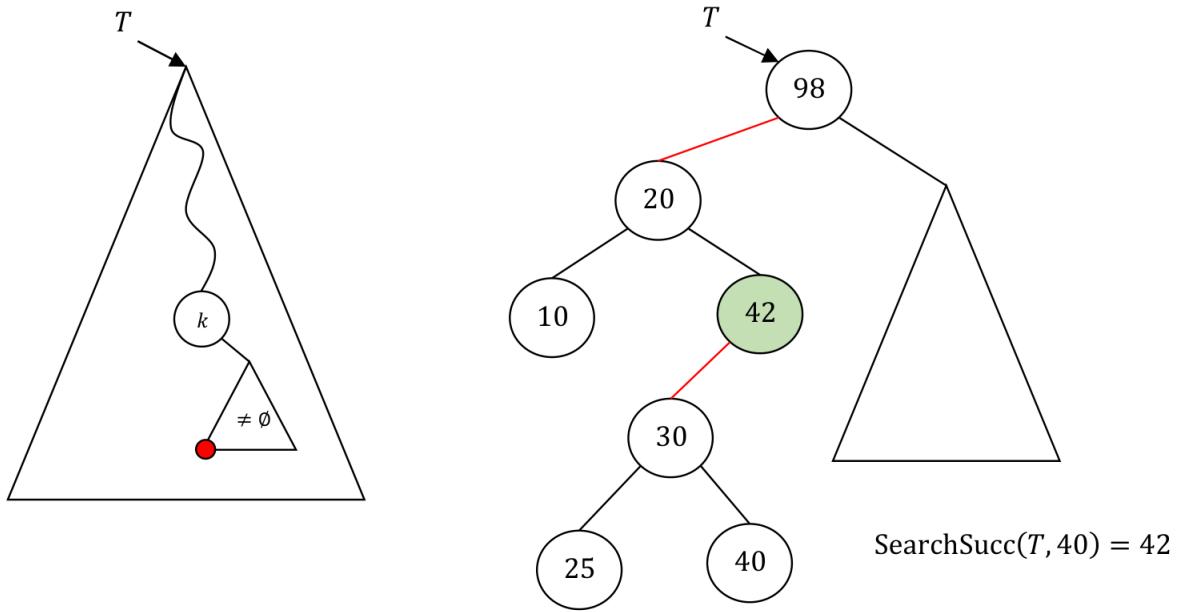


```

1 SearchSucc(T, k)
2   IF T != NIL THEN
3     IF T -> key <= k THEN
4       RETURN SearchSucc(T -> dx, k)
5     ELSE
6       x = SearchSucc(T -> sx, k)
7       IF x != NIL
8         RETURN x
9   RETURN T
  
```

Per la versione iterativa il ragionamento è diverso.

Se k è presente in T allora il successore sarà semplicemente il minimo del sottoalbero destro; tuttavia se tale sottoalbero non esiste significa che ho già passato il successore → esso sarà proprio l'ultimo nodo da cui sono sceso a sinistra lungo il percorso:



Si noti di come la versione ricorsiva svolga in maniera implicita questo ragionamento.

Inoltre, visto che dobbiamo ricordare un antenato è necessario utilizzare una variabile ausiliaria che conservi l'ultimo nodo da cui scendo a sinistra man mano che percorro l'albero

```

1 SearchSuccIter(T, k)
2   c = T // Candidato ad essere successore
3   s = NIL // L'effettivo successore
4   WHILE c != NIL or C -> key != k DO
5     IF c -> key < k THEN
6       c = c -> dx
7     ELSE // Devo scendere a sinistra
8       s = c
9       c = c -> sx
10    IF c = NIL or c -> dx = NIL THEN
11      RETURN s
12    ELSE // T != NIL e ho sottoalbero destro
13      RETURN SearchMinIter(c -> dx)

```

Il ragionamento per trovare il predecessore è duale (riportiamo quindi solo gli algoritmi)

```

1 SearchPred(T, k)
2
3   IF T != NIL THEN
4     IF T -> key >= k THEN
5       RETURN SearchPred(T -> sx, k)
6
7   ELSE
8     x = SearchPred(T -> dx, k)
9
10  IF x != NIL
11    RETURN x
12
13 RETURN T

```

```

1 SearchPredIter(T, k)
2   c = T
3   p = NIL
4   WHILE c != NIL OR c -> key != k DO
5     IF c -> key > k THEN
6       c = c -> sx
7     ELSE
8       p = c
9       c = c -> dx
10    IF c = NIL OR c -> sx != NIL THEN
11      RETURN p
12    ELSE
13      RETURN SearchMaxIter(c -> sx)

```

18.3 Cancellazione

Supponiamo di voler cancellare k da T e di aver già individuato k ; per eliminarlo devo assicurarmi di mantenere intatte le proprietà della struttura.

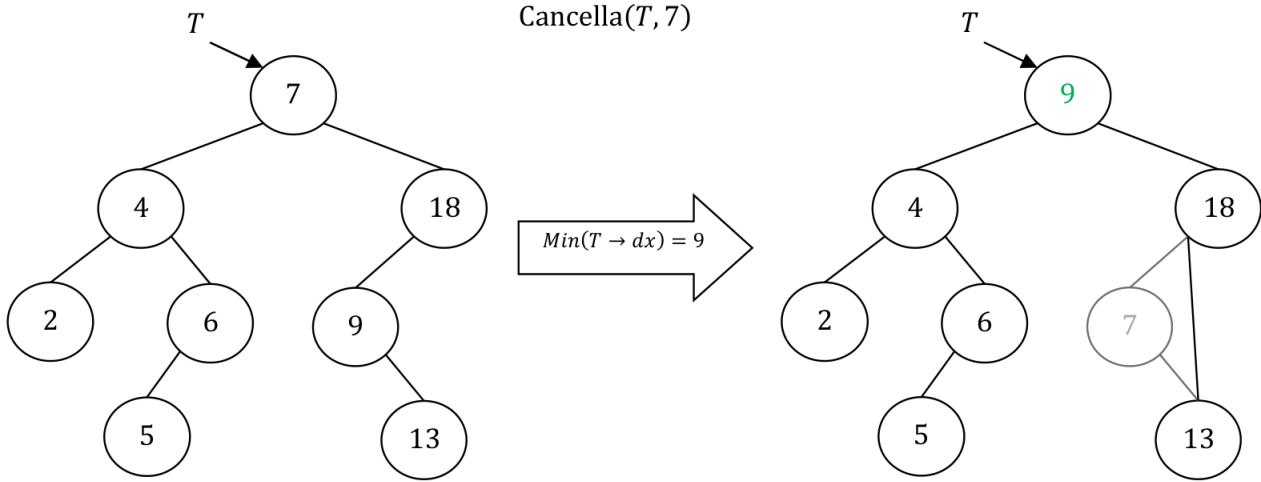
Se k è foglia allora i suoi sottoalberi sono entrambi vuoti (devo preoccuparmi solo di aggiornare il nodo padre settando tale foglia a **NIL**).

Banale è anche il caso in cui il nodo ha un solo figlio → Basta collegare il nodo padre a tale figlio (padre \rightarrow sx (o dx) = $son \rightarrow sx$ (o dx), dipende da quale puntatore è diverso da **NIL**).

Il problema lo si ha quando il nodo contenente k ha entrambi i sottoalberi non vuoti (dovrei collegare i 2 nodi del figlio all'unico riferimento libero del padre).

Il nostro scopo non è necessariamente quello di cancellare l'intero nodo contenente il dato, ma solo di rimuovere quest ultimo.

La soluzione è quella di sostituire il dato con quello di un altro nodo facile da cancellare (devo ovviamente scegliere un nodo che non mi distrugga la proprietà dell'**ABR** → Tale nodo è proprio il minimo del sottoalbero destro di k , o analogamente il massimo del sottoalbero sinistro); questo nodo non solo rende la sostituzione corretta, ma per la definizione di minimo siamo sicuri che questo nodo ha al più solo figlio destro:



Si noti che non basta la ricerca del minimo definita precedentemente poiché, non solo deve restituire il dato del minimo così da poterlo scambiare, ma deve anche modificare la struttura. Bisogna definire anche altri algoritmi per poter implementare il seguente algoritmo:

```

1 Cancella(T, k)
2 // La riga successiva non è più necessaria (andiamo a richiamare il successivo algoritmo
   sicuramente su un input non vuoto), ma la lasciamo per robustezza
3 IF T != NIL THEN
4   IF k < T -> key THEN
5     T -> sx = Cancella(T -> sx, k)
6   ELSE IF k > T -> key THEN
7     T -> dx = Cancella(T -> dx, k)
8   ELSE // T -> key = k
9     T -> dx = CancellaDataRoot(T)
10 RETURN T

```

```

1 CancellaDataRoot(T)
2 IF T != NIL THEN
3   IF T -> sx = NIL OR T -> dx = NIL THEN
4     IF T -> sx != NIL NIL THEN
5       x = T -> sx
6     ELSE
7       x = T -> sd
8       dealloca(T)
9     RETURN x
10 ELSE // Ho 2 figli
11   k = StaccaMin(T -> dx, T)
12   T -> key = k
13 RETURN T

```

```
1 StaccaMin(T, p)
2   IF T != NIL THEN
3     IF T -> sx != NIL THEN
4       RETURN StaccaMin(T -> sx, T)
5     ELSE // T contiene il minimo
6       k = T -> key
7       IF p -> sx = T THEN
8         p -> sx = T -> sx
9       ELSE
```

```
10    ||| p -> sx = T -> dx //
```

```
11    dealloca(T)
12    RETURN k
```

Questo caso non possiamo escluderlo in quanto chiamiamo la funzione passandogli gli input ($T \rightarrow dx, T$); Quindi se T contiene il minimo allora devo fare esattamente quello che abbiamo scritto alla linea 10

Questo algoritmo è nel caso peggiore lineare sull'altezza → Se l'albero è completo allora ottengo un tempo logaritmico come sperato.

Il problema che ci resta da risolvere è forzare un albero ad avere altezza logaritmica mantenendo al minimo il tempo di esecuzione necessario.

19 Lezione 22

19.1 Alberi bilanciati di ricerca

Definiamo dei vincoli aggiuntivi all'ABR in modo da garantire che l'altezza sia limitata superiormente da una funzione logaritmica (gli ABR sono facili da gestire e inserimento e cancellazione facili da implementare, ma hanno prestazioni poco prevedibili e potenzialmente basse).

La famiglia degli alberi che racchiude questa proprietà è quella degli **Alberi binari di ricerca bilanciati**.

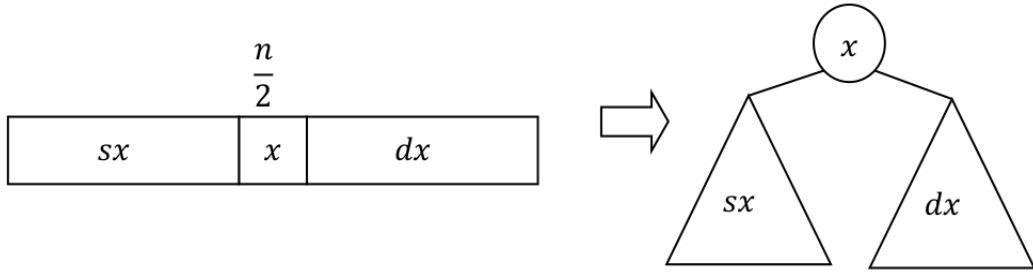
Per definizione, un albero T è bilanciato se $H(t) = O(\log n)$ con $n = |T|$

19.2 Alberi perfettamente bilanciati

Un albero T si dice perfettamente bilanciato se $\forall r \in T, ||x \rightarrow sx| - |x \rightarrow dx|| \leq 1^{27}$

È facile costruire un APB da una sequenza, infatti basta mettere l'elemento di mezzo in radice e suddividere allo stesso modo la sottosequenza di sinistra, che diverrà il sottoalbero sinistro, e la sottosequenza di destra, che sarà il sottoalbero destro:

²⁷ La differenza in valore assoluto tra la cardinalità del sottoalbero sinistro e quello destro è di al più uno.



È abbastanza evidente che questa costruzione rispetta la proprietà $| | x \rightarrow sx | - | x \rightarrow dx | | \leq 1$; inoltre se tale proprietà è rispettata è facile vedere che anche $H(t) = O(\log n)$ sia verificata.

Se ogni sottoalbero ha la metà dei nodi a sinistra e l'altra metà a destra significa che ad ogni livello l'insieme dei nodi radicati in quel livello è la metà di quello radicato al livello superiore.

Ad un livello i si hanno $\frac{n}{2^i}$ nodi, da cui $i = \log n$.

Si noti che $| | x \rightarrow sx | - | x \rightarrow dx | | \leq 1 \implies H(t) = O(\log n)$ (il viceversa non è detto che sia vero), visto che $H(t) = O(\log n)$ rappresenta un insieme con un numero maggiore di alberi.

Gli APB hanno prestazioni ottimali ($\log n$ garantito) ma inserimento e cancellazioni complesse (ribilanciamenti) proprio per la restrittività della proprietà

19.3 Alberi AVL

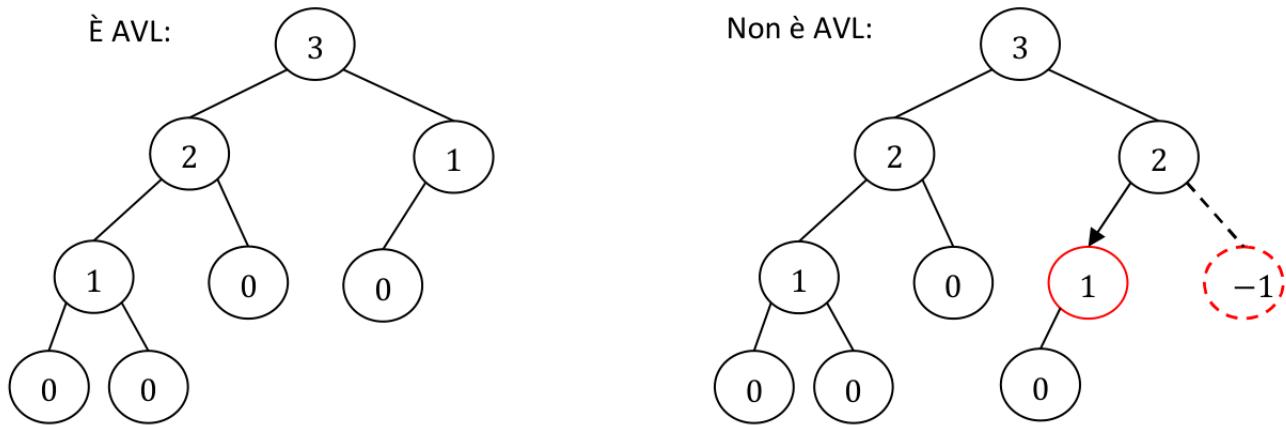
La classe di alberi AVL (Adelson-Velskii e Landis) rappresenta una classe di alberi bilanciati (non ottimale come la precedente) con buone prestazioni e gestione relativamente semplice.

È praticamente la versione più permissiva di APB (rilassa la sua proprietà)

Definizione

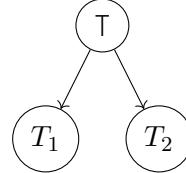
T è AVL se ABR e $\forall x \in T, |H(x \rightarrow sx) - h(x \rightarrow dx)| \leq 1$

Riportiamo 2 esempi scrivendo in ogni nodo l'altezza di quel sottoalbero ($T = \emptyset \implies H(T) = -1$):



Questa proprietà universale (vale per ogni nodo) è facile da descrivere in maniera ricorsiva. T è AVL $\iff T$ è ABR e

1. $T = \emptyset$
2. T (rappresentato a lato) $|H(T_1) - H(T_2)| \leq 1$ con T_1 e T_2 entrambi AVL



La variabilità negli AVL è abbastanza ampia, infatti con lo stesso numero di nodi è possibile generare molti AVL. Ma la cosa interessante è che ogni albero perfettamente bilanciato è AVL poiché è di fatto un albero completo (che rispetta la proprietà di AVL per definizione).

Siamo riusciti ad aumentare la tolleranza (gli AVL sono più permissivi degli APB); resta da dimostrare che anche per gli AVL $H(n) = O(\log n)$.

Nel caso degli AVL però non possiamo immediatamente definire una relazione tra l'altezza e il numero dei nodi (con n nodi possiamo infatti generare diversi AVL con altezza differente).

Per raggiungere il nostro obiettivo dobbiamo restringere la classe AVL in una particolare sottoclasse così da far valere la proprietà per tutti gli AVL.

Generalmente una proprietà valida per una sottoclasse non si estende alla superclasse, ma noi andremo a prendere gli alberi AVL con altezza peggiore possibile²⁸

19.3.1 AVL minimi

Fissato h , l'AVL minimo di altezza h è l'AVL T con $H(T) = h$ che ha il minimo numero di nodi possibile

Formalmente diremo che T è AVL minimo se $\forall T' \in \text{AVL}$ con $H(T') = H(T)$, $|T'| \geq |T|$ (qualsiasi altro AVL con stessa altezza non può avere meno nodi dell'AVL minimo).

²⁸È intuitivo che se per gli alberi peggiori vale $O(\log n)$ allora anche per tutti gli altri la relazione è verificata.

Una minimalità nel numero di nodi si traduce in una massimalità dell'altezza, ovvero se un albero T è un AVL minimo significa che qualsiasi altro albero con lo stesso numero di nodi avrà un altezza minore.

Formalmente

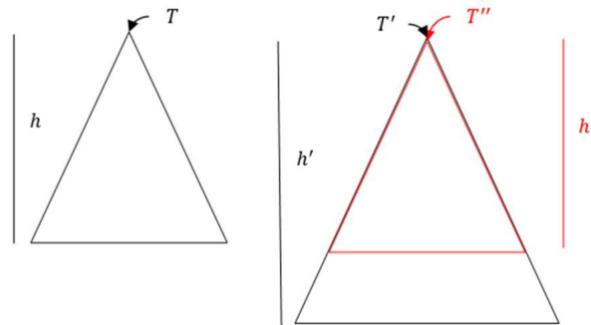
$$\text{se } T \in \text{AVL minimo con } |T| = n, \forall T' \in \text{AVL con } |T'| = n \implies H(T') \leq H(T)$$

Ciò significa che gli AVL minimi sono quelli con altezza peggiore per numero di nodi e quindi una relazione tra altezza e numero di nodi in questa classe farà da limite superiore per tutti gli AVL (che è il nostro obiettivo).

Dimostriamo tale proprietà per assurdo:

Supponiamo di avere un AVL minimo T e un AVL T' con altezza maggiore e stesso numero di nodi.

Se $h' > h$ allora posso togliere tutti i nodi sotto al livello h così da generare un albero T'' con altezza h e numero di nodi strettamente minore di T' (almeno un nodo devo averlo rimosso).

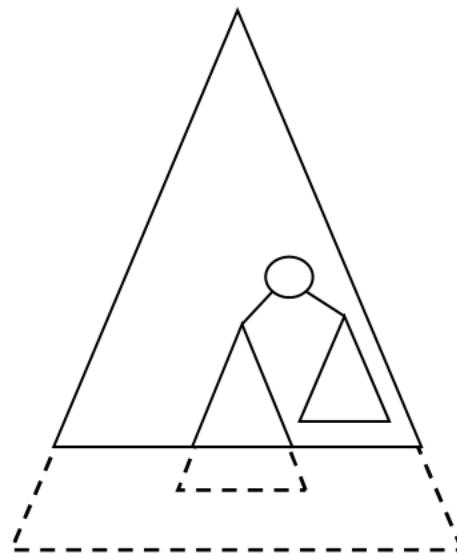


Quindi se dimostro che l'albero T'' così generato che ha un numero di nodi strettamente minore di n è ancora AVL avrò dimostrato che T (AVL minimo) è l'albero AVL con il massimo numero di nodi per altezza h .

Il fatto che T'' è ancora AVL è abbastanza evidente graficamente:

Supponiamo di aver eseguito il taglio descritto in figura. Ora visto che l'albero T' è AVL per definizione (ricordiamo di non aver toccato il sottoalbero destro durante il taglio) e quindi il sottoalbero sinistro è stato ridotto in altezza, è evidente che se già prima del taglio la relazione $sx' - dx' \leq 1$ era vera, nell'albero T'' non può che essere ancora rispettata:

$$\underbrace{sx''}_{\leq sx'} - dx' < dx' - dx' \implies sx'' - dx' < 1$$

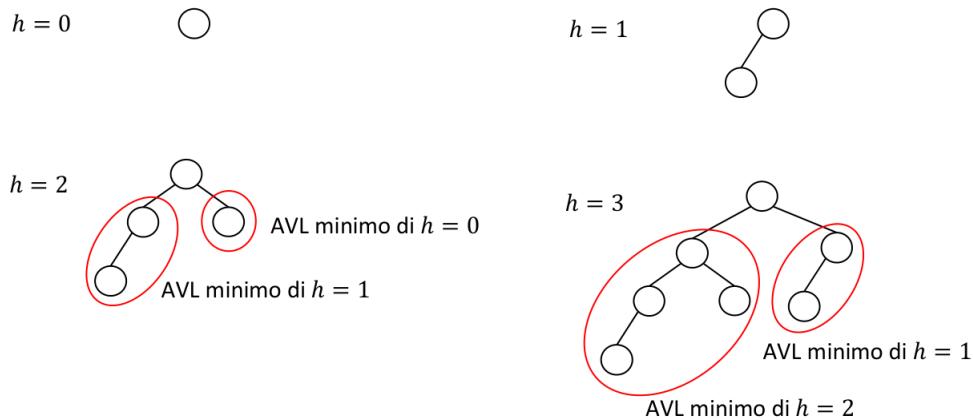


Tutto questo ragionamento ci permette di concentrarci solo sugli alberi AVL minimi, perché una proprietà valida per tale sotto-

classe risulterà vera per una qualsiasi AVL.

19.4 Relazione tra altezza e numero di nodi

Vediamo come sono strutturati gli AVL minimi in base all'altezza (ovviamente non sono gli unici AVL minimi con quella altezza, solo $h = 0$ ha un unico AVL minimo):



C'è quindi una regolarità tra gli AVL minimi, infatti un AVL minimo di altezza h è composto dalla radice e da 2 sottoalberi, in particolare un sottoalbero è un AVL di altezza $h - 1$ e l'altro di altezza $h - 2$.

Se per assurdo così non fosse, significherebbe che potrei togliere dei nodi in un sottoalbero, sia quello di altezza $h - 1$ (abbiamo supposto che tale sottoalbero non sia minimo) e renderlo AVL minimo di altezza $h - 1$, ma ciò significa che anche all'albero di altezza h posso togliere dei nodi senza modificarne l'altezza e ciò va contro alla definizione di AVL minimo.

A questo punto sia $NN(t)$ una funzione che dato l'albero restituisce il numero di nodi, per un qualsiasi albero si ha $NN(T) = 1 + NN(T \rightarrow sx) + NN(T \rightarrow dx)$

Fissato h il numero massimo di nodi di un AVL minimo è funzione di h (cosa che non è vera per un qualsiasi albero, infatti per quest ultimo si può dire che $h \leq NN(h) \leq 2^{h+1} - 1$).

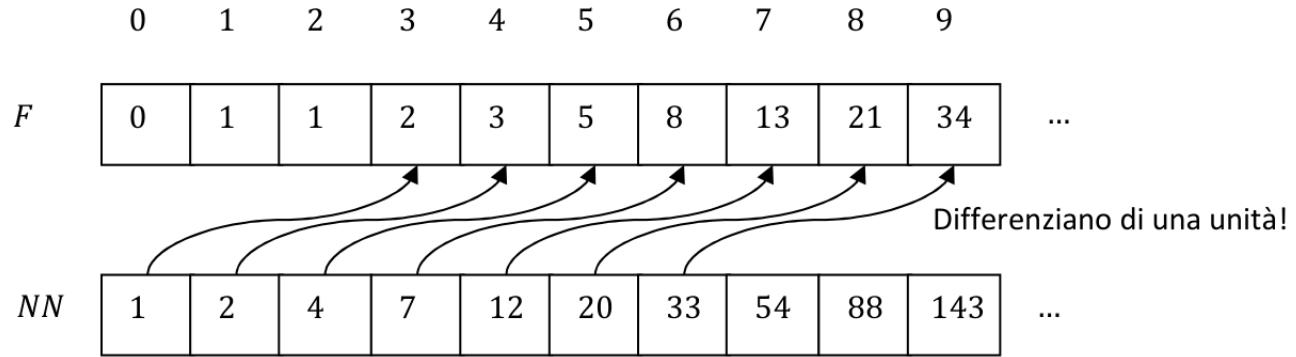
Dunque per un AVL minimo possiamo definire la seguente equazione di ricorrenza.

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

Una equazione molto simile alla sequenza di Fibonacci:

$$F(x) = \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{se } x = 1 \\ F(x - 1) + F(x - 2) & \text{se } x \geq 2 \end{cases}$$

Ma allora se troviamo una relazione tra le 2 equazioni possiamo sfruttare il fatto che la forma chiusa della sequenza di Fibonacci è nota in matematica



Dimostriamo per induzione che la proprietà $NN(h) = F(h + 3) - 1$ dedotta dalla precedente rappresentazione è vera. I casi base sono banalmente verificati:

- $NN(0) = F(3) - 1 \wedge NN(1) = F(4) - 1$.

Per il caso induttivo ($h \geq 2$) supponiamo la relazione vera per $h - 1$ e $h - 2$, ovvero che le relazioni

$$NN(h - 1) = F(h - 1 + 3) - 1 = F(h + 2) - 1 \text{ e } NN(h - 2) = F(h - 2 + 3) - 1 = F(h + 3) - 1$$

sono verificate. Allora:

$$\begin{aligned} NN(h) &= 1 + N(h - 1) + N(h - 2) = 1 + F(h + 2) - 1 + F(h + 1) - 1 = \\ &= \underbrace{F(h + 2) + F(h + 1)}_{\text{è per definizione } F(h + 3)} - 1 = F(h + 3) - 1 \end{aligned}$$

come volevasi dimostrare

La forma chiusa di Fibonacci è la seguente:

$$\begin{aligned}
 F(h) &= \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^2 - \underbrace{\left(\frac{1-\sqrt{5}}{2} \right)^2}_{\substack{\text{Compreso tra } -1 \leq 0 \\ \text{quindi per } h \rightarrow \infty, x^h \rightarrow \infty}} \right] \xrightarrow{\text{per } n \text{ sufficientemente grande}} F(h) \cong \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h \\
 \implies \sqrt{5} F(h) &= \left(\frac{1+\sqrt{5}}{2} \right)^h \implies \log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5} F(h))
 \end{aligned}$$

Ma allora anche $NN(h)$ sarà una equazione esponenziale rispetto h e di conseguenza h sarà logaritmica rispetto il numero di nodi $n = NN(h)$:

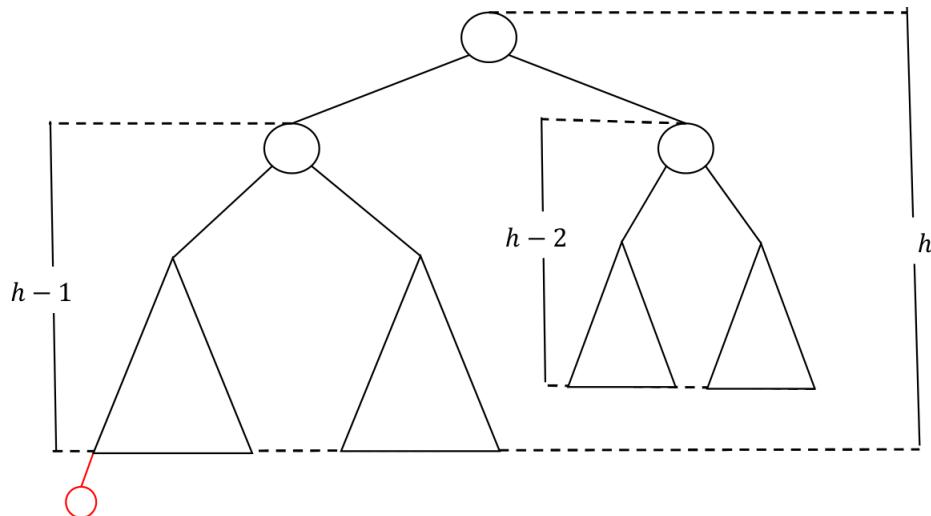
$$\begin{aligned}
 n &= \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \implies \sqrt{5}(n+1) = \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} \implies h+3 = \log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5}(n+1)) \\
 \implies \log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5}(n+1)) - 3 &= \frac{\log_2 (\sqrt{5}(n+1))}{\log_2 \left(\frac{1+\sqrt{5}}{2} \right)} - 3 = \Theta(\log n)
 \end{aligned}$$

Abbiamo così trovato la relazione tra altezza e numero di nodi, ovvero che $h = \Theta(\log n)$ nel caso degli AVL minimi; quindi, per qualsiasi AVL si ha che $h = O(\log n)$ come sperato.

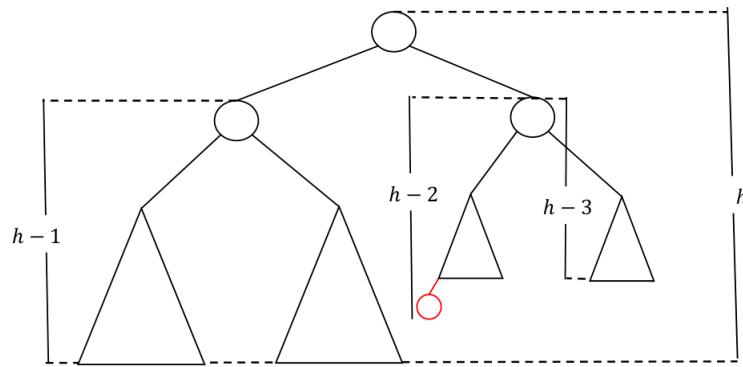
20 Lezione 23

20.1 Operazioni di modifica

Prendiamo un generico AVL, che sappiamo essere anche binario di ricerca, oltre che completo. Dunque, esiste un unico punto dove è possibile inserire un nuovo dato senza distruggere la struttura.



Supponiamo che il punto in cui sia inserito il dato sia nel sottoalbero estremo sinistro come in figura, ma allora dopo l'inserimento l'altezza dell'albero diventerà $h + 1$ con la conseguenza di aver distrutto la proprietà di AVL in quanto ora la radice avrà sottoalbero sinistro di altezza h e quello destro ancora di $h - 2$. Ragionamento analogo si può fare con la cancellazione; supponendo infatti di rimuovere il nodo evidenziato nella seguente situazione andrei a generare una violazione alla proprietà di AVL:



Dopo la cancellazione del sottoalbero sarà $h - 3$ e quindi avremo una differenza > 1 con il sottoalbero destro del figlio sinistro.

Di conseguenza è necessario un modo che mantenga la proprietà della struttura → Risulta complesso dal punto di vista computazionale risistemare la struttura se i punti di violazione sono molteplici.

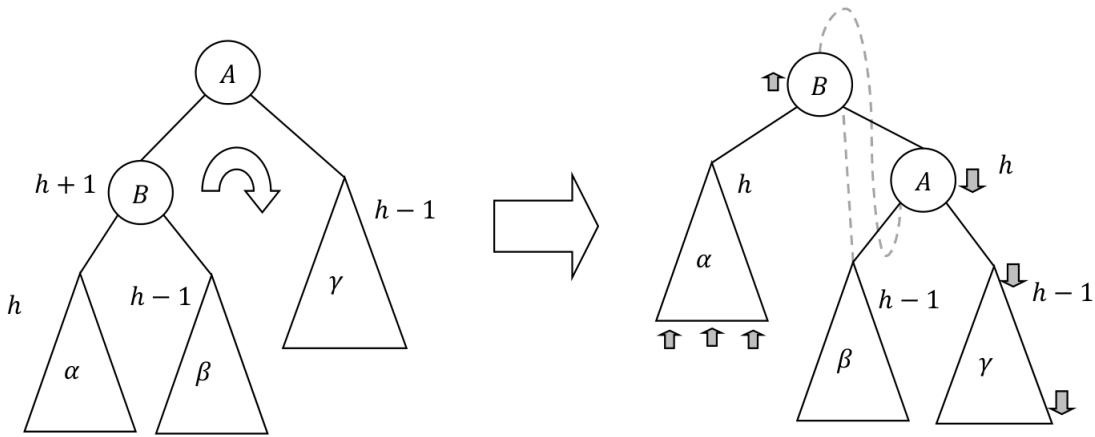
Possiamo garantire che ci sia un unico punto di violazione semplicemente aggiungendo a seguito di ogni singola operazione di modifica (o di cancellazione o inserimento), un controllo che nel momento in cui trovo una violazione la va a sistemare. Suddivideremo dunque l'operazione in:

$$AVL \xrightarrow{\text{modifica}} \text{non AVL} \xrightarrow{\text{bilancia}} AVL$$

Supponiamo di essere di fronte ad un inserimento nel sottoalbero α che viola la proprietà di AVL; per ribilanciare il peso dobbiamo accorciare di uno l'altezza del sottoalbero radicato in B aumentando quello del sottoalbero γ (non basta semplicemente cancellare dei nodi per ridurre il peso → è ovvio che il peso tolto da un sottoalbero deve essere posto in un altro sottoalbero).

Le proprietà di bilanciamento devono preservare anche delle proprietà di ordinamento e non solo quella delle altezze.

L'operazione che risolve il nostro problema è detta **rotazione singola sinistra** (chiamata così dalla rotazione che compie l'albero partendo dal figlio sinistro):



In questo modo viene rispettata la proprietà di AVL → Infatti α e γ non vengono toccati (la proprietà di ordinamento viene mantenuta); mentre β prima della rotazione era a destra di B e dopo viene spostato a sinistra di A²⁹

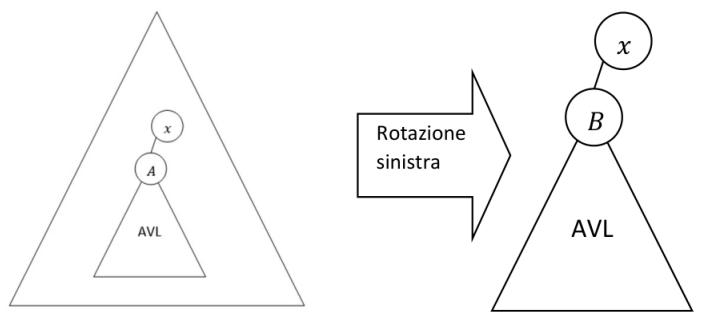
Per quanto riguarda le altezze, prima della rotazione era presente una violazione tra il sottoalbero sinistro di A (sottoalbero di altezza $h+1$ a causa dell'inserimento) e il suo sottoalbero destro (γ di altezza h); in seguito alla rotazione il risultato è di aver alzato il sottoalbero α e di aver abbassato γ , pertanto il sottoalbero sinistro della radice (che dopo la rotazione sarà B) avrà ora altezza h , così come il suo sottoalbero destro.

Interessante è che questa operazione non risolva soltanto il problema localmente → Supponendo di aver fatto la precedente operazione in un sottoalbero, l'operazione di rotazione risolve il problema della violazione delle proprietà AVL in maniera totale:

La proprietà di ordinamento viene ovviamente rispettata.

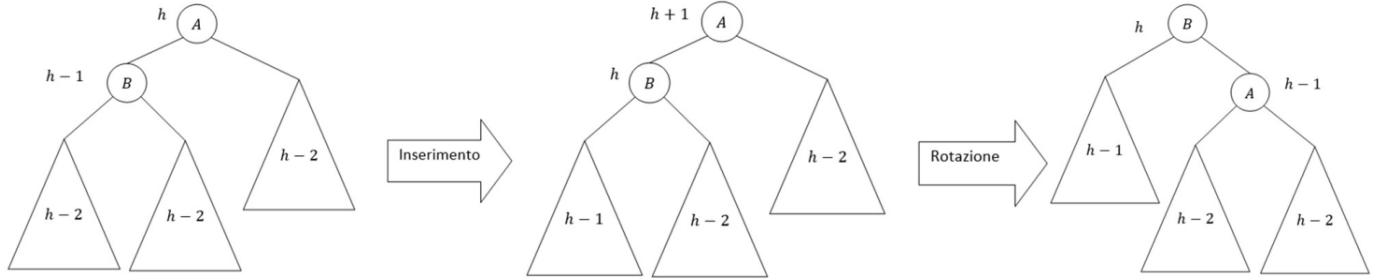
Prima della rotazione avevamo A a sinistra di x, quindi gli elementi di quell'albero erano tutti minori.

Pertanto anche dopo la rotazione tale proprietà è rispettata essendo $B < A > x$



²⁹ Sappiamo dalle proprietà di albero binario di ricerca che prima della rotazione $\forall i \in \beta, i \geq B$, ma essendo B a sinistra di A abbiamo $i \leq A$.

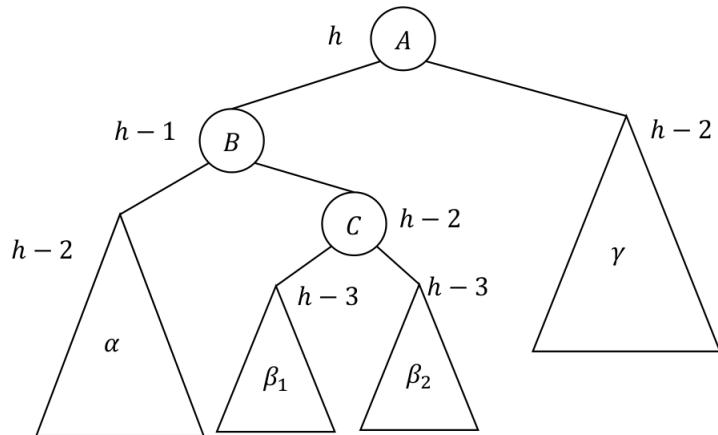
Inoltre, visto che l'altezza dell'albero dopo la rotazione resta invariata (uguale a quella prima dell'inserimento) chiaro che neanche la violazione sull'altezza venga propagata sugli antenati del sottoalbero ruotato. Dimostriamo graficamente questa proprietà:



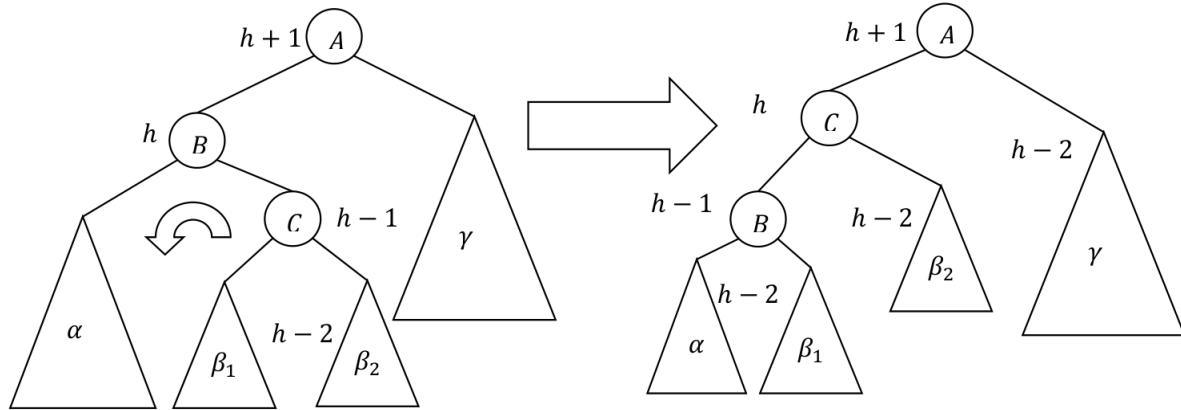
Abbiamo visto che la rotazione singola sinistra risolve il problema della violazione nel caso in cui venga inserito un elemento nel sottoalbero estremo sinistro (non è detto che un inserimento mi porti ad avere una violazione)

Se però avessimo inserito in β anziché in α non avremmo risolto nulla. Cosa abbastanza evidente, visto che con la rotazione a sinistra spostiamo il peso dal sottoalbero estremo sinistro al sottoalbero estremo destro (non toccando di fatto per niente il peso di β).

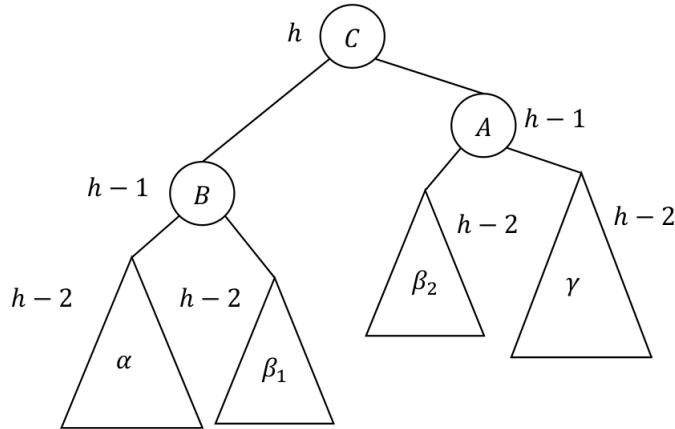
Analogamente al ragionamento fatto per la rotazione sinistra si potrebbe pensare che una rotazione destra sposterebbe il peso dall'estremo destro all'estremo sinistro. Nel dettaglio:



Supponiamo che dopo l'inserimento l'altezza di β aumenti di uno in entrambi i sottoalberi β_1 e β_2 .³⁰



La violazione non è stata ancora risolta, ma adesso è il sottoalbero estremo sinistro ad avere un peso maggiore rispetto all'estremo destro → Ci basta pertanto applicare una rotazione sinistra, ottenendo il seguente albero



Anche in questo caso, con una doppia rotazione non solo manteniamo la proprietà di ordinamento, ma anche la stessa altezza prima dell'inserimento.

Ne consegue che tutto il ragionamento fatto per la singola rotazione è ancora valido → Risolviamo la situazione in maniera totale (se queste operazioni sono state fatte in un sottoalbero T' di T , allora la violazione risolta in T' è risolta anche in T).

Possiamo concludere che se l'inserimento viene fatto a sinistra della radice A :

- ▶ Se si crea una violazione in α risolviamo con una singola rotazione sinistra;

³⁰È una casistica impossibile con un solo inserimento, ma puntiamo a rendere il ragionamento il più generale possibile così da mostrare che a prescindere da sottoalbero c in cui inseriamo la doppia rotazione risolve il problema.

- Se si crea una violazione in β risolviamo con una rotazione destra seguita da una rotazione sinistra;

Quindi, se l'inserimento crea una violazione in γ , si va semplicemente a considerare l'albero γ :

- La violazione è nel sottoalbero estremo sinistro → Risolvo con una singola rotazione sinistra;
- La violazione è nel sottoalbero estremo del figlio sinistro → Risolvo con una doppia rotazione.

Ricordiamo che la risoluzione locale nel sottoalbero risolve in maniera totale la violazione.

Abbiamo quindi coperto tutti i casi possibili; prima di implementare gli algoritmi bisogna notare che la violazione è individuabile solo conoscendo l'altezza in cui si trova quel determinato nodo.

Ciò rende necessario aggiungere in ogni nodo una cella di memoria che descriva l'altezza di quel nodo, con conseguenza di dover usare una memoria aggiuntiva pari a $n \log (\log n)$.³¹

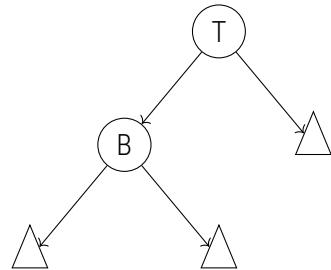
Si potrebbe pensare di calcolare in maniera ricorsiva l'altezza evitando di dover usare tutta quella memoria aggiuntiva (attraverso la linea $h(T) = 1 + \max(h(T \rightarrow sx), h(T \rightarrow dx))$); questa operazione avrebbe un tempo di esecuzione lineare sul numero di nodi (dobbiamo conoscerla per ogni nodo), andando quindi a vanificare tutti i nostri ragionamenti³²

```

1 RotazioneSingolaSx(T)
2   B = T -> sx
3   x = B -> dx
4   B -> dx = T
5   T -> sx = x
6   T -> h = 1 + max(Altezza(T -> sx), Altezza(T -> dx))
7   B -> h = 1 + max(T -> sx, Altezza(B -> dx))
8   RETURN B

```

Prima della rotazione



```

1 RotazioneDoppiaSx(T)
2   T -> sx RotazioneSingolaDx(T -> sx)
3
4
5   RETURN RotazioneSx(T)

```

```

1 Altezza(T)
2   IF T = NIL THEN
3     RETURN -1
4   ELSE
5     RETURN T -> h

```

Le operazioni di `RotazioneSingolaDx` e `RotazioneDoppiaDx` sono esattamente duali alle precedenti (ci basta scambiare `sx` con `dx` e viceversa).

³¹Dove n rappresenta il numero di nodi, $\log (\log n)$ deriva dall'utilizzo di una cella di memoria il valore massimo è un intero $h = \log n$.

³²Ricordiamo che stiamo facendo tutto questo per rendere le operazioni sugli alberi binari eseguibili in tempo lineare sull'altezza.

Riportiamo l'operazione di inserimento:

```
1 InsertAVL(T, k)
2   IF T != NIL THEN
3     // Se inserisco a destra la violazione potrà crearsi solo a destra
4     IF T -> key < k THEN
5       T -> dx = InsertAVL(T -> dx, k)
6       T = BilanciaDx(T)
7     // Se inserisco a sinistra la violazione potrà crearsi solo a sinistra
8     IF T -> key > k THEN
9       T -> sx = InsertAVL(T -> sx, k)
10      T = BilanciaSx(T)
11    ELSE // Creazione di una foglia
12      // Abbiamo praticamente un inserimento in un albero binario di ricerca con l'aggiunta di 3
13      // operazioni
14      T = AllocaNodoAVL()
15      T -> key = k
16      T -> sx = T -> dx = NIL
17      T -> h = 0
18
19 RETURN T
```

Resta solo da descrivere l'operazione di bilanciamento (vedremo solo quello a sinistra, visto che per quello a destra basta invertire sx con dx e viceversa).

Non è necessario avere il valore assoluto perché abbiamo inserito a sinistra → Se si crea una violazione sarà l'altezza del sottoalbero sinistro è aumentata di 1 (diventando troppo pesante rispetto al destro).

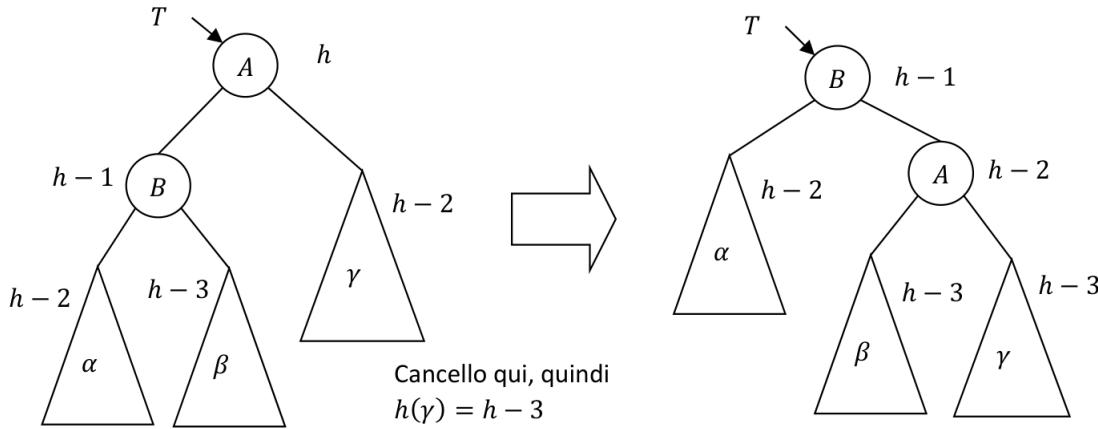
```
1 BilanciaSx(T)
2   IF Altezza(T -> sx) - Altezza(T -> dx) > 1 THEN
3     IF Altezza((T -> sx) -> ) > Altezza( (T -> sx) -> dx) THEN
4       RETURN RotazioneSingolaSx(T)
5     ELSE // La rotazione è in  $\beta$ 
6       RETURN RotazioneDoppiaSx(T)
7   ELSE // Se non abbiamo violazioni devo aggiornare solo l'altezza
8     T -> h = 1 + max(Altezza(T -> sx), Altezza(T -> dx))
9
10 RETURN T
```

Questo risultato ci permette di avere una risoluzione totale andando a eliminare la violazione localmente.

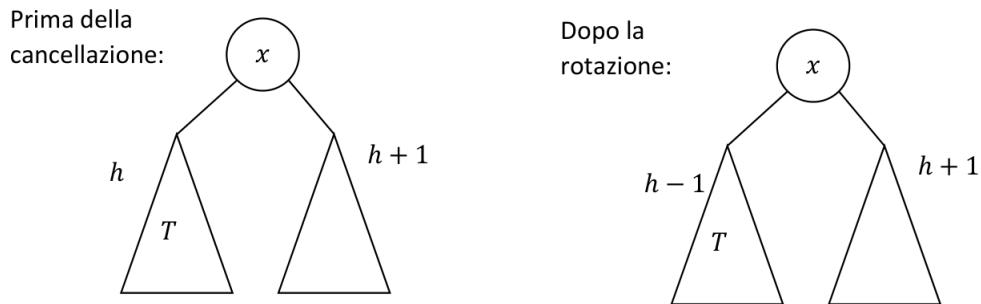
21 Lezione 24

21.1 Operazione di cancellazione in AVL

Sfortunatamente il discorso fatto per l'inserimento non vale per la cancellazione → Esiste la possibilità che la violazione possa essere propagata sui suoi antenati³³. Ad es:



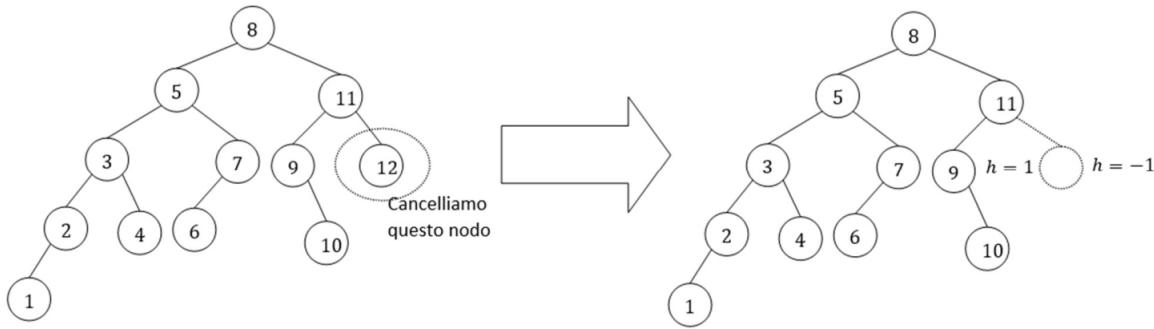
Dove T è un sottoalbero nella seguente situazione:



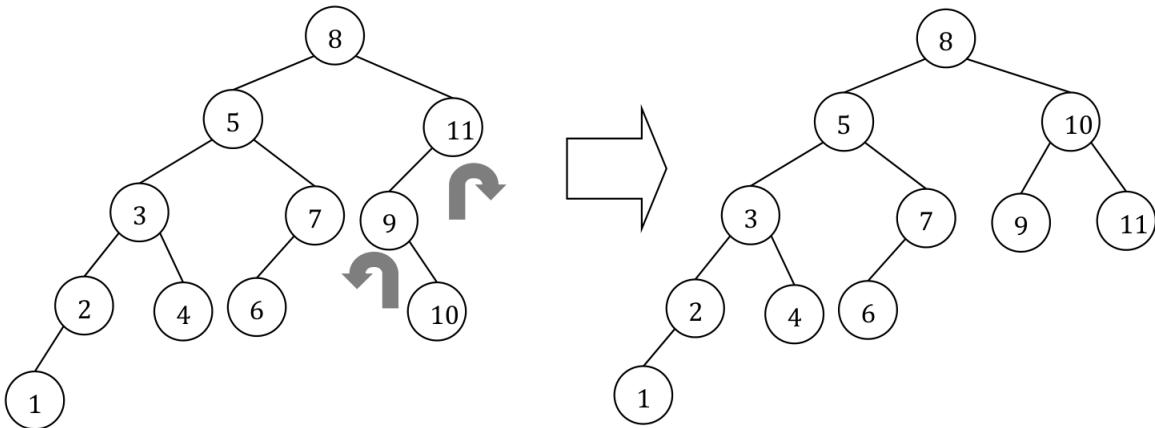
Pertanto se c'è una violazione allora dopo il bilanciamento l'altezza sarà sempre decrementata di uno rispetto all'altezza prima della cancellazione, e questo potrebbe propagare la violazione al padre (questo succede se prima della cancellazione valeva $|h(P \rightarrow sx) - h(P \rightarrow dx)| = 1$ con P che rappresenta il padre della dell'albero appena bilanciato).

Vediamo a questo punto quante violazioni sequenziali si possano avere nel caso peggiore. Sorvoliamo sulla differenza tra operazioni singole e doppie, nel caso peggiore la violazione si potrebbe propagare in maniera lineare rispetto all'altezza (quindi $\log n$ nel caso siano solo operazioni singole e $2 \log n$ per sole operazioni doppie). Prendiamo ad es il seguente AVL:

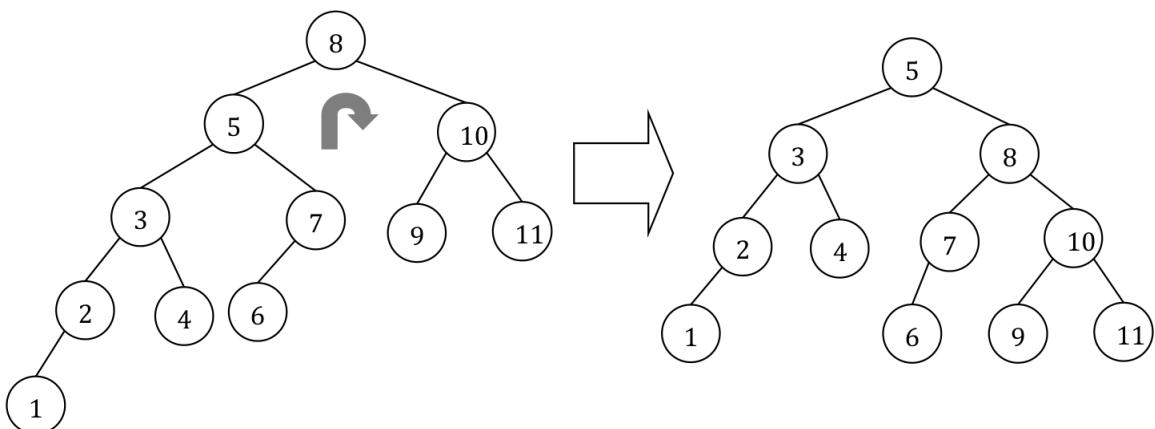
³³Ciò deriva dall'impossibilità di riportare l'altezza al valore prima della cancellazione.



Un primo bilanciamento è una doppia rotazione destra:



Questo albero ha ancora una violazione (il nodo 5 ha altezza 3, mentre il nodo 10 ha altezza 1), ne consegue che è necessario un altro bilanciamento attraverso una rotazione singola sinistra:



Dunque, l'albero di partenza di altezza $h = 4$ diventa, dopo 2 operazioni di rotazione, di $h = 3$.

Si noti che l'AVL dell'esempio precedente è un AVL minimo, questo tipo di AVL crea una violazione su qualsiasi nodo cancellato.

Se poi viene cancellata la foglia più in alto (quindi quella meno profonda) si crea la situazione descritta in precedenza, ovvero il caso peggiore dove sono necessarie tante operazioni di bilanciamento quanto è lungo il percorso più breve.

A differenza dell'inserimento, nella cancellazione dobbiamo verificare ogni volta che l'albero sia bilanciato durante la risalita (questo lavoro viene svolto anche dall'algoritmo di inserimento per come lo abbiamo definito, ma è possibile evitarlo attraverso l'uso di un flag booleano):

```
1 CancellaAVL(T, k) // È praticamente l'algoritmo di cancellazione per ABR
2   IF T != NIL THEN
3     IF T -> key > k THEN
4       T -> sx = CancellaAVL(T -> sx, k)
5       // Se cancelliamo a sinistra allora l' albero più pesante è il destro quindi usiamo
6       BilanciaDx
7       T = BilanciaDx(T)
8     ELSE IF T -> key < k THEN
9       T -> dx = CancellaAVL(T -> dx, k)
10      T = BilanciaSx(T)
11    ELSE
12      T = CancellaAVL_Root(T)
13
14 RETURN T
```

Definiamo quindi *CancellaAVL_ROOT*:

```
1 CancellaAVL_ROOT(T)
2   IF T != NIL THEN
3     IF T -> sx = NIL OR T -> dx = NIL THEN
4       IF T -> sx = NIL THEN
5         tmp = T -> dx
6       ELSE
7         tmp = T -> sx
8       dealloca(T)
9     RETURN tmp
10    ELSE
11      T -> key = StaccaAVL_Min(T -> dx, T)
12      T = BilanciaSx(T)
13
14 RETURN T
```

Si noti che non è necessario aggiornare le altezze perché nell'**IF** sono già corrette, mentre nell'**ELSE** se ne occuperà l'operazione di bilanciamento.

Anche per **StaccaAVL_Min** non ha bisogno di aggiornare le altezze perché lo fa correttamente **BilanciaDx**

```
1 StaccaAVL_Min(T)
2   IF T != NIL THEN
3     // La radice di T potrebbe cambiare per questo usiamo una variabile temporanea
4     IF T -> sx != NIL THEN
5       val = StaccaAVL_Min(T -> sx, T)
6       tmp = BilanciaDx(T)
7     ELSE
8       val = T -> key
9       tmp = T -> dx
10      // Non è necessaria in quanto sia in CancellaAVL_Root che CancellaAVL_Min abbiamo la
11      // certezza che il padre non sia mai nullo
12      // Questo controllo rende però possibile usare questo algoritmo in maniera indipendente (
13      // lo rende più robusto)
14      IF p != NIL THEN
15        IF T = p -> sx THEN
16          p -> sx = tmp
17        ELSE
18          p -> dx = tmp
19          dealloca(T)
20      RETURN val
```

Abbiamo terminato la discussione sugli AVL, la nostra prima struttura che ammette tutte le operazioni in tempo logaritmico. Ha praticamente la stessa complessità computazionale degli array ordinati ma ne migliora le operazioni di modifica.

21.2 Alberi Red-Black

Gli alberi Red-Black sono una struttura dati con proprietà simili agli AVL (anche tempo di esecuzione logaritmico) ma più tolleranti
→ Riduce il numero necessario di rotazioni per ribilanciare la struttura di fronte a delle modifiche.

Un albero RB è essenzialmente un albero di ricerca in cui:

- ▶ Le chiavi vengono mantenute solo nei nodi interni dell'albero;
- ▶ Le foglie sono costituite da speciali nodi **NIL** (nodi sentinella) il cui contenuto è irrilevante → Evitano di dover trattare diversamente i puntatori ai nodi dai puntatori **NIL**
 - ◊ Al posto di un puntatore **NIL** si usa un puntatore ad un nodo **NIL**;
 - ◊ Quando un nodo ha come figli nodi **NIL**, significa che si tratta di una foglia nell'albero binario di ricerca corrispondente.

La particolarità del RB è proprio quella di avere le foglie dell'albero senza dati; La cardinalità dell'albero è pari al numero di nodi interni (se si parla dal punto di vista di dati).

Il vincolo di bilanciamento è definito da un etichetta dei nodi → Ad ogni nodo è associato un colore (che può essere conservato in un bit) rosso o nero.

Il vincolo globale è espresso dalle seguenti proprietà:

1. Tutti i nodi hanno un colore (rosso o nero);
2. Le foglie (i nodi **NIL**) sono sempre e solo nere;
3. Ogni nodo rosso ha entrambi i figli neri (questo ci garantisce che i neri sono almeno la metà);
4. Ogni percorso da un nodo interno ad un nodo **NIL** ha lo stesso numero di nodi neri.

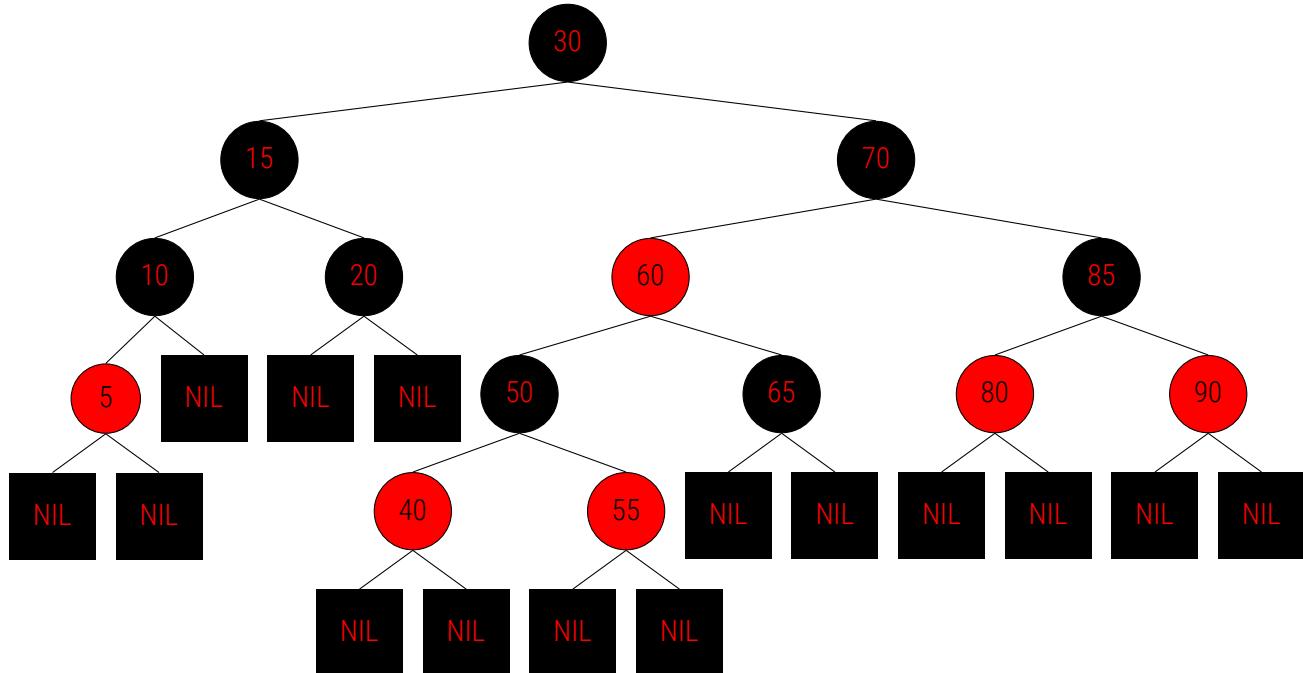
Questi vincoli (in particolare gli ultimi 2) garantiscono un altezza logaritmica sul numero dei nodi.

Altezza nera dell'albero radicato in x

Per altezza nera nell'albero radicato in x il numero di nodi neri di un percorso (che si estende a tutti i percorsi per la proprietà 4).

Inoltre dato che il colore di x è ininfluente non viene calcolato nell'altezza → Se ho 2 alberi con colore identico eccetto per la radice (uno con radice nera e l'altra rossa) questi avranno la stessa radice nera.

Di seguito riportiamo un esempio di albero RB:



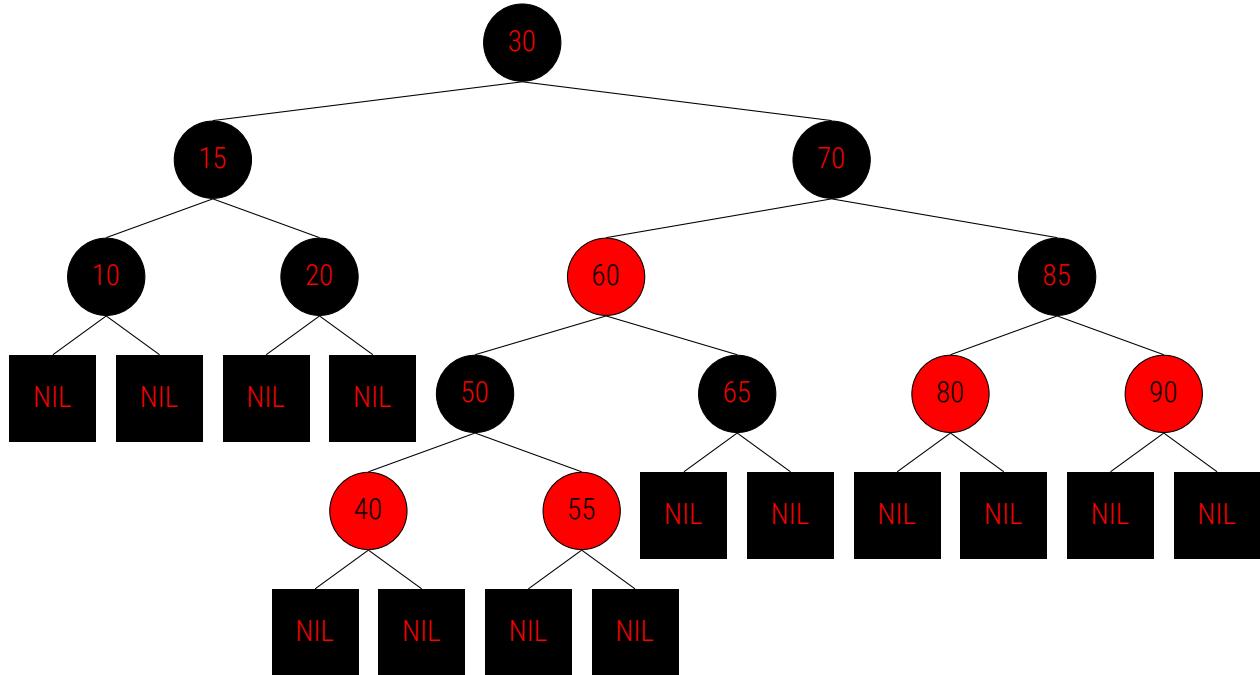
Il miglior modo per costruire un RB è a partire dal percorso più breve → I nodi di tale percorso vanno colorati tutti di nero (devo massimizzare il più possibile i nodi neri → Se nel percorso più breve mettessi dei nodi rossi allora non è detto che riuscirei a garantire la proprietà 4).

Una volta effettuato questa operazione ed aver determinato il numero di nodi neri di ogni percorso inizierò a colorare i percorsi restanti cercando di non violare la proprietà 3 e continuando a rispettare la proprietà 4.

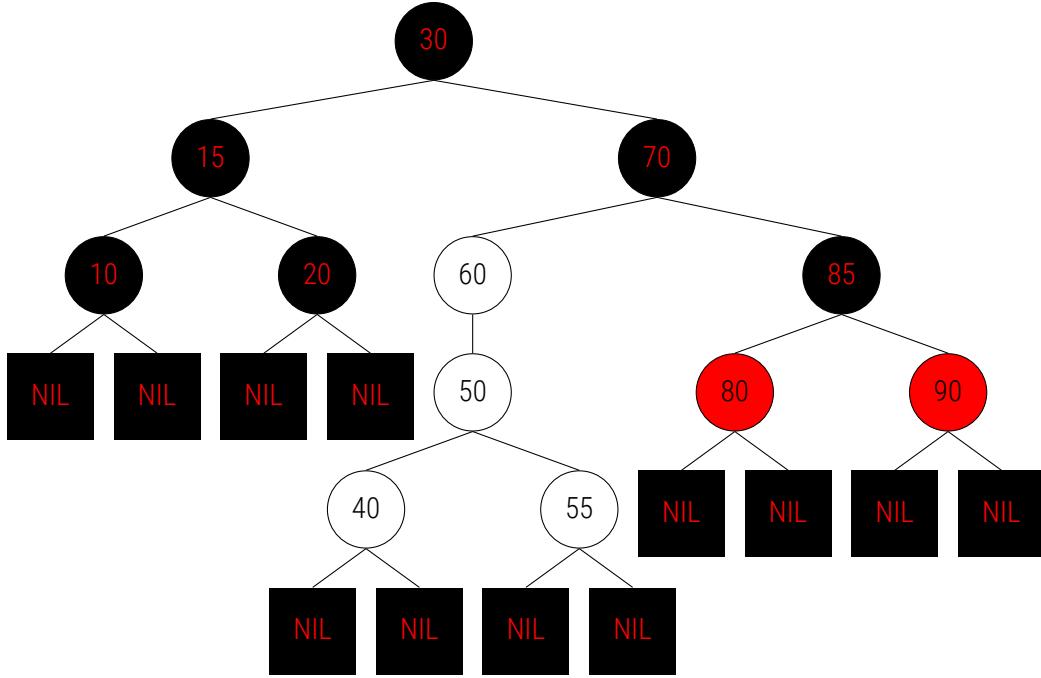
La colorazione di un albero non è necessariamente unica → Un esempio è l'albero pieno che ha il numero maggiore possibile di differenti colorazioni (posso farlo tutto nero, o alternando un livello nero con uno rosso, o ancora un livello rosso per ogni 2 neri, e così via) e man mano che sbilancia l'albero le variazioni di colorazione diminuiscono fino ad arrivare ad uno sbilanciamento tale da non permettere ulteriori colorazioni.

Ogni albero AVL è RB (ne è un esempio l'albero precedente). Si potrebbe dimostrare per induzione che è sempre possibile colorare un AVL (cosa che non vedremo).

Di seguito riportiamo invece un albero RB non AVL:

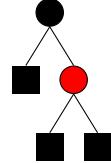
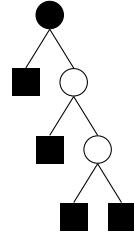


Di seguito invece riportiamo un albero, impossibile da colorare rispettando le proprietà di RB:



Gli alberi degeneri non banali non sono colorabili:

L'altezza nera è 1 e non posso utilizzare 2 nodi rossi adiacenti
Gli alberi degeneri più alti possono solo peggiorare la situazione



Questo è l'unico albero degenere RB (si noti che è anche un albero AVL)

22 Lezione 25

22.1 Dimostrazione che gli alberi Red-Black hanno altezza logaritmica sul numero di nodi

Sia n la cardinalità dell'insieme³⁴. Dimostriamo che vale la seguente relazione

$$\forall T \in RB \quad h(T) \leq 2 \log(n + 1) (\Theta(\log n))$$

Per dimostrare tale proprietà andremo a sfruttare una relazione tra il numero di nodi interni e l'altezza nera.

³⁴Che non coincide con il numero di nodi totali dell'albero, ma ne rappresenta solo i nodi interni.

Sia $n(x)$ il numero di nodi interni di un generico sottoalbero T_x , risulta che $n(x) \geq 2^{bh(x)} - 1$, dove $bh(x)$ rappresenta l'altezza nera.

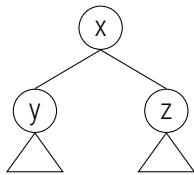
È possibile notare che questa relazione è simile al numero di nodi di un albero pieno $\rightarrow 2^{h+1} - 1$.

Se in un albero RB eliminiamo tutti i nodi rossi otteniamo proprio un albero pieno con altezza $bh(x)$ e non dovendo considerare il livello della foglia la relazione $n(x) \geq 2^{bh(x)} - 1$ è intuitivamente vera.

Andiamo a definire una rappresentazione più formale della precedente relazione (si procede per induzione sull'altezza).

Il caso base è $h_x = 0$ e quindi T_x è rappresentato solo dal nodo **NIL**; allora $n(x) = 0$ e $bh(x) = 0 \implies 2^{bh(x)} - 1 = 2^0 - 1 = 0$ e la relazione $n(x) \geq 0$ è soddisfatta.

Sia ora $h_x = h > 0$ e l'albero T_x =



Si noti che y e z sono certamente esistenti in quanto devono essere almeno foglie (nodi **NIL**)

Ma se $h_x = h$ allora $h_y < h$ e $h_z < h$; se l'altezza dei sottoalberi è strettamente minore dell'altezza di x possiamo applicare il caso induttivo (che ci permette di dire che per tutti i valori che vanno dal caso base ad h sono veri) e supporre la relazione vera per T_y e T_z , ovvero:

$$n(y) \geq 2^{bh(y)} - 1 \wedge n(z) \geq 2^{bh(z)} - 1$$

Dunque risulta:

$$n(x) = 1 + n(y) + n(z) \implies n(x) \geq 1 + 2^{bh(y)} - 1 + 2^{bh(z)} - 1 \implies n(x) \geq 2^{bh(y)} + 2^{bh(z)} - 1$$

A questo punto serve definire una relazione tra le altezze nere di y e z con l'altezza nera di x .

Sappiamo che $bh(y)$ ignora il colore di y , così come $bh(x)$ ignora il colore di x ma non quello di y , quindi a seconda del colore di y possiamo avere i seguenti casi:

- y è un nodo rosso $\rightarrow bh(x) = bh(y)$
- y è un nodo nero $\rightarrow bh(x) = bh(y) + 1$

Posso dunque concludere che $bh(x) - 1 \leq bh(y) \leq bh(x)$ e, allo stesso modo per il nodo z , $bh(x) - 1 \leq bh(z) \leq bh(x)$.

Sfruttando la monotonicità della funzione esponenziale è lecito scrivere:

$$2^{bh(x)-1} \leq 2^{bh(y)} \text{ e analogamente } 2^{bh(x)-1} \leq 2^{bh(z)}$$

Ritornando all'equazione principale con i seguenti passaggi abbiamo concluso la dimostrazione:

$$\begin{aligned} n(x) &\geq \underbrace{2^{bh(y)}}_{\geq 2^{bh(x)-1}} + \underbrace{2^{bh(z)}}_{\geq 2^{bh(x)-1}} - 1 \geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \implies \\ &\implies n(x) \geq 2 \cdot 2^{bh(x)-1} - 1 \implies n(x) \geq 2^{bh(x)} - 1 \end{aligned}$$

Questo discorso fatto per un generico sottoalbero varrà ovviamente anche per la radice dell'albero completo³⁵

Resta quindi solo da trovare una relazione tra altezza e altezza nera (lo scopo principale è quello di arrivare a $h = \Theta(\log n)$). Sappiamo che in un percorso ci devono essere almeno tanti neri quanti rossi (in caso contrario verrebbe violata la proprietà 3 dei RB), dunque:

$$\frac{h}{2} \leq bh \leq h \implies bh \geq \frac{h}{2} \implies 2^{bh} \geq 2^{\frac{h}{2}} \implies 2^{bh} - 1 \geq 2^{\frac{h}{2}} - 1$$

Ma allora come volevasi dimostrare:

$$n \geq 2^{bh} - 1 \geq 2^{\frac{h}{2}} - 1 \implies n + 1 \geq 2^{\frac{h}{2}} \implies \log(n + 1) \geq \frac{h}{2} \log 2 \implies h \leq 2 \log(n + 1)$$

22.2 Inserimento in un albero Red Black

L'algoritmo di inserimento deve preservare oltre ai vincoli di RB anche quelle di ordinamento; in altre parole deve sempre eseguire una ricerca binaria → Ciò significa che esiste una sola foglia dove inserire un determinato dato.

Il primo vincolo di RB è che il nodo abbia un colore, ma risulta evidente che se inserissimo un non nero questo genererebbe una violazione certa della proprietà 4; di conseguenza la migliore soluzione è quella di sostituire la foglia con un nodo rosso con 2 foglie.

Questo ci consente di dover bilanciare l'albero solo se il padre della foglia (e quindi del nuovo nodo rosso aggiunto) è rosso. Inoltre, tale metodo ci assicura che la proprietà 3 è l'unica che possa essere violata e la soluzione sarà quella di spostare verso l'alto i 2 nodi rossi fino alla radice, che andrà poi colorata di nero.

Questo processo funziona solo se la radice prima dell'inserimento è nera; in caso contrario avremo 2 violazioni e non più una (3 nodi rossi in successione), ma possiamo supporre sempre vera tale proprietà perché colorare la radice di nero non causa nessuna violazione alle proprietà di RB.

Riportiamo di seguito l'implementazione dell'inserimento supponendo che dopo l'algoritmo di inserimento ci sia un'istruzione che metta il colore della radice dell'albero a nero (abbiamo la certezza di operare solo su alberi RB con radice nera):

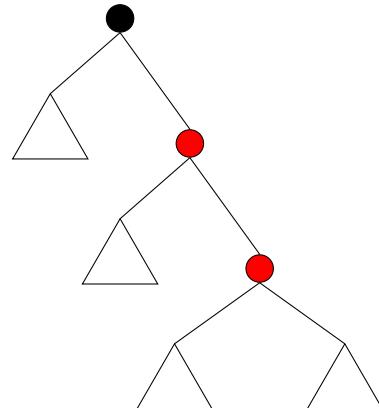
³⁵Quindi se si prende un albero **RB** con un'altezza nera pari a **bh** il numero di chiavi dell'albero è $n \geq 2^{bh} - 1$.

```

1 InsertRB(T, k)
2 // isNIL(T) è una funzione che restituisce vero se l'unico nodo presente è una foglia NIL
3 // (Non possiamo più utilizzare T != NIL proprio a causa dei nodi foglia di un RB)
4 IF !isNIL(T) THEN
5   IF T -> key > k then
6     T -> sx = InsertRB(T -> sx, k)
7     T = BilanciaRB_Sx(T)
8   ELSE IF T -> key < k then
9     T -> dx = InsertRB(T -> dx, k)
10 ELSE // Mi trovo in una foglia
11   x = creaNodoRB() // La funzione creaNodoRB crea il nodo con già le foglie NIL
12   x -> key = k
13   x -> c = R
14 RETURN x
15 RETURN T

```

L'idea del bilanciamento è abbastanza semplice; sappiamo che la radice è di colore nero e che l'unica violazione che può essere generata è della proprietà 3 → Pertanto per ogni nodo controllerà il figlio e il figlio di quest ultimo:

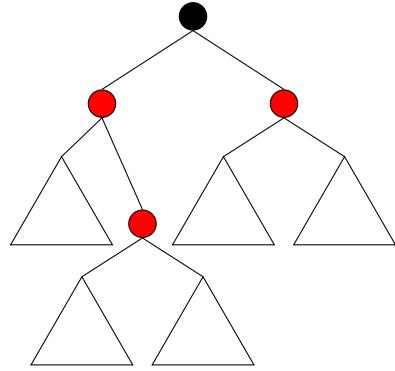
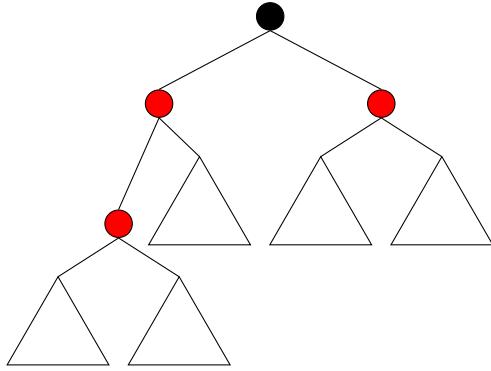


Il nodo che si accorge della violazione è evidentemente nero, perché altrimenti avremmo ben 2 violazioni (ricordiamo che un inserimento ne genera al massimo una)

Ci sono 3 possibili casi che si gestiscono in maniera differente. Di seguito riportiamo la gestione dei casi per:

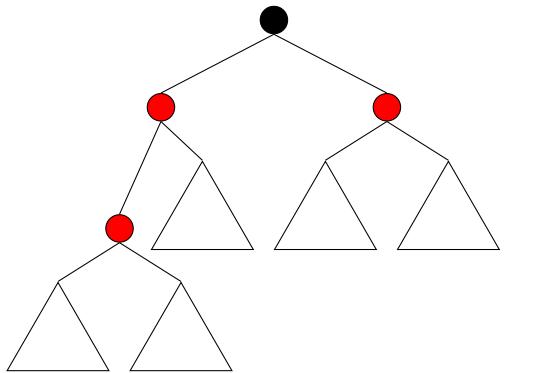
- ▶ Inserimento a sinistra (che genererà una violazione nel sottoalbero sinistro);
- ▶ Inserimento a destra (che è il duale → basta scambiare sx con dx).

Caso 1

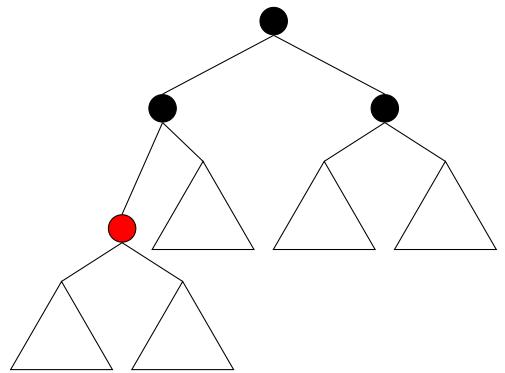


Anche se le precedenti rappresentazioni sono differenti (una ha la violazione a sinistra con il figlio sinistro e l'altra ha una violazione a sinistra con il figlio destro), entrambe attuano la stessa soluzione (andremo a considerarlo come caso unico).

Visto che il figlio destro della radice che si accorge della violazione è rosso basta cambiare i colori al livello della radice e il successivo per risolvere la violazione:

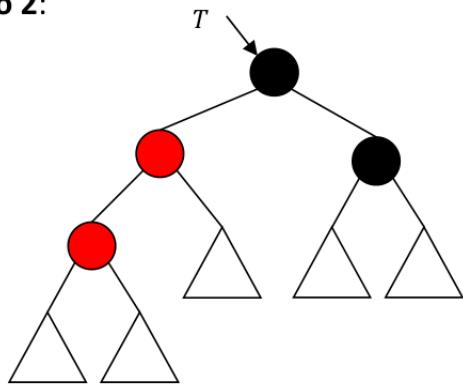
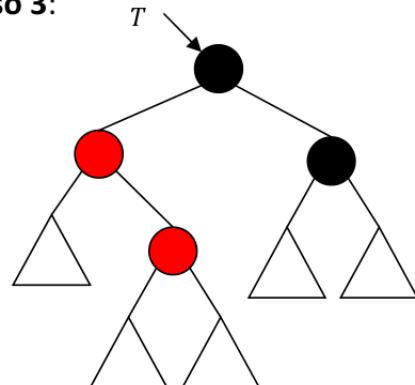


DIVENTA

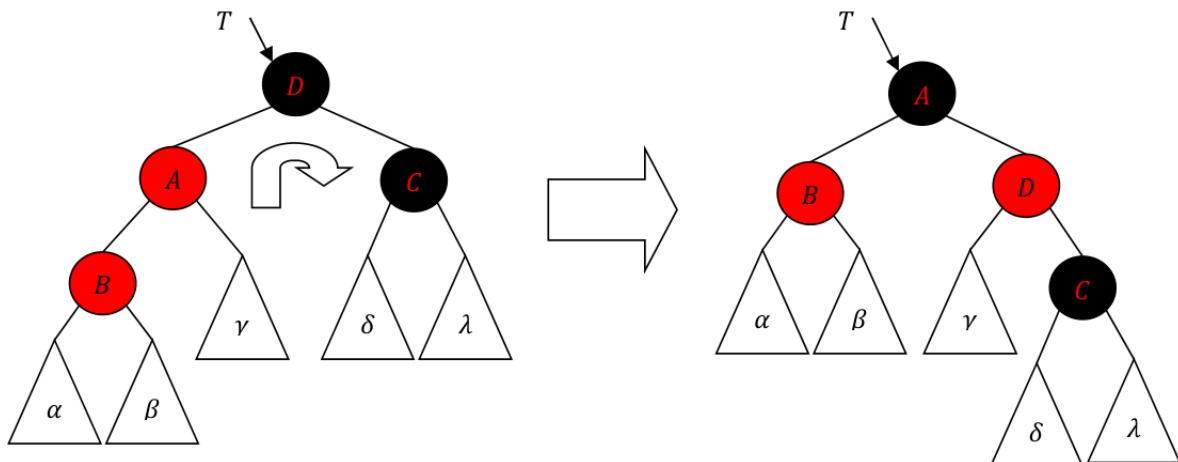


Il fatto che con questa operazione abbiamo risolto la violazione di tipo 3 è evidente (si noti inoltre di come non sia possibile violare in alcun modo le altre proprietà in quanto l'altezza nera resta invariata).

A questo punto se la radice di T è quella dell'albero totale ho risolto (andrà colorata di nero dopo l'inserimento); in caso contrario il padre del sottoalbero T potrebbe essere rosso ed in quel caso ho semplicemente propagato la violazione verso l'altro e dovrò ripetere la precedente operazione

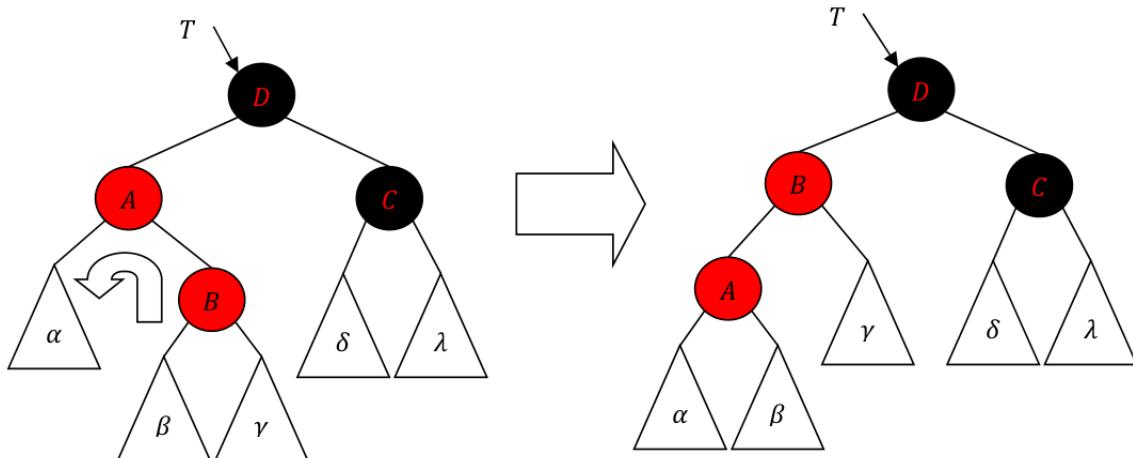
Caso 2:**Caso 3:**

Iniziamo dal caso 2. Per risolvere tale violazione proviamo ad eseguire la rotazione singola a sinistra già vista con gli alberi AVL:



Dopo la rotazione per risolvere la violazione è necessario invertire i colori tra A e D; inoltre essendo l'albero γ sottoalbero di un nodo rosso (e quindi la radice di γ è di colore nero) anche prima della rotazione siamo sicuri che il nuovo colore di D non introduce alcuna altra violazione.

Ci resta da analizzare il caso 3, a cui verrà applicata una rotazione doppia a sinistra:



Dopo la prima rotazione singola destra ci ritroviamo esattamente nella situazione del caso 2 → siamo certi che applicando un'altra rotazione singola a sinistra e aver cambiato il colore del nodo radice e del suo figlio destro abbiamo risolto la violazione anche per il caso 3.

Si noti che per entrambi i casi le violazioni vengono risolte totalmente senza il rischio di propagazione verso l'alto (la radice di T resta nera)

Le operazioni di bilanciamento quindi risolvono la violazione dopo al più 2 rotazioni (doppia rotazione del caso 3) perché anche se il caso 1 può propagare la violazione fino alla radice dell'albero totale, esso non fa alcuna rotazione.

```

1 BilanciaRB_Sx(T)
2   v = ViolazioneSx(T) // Restituisce il caso in cui ci troviamo
3   CASE v OF
4     // Caso_N_Sx(T) lo gestisce
5     1: T = Caso1Sx(T)
6     2: T = Caso2Sx(T)
7     3: T = Caso3Sx(T)
8   RETURN T

```

```

1 ViolazioneSx(T)
2   A = T -> sx
3   IF A -> c = R THEN
4     C = T -> dx
5     IF C -> c = R
6       IF (C -> sx) -> c = R OR (C -> dx) -> c = R THEN
7         RETURN 1
8     ELSE // C è nero
9       IF (C -> sx -> c) = R THEN
10        RETURN 2
11      ELSE IF (C -> dx -> c) = R THEN
12        RETURN 3

```

I seguenti algoritmi sono delle possibili implementazioni:

```

1 Caso1Sx(T)
2   T -> c = R
3
4   (T -> sx) -> c = N
5
6   (T -> dx) -> c = N
7
8   RETURN T

```

```

1 Caso2Sx(T)
2   A = T -> sx
3   gamma = A -> dx
4   A -> dx = T
5   T -> sx = gamma
6   A -> c = N
7   (A -> dx) -> c = R
8   RETURN A

```

```

1 Caso3Sx(T)
2   A = T -> sx
3   B = T -> dx
4   beta = B -> sx
5   B -> sx = A
6   A -> dx = beta
7   T -> sx = B
8   RETURN Caso2Sx(T)

```

23 Lezione 26

23.1 Cancellazione negli alberi RB

Sfortunatamente nella cancellazione non abbiamo modo di scegliere il colore del nodo da cancellare.

Se cancelliamo un nodo rosso non ci saranno violazione ai vincoli; al contrario nel cancellare un nodo nero viene sicuramente creata una violazione di tipo 4 → Questo vincolo essendo globale, è più difficile da sistemare rispetto a quello di tipo 3 visto per l'inserimento.

L'idea è quella di fingere che il colore nero venga spostato in un altro nodo:

- ▶ Se fosse stato rosso allora diventerebbe nero;
- ▶ Se fosse stato già nero (per non perdere un nodo nero) diventerebbe *doppio nero* (questo nuovo colore contribuirà di 2 e non più 1 all'altezza nera).

In questo modo ho convertito una violazione del vincolo globale (4) in una violazione del vincolo locale (2).

Nel caso in cui il *doppio nero* finisce in radice la soluzione è quella di colorare la radice di nero (ricordiamo che ciò non introduce nessuna violazione → le altezze nere di ogni percorso rimangono invariate).

L'algoritmo di cancellazione è quasi identico a quello visto per gli AVL; la differenza sta nelle procedure ausiliare. Supponiamo inoltre che dopo l'utilizzo del seguente algoritmo venga effettuata una istruzione che colori di nero la radice (risolve il caso del doppio nero in radice):

```

1 CancellaRB(T, k)
2   IF !isNIL(T) THEN
3     IF T -> key > k THEN
4       T -> sx = CancellaRB(T -> sx, k)
5       T = BilanciaCancSx(T)

```

```

6      IF T -> key < k THEN
7          T -> dx = CancellaRB(T -> dx, k)
8          T = BilanciaCancDx(T)
9      ELSE
10         T = CancRB_Root(T)
11     RETURN T

```

Sarà `CancRB_Root` ad occuparsi dell'individuazione e risoluzione della violazione.

Nel caso in cui questa si propaghi al padre allora interverranno le linee 5 o 8 una volta terminata la chiamata ricorsiva (se è la chiamata base allora risolveremo con la colorazione a nero della radice).

```

1 CancellaRB_Root(T)
2   IF !isNIL(T) THEN
3       IF !isNIL(T -> sx) OR !isNIL(T -> dx) THEN
4           Tmp = T
5           IF isNIL(T -> sx) THEN
6               T = T -> dx
7           ELSE
8               T = T -> sx
9           // Bisogna modificare il colore della radice
10          PropagaNero(Tmp, T)
11          dealloca(Tmp)
12      ELSE
13          k = StaccaMinRB(T -> dx, T)
14          T -> key = k
15          // T potrebbe avere una violazione nel sottoalbero destro
16          T = BilanciaCancDx(T)
17      RETURN T
18
19  /* ----- */
20 PropagaNero(x, y)
21   IF x -> c = N THEN
22       IF y -> c = R THEN
23           y -> c = N
24       ELSE
25           y -> c = DN

```

L'algoritmo di `StaccaMinRB(T, P)` ha una struttura identica quella vista per gli *AVL* (non aggiungeremo il controllo sulla variabile *T* visto che abbiamo la certezza che non sia mai un nodo foglia):

```

1 StaccaMinRB(T, P)
2   IF !isNIL(T -> sx) THEN
3       val = StaccaMinRB(T -> sx, T)
4       tmp = BilanciaCancSx(T)

```

```

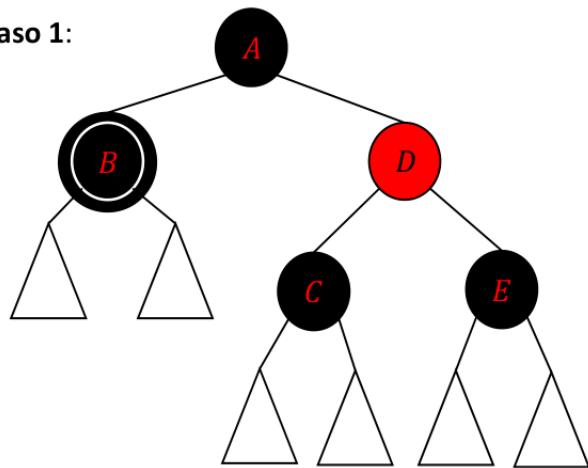
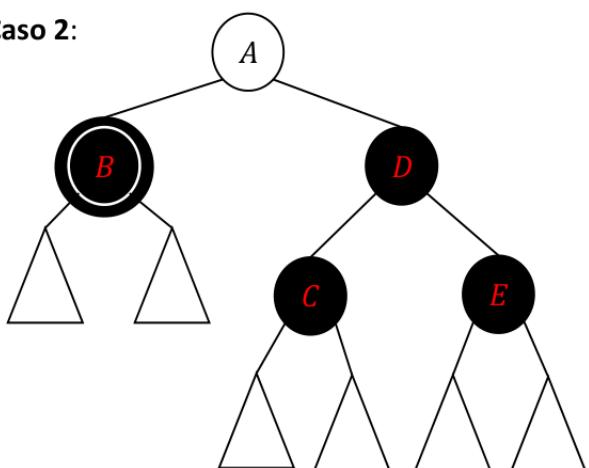
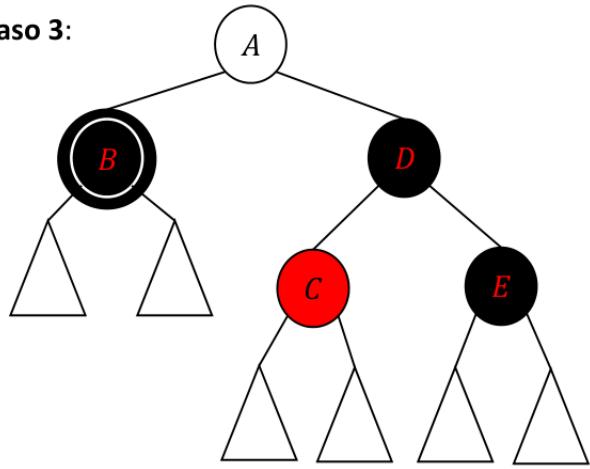
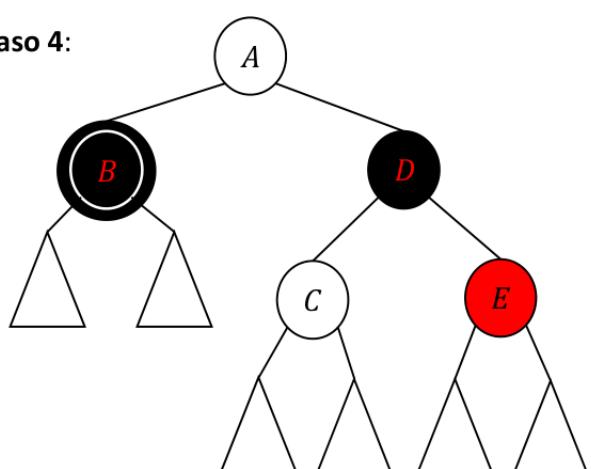
5      ELSE
6          val = T -> key
7          tmp = T -> dx
8          IF T = P -> sx THEN
9              P -> sx = tmp
10         ELSE
11             P -> dx = tmp
12             PropagaNero(T, tmp)
13             dealloca(T)
14         RETURN val

```

Passiamo ora a definire i vari casi del bilanciamento.

Per i seguenti ragionamenti considereremo che la cancellazione sia avvenuta nel sottoalbero sinistro (quindi l'eventuale doppio nero sarà a sinistra); ricordiamo che il ragionamento per `BilanciaCancDx` è duale (bisogna scambiare `sx` con `dx` e viceversa).

Per il bilanciamento ci sono 3 casistiche (se il nodo è bianco allora il comportamento è lo stesso a prescindere che tale nodo sia colorato di nero o rosso)

Caso 1:**Caso 2:****Caso 3:****Caso 4:**

Il caso 2 e 3 includono 2 possibilità (una per la radice rossa e una per quella nera), mentre il caso 4 ne include 4 (una per ogni combinazione di colore possibile tra A e C).

È evidente che per capire il caso in cui ci troviamo non serve analizzare la radice perché sappiamo che se il figlio destro è rosso allora la radice sarà nera.

Quindi se il sinistro è doppio nero allora basta controllare nel giusto ordine gli alberi del sottoalbero destro per capire il caso in cui ci troviamo:

```

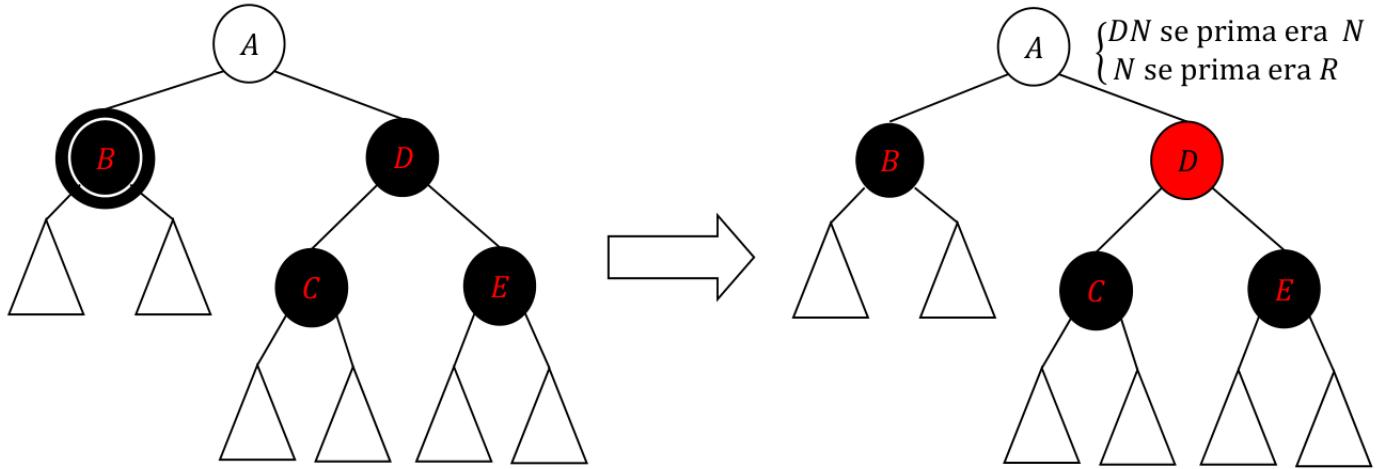
1 ViolazioneSx(S, D)
2   v = 0
3   IF S -> c = DN THEN
4     IF D -> c R THEN
5       v = 1
6     ELSE IF (D ->dx) -> c = N AND (D ->sx) -> c = N THEN
7       v = 2
  
```

```

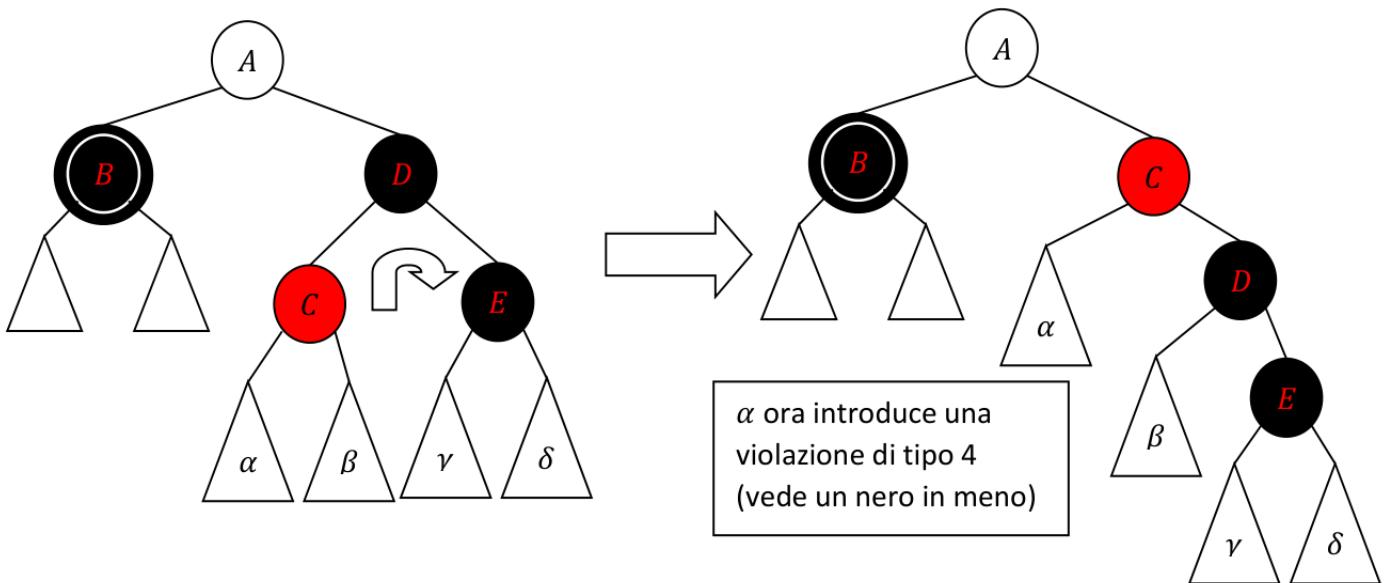
8     ELSE IF (D -> dx) -> c = N THEN
9         v = 3
10    ELSE // (D -> dx) -> c = R
11        v = 4
12    RETURN v

```

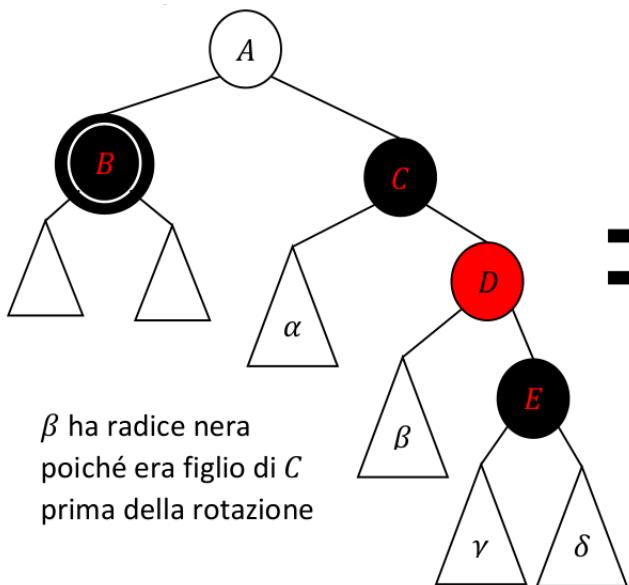
Analizziamo la risoluzione dei vari casi, partendo dal caso 2 (quello più semplice):



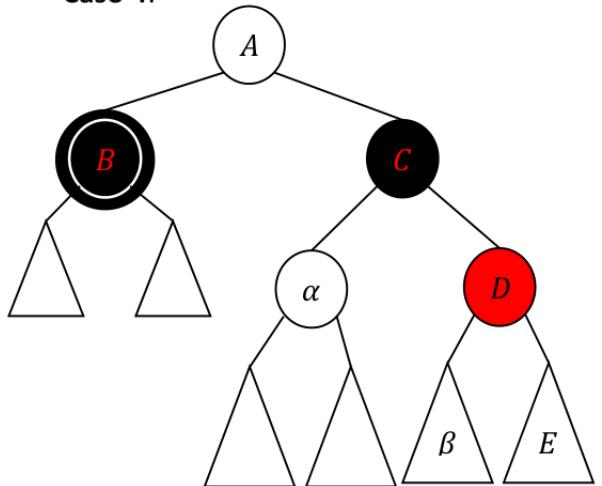
Per il caso 3 applichiamo una rotazione e un cambio di colore per portarci al caso 4:



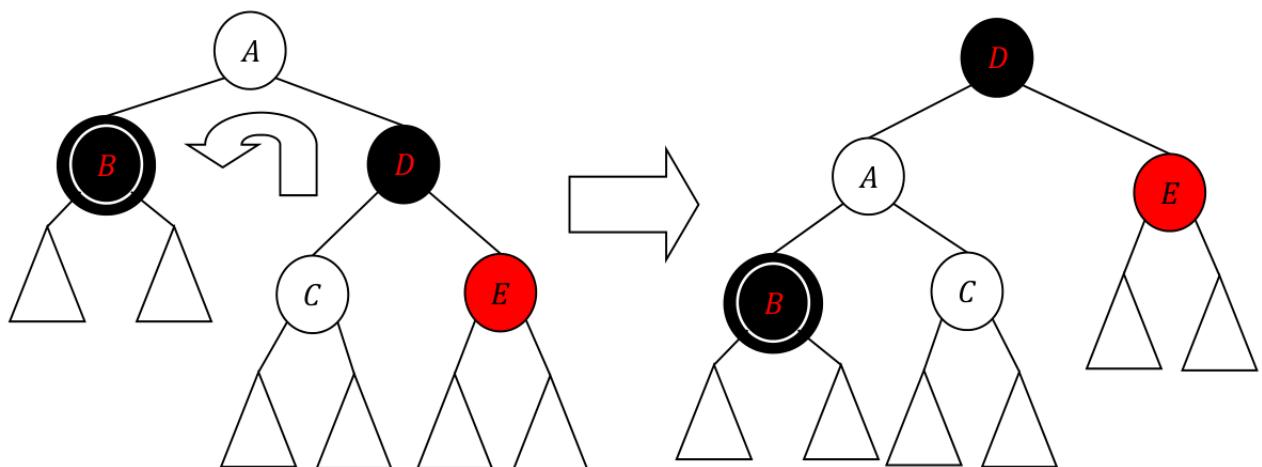
Risolviamo quindi la violazione ricolorendo i nodi:



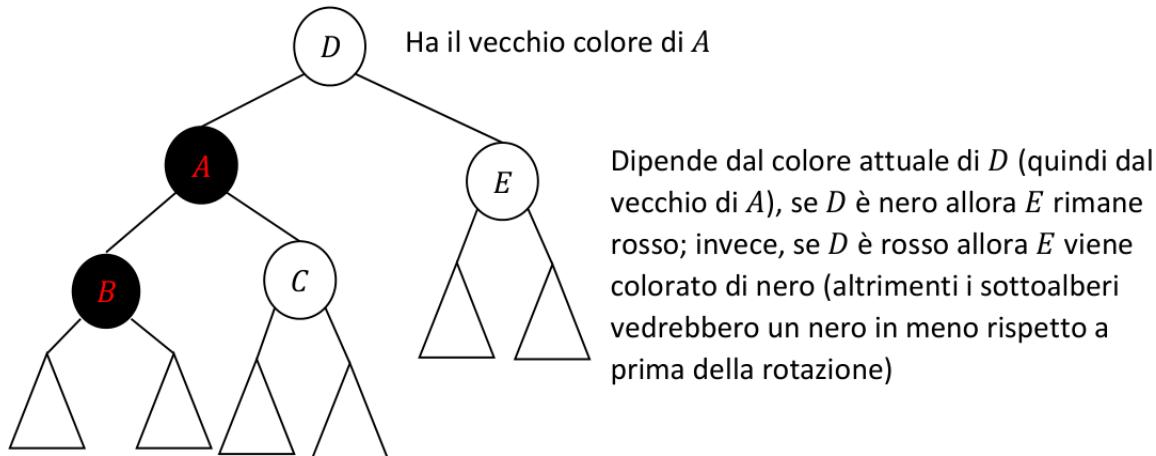
Caso 4:



Il caso 4 (che completa anche il caso 3) è risolvibile attraverso una rotazione destra ed un cambio di colore:



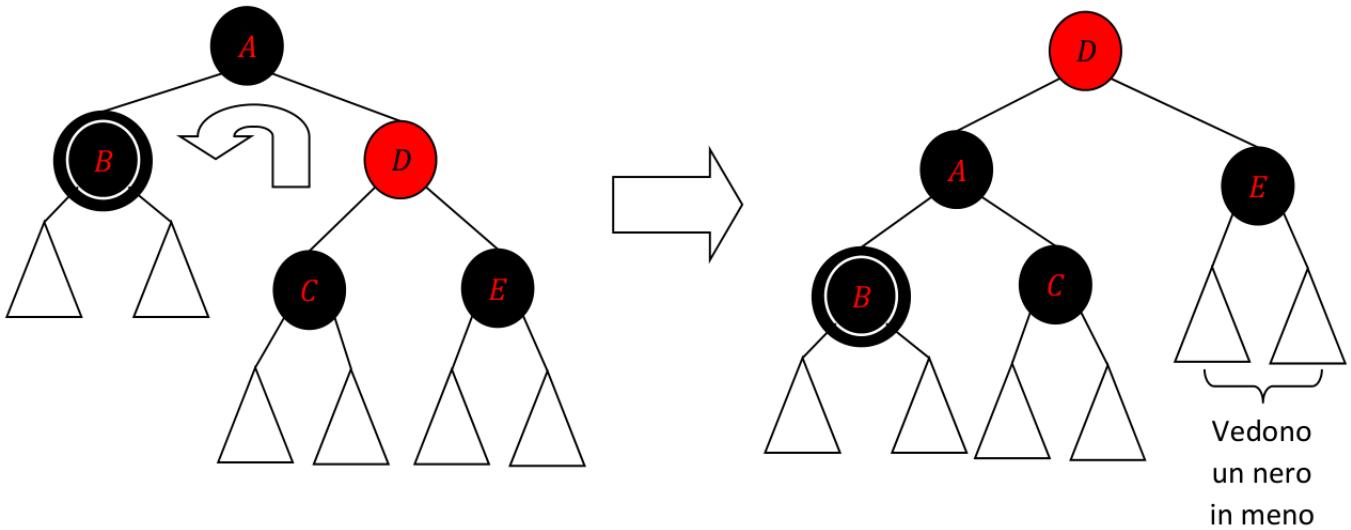
Dopo la rotazione il nodo B vede un nero in più, mentre i sottoalberi di E potrebbero vedere un nero in meno (dipende dal colore di A). Il vincolo 4 è risolvibile semplicemente ruotando i colori nel verso opposto alla rotazione effettuata (tenendo un nero in B)



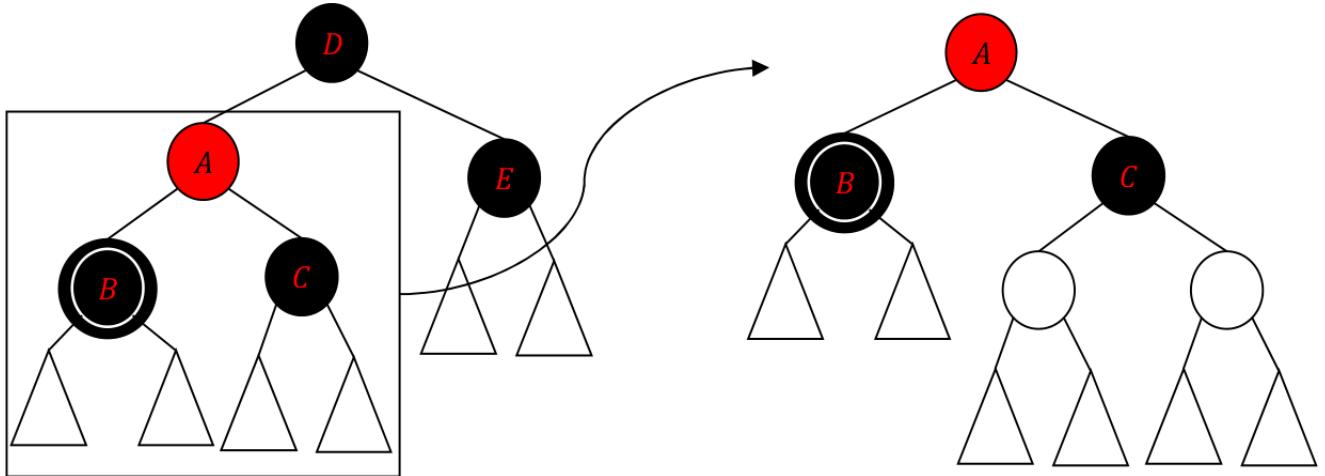
Una volta risolta la violazione del vincolo 4 (le altezze nere sono le stesse di quelle prima della rotazione) quindi si risolve del tutto anche la violazione di tipo 3 (senza introdurre nuove violazioni).

Il caso 3 e 4 sono risolutivi (non propagano la violazione come nel caso 2)

Resta ora da dimostrare solo il caso 1, che è quello con l'idea più contro intuitiva perché spinge la propagazione verso il basso:



A questo punto per risolvere la violazione del sottoalbero destro basta colorare la radice D in nero; otteniamo però una violazione nel sottoalbero sinistro (vedrebbe un nero in più) → Questa è risolvibile facilmente colorando il nodo A di rosso:



Ma quindi posso applicare **BilanciaCancSx(A)** per risolvere il problema (possiamo farlo perché ci troviamo nel caso 2/3/4, ma mai nel caso 1 e quindi non ho rischio di loop).

Inoltre potendoci trovare nel caso 2 con radice rossa abbiamo la certezza che la violazione venga risolta e non propagata; a questo si aggiungono i casi 3 e 4 (che ricordiamo essere risolutivi). Il caso 1 risolve in maniera totale la violazione con 2 bilanciamenti.

Di seguito riportiamo quindi il nostro algoritmo, basandoci sul fatto che sicuramente T non è un nodo **NIL**.

```

1 BilanciaCancSx(T)
2   v = ViolazioneSx(T -> sx, T -> dx)
3   CASE v OF
4     1: T = Caso1sx(T)
5       T -> sx = BilanciaCancSx(T -> sx)
6     2: T = Caso2sx(T)
7     3: T = Caso3sx(T)
8     4: T = Caso4sx(T)
9   RETURN T

```

Al termine di questo algoritmo l'unica violazione possibile è quella che la radice T sia di colore doppio nero (il seguito del caso 2 su radice nera); in caso contrario abbiamo risolto nell'albero totale

```

1 Caso1Sx(T)
2   T = RotazioneSingolaDx(T)
3   T -> c = N
4   (T -> sx) -> c = R
5   RETURN T

```

```

1 Caso2Sx(T)
2   (T -> dx) -> c = R
3   (T -> sx) -> c = N
4   PropagaNero(T -> sx, T)
5   RETURN T

```

```

1 Caso3Sx(T)
2   T -> dx = RotazioneSingolaSx(T)
3   (T -> dx) -> c = N
4   ((T -> dx) -> dx) -> c = R
5   T = Caso4sx(T)
6
7   RETURN T

```

```

1 Caso4Sx(T)
2   T = RotazioneSingolaSx(T)
3   (T -> dx) -> c = T -> c
4   T -> c = (T -> sx) -> c
5   (T -> sx) -> c = N
6   ((T -> sx) -> sx) -> c = N
7
8   RETURN T

```

È interessante notare che al fronte di una cancellazione il numero massimo di rotazione è 3 (caso 1 seguito da caso 3) → Pertanto i RB non solo sono più tolleranti, ma anche meno complessi (se si considera una rotazione come misura di complessità).

24 Lezione 27

24.1 Conversione Algoritmo Ricorsivo in Iterativo

24.1.1 Introduzione

La ricorsione è un meccanismo astratto (assente nel calcolatore). Di conseguenza, l'unico modo che ha un calcolatore di effettuare la ricorsione è iterando (ricordiamo che anche questa è assente nel linguaggio macchina, ma che può essere simulata attraverso *salti* e altre istruzioni).

Esistono poi casi in cui la ricorsione non utilizza memoria aggiuntiva (*ricorsione in coda*) → Una traduzione iterativa di questo tipo di ricorsione non necessita di uno stack.

In tutti gli altri casi è sempre necessario utilizzare uno stack in cui memorizzare delle variabili per simulare lo stack di attivazione delle chiamate di ricerca.

24.1.2 Memoria nella ricorsione

A patto di avere la memoria necessaria è sempre possibile trasformare un algoritmo ricorsivo in uno iterativo con la stessa potenza computazionale.

Sappiamo che durante una chiamata ricorsiva vengono memorizzate delle informazioni nello stack di attivazione, le informazioni principali riguardano le variabili usate dall'algoritmo ricorsivo.

Sia la ricorsione che l'iterazione non fanno altro che ripetere blocchi di codice; la differenza sostanziale sta nel fatto che l'iterazione condivide le variabili → ogni ripetizione usa e vede le stesse variabile (una modifica di una variabile in una ripetizione si ripercuote nella successiva).

Questo non accade nella ricorsione, dove ogni chiamata avrà una sua copia delle variabili in aree di memoria scollegate tra loro (al termine della chiamata l'area di memoria utilizzata viene ripulita → le modifiche in una chiamata ricorsiva non vengono viste dalle altre).

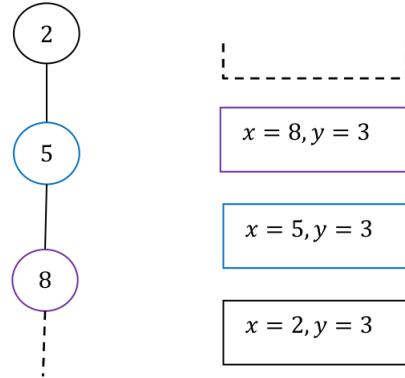
```

1   Ric(x)
2       y = 3
3       Ric(x + y)

```

Per la chiamata $Ric(2)$ si ha l'albero di ricorsione a destra.
Anche se non si nota ogni chiamata ricorsiva ha una sua variabile y .

Anche se hanno tutte lo stesso valore sono tutte variabili allocate in aree di memoria differenti.



È proprio questo meccanismo che dobbiamo tradurre in algoritmo iterativo (ciò implica memoria per ogni chiamata ricorsiva).

24.2 Algoritmo iterativo del meccanismo di ricorsione

Il meccanismo da implementare è il passaggio da una chiamata ricorsiva all'altra.

Sappiamo che una chiamata ricorsiva ha le proprie copie di variabili; pertanto prima di poter passare alla successiva chiamata ricorsiva deve salvare il suo contesto per poterlo riprendere successivamente (per salvare lo stato di una chiamata ricorsiva basta memorizzare le variabili che vengono usate prima della chiamata successiva).

Quando la chiamata figlia termina libera la memoria occupata e riprendo il controllo della chiamata precedente. Proprio perché chi riprendere prima il controllo è l'ultima chiamata ad essere effettuata è evidente che ci troviamo in una tipologia LIFO³⁶, pertanto per simulare questo meccanismo nell'iterazione è necessario fare affidamento della struttura dati stack.

Il numero di ripetizioni del blocco di codice in una chiamata ricorsiva dipende dall'input, perciò non possiamo sapere quante iterazioni deve fare un determinato algoritmo; ciò implica che non è possibile un ciclo `for`, ma bisogna implementare un ciclo `while`.

24.3 Struttura dell'algoritmo

Una chiamata ricorsiva o è nel suo stato iniziale (deve cominciare a effettuare il suo blocco di istruzioni) o deve riprendere l'esecuzione dal punto in cui è stata sospesa. Ciò significa che il blocco `while` sarà suddiviso in 2 parti principali e la sua condizione iniziale sarà definita dalle seguenti variabili:

- ▶ `Start` → flag booleano che indica se dobbiamo iniziare una nuova chiamata ricorsiva;
- ▶ ($st \neq \emptyset$) → Questo valore booleano determinerà la presenza o meno di chiamate ricorsive da riprendere.

Le suddette variabili andranno opportunamente gestite nel blocco `while`, il quale terminerà la sua esecuzione se non sono presenti chiamate ricorsive da cominciare (`Start` settato a false) o riprendere (stack vuoto).

³⁶ Abbreviazione di **last in, first out**. Metodo per gestire le strutture dati dove il primo elemento viene processato per ultimo, mentre l'ultimo viene processato per primo.

Il blocco while sarà così strutturato (in seguito vedremo come andranno inizializzate le variabili):

```
1 WHILE start OR st != NIL DO
2   IF start THEN
3     // Simulo l'inizio della nuova chiamata ricorsiva
4   ELSE
5     // Ripristino il contesto della vecchia chiamata
6     // prendendolo dallo stack e riprendo l'esecuzione
```

La complessità della ripresa di una chiamata ricorsiva sta nel comprendere in quale punto riprendere l'esecuzione, perché se un algoritmo fa diverse chiamate ricorsive vuol dire che ho più punti in cui le chiamate vengano sospese e altrettanti punti in cui queste vengano riprese

24.4 Esempi di traduzione

24.4.1 PrintTree

Partiamo con un algoritmo che prende in input un albero binario e ne stampa i nodi in post order. La sua implementazione ricorsiva è la seguente:

```
1 PrintTree(T)
2   IF T != NIL THEN
3     PrintTree(T -> sx)
4     PrintTree(T -> dx)
5     PrintTree(T -> key)
```

Ogni chiamata ricorsiva avrà una propria copia di T , ciò significa che serve una variabile da poter modificare con il valore del parametro di input della chiamata ricorsiva così da non perdere il riferimento all'albero T .

```
1 PrintTreeIter(T)
2   cT = T // current T
3   st = NIL // Simula lo stack di attivazione
4   start = true // Variabile che rende vera la condizione del ciclo (inizializza il ciclo)
5   last = NIL // Si occupa di capire la chiamata da riprendere
6   WHILE (start OR st != NIL) DO
7     IF start THEN
8       // Iniziamo con la prima istruzione
9       IF cT != NIL THEN
10         // Dobbiamo sospendere l'attuale chiamata ricorsiva
11         st = Push(st, cT) // Salvo il contesto
12         cT = cT -> sx // Simulo il passaggio dei parametri
13         start = true // Devo cominciare una nuova chiamata ricorsiva
14     ELSE
15       start = false // Non devo cominciare una nuova chiamata ricorsiva
16       last = cT // Memorizzo l'input dell'attuale chiamata ricorsiva in modo da capire quale chiamata ricorsiva vada ripresa
```

```

17     ELSE // ripristino il contesto
18         cT = Top(st)
19         /*
20             Per sapere se mi trovi nella prima o seconda chiamata ricorsiva controllo l'input
21             della chiamata figlia, perchè se T è foglia è possibile che sia
22             sx che dx siano NIL (quindi devo differenziare)
23         */
24         IF last != ct -> dx AND ct -> dx != NIL
25             // Simulo la seconda chiamata
26             cT = cT -> dx
27             start = true // Ridondante come la linea 13
28         ELSE IF last != ct -> dx THEN
29             /*
30                 In questo punto dovrei simulare la chiamata ricorsiva
31                 PrintTree(T -> dx = NIL) ma in questo algoritmo non serve
32                 perchè non devo fare nulla (cosa non sempre vera per tutti gli algoritmi)
33             */
34         ELSE
35             Print(cT -> key)
36             // Simulo la terminazione
37             last = cT
38             start = false
39             Pop(st)

```

24.4.2 Altezza

Di seguito riportiamo un algoritmo ricorsivo che prende in input un albero e ne restituisce l'altezza

```

1 Altezza(T)
2     IF T != NIL THE
3         sx = Altezza(T -> sx)
4         dx = Altezza(T -> dx)
5         h = 1 + max(sx, dx)
6         RETURN h
7     RETURN -1

```

Di seguito la sua versione iterativa

```

1 AltezzaIter(T)
2     cT = T
3     st = NIL
4     st_sx = NIL
5     ret = -1 // In questo modo viene coperto anche il caso Altezza(NIL)
6     start = true
7     WHILE (start OR st != NIL) DO

```

```

8     IF start THEN
9         IF cT != NIL THEN
10            st = push(st, cT)
11            cT = cT -> sx
12        ELSE // Caso base
13            ret = -1
14            start = false
15            last = cT
16        ELSE
17            cT = Top(st)
18        IF last != cT -> dx AND cT -> dx != NIL THEN
19            sx = ret
20            st_sx = Push(st_sx, sx)
21            cT = cT -> dx
22            start = true
23        ELSE IF last != cT -> dx THEN
24            // Sto tornando dalla prima e devo assegnare il risultato a sx
25            sx = ret
26            // Simulo la chiamata Altezza(T -> dx = NIL)
27            dx = - 1
28            // Procedo con le altre istruzioni
29            h = 1 + max(sx, dx)
30            ret = h
31            // Termino la chiamata
32            last = cT
33            start = false
34            Pop(st)
35        ELSE
36            dx = ret
37            sx = Top(st_sx)
38            h = 1 + max(sx, dx)
39            ret = h
40            last = cT
41            start = false
42            Pop(st)
43            Pop(st_sx) // Solo in questo caso devo liberare la memoria di sx corrente
44    RETURN ret

```

È interessante notare che le variabili che devo salvare durante la sospensione di una chiamata sono solo e soltanto quelle che vengono lette durante la chiamata successiva (per questo motivo abbiamo utilizzato **st_sx**)

25 Lezione 28

25.1 QuickSort

```
1 QuickSort(A, p, r)
2   IF p < r THEN
3     // Partiziona è un algoritmo ricorsivo (possiamo usarlo quindi come chiamata a funzione)
4     // Se non lo fosse stato saremmo stati costretti a tradurlo in ricorsivo per poterlo usare
5     q = Partiziona(A, p, r)
6     QuickSort(A, p, q)
7     QuickSort(A, q + 1, r)
```

Sullo stack metto tutti quei valori definiti prima di una chiamata ricorsiva e letti dalla chiamata successiva (se è solo scritta prima o letta dopo non è necessario lo stack³⁷)

Nel nostro caso le uniche variabili necessarie da memorizzare su uno stack sono q ed r ; infatti p viene perso dopo la prima chiamata ricorsiva (non viene utilizzato nella seconda).

Resta da scegliere un parametro che ci consente di riconoscere da quale chiamata ricorsiva siamo tornati → Gli unici parametri che ci permettono di fare ciò sono p ed r , avendo la certezza che $p \leq q < r$.

Tra questi useremo r (in quanto p viene perso).

Se gli r tra una chiamata ricorsiva figlia e una padre sono diversi allora è sicuramente terminata la prima chiamata, mentre se sono uguali siamo alla fine della seconda.

Ricapitolando, avremo nello stack q ed r , mentre come *last* usiamo il terzo parametro.

```
1 QuickSortIter(A, p, r)
2   cp = p
3   cr = r
4   st_r = st_q = NIL
5   last = -1 // ci serve un valore non valido e sappiamo che sia p che q sono maggiori di 0
6   start = true
7   /*
8     Questo controllo è sufficiente perchè i 2 stack sono sincronizzati (uno vale l'altro)
9     Ciò ovviamente non è detto che valga per qualsiasi algoritmo; potrei avere infatti degli
10    stack che si riempiono in maniera diversa
11    NB Ci sarà sempre uno stack che si riempie in base alle chiamate ricorsive
12   */
13  WHILE (start OR st_r != NIL) DO
14    IF start THEN
15      IF q < cr THEN
16        q = Partiziona(A, cp, cr)
```

³⁷ Utilizzare stack superflui è un inutile spreco di spazio che influenza negativamente su un eventuale esercizio d'esame.

```

17     st_q = Push(st_q, q)
18     st_r = Push(st_r, cr)
19     cr = q
20 ELSE
21     last = cr
22     // Forziamo la risalita
23     start = false
24 ELSE // ripristiniamo il contesto (abbiamo perso cp)
25     cr = Top(st_r)
26     q = Top(st_q)
27     // Riconosco in quale punto ci troviamo
28 IF cr != last THEN
29     cp = q + 1
30     // cr non varia
31     start = true
32 ELSE // Sono tornato dalla seconda chiamata
33     // Aggiorno last e pulisco lo stack
34     last = cr // Istruzione obsoleta
35     start = true // Istruzione ridondante
36     st_q = Pop(st_q)
37     st_r = Pop(st_r)

```

```

1 Count(A, p, r, k)
2     x = 0
3     IF p <= r THEN
4         q = (p + r) / 2
5         IF A[q] = k THEN
6             x = 1
7             x = x + count(A, p, q - 1, k)
8             x = x + count(A, q + 1, r, k)
9     RETURN x

```

Questo algoritmo conta il numero di k presenti in un array A ma in realtà a noi non interessa cosa faccia → La traduzione in iterativo trascende il funzionamento in sé (è più legato alla sintassi del linguaggio).

Lo scopo è quello di simulare l'algoritmo ricorsivo e non creare un algoritmo iterativo che faccia la medesima cosa (anche se risulta più efficiente l'esercizio verrà considerato sbagliato).

Prestiamo attenzione al nostro algoritmo:

- ▶ $q < r$ sicuramente; di conseguenza anche $q - 1 < r$ e pertanto possiamo usare r come $last$;
- ▶ Analogamente a *QuickSort*, abbiamo bisogno di uno stack per r e uno per q ;
- ▶ Anche x viene scritto e letto in entrambe le chiamate (bisogna tenerne traccia utilizzando uno stack); Inoltre dobbiamo tener conto anche del fatto che x cambia nel tempo (viene assegnato nella riga 6, letto dalla prima chiamata ricorsiva e

riassegnato dopo la stessa, infine viene riassegnato anche nella riga 8 dopo la seconda chiamata ricorsiva).

```
1 CountIter(A, p, r, k)
2   cp = p
3   cr = r
4   st_r = st_q = st_x = NIL
5   start = true
6   WHILE (start OR st_r != NIL) DO
7     IF start THEN
8       x = 0
9       IF cp < cr THEN
10         q = (cp + cr) / 2
11         IF A[q] = k THEN
12           x = 1
13           // In questo punto perdere i valori dell'attuale chiamata ricorsiva
14           st_r = Push(st_r, cr)
15           st_q = Push(st_q, q)
16           st_x = Push(st_x, x)
17           cr = q - 1 // L'input cambia nella chiamata successiva
18     ELSE
19       ret = x
20       last = cr
21       start = false
22   ELSE
23     // Ripristino il contesto a prescindere da quale chiamata torno le variabili da
24     // ripristinare
25     // (sono le stesse. NB: Non è detto che valga per altri algoritmi)
26     cr = Top(st_r)
27     q = Top(st_q)
28     x = Top(st_x)
29     IF last != cr THEN
30       x = x + ret // Simula l'istruzione di RETURN
31       // In st_x c'è un valore obsoleto e non quello aggiornato
32       st_x = Pop(st_x)
33       st_x = Push(st_x, x)
34       // Assegno gli input che cambiano e comincio la nuova chiamata
35       cp = q + 1
36       start = true
37   ELSE // Torno dalla seconda chiamata
38     x = x + ret
39     ret = x
40     last = cr
41     start = false
42     st_r = Pop(st_r)
43     st_q = Pop(st_q)
44     st_x = Pop(st_x)
```

```
RETURN x
```

È interessante notare che, come valore di ritorno avremmo potuto mettere anche *ret* perché al termine del while avranno lo stesso valore (tuttavia conviene sempre separare il valore di ritorno della chiamata principale con i *return* tra chiamata padre e figlia onde evitare possibili errori).

25.2 Algoritmo sadico

```

1  Algo(A, p, r, L)
2      x = L
3      IF p <= r THEN
4          q = (p + r) / 2
5          L' = AllocaNodo()
6          L' -> key = A[q]
7          IF A[q] % 2 = 0 THEN
8              L' -> next = Algo(A, q + 1, r, L)
9              x = Algo(A, p, q - 1, r, L')
10         ELSE
11             L' -> next = Algo(A, q - 1, r, L)
12             x = Algo(A, q + 1, r, L')
13
RETURN x

```

È più complesso discriminare le chiamate ricorsive; fortunatamente le copie vengono suddivise dalla condizione $A[q] \% 2 = 0$, che insieme a *last* ci danno tutte le informazioni per poter disambiguare le 4 chiamate ricorsive.

Ciò è possibile solo perché l'array *A* non viene modificato durante l'algoritmo e quindi abbiamo la certezza che $A[q]$ abbia lo stesso valore sia per una chiamate che per l'altra della stessa coppia (in altri algoritmi dove *A* cambia basta usare uno stack dove salvare *A* per risolvere l'ambiguità, in altri ancora magari basta usare semplicemente più *last*).

```

1  AlgoITer(A, p, r, L)
2      cL = L
3      cp = p
4      cr = r
5      st_r = st_p = st_L' = NIL
6      start = true

```

L non viene letto dopo una chiamata ricorsiva e quindi *st_L* non serve, lo stesso vale per *x* che viene assegnato in tutte le chiamate ricorsive. Inoltre, visto che abbiamo bisogno sia di *st_r* che di *st_p* possiamo risparmiarci *st_q*; infatti basta calcolare per ogni chiamata *q* corrente tramite l'operazione scritta nella linea 4.

```

1  WHILE (start OR st_r != NIL) DO
2      IF start THEN
3          x = L
4          IF cp <= cr THEN

```

```

5   q = (cp + cr) / 2
6   L' = AllocaNodo()
7   L' -> key = A[q]
8   // Devo distinguere quale coppia far partire (nello stack metto però le stesse cose)
9   st_r = Push(st_r, cr)
10  st_p = Push(st_p, cp)
11  st_L' = Push(st_L', L')
12  IF A[q] % 2 = 0 THEN
13      cp = q + 1
14  ELSE
15      cp = q - 1
16  ELSE
17      ret = x
18      start = false
19      last = cr
20 ELSE
21     cp = Top(st_p)
22     cr = Top(st_r)
23     L' = Top(st_L')
24     q = (cp + cr) / 2
25     IF A[q] % 2 = 0 THEN
26         IF last != cr THEN
27             x = ret
28             // ret = x
29             last = cr
30             start = false
31             st_p = Pop(st_p)
32             st_r = Pop(st_r)
33             st_L' = Pop(st_L')
34     ELSE // È terminata la prima
35         L' -> next = ret
36         // Cambiano i parametri 3 e 4
37         cr = q - 1
38         cL = L'
39         start = true
40 ELSE // Siamo nella seconda coppia
41     IF last != cr THEN
42         L' -> next = ret
43         // Cambiano i parametri 2 e 4
44         cp = q + 1
45         cL = L'
46         start = true
47     ELSE // È terminata la seconda
48         x = ret
49         // ret = x
50         last = cr

```

```

51     start = false
52     st_p = Pop(st_p)
53     st_r = Pop(st_r)
54     st_L' = Pop(st_L')
55 RETURN x

```

NB: Se nell'assurdo caso ci si ritrovi a dover mettere un RETURN all'interno del while stiamo sicuramente sbagliando

26 Lezione 29

26.1 Esercizio

Gli esercizi di questa tipologia sono facilmente generabili (basta creare un algoritmo ricorsivo a caso e tradurlo). Di seguito ne proponiamo un esempio

```

1 Esercizio(T, k1, k2, P)
2   x = 0
3   IF T != NIL THEN
4     IF T -> key < k1 THEN
5       x = Algo(T -> dx, k1, k2, T)
6     ELSE IF T -> key > k2 THEN
7       x = Algo(T -> sx, k1, k2, T)
8     ELSE
9       x = Algo(T -> dx, k1, k2, T)
10    x = x + Algo(T -> sx, k1, k2, T)
11    IF P != NIL THEN
12      IF T = P -> sx THEN
13        P -> sx = cancellaRoot(T)
14      ELSE
15        P -> dx = cancellaRoot(T)
16      x = x + 1
17 RETURN x

```

Suggerimenti: Nello stack dobbiamo salvare P e T ed anche x , perché dopo la riga 9 ho bisogno di salvarlo prima di cominciare la seconda chiamata ricorsiva (ovviamente non ho bisogno di $k1$ e $k2$, perché non cambiano durante la loro esecuzione e pertanto uso direttamente $k1$ e $k2$).

Soluzione: Possiamo notare che dall'algoritmo ricorsivo che anche se P e T sono 2 variabili differenti, esse sono strettamente legate tra loro. Possiamo quindi utilizzare un unico stack per entrambe, infatti mettendo P nello stack prima di T posso raggiungere P semplicemente risalendo una volta in più nello stack (è più chiaro con il codice alla mano), perché ogni chiamata ricorsiva porta in sé sia il nodo figlio che il padre.

Il precedente ragionamento comporta però delle modifiche nella struttura che conosciamo → siamo obbligate a dover inizialmente (prima del ciclo `while`) mettere il parametro P già nello stack e quindi non possiamo utilizzare il controllo $st \neq \emptyset$ per uscire

dal **while**.

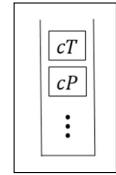
Fortunatamente abbiamo la certezza che P sia esattamente il proprio elemento inserito nello stack e determina in maniera univoca la prima chiamata ricorsiva (in tutte le altre avrà i suoi discendenti); dunque, se la cima dello stack contiene P dobbiamo terminare l'esecuzione del ciclo **while**.

```
1 AlgoIter(T, k1, k2, P)
2   cT = T
3   cP = P
4   st_x = NIL
5   st_T = push(st_T, P)
6   start = true
7   WHILE (start OR Top(st_T) != P) DO
8     IF start THEN
9       x = 0
10    IF T != NIL THEN
11      // Una chiamata ricorsiva la farà sicuramente
12      st_T = Push(st_T, T)
13      cP = cT // Il quarto parametro è sempre lo stesso in ogni chiamata
14      // Le righe 15-18 sono analoghe a
15      IF cT -> key < k1 THEN
16        cT = cT -> dx
17      ELSE IF cT -> key > k2 THEN
18        cT = cT -> sx
19      ELSE
20        cT = cT -> dx
21      ELSE
22        ret = x
23        start = false
24        last = cT
25      ELSE
26        cT = top(st_T)
27        // Se mi trovo in una chiamata singola devo effettuare la medesima cosa
28        IF cT -> key < k1 OR cT -> key > k2 THEN
29          x = ret
30          st_T = Pop(st_T)
31          // start = false
32          last = cT
33        ELSE
34          IF last != cT -> sx AND cT -> sx != NIL THEN
35            // Torno da destra (prima chiamata ricorsiva) e sinistra non è NIL
36            x = ret
```

```

35         st_x = Push(st_x, x)
36         cP = cT
37         cT = cT -> sx
38         start = true
39     ELSE IF last != ct -> sx THEN
40         // Torno da destra e il sinistro è NIL
41         x = ret // Basta per simulare il caso base, infatti x = x + 0
42     ELSE // In questo punto avrà last = cT -> sx ma questo potrebbe essere NIL
43         IF cT -> sx = NIL THEN
44             x = ret
45         ELSE
46             x = Top(st_x) + ret
47             st_x = Pop(st_x)
48             cT = Top(st_T)
49             // Dovrà in ogni caso ripulire st_T

```



```

50         st_T = Pop(st_T) // Alla riga 48 avrà il seguente stato:
51         cP = Top(st_T)
52         IF cP != NIL THEN
53             IF cT = cP -> sx THEN
54                 cP -> sx = cancellaRoot(cT)
55             ELSE
56                 cP -> dx = cancellaRoot(cT)
57                 ret = x
58                 // start = false
59         st_T = Pop(st_T) // In seguito al while abbiamo ancora P in st_T
60         RETURN x

```

26.2 Ricorsione in coda

Abbiamo già accennato ad algoritmi che ricorsivi per cui esiste una versione iterativa che non fa uso di stack

Questo tipo di algoritmi sono possibili solo nei casi in cui si conosce esattamente la direzione da seguire → Ovvero presentano un'unica chiamata ricorsiva da dover fare.

Un esempio sono gli algoritmi di inserimento e cancellazione su alberi binari di ricerca poiché si ha un unico percorso da dover seguire (non funzionerebbe quindi un generico albero binario dove non si hanno proprietà di ordinamento tale tra gli elementi).

Descriviamo un algoritmo per un ABR T che cancella un nodo con chiave k in maniera iterativa e senza uso di stack.

La soluzione è banalmente quella di usare una variabile dove memorizzare il padre del nodo da cancellare, che aggiorneremo durante la discesa della ricerca binaria:

```

1 CancellaIter(T, k)
2   cT = T
3   cP = NIL
4   WHILE (cT != NIL AND cT -> key != k) DO
5     cP = cT
6     IF cT -> key < k THEN
7       cT = cT -> dx
8     ELSE
9       cT = cT -> sx
10    IF cT != NIL THEN
11      IF cP != NIL THEN
12        IF cT = cP -> sx THEN
13          cP -> sx = cancellaRoot(cT)
14        ELSE
15          cP -> dx = cancellaRoot(cT)
16    ELSE // Dobbiamo cancellare la radice
17      T = cancellaRoot(T)
18      // cT = T
19  RETURN T

```

L'algoritmo ricorsivo che più si avvicina all'algoritmo precedente è il seguente:

```

1 CancellaRC(T, k, p)
2   IF T != NIL THEN
3     IF T -> key > k THEN
4       CancellaRC(T -> sx, k, T)
5     ELSE IF T -> key < k THEN
6       CancellaRC(T -> dx, k, T)
7     ELSE // T contiene k
8       IF p != NIL THEN
9         IF T = p -> sx THEN
10           p -> sx = CancellaRoot(T)
11         ELSE
12           p -> dx = CancellaRoot(T)

```

Si noti che manca il caso in cui T è radice dell'albero, ma sarebbe sbagliato definirlo in quell'algoritmo poiché cambierebbe il puntatore di T senza che il chiamante possa accorgersene (e quest'ultimo andrebbe poi ad utilizzare la variabile T senza conoscerne il reale contenuto).

In questo caso l'algoritmo predente andrebbe usato dal seguente algoritmo

```

1 Cancella(T, k)
2   IF T -> key != k THEN
3     CancellaRC(T, k, NIL)
4   ELSE

```

```

5   T = cancellaRoot(T)
6   RETURN T

```

In ogni caso, ritornando a **CancellaRC** si può notare che ogni chiamata ricorsiva viene effettuata prima del **RETURN**, tale priorità è proprio quella che deve avere ogni algoritmo per essere definito ricorsivo in coda; oltre a quello di avere al più una sola chiamata ricorsiva per ogni chiamata.

Se non ci sono altre istruzioni di lettura dopo una chiamata ricorsiva (non bisogna rispondere il contesto della chiamata precedente) e c'è al più una chiamata ricorsiva allora l'algoritmo è detto ricorsivo in coda; dunque, non ha bisogno di stack per gestire la sua esecuzione.

NB: Se invece di cancellare soltanto il nodo volessimo restituire la profondità in cui si trovato il nodo cancellato l'algoritmo ricorsivo sarà il seguente:

```

1 Canc(T, k, p)
2   x = 0
3   IF T != NIL THEN
4     IF T -> key > k THEN
5       x = 1 + Canc(T -> sx, k, T)
6     ELSE IF T -> key < k THEN
7       x = 1 + Canc(T -> dx, k, T)
8     ELSE
9       IF p != NIL THEN
10         IF T = p -> sx THEN
11           p -> sx = CancellaRoot(T)
12         ELSE
13           p -> dx = CancellaRoot(T)
14
RETURN x

```

Questo è ben lontano dall'essere ricorsivo in coda. Visto che segue un unico percorso è possibile renderlo ricorsivo in coda semplicemente aggiungendo un nuovo parametro che tenga traccia del livello in cui si trova tale nodo, tale parametro andrà banalmente incrementato ad ogni chiamata:

```

1 Canc(T, k, p, 1)
2   IF T != NIL THEN
3     IF T -> key > k THEN
4       RETURN Canc(T -> sx, k, T, 1 + 1)
5     ELSE IF T -> key < k THEN
6       RETURN Canc(T -> dx, k, T, 1 + 1)
7     ELSE
8       IF p != NIL THEN
9         IF T = p -> sx THEN
10           p -> sx = CancellaRoot(T)
11         ELSE
12           p -> dx = CancellaRoot(T)

```

```
RETURN 1
RETURN -1
```

27 Lezione 30

27.1 Grafi

27.1.1 Definizioni

I grafi sono uno strumento di rappresentazione (modellazione) di problemi molto importanti → È infatti la soluzione a molti problemi reali (possiamo ricondurli a opportuni problemi su grafi).

Matematicamente è definito come una coppia di insiemi $G = (V, E)$, dove:

- ▶ $V \rightarrow$ insieme di nodi chiamati *vertici*;
- ▶ $E \subseteq V \times V$; relazione binaria tra gli elementi di V (è praticamente un insieme di coppie di vertici, detto insieme degli *archi*).

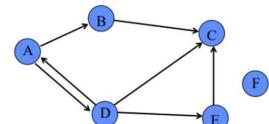
Sia $G = (V, E)$ grafo, allora l'insieme degli archi possibili sono $2^{|V|^2}$ combinazioni; infatti è possibile definire il collegamento tra 2 vertici con un singolo bit (1 se esiste un arco, 0 altrimenti).

27.1.2 Tipi di grafo

I grafi possono esprimere sia relazioni simmetriche (grafo non orientato), che antisimmetriche (grafo orientato).

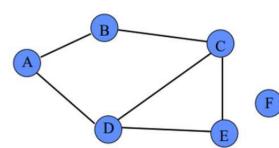
Un **grafo orientato** G è una coppia (V, E) dove:

- ▶ V è un insieme detto insieme dei vertici
 - ◊ $V = \{A, B, C, D, E, F\}$
- ▶ E è una relazione binaria tra vertici
 - ◊ $E = \{(A, B), (A, D), (B, C), (D, C), (E, C), (D, E), (D, A)\}$



Un **grafo non orientato** G è una coppia (V, E) dove:

- ▶ V è un insieme detto insieme dei vertici
 - ◊ $V = \{A, B, C, D, E, F\}$
- ▶ E è un insieme di coppie *non ordinate* di vertici ($(A, B) = (B, A)$)
 - ◊ $E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$



In linea di principio se la relazione è simmetrica posso usare sia un grafo non orientato, sia un grafo orientato; se la relazione è

antisimmetrica allora posso utilizzare solo grafi orientati.

Nonostante sia meno potente, il grafo non orientato è utile in ambito informatico poiché rende più semplice la soluzione di alcuni problemi.

In alcuni casi, gli archi hanno un **peso** (o costo) associato. Il costo può essere rappresentato da una funzione di costo $c : E \rightarrow \mathbb{R}$.

Se un arco ha un peso allora il grafo viene detto **pesato**. Quando tra 2 vertici non esiste un arco, si dice che il costo è infinito.

27.1.3 Grado di un vertice

Si noti che un albero può essere considerato un particolare tipo di grafo dove le relazioni sono molto più restrittive. Nonostante ciò la nozione di grado definita per un albero (un nodo ha grado n se ha n figli) vale anche per i grafi:

- ▶ In un grafo non orientato il grado di un vertice è il numero di archi che da esso si dipartono;
- ▶ In un grafo orientato differenziamo tra grado *entrante* e grado *uscente*
 - ◊ Il **grado entrante** di un vertice è il numero di archi **incidenti** in esso (l'arco (u, v) è indicente da w in v , ovvero la freccia parte da w e arriva in v);
 - ◊ Il **grado uscente** di un vertice è il numero di archi uscenti da esso;
- ▶ In un grafo orientato il grado di un vertice è la somma del suo grado entrante e del suo grado uscente.

27.1.4 Sottografo

Sia $G = (V, E)$ un grafo; un grafo $G' = (V', E')$ è sottografo di G se, e solo se:

$$V' \subseteq V \text{ e } E' \subseteq E \cap (V' \times V')$$

Abbiamo quindi che $E' \subseteq E$ e $E' \cap (V' \times V')$ contemporaneamente (Scrivere solo una delle 2 non definisce necessariamente un grafo).

Sia $G = (V, E)$ un grafo e $V' \subseteq V$ un insieme di vertici. Il sottografo di G **indotto** da V' è il più grande sottografo di G (ovvero $E' = E \cap (V' \times V')$)

Il concetto di sottografo è importante perché ci permette di decomporre il problema su grafi in sottoproblemi della stessa natura.

27.1.5 Percorso

Un **percorso** nel grafo è una sequenza di vertici $\langle v_0, v_1, \dots, v_k \rangle$ (non necessariamente il numero di vertici è finito) tale che:

$$(v_i, v_{i+1}) \in E, \forall 0 \leq i < k \text{ (dove } k = |V|)$$

Si noti che tale definizione rende il percorso più breve il cosiddetto percorso vuoto → un percorso composto da un singolo vertice.

Infatti $(v_i, v_{i+1}) \in E, \forall 0 \leq i \leq 0$ è banalmente vera (essendo il dominio di tale proprietà vuoto, non esiste nessun elemento che possa contraddirre tale proprietà universale).

Escludendo questo percorso, tutti gli altri devono contenere almeno un arco.

Un percorso si dice **semplice** se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza), eccetto al più il primo e l'ultimo che possono coincidere (ad es $v_0v_3v_2v_4v_0$ è semplice, $v_0v_1v_2v_3v_2v_6$ no).

La **lunghezza** di un percorso è pari al numero di vertici meno 1 (ovvero il numero di archi di un percorso).

La **distanza** tra 2 vertici è pari alla lunghezza del percorso minimo tra questi 2 vertici³⁸.

27.1.6 Raggiungibilità

Siano u, v vertici. Diremo che u è **raggiungibile** da v in G se, e solo se, esiste un percorso in G che parte da v e termina in u .

La relazione di raggiungibilità è transitiva (se x è raggiungibile da y e y è raggiungibile da z , allora x sarà raggiungibile da z) e riflessiva (basta pensare al percorso vuoto).

La verifica di raggiungibilità ci permette di risolvere molti problemi sui grafi. La relazione di raggiungibilità è **chiusura transitiva e riflessiva di E** . Sia R_E la chiusura transitiva e riflessiva di E ; essa è così definita:

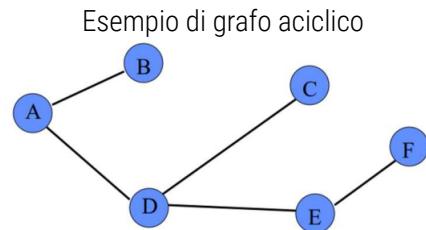
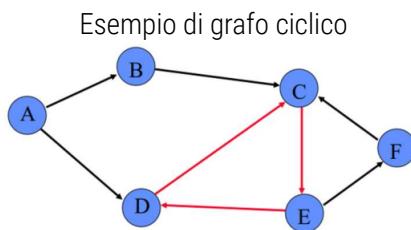
$$R_E = \underbrace{(v, v) \mid v \in V}_{\text{riflessività}} \cup \underbrace{(v, u) \mid \exists w \in V : (w, u) \in E \wedge (v, w) \in R_E}_{\text{transitività}}$$

NB: Questa è una definizione ricorsiva ben definita, con caso base il percorso vuoto.

27.1.7 Grafi ciclici e aciclici

Un **ciclo** in un grafo è un percorso di lunghezza almeno uno tale che inizi e termini con lo stesso vertice.

Un grafo senza cicli è detto **aciclico**; un grafo con uno o più cicli è detto **ciclico**



³⁸ Si noti che il concetto di livello per un albero non è altro che la distanza tra la radice ed un nodo situato in questo livello.

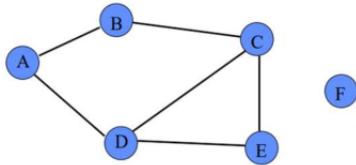
28 Lezione 31

28.1 Grafi connessi e grafi completi

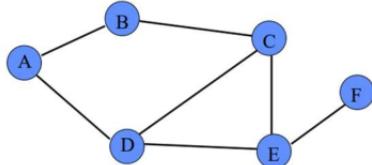
Se G è un grafo non orientato, diciamo che G è **connesso** se esiste un percorso da ogni vertice ad ogni altro vertice.

Se G è un grafo orientato, diciamo che G è **fortemente connesso** se esiste un percorso da ogni vertice ad ogni altro vertice.

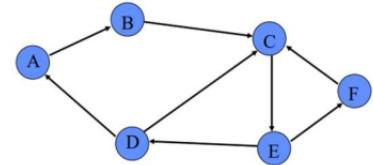
Non connesso (o **sconnesso**):



Connesso:

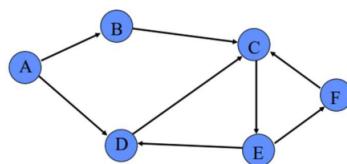


Fortemente connesso:



Se G è un grafo orientato non fortemente connesso, ma il grafo non orientato sottostante (cioè senza la direzione degli archi) si, allora diremo che G è **debolmente connesso**.

Il grafo a destra non è fortemente connesso (non esiste un percorso da D ad A), ma è debolmente connesso.



28.2 Altri concetti

Un **grafo completo** è un grafo che ha un arco tra ogni coppia di vertici.

Un **albero libero** è un grafo non orientato connesso, aciclico. Se un grafo non orientato è aciclico ma sconnesso, prende il nome di foresta (contiene più alberi).

28.3 Rappresentazioni concrete di grafi

Un grafo non ha una sorgente con cui poter esplorare tutta la struttura come la radice di un albero o la testa di una lista.

Ciò si traduce in:

- ▶ Rappresentazione a **matrice di adiacenza**;
- ▶ Rappresentazione a **liste di adiacenza**;

28.3.1 Matrice di adiacenza

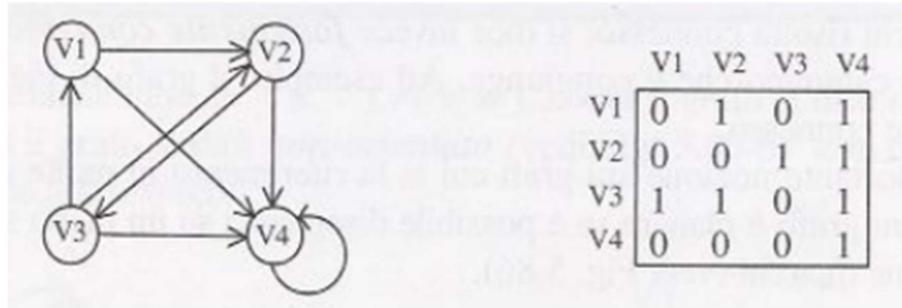
Una delle possibili implementazioni è quella di descrivere l'insieme E attraverso la funzione caratteristica (che restituisce 1 se l'elemento in input appartiene all'insieme, 0 altrimenti).

Dunque, essendo $E \subseteq V \times V$, la funzione caratteristica avrà dominio $V \times V$. Questo ragionamento vale solo per insiemi finiti, infatti dovendo generare tutte le combinazioni e associare a ognuna il valore 0 o 1 è evidente che siamo limitati dalla memoria del nostro calcolatore.

La rappresentazione più naturale è attraverso una matrice:

$$\begin{matrix} & 1 & \dots & |V| \\ \begin{matrix} 1 \\ \vdots \\ |V| \end{matrix} & \left[\begin{matrix} : & M(i,j) & : \\ \vdots & \dots & \vdots \end{matrix} \right] & M(i,j) = \begin{cases} (i,j) \mapsto 1 & \text{se } (i,j) \in E \\ (i,j) \mapsto 0 & \text{se } (i,j) \notin E \end{cases} \end{matrix}$$

Esempio:



Quindi ad ogni vertice viene associato un intero, anche se i vertici sono definiti con dei nomi simbolici, basta associare ad ogni nome un valore intero da 0 a $|V| - 1$ (se abbiamo bisogno dei nomi simbolici basta utilizzare un array dove gli indici rappresentano l'intero e nella cella il nome simbolico associato a quell'intero).

Questa è la rappresentazione più utilizzata nella teoria dei grafi ma non la più efficiente. Il motivo è abbastanza evidente → Questa rappresentazione necessita di uno spazio in memoria pari a $|V|^2$ a prescindere dal numero di archi presenti.

Idealmente, essendo un grafo una coppia di insieme ci aspettiamo che la dimensione per la rappresentazione sia $\Theta(|V|, |E|)$ e non $\Theta(|V|)$. Se ad esempio abbiamo il seguente grafo:

$$[v_1] \rightarrow [v_2] \rightarrow \dots \rightarrow [v_n]$$

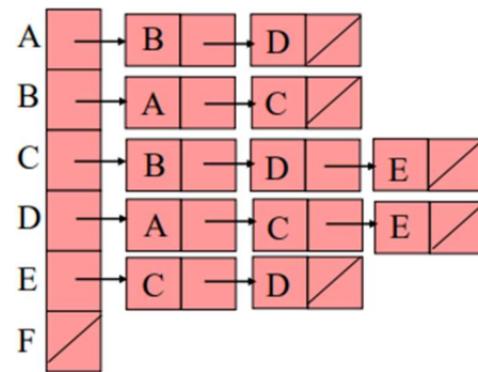
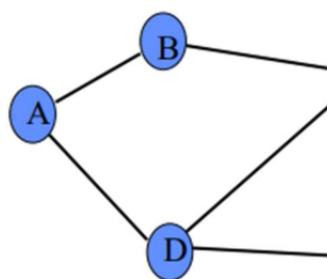
Dove $|V| = n$ e $|E| = n - 1$. Vogliamo rappresentare tale grafico in uno spazio in memoria pari a $2n - 1$ e non n^2 come avviene nella rappresentazione con matrice di adiacenza.

Questo spreco di memoria è dovuto al fatto che questo tipo di rappresentazione da informazioni su ogni elemento del dominio (archi esistenti compresi), ma ha comunque dei vantaggi sul tempo di esecuzione di alcuni algoritmi; infatti, aggiungere o rimuovere un arco è un'operazione costante, così come l'accesso ad un elemento (verificare se 2 vertici sono adiacenti). Tuttavia aggiungere un vertice è un'operazione molto costosa, poiché richiede l'allocazione di una nuova matrice e la copia degli elementi presenti nell'altra (tempo quadratico).

28.3.2 Liste di adiacenza

L'informazione più interessante è sapere quali coppie sono in E perché a noi interessa muoverci tra i vertici attraverso gli archi (tramite i percorsi). Una rappresentazione di questo tipo è quella a **liste di adiacenza** (che è molto più efficiente dal punto di vista della memoria).

A ogni vertice associa i vari archi uscenti, abbiamo una struttura dati che rappresenta l'insieme degli archi (usualmente un array, ma volendo anche una lista, con i vantaggi e svantaggi che ognuna delle 2 rappresentazioni comporta), nel quale ad ogni elemento è associata una struttura dati che rappresenta gli adiacenti di quel vertice (di solito è una lista ma è possibile usare un albero). Formalmente una lista di adiacenza $L(v)$ con $v \in V$ è una lista di w , tale che $(v, w) \in E$. Ad es.



Tale rappresentazione ha una occupazione in memoria di $\Theta(|V|, |E|)$ che è proprio il nostro obiettivo.

Scegliere questa modalità ovviamente comporta un aumento del tempo di esecuzione per alcune operazioni ma una riduzione per altri.

Aggiungere un arco (i, j) infatti impiega un tempo lineare sul numero di adiacenti del vertice v_i ; aggiungere un vertice avviene in tempo lineare $|V|$ e non più $|V|^2$

Tale rappresentazione è ottimale dal punto di vista della memoria, ma paga per le operazioni di **inserimento/cancellazione** di un arco.

In ogni caso tale rappresentazione è la più efficiente per un grafo generico (casi particolare possono avere rappresentazioni più efficienti che sfruttino le proprietà di quel particolare grafo).

Nei nostri algoritmi assumeremo sempre e solo una delle 2 rappresentazioni descritte, nella quale la lista di adiacenza sarà la più efficiente → Di nostro interesse è conoscere gli adiacenti di un vertice (nella matrice di adiacenza tale operazione impiega tempo lineare su $|V|$, poiché dobbiamo scorrere tutta la riga di quel vertice).

28.4 Visita in ampiezza

Il problema di raggiungibilità tra vertici si può risolvere tramite algoritmi che esplorano la topologia di un grafo (determinata dagli archi che sono nel grafo).

La visita in un grafo deve praticamente esplorare tutti i percorsi → Ciò rende necessario gestire i cicli di un grafo (altrimenti l'algoritmo non terminerebbe) e assicurarsi di visitare ogni vertice una sola volta (altrimenti avremo un tempo di esecuzione esponenziale, pari al numero di percorsi in un grafo, ovvero n^n !).

NB: Qualsiasi algoritmo che visiti tutti i percorsi semplici di un grafo avrà nel caso peggiore costo esponenziale, poiché il fattoriale è compreso tra 2^n e n^n ($2^n \leq n! \leq n^n$).

La visita in profondità è definita dal concetto di percorso, mentre quella in ampiezza dal concetto di livello.

Per un grado il livello è definito in maniera simile ad un albero → Infatti, mentre per un albero il livello è praticamente la distanza tra la radice e il nodo di quel livello, per un grafo un livello è la distanza (percorso più breve) tra un vertice e la sua sorgente (il vertice che abbiamo scelto come *radice*).

Per motivi che poi saranno chiari, assoceremo 3 stati ad ogni vertice:

- ▶ Vertici non ancora incontrati (*visti*), codificato con il colore *bianco*;
- ▶ Vertici *visti*, ma non ancora visitati , codificato con il colore grigio;
- ▶ Vertici già visitati , codificato con il colore nero;

Questo tipo di codifica è fondamentale per garantire al più una visita per ogni vertice.

28.5 Algoritmo BFS

Scriviamo un algoritmo di visita in ampiezza per grafi. Prima definiamo una funzione che inizializzi il colore di ogni vertice a bianco. Si noti che per codificare i 3 colori bastano 4 bit (ad es. $00 = b$, $01 = g$, $10 = n$); quindi basta un array per avere la relazione **vertice-colore** (i vertici saranno gli indici dell'array e nella cella avremo il colore *b,g* oppure *n*).

```
1 Init(G)
2   FOR EACH v IN V DO
3     Color[v] = b
```

La visita in ampiezza visiterà tutti i vertici raggiungibili da una sorgente S (l'algoritmo **Adj(v)** restituisce la lista di adiacenti di v)

```
1 VisitaAmpiezza(G, s)
2   Init(G)
3   Q = Accoda(Q, s)
4   Color[s] = g
5   // Aggiungo tutti gli adiacenti non ancora incontrati nella coda
6   // (Mettendo il colore a grigio prima mi assicuro che non accoderà mai lo stesso elemento più di
    // una volta)
```

```

7   WHILE Q != NIL DO
8       v = Testa(Q)
9       FOR EACH u in Adj(v) DO
10          IF Color[u] = b THEN
11              Color[u] = g
12              Q = Accoda(Q, u)
13      Visita(v)
14      Q = Decoda(Q)
15      Color[v] = n

```

Questo algoritmo garantisce sia la terminazione che un tempo asintotico lineare sugli adiacenti di s . Nel caso peggiore avremo un $O(|V|, |E|)$ poiché esploro i vertici al più una volta → Di conseguenza un arco o non viene mai esplorato o viene esplorato una volta sola. In particolare, è grazie all'array di colori che il tempo richiesto è lineare:

$$T_{BFS}(|V|, |E|) = O(|V|, |E|)$$

29 Lezione 32

29.1 Calcolo distanze e percorsi minimi

La *BFS* visita il grafo in ordine crescente di distanza da s , quindi sostanzialmente ogni vertice viene raggiunto tramite il percorso più corto (ovvero quello che determina la distanza); dunque, l'idea sottostante è che questo algoritmo segue i percorsi minimi.

Dunque, qualsiasi problema in cui si debba calcolare le distanze (e di conseguenza i percorsi minimi) dei vertici di G da un vertice dato in ingresso (la sorgente) può essere risolto dalla visita in ampiezza. Prima di mostrare una versione di *BFS* che calcoli anche le distanze, discutiamo prima l'idea dietro questo algoritmo.

Dato un qualsiasi grafo $G = (V, E)$ e 2 vertici v e u , chiameremo $\delta(v, u)$ la distanza tra v e u ; quindi la funzione δ restituisce la lunghezza del percorso più corto che parte da v ed arriva a u .

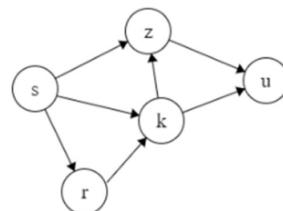
L'idea per il calcolo delle distanze è quella di definire un algoritmo che calcoli i valori della funzione δ da un vertice fissato; ovvero $\delta(s, u) \forall u \in V$. Questo si può ottenere riempiendo un array d tale che $\forall u \in V$, al termine dell'algoritmo si abbia $d[u] = \delta(s, u)$

Ovviamente possono esserci più percorsi minimi, ad es, per il grafo di destra abbiamo

s e u percorsi minimi, ma indipendentemente dal numero di percorsi minimi la distanza è un concetto ben definito.

Infatti $\delta(s, u) = 2$ e qualunque percorso non può avere distanza minore di 2

Si osservi che nel momento in cui si raggiunge un vertice bisogna memorizzare i vertici che ci hanno permesso di arrivare a quel vertice (praticamente salviamo il percorso) per poter visitare il grafo a distanze crescenti.



Inizializzeremo quindi una coda con $\{s\}$; poi supponiamo di scoprire da s i vertici r, k, z in questo ordine \rightarrow la coda quindi sarà $\{r, k, z\}$ (s verrà rimosso una volta visitato) ed a ciascuno di questi verrà associato il vertice s come colui che li ha scoperti.

Poi passa ad r (testa della coda) che viene tolto dalla coda senza scoprire nulla.

Nuova coda $\{k, z\}$; passiamo a k che scopre u e lo mette in coda $\{z, u\}$. A questo punto i vertici non aggiungono nulla di nuovo e vengono semplicemente rimossi dalla coda.

Termineremo l'algoritmo con le seguenti informazioni:

$$s \rightarrow r, s \rightarrow k, s \rightarrow z, k \rightarrow u (\rightarrow = \text{scopre})$$

In pratica memorizziamo gli archi $(s, r), (s, k), (s, z), (k, u)$.

Si noti che concatenando l'arco (s, k) con (k, u) otteniamo la sequenza sku (che è esattamente uno dei percorsi minimi che partono da s e raggiungono u).

Questo ragionamento ci fa capire che basta ricordarci gli archi che scopre la *BFS* per ottenere i percorsi minimi (e di conseguenza le distanze) del grafo dalla sorgente. Quindi per risolvere il nostro problema basta raffinare l'algoritmo della *BFS*:

- $d[v] = \delta(s, u)$ dove la funzione $d \rightarrow \mathbb{N} \cup \{\infty\}$
 - ❖ Il valore ∞ è usato per convenzione per definire la distanza tra 2 vertici non raggiungibili.
- $p[u] = v$: vettore p che associa al vertice u il vertice v che ha permesso la sua scoperta
 - ❖ La funzione $p : V \rightarrow \cup \{NIL\}$ con **NIL** che rappresenta il vertice associato dalla sorgente (la sorgente non viene scoperta da nessun vertice) o dai vertici non raggiungibili dalla sorgente.

Date queste premesse, abbiamo bisogno di nuovi valori di inizializzazione:

```

1  Init(G)
2    FOR EACH x IN V DO
3      Color[x] = 'b'
4      d[x]     = ∞
5      p[x]     = NIL

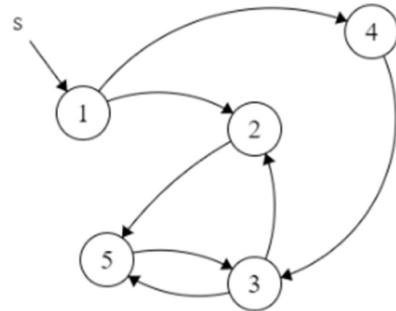
```

Dunque, al precedente algoritmo **VisitaAmpiezza** dobbiamo aggiornare i valori precedenti ogni volta che scopriamo un vertice:

```

1   BFS(s, G)
2     Init(G)
3     F = Accoda(F, s)
4     Color[s] = 'g'
5     d[s] = 0
6     // p[s] = NIL
7     WHILE F != NIL DO
8       x = Testa(F)
9       FOR EACH y in Adj[x] DO
10         IF Color[y] = 'b' THEN
11           Color[y] = 'g'
12           d[y] = 1 + d[x]
13           p[y] = x
14           F = Accoda(F, y)
15     F = Decoda(F)
16     Color[x] = 'n'

```



29.2 Correttezza della BFS

Facciamo una prima verifica di correttezza con un esempio, descriviamo quindi il grafo precedente attraverso 2 tabelle che analizzano i valori degli elementi degli array ad ogni iterazione del while (l'iterazione numero 0 indica che il ciclo while non è ancora iniziato, dalla linea 3 a 6):

	0	1	2	3
$d[1]$	0	0	0	0
$d[2]$	∞	1	1	1
$d[3]$	∞	∞	∞	2
$d[4]$	∞	1	1	1
$d[5]$	∞	∞	2	2

	0	1	2	3
$p[1]$	NIL	NIL	NIL	NIL
$p[2]$	NIL	1	1	1
$p[3]$	NIL	NIL	NIL	4
$p[4]$	NIL	1	1	1
$p[5]$	NIL	NIL	2	2

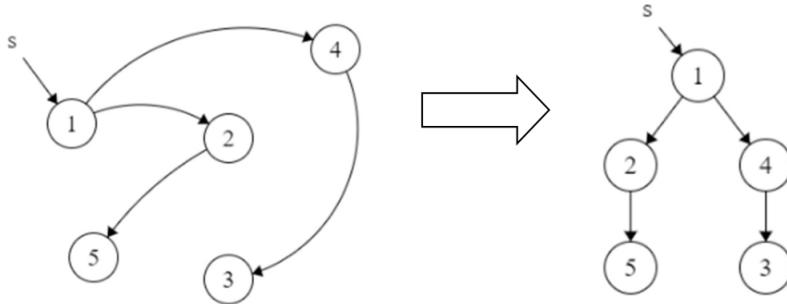
	0	1	2	3	4	5
F	(1)	(2,4)	(4,5)	(5,3)	3	0

Si noti che dopo la terza iterazione non verrà fatto alcun aggiornamento. si va solo a pulire la coda.

Dunque alla fine dell'algoritmo verranno percorsi i seguenti archi:

$$(1,2), (1,4), (2,4), (4,3)$$

Più precisamente, del grafo G andremo a memorizzare il seguente sottografo, detto sottografo dei percorsi minimi di s .



Il precedente sottografo è praticamente un albero con radice la sorgente s , con le distanze come percorsi.

Ad es. il percorso che va dalla radice alla figlia 5 ha distanza 2 che è proprio il percorso minimo del grafo G da 1 a 5 (tutti gli altri percorsi non possono essere più corti).

Ovviamente non basta esporre un singolo esempio per poter confermare la correttezza dell'algoritmo.

Dunque per poter dire che l'algoritmo risolve correttamente il problema delle distanze dobbiamo innanzitutto dimostrare che la *BFS* visiti **tutti** i nodi raggiungibili dalla sorgente e che le distanze siano aggiornate correttamente (ovvero che la *BFS* scopre i vertici entro il *turno* della distanza del vertice).

Formalmente dobbiamo dimostrare che al termine di $BFS(G, s)$ saranno rispettate le seguenti proprietà:

- A Ogni vertice $v \in V$ raggiungibile da s verrà prima o poi visitato;
- B $\forall v \in V, d[v] = \delta(s, v)$;
- C Se v è raggiungibile da s , un percorso minimo da s a v è ottenuto concatenando un percorso minimo da s a $p[v]$ con l'arco $(p[v], v)$ (si noti che il predecessore di v è proprio il vertice che scopre v).

Per dimostrare le precedenti assunzioni dobbiamo sfruttare delle proprietà dei grafi.

1. $(u, v) \in E \implies \delta(s, v) \leq \delta(s, u) + 1$ Questa proprietà delle distanze è abbastanza ovvia; potremmo infatti avere i seguenti casi:

- u non è raggiungibile da s . Allora $\delta(s, v) < \infty$ se v è raggiungibile, altrimenti risulta $\infty = \infty$. La proprietà è banalmente verificata in entrambi i casi;
- Se u è raggiungibile da s allora esiste anche un percorso minimo da s a u (al più togliamo l'arco (u, v)). È ovvio che il percorso minimo da s a v non può essere più lungo di quello che segue il percorso minimo da s a u concatenato all'arco (u, v) ($\delta(s, u) + 1$). Può essere solo più corto → la proprietà $\delta(s, v) \leq \delta(s, u) + 1$ è verificata anche in questo caso.

2. In ogni momento, vale che $\forall v \in V, d[v] \geq \delta(s, u) \rightarrow$ le stime possono essere fatte solo per eccesso (questa proprietà riguarda il funzionamento di *BFS*) Anche questa proprietà è semplice da verificare, gli unici momenti in cui

cambia la stima sono quelli concomitanti all'accodamento del vertice. Basta quindi verificare la suddetta proprietà solo nei momenti in cui il vertice viene inserito nella coda (gli altri vertici avranno la stessa stima)

- ▶ Il caso base è $F = \{s\}$ (numero di inserimenti 1), dove $d[s] = 0 = \delta(s, s)$ (è proprio la distanza di s), mentre $\forall v \in V \setminus \{s\}$, abbiamo $d(v) = \infty \geq \delta(s, v)$ (sicuramente verificata)
- ▶ Per il caso induttivo supponiamo che il numero di inserimenti in coda sia $k > 1$; dunque per ipotesi induttiva al $k - 1$ inserimento la proprietà è supposta vera:

$$\forall v \in V, d \geq \delta(s, v)$$

Al k -esimo inserimento la coda passerà da $F = (v_1, \dots, v_r)$ a $F = (v_1, \dots, v_r, v_{r+1})$; Dunque v_{r+1} è stato appena scoperto (nell'algoritmo avrà nome y) dal vertice in cima alla coda v_1 (x nell'algoritmo).

Ma per ipotesi induttiva abbiamo $d[v_1] \geq \delta(s, v_1)$, quindi poiché nell'algoritmo abbiamo l'istruzione $d[v_{r+1}] = d[v_1] + 1$ (tutti gli altri vertici mantengono la stima precedente, che sono verificate per ipotesi induttiva).

Essendo $d[v_{r+1}] = d[v_1] + 1 \geq \delta(s, v_1) + 1$ per la priorità 1 (l'arco v_1, v_{r+1} esiste altrimenti non avremmo potuto scrivere $v_{r+1} \rightarrow \delta(s, v_1) + 1 \geq \delta(s, v_{r+1}) \geq d[v_{r+1}]$ come volevasi dimostrare)

3. Se $F = (v_1, v_2, \dots, v_k)$ allora:

- I. $d[v_i] \leq d[v_{i+1}] \forall 1 \leq i < k$ (i vertici in coda o hanno la stessa stima o sono più grandi);³⁹
- II. $d[v_k] \leq d[v_1] + 1$ (gli estremi hanno una stima che differisce al più di 1)⁴⁰. Dimostriamo tale proprietà per induzione sul numero di momenti in cui la coda cambia (ovvero, quando aggiungo o tolgo un elemento dalla coda)

Caso base: $F = (s)$

La prima operazione può essere solo l'accodamento della sorgente che è banalmente vera.

Caso induttivo

Con la z -esima operazione in coda possiamo avere o un accodamento o un decodamento.

Nel primo caso passeremo da $F = (v_1, v_2, \dots, v_k)$ (verificata per induzione) a $F = (v_1, v_2, \dots, v_k, v_{k+1})$ dove l'unica stima che può confutare la proprietà è quella di v_{k+1} (le altre non cambiano e quindi sono verificate), ma durante l'accodamento abbiamo $d[v_{k+1}] = d[v_1]$ e quindi la proprietà II è verificata.

Per la proprietà I bisogna verificare solo che $d[v_k] \leq d[v_{k+1}]$, ma prima della z -esima operazione abbiamo per ipotesi induttiva $d[v_k] \leq d[v_1] + 1$ e $d[v_1] \leq \dots \leq d[v_k]$; dunque $d[v_k] \leq d[v_1] + 1 = d[v_{k+1}]$ (per la linea 12 di BFS).

Se invece la z -esima operazione è di decodamento passiamo da $F = (v_1, v_2, \dots, v_k)$ a $F = (v_2, \dots, v_k)$, ma per ipotesi di induzione $d[v_1] \leq d[v_2] \leq \dots \leq d[v_k]$ e poiché il decodamento non cambia nessuna stima la proprietà $d[v_i] \leq d[v_{i+1}] \forall 1 \leq i < k$ è banalmente verificata. Per la proprietà II dobbiamo invece verificare che $d[v_k] \leq d[v_2] + 1 \rightarrow$ prima del decodamento si ha $d[v_1] \leq d[v_2] \implies d[v_1] + 1 \leq d[v_2] + 1$ e $d[v_k] \leq d[v_1] + 1$. Per transitività risulta $d[v_k] \leq d[v_1] + 1 \leq d[v_2] + 1$ (come volevasi dimostrare).

³⁹Operazione universale su un insieme vuoto.

⁴⁰Abbiamo $0 \leq 1$.

Possiamo adesso dimostrare le 3 proprietà (A, B, C) che verificano la correttezza della BFS.

Ragioniamo per induzione su distanze crescenti, suddividiamo il nostro insieme $V = V_0 \cup V_1 \cup \dots \cup V_k \cup V_\infty$; dove V_i contiene tutti i vertici a distanza i dalla sorgente. Se prendiamo il nostro grafo d'esempio avremo:

$$V_0 = \{1\}, V_1 = \{2, 4\}, V_2 = \{5, 3\} \text{ e } V_\infty = \emptyset$$

L'idea è quella di usare l'induzione sulle classi di equivalenza precedentemente descritte ($i \in \mathbb{N}$), attraverso queste le proprietà A, B, C descritte diventano:

$$\forall v \in V_i$$

- A Esiste un istante in cui v viene messo in coda (e quindi viene colorato di grigio)
- B Quando v viene messo in coda $d[v] = i (= \delta(s, v))$
- C Se $v \neq s$, un percorso minimo da s a v è ottenibile concatenando il percorso minimo da s a $p(v)$ con l'arco $(p(v), v)$

Dunque la verifica di correttezza dell'algoritmo BFS si riduce a dimostrare le suddette proprietà. Il caso base sarà $i = 0$, ovvero $V_0 = \{s\}$.

Il momento in cui viene messa la sorgente in coda la sua stima è 0 che è evidentemente la stima corretta (A e B sono verificate e anche C è banalmente vera essendo la premessa falsa). Poiché la stima del vertice può cambiare solo quando questo viene messo in coda e, poiché ciò avviene al più una volta (si ricorda che nessun vertice può essere accodato 2 volte), siamo sicuri che una volta stabilita la stima di un vertice, questa resterà così fino al termine dell'algoritmo.

Caso induttivo

Sia V_z con $z > 0$ (V_z può contenere tanti vertici) e sia $v \in V_z$ un arbitrario vertice a distanza z .

Se $v \in V_z$ allora esiste un percorso di lunghezza z da s a v ; siccome $z > 0$ è evidente che $s \neq v$ e che c'è almeno un arco tra i 2.

Sia u il predecessore di v (vale anche per $u = s$); sappiamo che $u \in V_{z-1}$ e quindi per ipotesi induttiva, esiste un momento in cui u viene messo in coda (A) e $d[u] = \delta(s, u)$ (B). Inoltre se $u \neq s$ esiste un percorso minimo da s a u ottenuto concatenando il percorso minimo da s a $p(u)$ con l'arco $(p(u), u)$ (C).

Il fatto che v venga scoperto da questo u si dimostra ragionando su un qualsiasi $l \in V_r$; vediamo quindi se tale l possa scoprire V (il nostro scopo è assicurarsi che $r = z - 1$):

- Se $r < z - 1$ allora l non potrà mai scoprire z perché dovrebbe esistere un percorso minimo da s a l che attraverso un arco raggiungesse v (assurdo perché $r + 1 < z - 1 + 1 \rightarrow r < z$);

- $r > z - 1$ viene escluso dalla proprietà di monotonia della coda (proprietà 3) → Infatti se l scopre v allora significa che l è in cima alla coda, ma ciò è assurdo perché la stima dei vertici che vengono messi in coda non può decrescere (o resta uguale o aumenta); di conseguenza prima che un vertice della classe V_i possa entrare in coda, tutti i vertici di V_{i-1} devono essere già entrati in coda (in caso contrario non entreranno mai più in coda).
 - ❖ Quindi prima che $l \in V_r$ possa entrare in coda, tutti i vertici V_{z-1} devono passare per la coda. Allora il vertice $u \in V_{z-1}$ entra in coda prima di l , con la conseguenza che u arriva alla testa della coda prima di l e quindi sarà proprio u a scoprire v .

Dunque, quando u scoprirà v risulterà $d[v] = d[u] + 1 = \delta(s, u) + 1 = z - 1 + 1 = z$ (per ipotesi induttiva), quindi la A e la B sono rispettate (ricordiamo che $v \in V_z$), ovviamente anche la C è vera perché la BFS assegnerà a $p[v] = u$ e il percorso minimo sarà proprio il percorso minimo da s a $p[v]$ concatenando con l'arco $(p[v], v)$.

A questo punto restano solo i vertici non raggiungibili, ma di questi dovremmo solo garantire che le distanze siano corrette, ma la loro distanza sarà quella iniziale che è ∞ , esattamente il valore corretto.

29.3 Algoritmo del percorso minimo

Abbiamo dimostrato che l'algoritmo della BFS precedentemente descritto calcola le distanze corrette e i percorsi minimi sono ottenibili nel modo descritto nella proprietà C.

Ciò significa che posso implementare un algoritmo che sfrutta la sopracitata proprietà per calcolare il percorso minimo di un vertice dalla sorgente nel seguente modo

```

1 PercorsoMinimo(G, s, v)
2   BFS(G, s) // visita tutti i vertici raggiungibili da s
3   IF Color[v] = 'n' THEN
4     // v è raggiungibile da s
5     StampaPercorsoMinimo(s, v, p)

```

StampaPercorsoMinimo è un algoritmo ricorsivo che prende in ingresso s , v è l'array dei predecessori e concatena ricorsivamente il percorso del predecessore di v con l'arco $(p[v], v)$:

```

1 StampaPercorsoMinimo(s, v, p)
2   IF s = v THEN
3     print(s) // Percorso vuoto
4   ELSE
5     StampaPercorsoMinimo(s, p[v], p)
6     print(v) // Questo aggiunge l'arco (p[v], v)

```

La complessità di questo algoritmo è al massimo il numero di vertici.

Il costo dell'algoritmo principale **PercorsoMinimo** sarà semplicemente il costo della BFS ($O(|V| + |E|)$) più l'eventuale costo di **StampaPercorsoMinimo**:

$$T_{PercorsoMinimo}(|V| + |E|) = O(|V| + |E|) + O(V) = O(|V| + |E|)$$

30 Lezione 33

30.1 Visita in profondità

La visita in profondità è definibile in maniera naturale grazie alla definizione di percorso. Ha infatti gli stessi problemi di terminazione e efficienza riscontrati nella visita in ampiezza, e come questa, si risolvono utilizzando la colorazione.

Se per alcuni problemi è indifferente il tipo di visita utilizzato, ci sono dei problemi come il calcolo dei percorsi minimi o la scoperta di cicli di un grafo che possono essere utilizzati solo con un determinato tipo di visita.

Per il primo infatti l'unica soluzione corretta è la *BFS*; per il secondo si può procedere solo tramite *DFS* (Depth First Search).

L'idea di visita in profondità è quella di avere un algoritmo che in maniera ricorsiva si chiama sugli adiacenti → Visitare tutto ciò che è raggiungibile da un nodo significa visitare tutto ciò che è raggiungibile dai suoi adiacenti più il nodo stesso (è naturale l'implementazione ricorsiva).

Il problema dei cicli è ovviamente ricorrente anche nella visita in profondità; quindi anche in questo tipo di visita bisogna evitare i cicli per terminare l'algoritmo e visitare ogni vertice una e una sola volta così da rendere l'algoritmo esponenziale (nel caso peggiore infatti avremmo tempo esponenziale).

Il tipo di applicazioni della visita in profondità in generale necessita la visita di tutto il grafo; dunque, definiremo un algoritmo che non si limita ad una singola sorgente come abbiamo visto per la visita in ampiezza (il tipo di applicazioni per quest'ultima è sensata solo se applicata per una singola sorgente).

Prima di passare all'algoritmo di *DFS* diamo una prima implementazione di algoritmo di visita in *post/pre* order. Poi definiremo la *DFS* in post order (che è quella più utilizzata).

```

1 VisitaPostOrder(G, s)
2   Color[s] = 'g'
3   FOR EACH v IN Adj[s] DO
4     IF Color[v] = 'b' THEN
5       VisitaPostOrder(G, v)
6     /*
7       In questo punto ho visitato tutti i
8       vertici raggiungibili dalla sorgente
9       s
10    */
11   Visita(s)
12   Color[s] = 'n'
```

```

1 VisitaPreOrder(G, s)
2   Color[s] = 'g'
3
4   Visita(s)
5
6   Color[s] = 'n'
7
8   FOR EACH v IN Adj[s] DO
9
10     IF Color[v] = 'b' THEN
11
12       VisitaPreOrder(G, v)
```

30.2 Algoritmo DFS

Come per la *BFS*, anche con la *DFS* è possibile associare ad ogni vertice delle informazioni aggiuntive sfruttando un array:

- ▶ $p[v]$ → Associa al vertice v il suo predecessore. Quindi con $p[v] = s$ indica che il vertice v è stato scoperto da s ;
- ▶ $d[v]$ → Rappresenta il tempo in cui il vertice v è stato scoperto;
- ▶ $f[v]$ → Rappresenta l'istante di tempo in cui ho finito di visitare quel vertice.

Di seguito riportiamo l'algoritmo *DFS* (supponiamo tempo variabile globale)

```
1 DFS_Visit(G, s)
2     Color[s] = 'g'
3     tempo = tempo + 1
4     d[s] = tempo
5     FOR EACH v in Adj[s] DO
6         IF Color[v] = 'b' THEN
7             p[v] = s
8             DFS_Visit(G, v)
9         // Operazioni con s
10    Color[s] = 'n'
11    tempo = tempo + 1
12    f[s] = tempo
```

È interessante notare che per come abbiamo gestito la variabile *tempo* non esisterà mai un valore uguale per i vettori *d* e *f* (né tanto meno all'interno dello stesso array, né tra celle dei 2 array).

Il precedente algoritmo verrà utilizzato dal seguente in modo da visitare completamente il grafo:

```
1 DFS(G)
2     Init(G)
3     tempo = 0
4     FOR EACH s in V DO
5         IF Color[s] = 'b' THEN
6             DFS_Visit(G, s)
```

Si noti che durante la *DFS_Visit* posso avere vertici *bianchi*, *grigi* e *neri*, ma durante la chiamata *DFS* i vertici o sono *bianchi* o *neri* → Questo è dovuto al fatto che la *DFS_Visit* colora in nero tutti i vertici che scopre (quando termina tutto il percorso viene annerito in ordine inverso della scoperta dei vertici).

30.3 Terminazione e complessità

Questo algoritmo termina per lo stesso motivo per cui la *DFS* terminava → Il fatto che non sia possibile visitare 2 volte lo stesso vertice ci garantisce la terminazione.

Se per assurdo visitassi un vertice più di una volta allora esisterebbe un primo istante in cui il vertice venga colorato in grigio per la prima volta e in un secondo istante in cui venga di nuovo visitato.

Ciò significa che ci saranno 2 momenti in cui chiamo la `DFS_Visit` sullo stesso vertice, che è assurdo visto che il vertice da grigio non può mai ritornare bianco (al massimo diventa nero); pertanto il vertice incontrato per la seconda volta è già grigia.

Quindi o viene eseguita la `DFS_Visit` in `DFS_Visit` oppure la `DFS_Visit` in `DFS`; in entrambi i casi ho come *guardia* un test sul colore del vertice (grigio dopo la prima visita).

Ciò si traduce nell'avere un numero di chiamate ricorsivo pari al numero di vertici del grafo. L'albero di ricorrenza di `DFS_Visit` al più genera una chiamata per ogni vertice; `DFS` ci garantisce una chiamata di `DFS_Visit` su ogni vertice → Quindi non solo non potrò avere una chiamata ricorsiva sullo stesso vertice nel singolo albero di ricorrenza, ma non potrò averla nemmeno per alberi di ricorrenza diversi (sarà più chiaro in seguito con un esempio pratico).

Poiché scopro tutti i vertici del grafo e ogni vertice viene visitato una sola volta significa che anche un arco viene scoperto una sola volta.

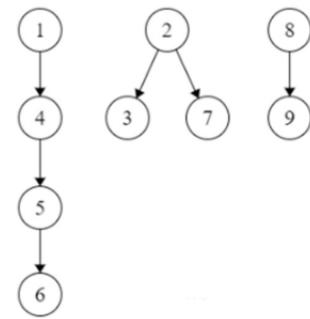
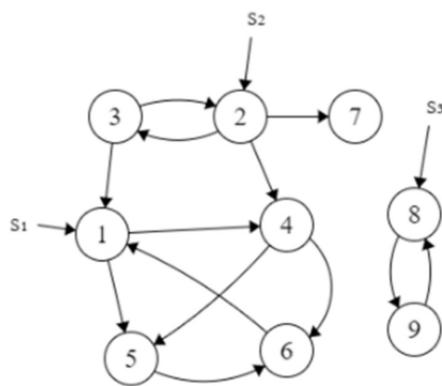
Anche se non posso dare un tempo di esecuzione ad un singolo albero di ricorrenza posso dire che l'algoritmo di `DFS` ha tempo lineare sul numero di vertici e di archi:

$$T_{DFS}(|V|, |E|) = \Theta(|V|, |E|)$$

(è sicuramente un Θ perché non lascio nessun vertice bianco)

Per una migliore comprensione di seguito riportiamo un esempio di degli alberi di ricorrenza della `DFS` per un grafo:

Il grafo a sinistra genererà la seguente foresta di alberi di ricorrenza



Questo rappresenta anche il sottografo dei predecessori dove le radici sono gli unici vertici con

$$p[v] = NIL$$

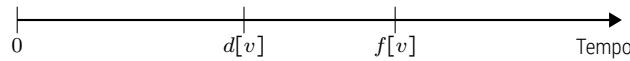
Si noti che mentre la visita in ampiezza non garantisce che il percorso dalla sorgente ad un vertice raggiungibile è il percorso minimo, nella visita in profondità ciò non è vero. Infatti il percorso minimo da 1 a 5 nell'esempio è di lunghezza 1, ma nel sottografo dei predecessori seguiamo il percorso 1, 4, 5 che ha lunghezza 2.

Dunque, la *DFS* non può essere usata per risolvere il problema dei percorsi minimi, tuttavia sarà utile per verificare se un grafo sia ciclico o meno.

30.4 Teorema della struttura a parentesi

Il teorema è conseguenza diretta delle proprietà della *DFS* (descritte di seguito). Tali proprietà anche se astratte hanno un impatto diretto sullo studio della correttezza di algoritmi che usano la *DFS*.

Descriviamo ora il seguente asse temporale



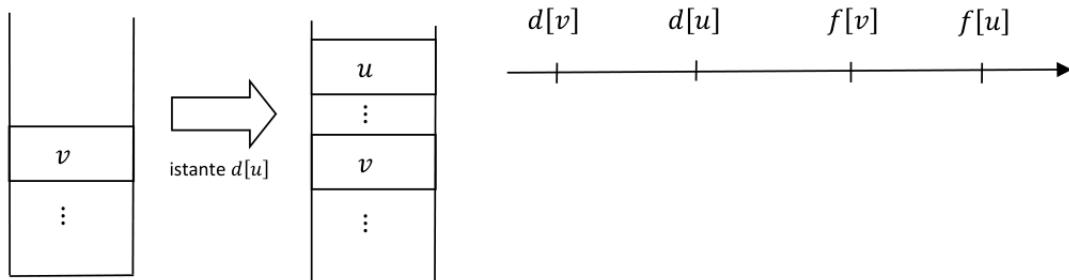
In questo asse ad ogni vertice è associato un unico intervallo $[d[v], f[v]]$ che non potrà mai essere modificato in seguito all'inizializzazione (per come è definito l'algoritmo). Tale intervallo viene definito come **intervallo di scoperta** e rappresenta la durata in cui il vertice è stato attivo.

Teorema

$\forall u, v \in V$ con $u \neq v$ vale **una ed una sola** delle seguenti proprietà

- $d[v] < d[u] < f[u] < f[v]$;
- $d[u] < d[v] < f[v] < f[u]$;
- $d[v] < f[v] < d[u] < f[u]$;
- $d[v] < f[u] < d[u] < f[v]$;

In pratica se i 2 intervalli non sono disgiunti allora uno è dentro l'altro ($[]$) è lecito, $([)$ no). Dimostreremo questo teorema concentrandoci sulle parentesizzazioni non possibili → supponiamo che non sia possibile $d[v] < d[u] < f[v] < f[u]$;



Ma affinché possa eseguire l'istante $f[v]$ deve essere la chiamata di v ad avere il controllo → Questo significa che in cima allo stack ci sia v ; questo può accadere in 2 modi:

- ▶ C'è una nuova chiamata $\text{DFS_Visit}(G, v)$ (Questo non è possibile per l'algoritmo *DFS* descritto, che non permette di scoprire più di una volta lo stesso vertice);
- ▶ La chiamata ricorsiva su v rappresentata nello stack riprende il controllo → devo terminare tutte le chiamate in cima allo stack fino a v .

Dunque l'unica possibilità è che le chiamate in cima allo stack terminano con la conseguenza di dover prima terminare la chiamata ad u e quindi avere $d[v] < d[u] < f[u]$ (che va contro la nostra supposizione). La dimostrazione che anche la parentesizzazione $d[u] < d[v] < f[u] < f[v]$ non sia possibile è analoga.

Abbiamo in questo modo che le uniche possibilità siano proprio le assunzioni descritte nel teorema

Una diretta conseguenza di tale teorema è che se un intervallo è dentro un altro allora c'è una discendenza diretta nel sottografo dei predecessori. Si noti che se non c'è nessuna discendenza diretta non vuol dire che non ci siano archi tra i vertici; significa solo che la *DFS* non li ha seguiti.

31 Lezione 34

31.1 Foresta DF

Consideriamo il grafo dei predecessori $G^P = (V, E^P)$ dove

$$E^P = \{(p[v], v) \mid v \in V \wedge p[v] \neq NIL\}$$

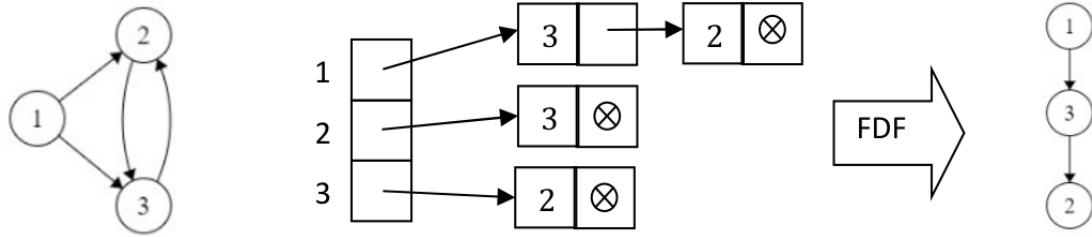
(prendo tutti i vertici e solo gli archi scoperti della *DFS*)

G^P è una foresta poiché le sorgenti non hanno archi entranti (fungono da radice dell'albero) e ogni nodo può essere scoperto da un unico nodo (quindi ho un albero);

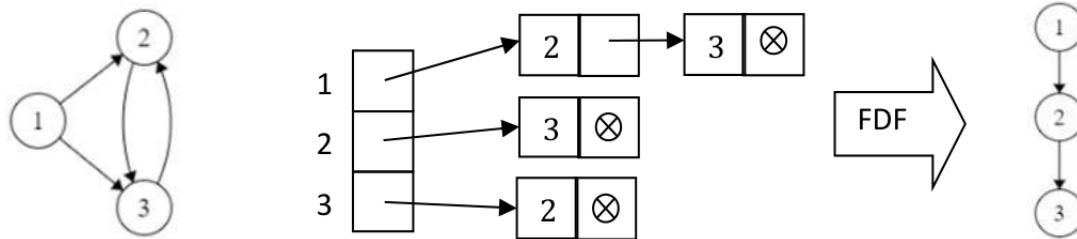
G^P viene anche conosciuto come *Forest DF* (abbreviato in *FDF*).

Individuiamo delle correlazioni tra il comportamento dell'algoritmo e il grafo dei predecessori che la *DFS* genera. Ovviamente per un grafo non esiste un'unica foresta, infatti questa dipende da come viene rappresentato concretamente il grafo.

Di seguito riportiamo un esempio, a partire da un grafo con la sua lista di adiacenza:



Se cambiamo la lista di adiacenza (il grafo resta invariato) cambierà anche la foresta:



Quindi, data una rappresentazione del grafo posso dedurre la sua foresta *DF*, ma se mi è dato solo un grafo non ho modo di prevedere quali percorsi la *DFS* esplorerà (a meno che non venga deciso arbitrariamente) → Non posso quindi conoscere la *FDF*. L'unica certezza è che la *DFS* visiterà tutti i vertici e la *FDF* avrà determinate proprietà.

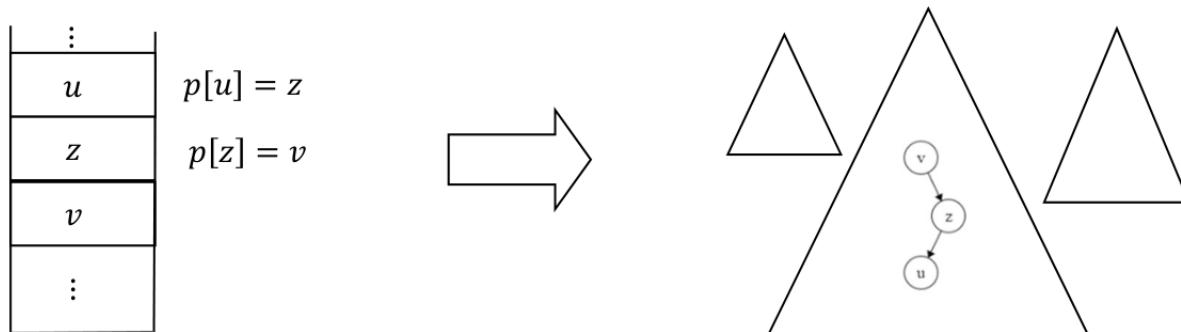
Abbiamo già dimostrato che la struttura della foresta è determinata dai tempi memorizzati in $d[v]$ e $f[v]$.

Possiamo esprimere una prima proprietà della *FDF*, al termine della *DFS* in G :

$\forall u, v \in V$ con $u \neq v$:

- u è discendente di v in *FDF* $\iff d[v] < d[u] < f[u] < f[v]$

Questo implica che per il seguente stack di attivazione avremmo una specifica *FDF*:



(questo dimostra anche l'implicazione \Leftarrow per diretta conseguenza del teorema della struttura a parentesi)

Dimostriamo l'implicazione \Rightarrow per induzione:

- Se u è discendente da v allora il percorso ha lunghezza almeno 1 (essendo $v \neq u$).

Sia proprio $|\pi| = 1$ il nostro caso base (banalmente vero) \rightarrow Per come viene descritta la DFS, sarà $p[u] = v$ con la conseguenza che $d[u] = d[v] + 1$ (v viene scoperto prima di u e quindi il suo tempo sarà sicuramente minore) e finché non termina la chiamata di u non potrà terminare nemmeno quella di v :

$$f[v] > f[u] \text{ (ovvero: } d[v] < d[u] < f[u] < f[v])$$

Per $|\pi| = k > 1$ abbiamo una situazione del seguente tipo:

Supponiamo che z sia il predecessore di u ; per ipotesi induttiva avremo

$$d[v] < d[z] < f[z] < f[v]$$

poichè se $p[u] = z$ allora la lunghezza del percorso da v a z è $k - 1$

Il fatto che $p[u] = z$ implica anche $d[z] < d[u] < f[z]$ e sempre per il teorema di chiusura a parentesi l'unica relazione possibile (essendo u discendente di z) è

$$d[z] < d[u] < f[u] < f[z]$$

Dunque per transitività $d[v] < d[u] < f[u] < f[v]$ come volevasi dimostrare

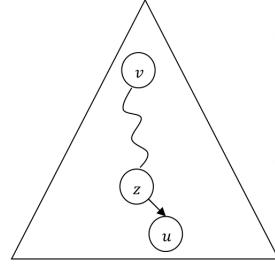
31.2 Teorema del percorso bianco

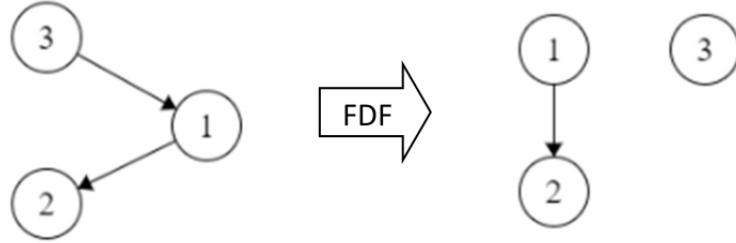
Dato un grafo $G = (V, E)$ e $u, v \in V$, al termine della DFS varrà la seguente proprietà:

- u è discendente di v in $FDF \iff$ al tempo $d[v]$, $\exists \pi$ in G da v a u fatto di vertici bianchi.

NB: Si potrebbe pensare che u è raggiungibile da $v \iff$ se esiste un percorso nella FDF . In realtà è vera solo l'implicazione \Leftarrow , infatti se non esiste un percorso in FDF non è detto che non esista un percorso nel grafo.

Di seguito un banale esempio:





Dimostriamo l'implicazione \Leftarrow del teorema. Abbiamo che al termine della *DFS* u è discendente di v in *FDF*.

Poniamoci in un vertice $z \neq v$ che si trova nel percorso tra v e u all'istante $d[v]$. Se dimostriamo che z è bianco allora anche tutti i vertici dopo z (u compreso) sono bianchi.

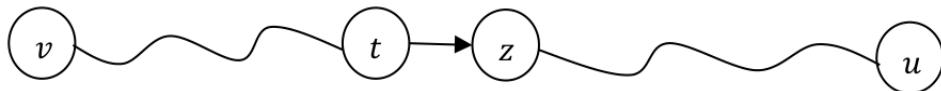
Per la proprietà vista in precedenza si ha $d[v] < d[z] < f[z] < f[v]$, ma se $d[v] < d[z]$ significa che v viene scoperto prima di z , ergo z è bianco all'istante $d[v]$. Ma allora è evidente che all'istante $d[v]$ esiste un percorso π di vertici bianchi fino ad u .

Dimostriamo l'implicazione \Rightarrow . Abbiamo che al tempo $d[v]$ esiste un percorso π tutto bianco nel grafo G .

Dimostriamo che ogni vertice di π discende da v in *FDF* (questa è una assunzione più forte di u è discendente di v in *FDF*; quindi sarà vera anche quest'ultima).

Supponiamo per assurdo che esista un vertice nel percorso π che non sia discendente di v , supponiamo sia z il primo vertice che non sia discendente di v ($z \neq v$ altrimenti sarebbe discendente).

Dunque, esiste un predecessore di z , chiamiamolo t , che discenderà da v (altrimenti z non sarebbe il primo vertice non discendente da v).



Abbiamo 2 casi possibili:

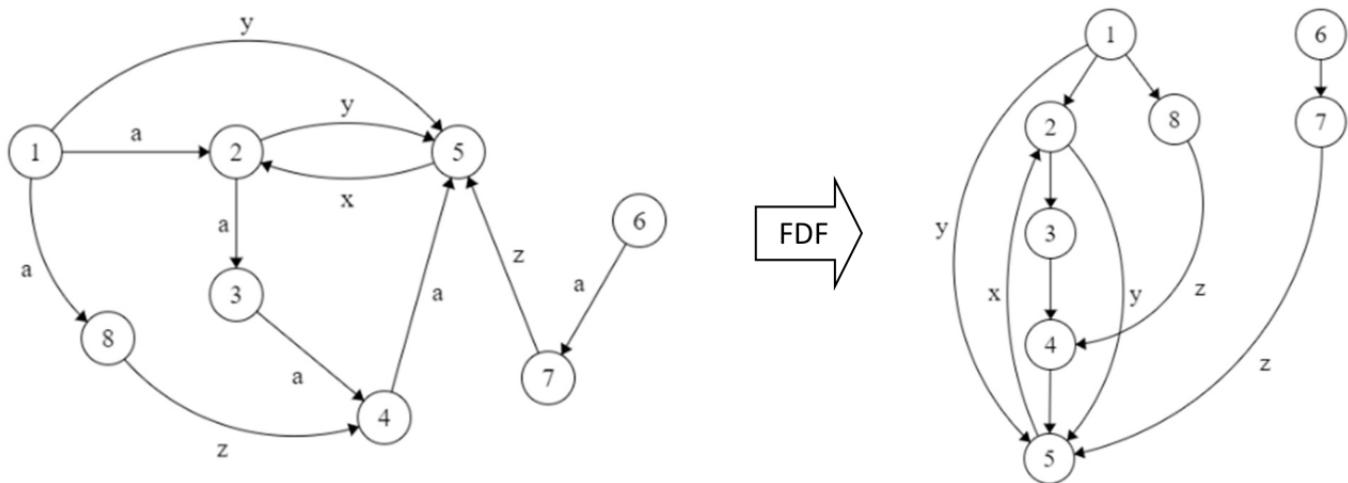
- ▶ $t = v \rightarrow$ Siamo nella situazione $[v] \rightarrow [z]$ con z bianco e adiacente a v . Ma allora la visita *DFS* prima di terminare v guarderà tutti i suoi adiacenti; quindi se z è nero (sappiamo solo che all'istante $d[v]$ è bianco, quindi potrebbe essere stato colorato nel frattempo) le uniche possibilità sono $d[v] < f[v] < d[z]$ e $d[v] < d[z] < f[z] < f[v]$
 - ◊ La prima contraddice il fatto che v e z siano adiacenti. Pertanto l'unica possibile è la seconda con la conseguenza che z sia discendente di v .
- ▶ $t \neq v \rightarrow$ Poiché t è discendente di v abbiamo $d[v] < d[t] < f[t] < f[v]$, ma se z non è discendente di v l'unica possibilità è che $f[v] < d[z]$ il che è assurdo per lo stesso ragionamento fatto in precedenza

- ◊ Infatti essendo t adiacente a z dobbiamo avere $d[t] < d[z] < f[z] < f[t]$ e quindi per transitività $d[v] < d[z] < f[z] < f[v]$, ovvero z è discendente di v .

Questo teorema garantisce che la prima sorgente percorrerà tutti i vertici raggiungibili da tale sorgente, sulle sorgenti successive non è detto (potremmo imbatterci in vertici già scoperti dalle sorgenti precedenti).

31.3 Tipi di archi nella DFS

Supponiamo di avere l'ordine naturale degli interi sia per l'array dei vertici che per la lista di adiacenza:



si noti che solo gli archi etichettati con a sono gli archi che realmente esistono nella FDF (gli altri appartengono al grafo, ma non alla foresta DF). Per quanto riguarda l'arco etichettato con x esso, se inserito nella foresta, sarebbe un arco che da un nodo arriva ad un suo antenato, mentre y va da un antenato ad un suo discendente non diretto.

Infine, gli archi etichettati con z , nella FDF connettono 2 nodi che non hanno nessuna relazione tra loro (sono sottoalberi disgiunti).

Queste sono tutte e sole le categorie di archi che è possibile discriminare in una FDF :

- Gli archi a sono detti **archi dell'albero**;
- Gli archi x vengono denominati **archi di ritorno** (da un nodo ad un antenato);
- Gli archi y sono gli **archi in avanti** (da un nodo ad un suo discendente non diretto);
- Gli archi z vengono detti **archi di attraversamento** (2 sottoalberi distinti).

Questi 4 tipi di archi sono riconoscibili dalla DFS ; è possibile scrivere una variante che associa il nome dell'arco durante la visita. Ma questo significa che esistono delle condizioni verificabili localmente tra il passaggio di un nodo ad un altro, ovvero, percorrendo un arco (v, u) :

- ▶ Se u è bianco all'ora è banalmente un arco dell'albero;
- ▶ Gli archi di ritorno sono quelli che partono da un discendente ed arrivano ad un antenato
 - ◊ Ma se (v, u) è un arco di ritorno significa che $d[v] < d[u] < f[u] < f[v]$ con u che non può essere bianco
 - Dunque, u è grigio (mi trovo in un percorso che da u mi ha riportato ad u attraverso v);
- ▶ Se u è nero posso avere 2 possibili casi
 1. $d[v] < d[u] < f[u] < f[v] \rightarrow$ Siamo di fronte ad un arco in avanti (u è stato già scoperto da un percorso partito da v prima di scoprire l'arco (v, u));
 2. $d[u] < f[u] < d[v] < f[v] \rightarrow (v, u)$ è un arco di attraversamento.

Si noti che è possibile distinguere i 2 archi semplicemente confrontando il valore $d[v]$ con $d[u]$ (oppure confrontando il valore $d[v]$ con $f[u]$).

```

1 DFS_Visit(G, v)
2   Color[v] = 'g'
3   d[s] = tempo = tempo + 1
4   FOR EACH u in Adj[v] DO
5     IF Color[u] = 'b' THEN
6       // (v, u) è arco dell'albero
7       p[u] = v
8       DFS_Visit(G, u)
9     ELSE IF Color[u] = 'g' THEN
10      // (v, u) è arco di ritorno
11    ELSE
12      IF d[v] < f[u] THEN
13        // (v, u) è arco in avanti
14      ELSE
15        // (v, u) è arco di attraversamento
16   Color[v] = 'n'
17   f[v] = tempo = tempo + 1

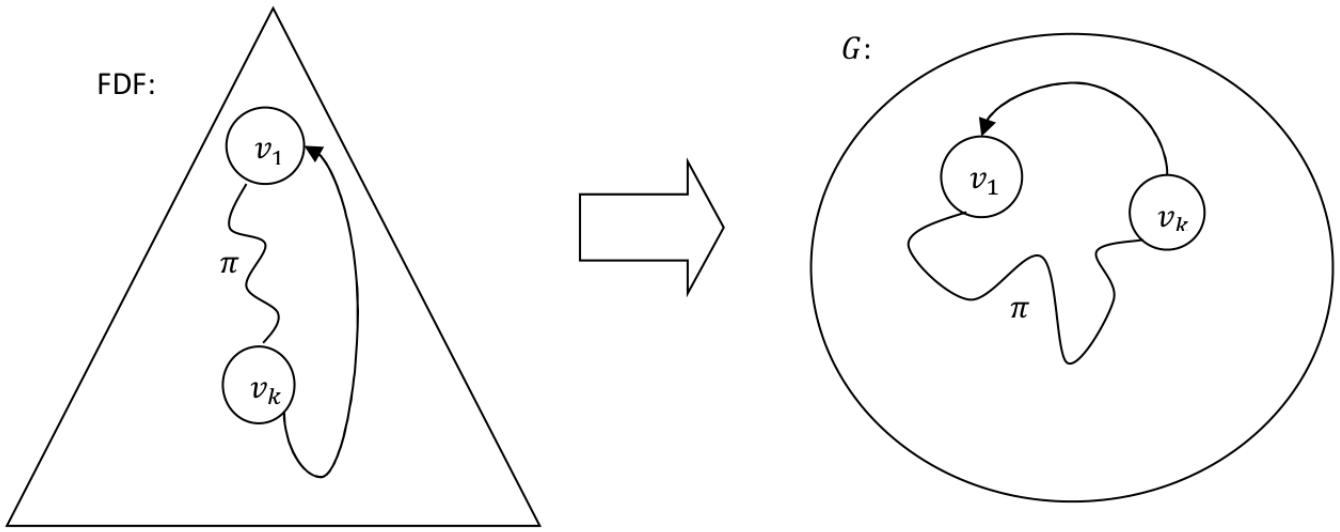
```

La distinzione di tali archi è molto importante in alcuni contesti (come ad es. la verifica della presenza di cicli in un grafo).

La verifica dell'aciclicità si risolve individuando un percorso non semplice $v_1 v_2 \dots v_k v_1$. Questo problema è banalmente risolvibile dal precedente algoritmo; poiché accorgersi della presenza di un ciclo equivale a trovare un arco di ritorno.

31.4 Verifica della ciclicità di un grafo

Per definizione, un arco di ritorno consiste in un nodo nella foresta DF che arriva ad un suo antenato:



Questo dimostra che se c'è un arco di ritorno allora nel grafo esiste un ciclo; il nostro obiettivo è dimostrare che se esistono cicli nel grafo l'algoritmo se ne accorge (questo si traduce nel dire che se non esistono cicli nel grafo l'algoritmo è in grado di dire che tale grafo è aciclico). Dimostriamo che se il grafo presenta un ciclo allora incontrerà un nodo grigio.

Supponiamo che il percorso ciclico sia $v_1 v_2 \dots v_k v_1$ e supponiamo che il primo vertice di quel percorso incontrato dalla *DFS* sia proprio v_1 , ciò significa che all'istante $d[v_1]$ tutti i vertici v_2, \dots, v_k sono bianchi.

Ma quindi, la *DFS* visiterà il percorso π fino a v_k , quindi dovrà visitare tutti gli adiacenti, compreso v_1 che sarà grigio poiché dovrà terminare dopo di v_k .

Dunque, siccome la *DFS* è in grado di accorgersi di eventuali cicli, è possibile scrivere un algoritmo che restituisce 0 (o **false**) se il grafo è ciclico; 1 (o **true**) altrimenti:

```

1 Aciclico(G)
2   Init(G)
3   FOR EACH s in V DO
4     IF Color[s] = 'b' THEN
5       v = DFS_Visit(G, s)
6       IF v = 0 THEN
7         RETURN 0
8   RETURN 1

```

```

1 DFS_Visit(G, s)
2   Color[s] = 'g'
3   FOR EACH v in Adj[s] DO
4     IF Color[v] = 'b' THEN
5       v = DFS_Visit(G, v)
6     IF v = 0 THEN
7       RETURN 0
8     ELSE IF Color[v] = 'g' THEN
9       RETURN 0
10    Color[s] = 'n'
11    RETURN 1

```

L'aciclicità può essere verificata in tempo lineare sulla dimensione del grafo.

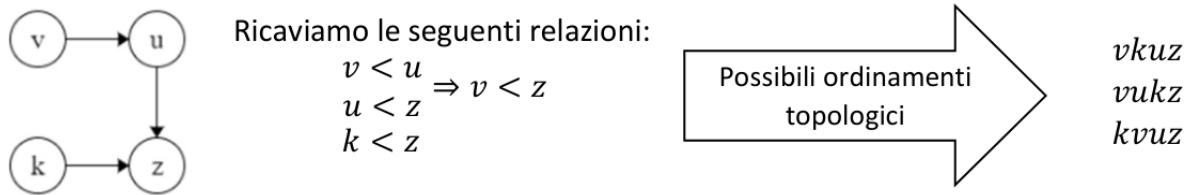
NB: Con la *BFS* non è possibile tale verifica → Il colore grigio non implica aciclicità.

31.5 Ordinamento Topologico

Per ordinamento topologico si intende una relazione d'ordine parziale (possono esserci casi in cui sia totale), definita nel seguente modo:

Dato $G = (V, E)$, un ordinamento topologico (non è detto che sia unico) di G è una permutazione π di V tale che:

$\forall (v, u) \in E, v$ precede u in π ($v < u$). Di seguito riportiamo un esempio:



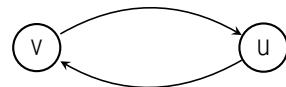
Questi vari ordinamenti sono determinati dal fatto che non esista nessuna relazione tra k e i vertici u, v .

L'importante è che k venga prima di z e ci sia l'ordinamento relativo vuz (qualsiasi altra permutazione al di fuori delle 3 descritte non rispetta la definizione di ordinamento topologico).

In linea di principio potremmo avere tanti ordinamenti topologici quante sono le permutazioni dei vertici nel grafo (senza archi), ma è anche possibile avere grafi con un unico ordinamento topologico (grafo in sequenza $v \rightarrow u \rightarrow x \rightarrow z$, unico ordinamento topologico $\rightarrow vuxz$)

Esistono grafi che non hanno nessun ordinamento topologico:

Infatti, vu non rispetta l'arco (u, v) , simmetricamente uv non rispetta l'arco (v, u)



Abbiamo le seguenti proprietà:

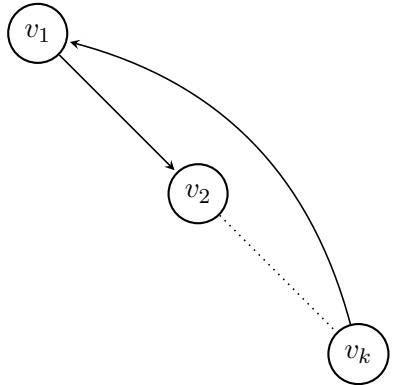
- G è aciclico $\iff \exists$ ordinamento topologico;
- G è ciclico $\iff \nexists$ ordinamento topologico (negazione della precedente).

Dimostriamo che con un grafo ciclico non può esistere nessun ordinamento topologico:

Ipotizziamo di avere il ciclo riportato a lato; supponiamo per assurdo che esista un ordinamento topologico. Dunque i seguenti vertici possono essere messi in una sequenza ordinata del seguente tipo:

$v_1 \ v_2 \dots v_k$

Ma l'esistenza dell'arco (v_k, v_1) implica che il vertice v_1 debba essere messo dopo il vertice v_k , però è assurdo che un vertice sia presente contemporaneamente in 2 punti distinti di una sequenza.



Di conseguenza, non può esistere alcun ordinamento topologico per un grafo ciclico.

32 Lezione 35

32.1 Ordinamento Topologico

Dimostriamo ora che per ogni grafo aciclico esiste almeno un ordinamento topologico. Sia $G = (V, E)$ orientato e aciclico.

Dimostriamo che esista una permutazione π con π ordinamento topologico di G . Procederemo quindi con una dimostrazione costruttiva → Mostriamo quindi un modo per costruire tale permutazione⁴¹. È interessante notare che tale dimostrazione conterrà in sé anche l'idea di come implementare l'algoritmo.

Sfrutteremo le seguenti proprietà:

1. $G = (V, E)$ è aciclico $\iff \exists v \in V$ che ha zero archi entranti

NB: Esistono anche grafi ciclici che rispettano queste proprietà, ad es $[1] \rightarrow [2] \Rightarrow [3]$.

Questa proprietà è universale per tutti i grafi; quindi possiamo procedere per assurdo (non abbiamo una definizione induttiva).

Supponiamo che $\forall v \in V$ ci sia almeno un arco entrante; sia $v_1 \in V$, visto che ha almeno un grado entrante è lecito supporre che ci sia una sorgente $v_2 \neq v_1$ che arriva a v_1 (nel caso in cui $v_2 = v_1$ siamo nel caso di un grafo ciclico).

Analogamente per v_2 esiste una sorgente $v_3 \neq v_2 \neq v_1$ che arriva a v_2 . Ora visto che si sta lavorando su insiemi finiti, possiamo iterare questo procedimento fino al vertice v_n con $n = |V|$, ma anche questo dovrebbe avere un arco entrante che proviene dal vertice v_i con $1 \leq i < n$ rendendo tale grafo ciclico (abbiamo raggiunto il nostro assurdo e quindi dimostrato questa proprietà);

2. $G = (V, E)$ è aciclico e G' è sottografo di $G \implies G'$ è aciclico

Supponiamo per assurdo che G' sia un sottografo *ciclico* di un grafo G aciclico. Ma se G' è ciclico significa che esistono dei vertici v_1, \dots, v_k che formano un ciclo; ora essendo G' sottografo di G , quest'ultimo dovrà contenere tutti gli archi e i vertici di G' . Di conseguenza conterrà anche il ciclo formato dai vertici v_1, \dots, v_k e ciò è assurdo perché G è aciclico.

⁴¹Uno dei modi per dimostrare che qualcosa esista è mostrare come si costruisca.

Possiamo quindi procedere con la dimostrazione.

All'inizio della sequenza π ci dovrà essere un vertice v_1 che non ha archi entranti, altrimenti esisterebbe un vertice v con l'arco (v, v_1) che dovrebbe essere messo prima di v_1 in π . Tale vertice esiste per la proprietà 1, quindi per ora abbiamo $\pi = v_1$.

A questo punto, essendo v_1 primo, tutti i suoi archi andranno nella posizione giusta (i vertici adiacenti a v_1 saranno disposti dopo v_1 nella sequenza); quindi posso costruire un nuovo grafo:

$$G_1 = (V \setminus \{v_1\}), \exists \{(v_1, v) \in E \mid v \in V\}^{42}$$

Tale G_1 sarà aciclico per la proprietà 2 → Pertanto esiste un

$$v_2 \in V \setminus \{v_1\}$$

Con 0 archi entranti che può essere messo correttamente in seconda posizione

$$\pi = v_1 v_2$$

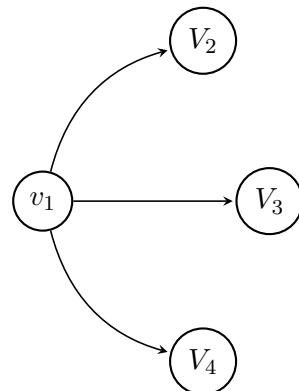
Da G_1 creiamo un nuovo grafo G_2 togliendo v_2 e i suoi archi uscenti. Tale ragionamento può essere iterato fino alla generazione di una permutazione $\pi = v_1 v_2 \dots v_k$ (ci fermiamo alla creazione di un sottografo vuoto) che sarà un ordinamento topologico di G .

32.2 Algoritmo del grafo entrante

Usando la procedura informale descritta precedentemente possiamo implementare un algoritmo che definisca un ordinamento topologico per un qualsiasi grafo aciclico. Questa idea la si può realizzare in tempo lineare su $|V| + |E|$ con delle piccole accortezze:

- ▶ Poniamoci nella situazione di dover cancellare v_1 nel seguente grafo

Invece di cancellare realmente il nodo v_1 possiamo sfruttare una struttura dati dove inserire il grado entrante di ogni vertice (nel nostro grafo → $v_1 = 1, v_2 = v_3 = v_4 = 2$); un vertice lo si cancella semplicemente decrementando di 1 il grado entrante dei suoi adiacenti ($v_2 = v_3 = v_4 = 0$, è come se avessi creato un sottografo $G_1 = (V = \{v_2, v_3, v_4\}, E = \emptyset)$). In questo modo possiamo evitare di dover generare i sottografi (risparmiando anche memoria)



⁴²In pratica eliminiamo il vertice v_1 e tutti gli archi che partono da essi.

- Possiamo evitare anche di dover cercare ogni volta una sorgente con grado entrante 0 semplicemente con una ricerca lineare sui vertici di V dove vado a salvarmi in una struttura datti (una coda fa proprio al caso nostro) tutti i vertici incontrati che non hanno archi entranti;

Senza queste ottimizzazioni avremmo dovuto creare un nuovo sottografo ogni volta che andassimo a cancellare un vertice, nel quale cercare un vertice con grado entrante 0. Un algoritmo di questo tipo avrebbe tempo:

$$\Omega(|V|^2 + |V||E|)^{43}$$

```

1 OrdinamentoTopologico(G)
2   Q = NIL
3   // Associamo ad ogni vertice un grado entrante
4   GradoEntrante(G, GE) // Dove GE è un array
5   // Inizializziamo la coda con i vertici che hanno grado entrante pari a 0
6   FOR EACH v in V DO
7     IF GE[v] = 0 THEN
8       Q = Accoda(Q, v)
9   WHILE Q != NIL DO
10    v = Testa(Q)
11    // v sarà il prossimo vertice nell'ordinamento topologico
12    print(v)
13    // Decido il grado entrante negli adiacenti
14    FOR EACH u in Adj[v] DO // Il contenuto di questo loop è Θ(1)
15      GE[u] = GE[u] - 1
16      // Controllo se aggiungerlo in coda
17      IF GE[u] = 0 THEN
18        Q = Accoda(Q, u)
19    Q = Decoda(Q)

```

Il ciclo `while` viene eseguito una sola volta per vertice, mentre il ciclo `for` al suo interno viene effettuato una sola volta per ogni vertice adiacente a v . Pertanto tutte le esecuzioni del ciclo `for` sono un $\Theta(|V| + |E|)$. Ci resta da vedere l'algoritmo di `GradoEntrante` e studiarne la sua complessità:

```

1 GradoEntrante(G, GE)
2   FOR EACH v IN V DO
3     GE[v] = 0
4   FOR EACH v IN V DO
5     FOR EACH u in Adj[v] DO
6       GE[u] = GE[u] + 1

```

È facile da notare che la complessità di questo algoritmo sulla dimensione del grafo:

⁴³Potrebbe essere addirittura cubico nel caso in cui $E = V \times V$.

$$\Theta(|V| + |E|)$$

Dunque possiamo concludere che il costo complessivo di **OrdinamentoTopologico** è proprio

$$\Theta(|V| + |E|)$$

32.3 Algoritmo con DFS

Al problema precedente esiste anche una soluzione che sfrutta la *DFS* → So che appena un vertice non ha più vincoli da soddisfare posso metterlo arbitrariamente alla fine.

Utilizziamo quindi un approccio simmetrico al precedente dove, grazie alla *DFS*, costruiamo l'ordinamento topologico dalla fine all'inizio; intuitivamente esiste sicuramente un vertice con grado entrante 0 (verifica speculare alla proprietà 1). Tale vertice andrà alla fine di π .

L'idea è quella di mettere, dopo aver messo i vertici con grado uscente 0, quelli con grado uscente pari ad 1 e così via⁴⁴

```

1 OT_DFS(G)
2   OT = NIL // Usiamo uno stack
3   Init(G) // Solita funzione di inizializzazione
4   FOR EACH v IN V DO
5     IF Color[v] = 'b' THEN
6       OT = OT_DFS_Visit(G, v)
7   RETURN OT // Letto dalla testa darà il nostro ordinamento

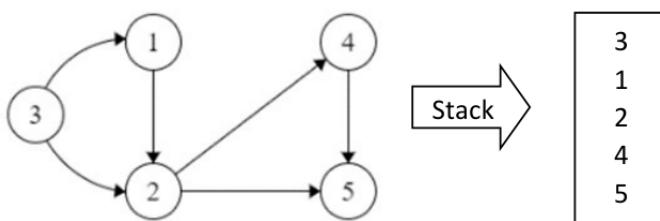
```

```

1 OT_DFS_Visit(G, v)
2   Color[v] = 'g'
3   FOR EACH u in Adj[v] DO
4     IF Color[u] = 'b' THEN
5       OT = OT_DFS_Visit(G, u)
6   OT = Push(OT, v)
7   Color[v] = 'n'
8   RETURN OT

```

Per il seguente grafo avremo i risultati aspettati:



Se leggiamo dall'alto verso il basso:
 $\pi = 3 \ 1 \ 2 \ 4 \ 5$

⁴⁴Si noti annerendo un vertice esso può essere messo prima di tutti gli altri già anneriti nell'ordinamento.

Verifichiamo che tale algoritmo sia corretto dimostrato che dato $G = (V, E)$ al termine di $\text{OT_DFS}(G)$, lo stack OT è tale che $\forall (v, u) \in E, v$ sta sopra u in OT (questa è la formulazione della proprietà di essere ordinamento topologico in una sequenza che va dall'alto verso il basso⁴⁵)

Si noti che v sta sopra in $OT \iff f[v] > f[u] \rightarrow$ Mettiamo sullo stack quando la DFS annerisce il vertice e quindi inizializza il suo tempo di fine visita (le istruzioni sono concomitanti nella DFS).

Dimostriamo allora che $\forall (v, u) \in E, f[v] > f[u]$ al termine di $\text{OT_DFS}(G)$.

Sappiamo che ogni arco verrà attraversato dalla DFS , quindi prendendo un arbitrario arco $(v, u) \in E$, quando la DFS lo attraversa posso avere 3 casi (v è sempre attualmente attivo, quindi grigio):

1. u è bianco $\rightarrow d[v] < d[u] < f[v] < f[u]$ per il teorema della struttura a parentesi
 - Questo caso è quindi verificato essendo $f[v] > f[u]$;
2. u è nero \rightarrow Significa che u è già terminato prima di v
 - Anche in questo caso $f[v] > f[u]$; più precisamente $d[u] < f[u] < d[v] < f[v]$
3. u è grigio \rightarrow Significa che nel grafo c'è un ciclo
 - Questo caso non può verificarsi in un grafo aciclico (l'unico caso problematico dove $d[u] < d[v] < f[v] < f[u]$)

32.4 Componenti fortemente connesse

Il concetto di componente fortemente connessa (in breve *CFC*) è definibile dal concetto di grafo fortemente connesso.

Un grafo $G = (V, E)$ è **fortemente connesso** $\iff \forall v, u \in V, v$ raggiunge u e u raggiunge b (un grafo è fortemente connesso se tutte le coppie di vertici sono reciprocamente raggiungibili).

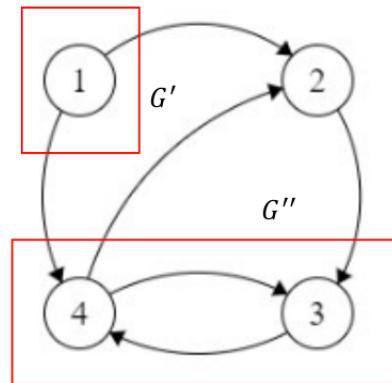
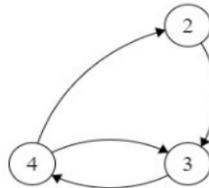
Una **componente fortemente connessa** è un sottografo massimale fortemente connesso di G (il più grande sottografo fortemente connesso di G , se prendiamo un sottografo più grande questo non è fortemente connesso).

⁴⁵ *stare prima* \equiv *stare sopra*.

Prendiamo, ad es, il seguente grafo G (non fortemente connesso)

Il sottografo G' è fortemente connesso, ma non è una CFC così come il sottografo G''

L'unica CFC di G (non è detto che ci sia un'unica CFC per ogni grafo) è la seguente:



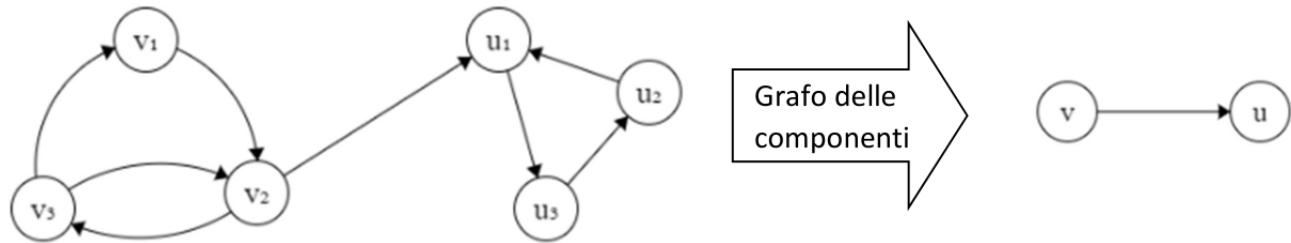
Ogni grafo contiene almeno una componente fortemente connessa; in particolare:

- ▶ Un grafo aciclico ha tante componenti fortemente connesse quanti sono i vertici
 - ❖ Poiché nessuna coppia di vertici è mutualmente raggiungibile, ogni vertice è una CFC;
- ▶ Una componente fortemente connessa ha un'unica CFC che è lei stessa.

Il concetto di componente fortemente connessa, più precisamente la mutua raggiungibilità (\sim) è una relazione di equivalenza:

- ▶ Riflessiva ($v \sim v$);
- ▶ Simmetrica ($u \sim v \implies v \sim u$);
- ▶ Transitività ($u \sim v \wedge v \sim t \implies u \sim t$).

Questo significa che se il mio interesse è solo la raggiungibilità posso far collassare tutti i vertici equivalenti di un grafo in un unico vertice; ovvero posso creare un grafo G' da G con vertici le CFC di G e tale grafo, detto **grafo delle componenti**, è equivalente in G per raggiungibilità (occupa meno spazio). Ad es:



NB: Il grafo delle componenti è sempre aciclico (se ad es ci fosse un ciclo tra 2 CFC v e u , queste collasserebbero in una singola CFC, poiché ogni vertice mutualmente raggiungibile con ogni vertice di u).

Dunque, dal concetto di *CFC* è possibile ridurre la complessità del grafo tramite il grafo delle componenti.

33 Lezione finale

33.1 Proprietà della CFC

1. Sia $C = (V', E')$ una *CFC* di $G = (V, E)$ (C sottografo di G) e siano $u, v \in V'$ → Ogni percorso tra u e v (in entrambe le direzioni) non può uscire da C ⁴⁶

La validità di tale proprietà è abbastanza naturale; supposto un percorso da u a v che passi attraverso $z \notin C$ si trova facilmente un assurdo:

u e v sono mutuamente raggiungibili → Esiste un percorso π_3 che va da v a u ed un percorso π_1, π_2 che va da u a v attraverso z .

Abbiamo che anche il percorso π_3, π_1 va da v a z , mentre il percorso π_2 va da z a v .

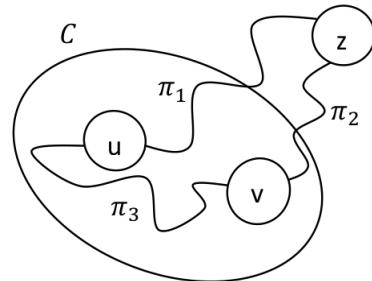
Quindi v e z sono mutualmente raggiungibili → z deve appartenere alla stessa componente per definizione

Questa proprietà ci permette di introdurre la successiva proprietà, che mette in relazione la *DFS* con le *CFC*:

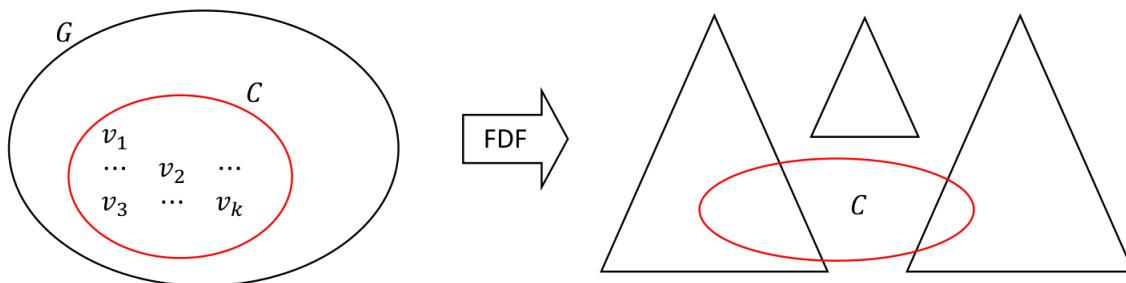
2. Dato $G = (V, E)$ ed eseguita una *DFS* arbitraria (può partire da qualsiasi sorgente) su G , avremo al termine che tutti i vertici di una qualche *CFC* saranno contenuti dentro lo stesso albero nella foresta *DF*.

Nota

Tale proprietà non garantisce che ogni albero sia una componente (potrei avere più *CFC* in uno stesso albero della *FDF*)



Dimostriamo anche questa proprietà per assurdo, supponendo che al termine di una *DFS*, si sia creata una foresta con 2 vertici della stessa *CFC* C in 2 alberi distinti



Per definizione sappiamo che ogni coppia di vertici in C è mutuamente raggiungibile (qualsiasi vertice può andare in ogni

⁴⁶Ogni percorso da u a v e viveversa è necessariamente contenuto nella componente.

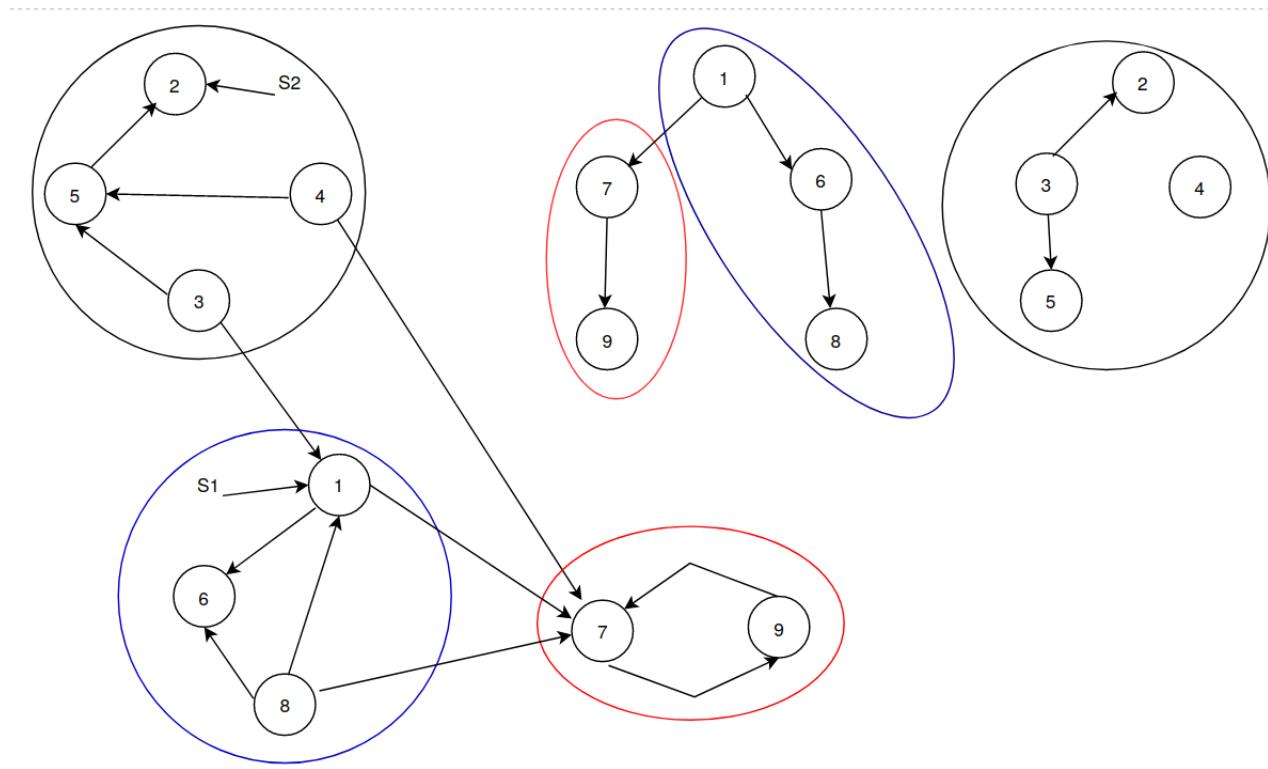
altro vertice). Ipotizziamo di eseguire una *DFS* su G : sappiamo che prima o poi visiterà il primo vertice in C . Possiamo dunque fare le seguenti osservazioni:

- ① Supponiamo che il primo vertice scoperto in C dalla *DFS* sia v_1 ;
- ② Sappiamo che esiste un percorso da v_1 ad ogni altro vertice di C ;
- ③ Ogni percorso da v_1 in C non può uscire da C (per la proprietà 1)

Dunque, possiamo dedurre che al tempo $d[v_1]$ tutti i vertici di C (escluso v_1) sono bianchi → abbiamo supposto che v_1 sia il primo vertice scoperto di C . Da questa conclusione (unita alle osservazioni ② e ③) segue che al tempo $d[v_1]$ esiste un percorso tutto bianco da v_1 a ciascun vertice di C .

Ciò significa che possiamo applicare il *teorema del percorso bianco* e concludere che ogni vertice di C diventerà discendente di v_1 nella *FDF* → Tutti i vertici di C saranno nello stesso albero di derivazione (in questo modo troviamo l'assurdo).

La proprietà 2 ci garantisce che la *DFS* preservi l'integrità delle *CFC*. Per una migliore comprensione forniamo un esempio di cosa succeda se si applica la *DFS* al seguente grafo:



Si noti che il grafo contiene 3 *CFC* ma la sua *DFS* è composta da soli 2 alberi.

Da questo esempio si comprende che la *DFS* non può garantire che ogni albero rappresenti una singola componente → Non possiamo quindi risolvere il problema delle componenti in maniera ovvia.

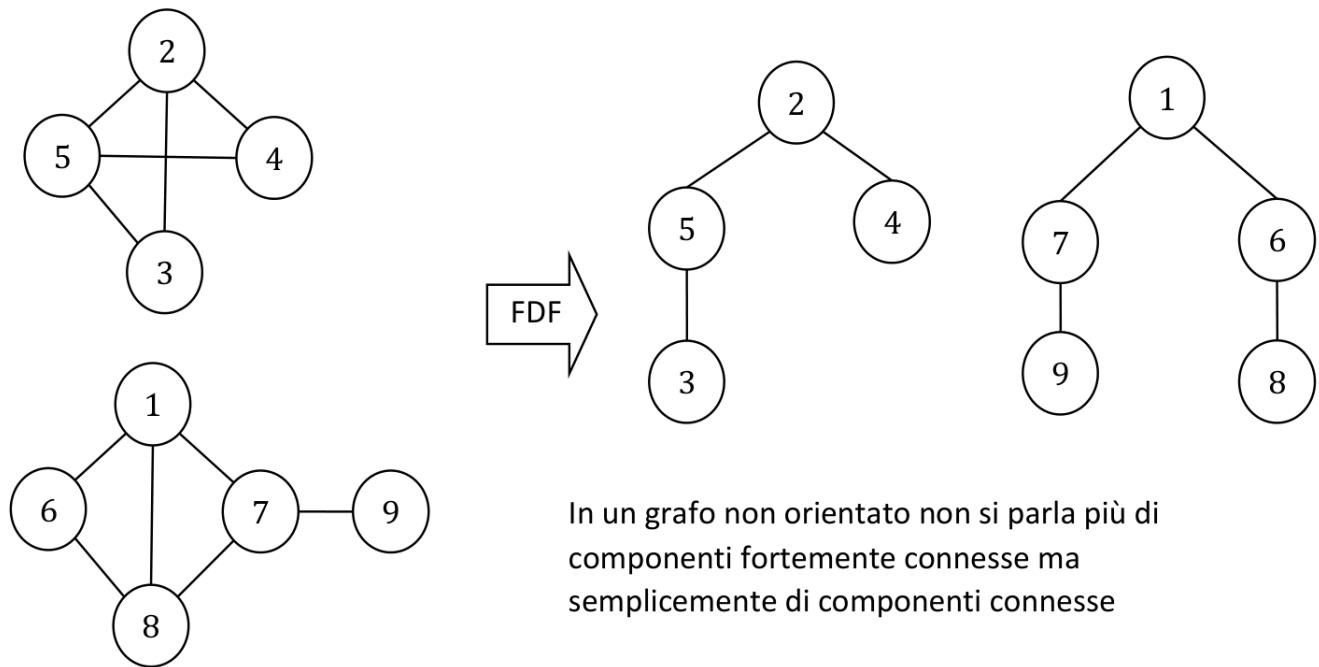
33.2 Calcolo delle CFC

Il problema da risolvere è quello di trovare le *CFC*; più precisamente i vertici che compongono ciascuna componente così da costruire il sottografo indotto dalle *CFC*.

Sappiamo che tutti i vertici di un certo albero della *FDF* sono raggiungibili dalla radice di quell'albero (proprietà della *FDF*, e degli alberi in generale). La sola informazione di raggiungibilità non basta per poter dire che un nodo faccia parte o meno di una determinata componente (basta pensare all'esempio precedente).

Se un nodo di un sottoalbero avesse un percorso formato di archi di ritorno verso la radice della componente *C* potremmo affermare che esso appartenga alla stessa *CFC* *C*. Ci resta da capire come ottenere tale informazione.

Esistono dei tipi di grafi dove la sola informazione della raggiungibilità basta a risolvere il problema del calcolo delle componenti → parliamo proprio dei grafi non orientati; infatti se *G* è un grafo non orientato allora u raggiunge $v \iff v$ raggiunge u (la raggiungibilità è sinonimo di mutua raggiungibilità); di conseguenza l'informazione che una *DFS* fornisce è sufficiente per la risoluzione del problema.



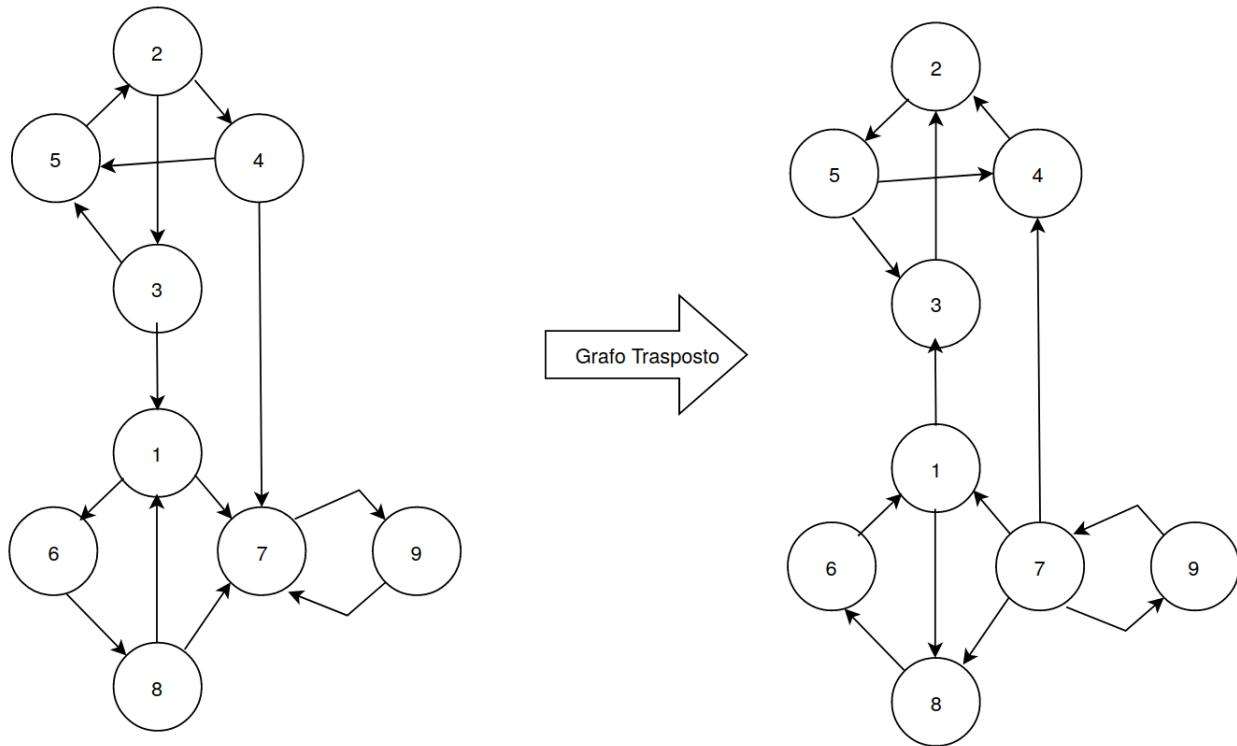
Ritornando al grafo orientato, dobbiamo risolvere il problema delle possibilità di avere più componenti in uno stesso albero. Dobbiamo quindi recuperare l'informazione della raggiungibilità in ambo i sensi.

Conoscere quali vertici possono raggiungere la radice (che raggiunge tutti i nodi del suo albero) potrebbe essere molto costoso se si usasse un algoritmo brute force.

Risolviamo il problema in tempo lineare grazie alla definizione di grafo trasposto:

Grafo trasposto

Per grafo trasposto si intende un grafo in cui tutti gli archi hanno direzione inversa rispetto a quelli del grafo originario



Se effettuassi una $\text{DFS_Visit}(G^T, 2)$ raggiungerei i vertici 5, 3, 4, ovvero tutti i vertici della *CFC* in cui 2 fa parte. Se in seguito ci fosse una $\text{DFS_Visit}(G^T, 1)$ otterremmo anche i vertici di cui 1 fa parte (3 lo trova nero per la visita precedente)

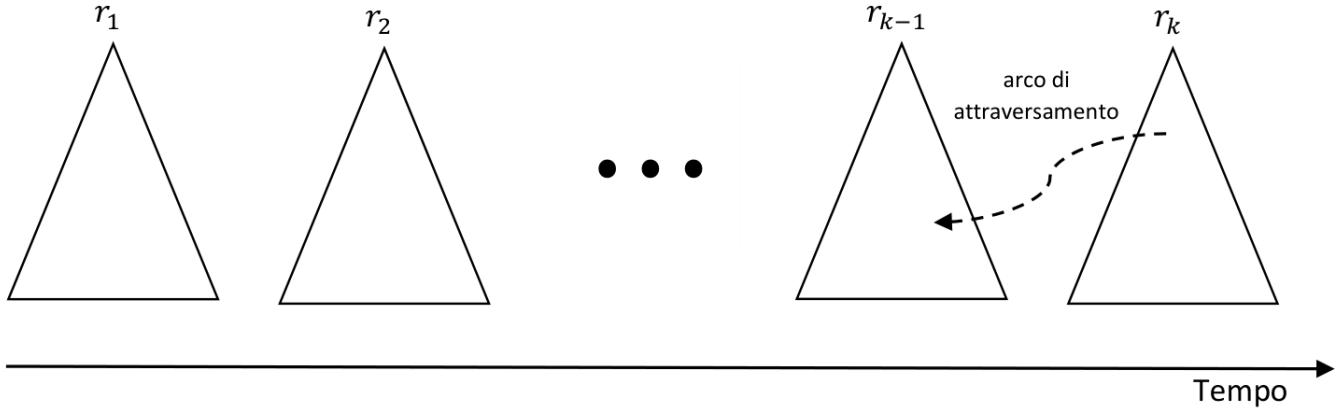
In generale, sia $G = (V, E)$, essendo:

$$G^T = (V^T, E^T) \text{ con } V^T = V \text{ e } E^T = \{(v, u) \mid (u, v) \in E\}$$

È chiaro che se in G esiste un percorso da v ad u allora lo stesso percorso letto nella direzione opposta è un percorso in G da u a v (e viceversa).

È importante ricordare che non è detto che i vertici raggiungibili nel grafo trasposto attraverso una *DFS* appartengano a priori alla stessa componente (ad es. nel caso precedente se partissimo da 1, l'albero generale conterebbe anche la *CFC* di 2) → Bisogna trovare un ordine di visita per i vertici in maniera tale da poter discriminare le diverse *CFC* del grafo.

Riassumendo, una *DFS* su G genererà la seguente *FDF*:



Per quanto riguarda la DFS su G^T (sapendo che per la proprietà 2 posso avere più CFC in uno stesso albero) dobbiamo evitare gli archi di attraversamento in modo tale da evitare di inserire nello stesso albero vertici di componenti diverse.

Non si può ovviamente eliminare gli archi di attraversamento in G^T (non si avrebbe più il grafo trasposto di G , e oltretutto dispendioso). L'unica soluzione è fare in modo che quando si esplori un arco di attraversamento esso risulti *innocuo* (il vertice di arrivo è nero).

Assumiamo che un arco (v, u) sia di attraversamento \rightarrow Sappiamo che nel momento in cui la DFS lo esplora, u sarà già nero e $d[u] < d[v]$ (per definizione di arco di attraversamento).

Pertanto gli archi di attraversamento che $DFS(G)$ può individuare possono andare solo su alberi già costruiti (nell'esempio da r_k a r_{k-1}). Dunque non esistono archi attraversamento che vadano da r_i a r_j con $i < j$.

Pertanto se nella $DFS(G^T)$ parto dalla radice r_k non potrò mai trovare un arco che mi porti ad un altro albero⁴⁷

Basta quindi visitare i vertici in $DFS(G^T)$ esattamente nell'ordine inverso di $DFS(G)$ in modo da risolvere, oltre al problema degli attraversamento, anche quello di avere più CFC nello stesso albero (l'arco di attraversamento resta *innocuo* e le CFC si suddivideranno ognuna in un albero distinto).

Descriviamo adesso l'algoritmo, che sarà evidentemente una sequenza di 2 DFS e la costruzione del grafo trasposto:

```

1 DFS1(G)
2   Init(G)
3   O = NIL // Stack dove andare a salvare i vertici nel giusto ordine
4   FOR EACH v in V DO
5     IF Color[v] = 'b' THEN
6       O = DFS1_Visit(G, v, O)
7   RETURN O
8
9 /* ----- */

```

⁴⁷ Si noti che nel grafo trasposto gli archi di attraversamento cambieranno di segno, quindi nell'esempio andrà da r_{k-1} a r_k .

```

10 DFS1_Visit(G, v, 0)
11   Color[v] = 'g'
12   FOR EACH u in Adj[v] DO
13     IF Color[u] = 'b' THEN
14       O = DFS1_Visit(G, u, 0)
15     Color[v] = 'n'
16     O = Push(O, v)
17   RETURN O

```

Utilizzeremo un array CFC[v] che ad ogni vertice assocerà un valore interno (i vertici con lo stesso valore apparterranno alla stessa CFC)

```

1 DFS2(Gt, 0)
2   Init(Gt)
3   c = 1
4   FOR EACH v in V DO
5     IF Color[v] = 'b' THEN
6       O = DFS2_Visit(Gt, v, c)
7
8   /* ----- */
9   DFS1_Visit(Gt, v, 0)
10  Color[v] = 'g'
11  CFC[v] = c
12  FOR EACH u in Adj_t[v] DO
13    IF Color[u] = 'b' THEN
14      DFS2_Visit(Gt, u, c)
15  Color[v] = 'n'

```

La seguente funzione costruirà il grafo trasposto:

```

1 GrafoTrasposto(G)
2   Vt = NIL
3   Et = NIL
4   // Vt = V
5   FOR EACH v IN V DO
6     Vt = add(Vt, v)
7   // Aggiungo gli archi
8   FOR EACH v IN V DO
9     FOR EACH u IN Adj[v] DO
10       // (v, u) appartiene a E
11       Et = add(Et, (u, v))
12       // Nel caso di liste avremo insert(Adj_t[u], v)
13   Gt = costruisciGrafo(Vt, Et)
14   RETURN Gt

```

Visto che l'algoritmo principale sarà un susseguirsi di algoritmi con complessità $\Theta(|V| + |E|)$, la sua complessità sarà esattamente $\Theta(|V| + |E|)$:

```
1 CalcoloCFC(G)
2   O = DFS1(G)
3   Gt = GrafoTrasposto(G)
4   DFS2(Gt, O) // Alla fine avremo CFC[v] = CFC[u]  $\iff$  v, u  $\in$  C (Stessa CFC)
```