

Algoritmi e Strutture Dati

A.A. 2022-2023

Docente: M. Benerecetti

Dispense tratte dal corso di Informatica a cura dello studente **S. Cerrone**

Si ringrazia **F. Formicola** per l'attenta revisione

Sommario

| | |
|---|-----------|
| 1. Introduzione | 6 |
| Programma del corso | 6 |
| Prerequisiti | 6 |
| <i>Sommatorie notevoli</i> | 6 |
| Nozioni di base: nozione di algoritmo; correttezza di un algoritmo | 6 |
| Macchina di Turing | 7 |
| Complessità degli algoritmi | 8 |
| Tempo di esecuzione e modello RAM | 8 |
| Un problema di conteggio | 9 |
| <i>Soluzione 1</i> | 9 |
| <i>Soluzione 2</i> | 10 |
| <i>Soluzione 3</i> | 10 |
| <i>Soluzione 4</i> | 10 |
| <i>Conclusioni</i> | 11 |
| 2. Notazione asintotica..... | 11 |
| Limite superiore asintotico..... | 11 |
| Limite inferiore asintotico | 12 |
| Limite asintotico stretto | 12 |
| Interpretazione dei limiti asintotici | 12 |
| Esempi sulla notazione asintotica | 12 |
| Proprietà dei limiti asintotici | 13 |
| <i>Esempio d'uso della proprietà transitiva</i> | 13 |
| Teorema sulla notazione asintotica..... | 14 |
| <i>Dimostrazione della terza implicazione</i> | 14 |
| <i>Esempi di applicazione del teorema</i> | 15 |
| <i>Esercizi</i> | 16 |
| Metodologia di risoluzione di una tipologia di esercizi | 16 |
| <i>Confutare una implicazione</i> | 16 |
| 3. Tempo di esecuzione e ricorrenze | 17 |
| Somma massima di una sottosequenza contigua | 17 |
| <i>Soluzione 1</i> | 17 |
| <i>Soluzione 2</i> | 19 |
| <i>Soluzione 3</i> | 19 |
| Approccio incrementale ed approccio ricorsivo..... | 21 |
| Un primo albero di ricorrenza | 21 |
| Forma generale di equazioni di ricorrenza con funzioni a un solo parametro | 21 |
| <i>Esempio 1: equazione di ricorrenza</i> | 22 |
| <i>Esempio 2: equazione di ricorrenza con input esponenziale</i> | 24 |
| <i>Esempio 3: chiamate ricorsive con input diversi</i> | 25 |
| <i>Esempio 4: limite superiore ed inferiore che crescono in maniera diversa</i> | 26 |
| 4. Algoritmi di ordinamento | 28 |
| Ordinamento di una sequenza | 28 |
| <i>Approccio 1</i> | 28 |
| <i>Approccio 2</i> | 29 |
| Insertion Sort | 29 |
| <i>Analisi</i> | 30 |

| | |
|--|-----------|
| Merge Sort..... | 32 |
| <i>Correttezza dell'algoritmo</i> | 33 |
| <i>Algoritmo Merge e occupazione in memoria</i> | 33 |
| <i>Tempo di esecuzione</i> | 34 |
| Selection Sort..... | 35 |
| <i>Alberi binari pieni</i> | 36 |
| <i>Alberi binari completi</i> | 37 |
| <i>Alberi heap</i> | 38 |
| HeapSort..... | 38 |
| <i>Analisi di HeapSort e algoritmo completo</i> | 42 |
| Quick Sort | 42 |
| <i>Partiziona</i> | 44 |
| <i>Analisi asintotica</i> | 45 |
| <i>Caso medio</i> | 47 |
| <i>Tecnica per validare un'equazione di ricorrenza</i> | 49 |
| <i>Maggiorazione di una sommatoria</i> | 50 |
| <i>Conclusioni su QuickSort</i> | 50 |
| Problema generale sull'ordinamento..... | 51 |
| <i>Alberi di decisione</i> | 51 |
| <i>Algoritmi di ordinamento e alberi di decisione</i> | 52 |
| <i>Dimostrazione del teorema sull'ordinamento</i> | 53 |
| 5. Strutture dati elementari..... | 55 |
| Struttura dati concreta e struttura dati astratta | 55 |
| <i>Operazioni su una struttura dati</i> | 55 |
| Array (non) ordinato..... | 56 |
| <i>Ricerca binaria</i> | 56 |
| Liste..... | 57 |
| <i>Definizione formale di lista e algoritmo di ricerca</i> | 58 |
| <i>Operazioni su liste</i> | 58 |
| Alberi binari | 60 |
| <i>Visite in profondità</i> | 61 |
| <i>Visita in ampiezza (BFS)</i> | 61 |
| <i>Memoria aggiuntiva</i> | 62 |
| 6. Alberi Binari di Ricerca (ABR) | 63 |
| Definizione di ABR | 63 |
| <i>Definizione ricorsiva</i> | 64 |
| Operazioni su alberi binari..... | 64 |
| <i>Ricerca</i> | 64 |
| <i>Inserimento</i> | 65 |
| <i>Ricerca del minimo e del massimo</i> | 66 |
| <i>Ricerca del successore e del predecessore</i> | 66 |
| <i>Cancellazione</i> | 68 |
| Alberi Bilanciati di Ricerca | 70 |
| <i>Alberi Perfettamente Bilanciati (APB)</i> | 70 |
| Alberi AVL | 71 |
| <i>AVL minimi</i> | 71 |
| <i>Relazione tra altezza e numero di nodi</i> | 72 |
| <i>Operazione di inserimento in AVL</i> | 74 |

| | |
|--|------------|
| Operazione di cancellazione in AVL | 79 |
| Alberi Red-Black..... | 81 |
| Dimostrazione che i RB hanno altezza logaritmica sul numero di nodi..... | 84 |
| Inserimento in un RB..... | 85 |
| Cancellazione nei RB..... | 88 |
| 7. Conversione Algoritmo Ricorsivo in Iterativo | 94 |
| Introduzione | 94 |
| Memoria nella ricorsione..... | 94 |
| Algoritmo iterativo del meccanismo di ricorsione | 95 |
| Struttura dell'algoritmo..... | 95 |
| Esempi di traduzione | 95 |
| PrintTree | 95 |
| Altezza | 96 |
| QuickSort | 97 |
| Count | 98 |
| Algoritmo sadico..... | 100 |
| Esercizio svolto..... | 101 |
| Ricorsione in coda | 103 |
| 8. Grafi..... | 105 |
| Definizioni sui grafi | 105 |
| Tipi di grafi..... | 105 |
| Grado di un vertice | 105 |
| Sottografo..... | 106 |
| Percorso | 106 |
| Raggiungibilità..... | 106 |
| Grafici ciclici e aciclici | 106 |
| Grafici connessi | 107 |
| Altri concetti | 107 |
| Rappresentazioni concrete di grafi..... | 107 |
| Matrice di adiacenza | 107 |
| Liste di adiacenza..... | 108 |
| Visita in ampiezza | 109 |
| Algoritmo BFS..... | 109 |
| Calcolo distanze e percorsi minimi | 110 |
| Correttezza della BFS | 111 |
| Algoritmo del percorso minimo | 114 |
| Visita in profondità | 114 |
| Algoritmo DFS..... | 115 |
| Terminazione e complessità | 115 |
| Teorema della struttura a parentesi..... | 116 |
| Foresta DF..... | 117 |
| Teorema del percorso bianco | 119 |
| Tipi di archi nella DFS..... | 120 |
| Verifica della ciclicità di un grafo..... | 121 |
| Ordinamento Topologico..... | 122 |
| Algoritmo del grado entrante..... | 123 |
| Algoritmo con DFS | 124 |
| Componenti Fortemente Connesse | 125 |

| | |
|---|------------|
| <i>Proprietà delle CFC</i> | 126 |
| <i>Calcolo delle CFC</i> | 128 |
| 9. Esercizi svolti | 132 |
| Regole utili per gli esercizi di analisi asintotica | 132 |
| <i>Derivate elementari e regole di derivazione</i> | 132 |
| <i>Proprietà logaritmi</i> | 132 |
| <i>Sommatorie notevoli</i> | 132 |
| Tutorato ASD | 132 |
| <i>Lezione 1 del 14/11/2022</i> | 133 |
| <i>Lezione 2 del 21/11/2022</i> | 135 |
| <i>Lezione 3 del 28/11/2022</i> | 136 |
| <i>Lezione 4 del 05/12/2022</i> | 138 |
| <i>Lezione 5 del 12/12/2022</i> | 140 |
| <i>Lezione 6 del 19/12/2022</i> | 142 |
| <i>Lezione 7 del 16/01/2023</i> | 145 |
| <i>Lezione 8 del 23/01/2023</i> | 148 |
| Svolti da me | 152 |
| <i>Esercizi assegnati dal tutor</i> | 152 |
| <i>Esercizio 1 del 21/07/2022</i> | 154 |
| <i>Esercizio 1 del 13/01/2021</i> | 155 |
| <i>Esercizio 1 del 04/03/2020</i> | 155 |
| <i>Esercizio 1 del 21/06/2022</i> | 156 |
| <i>Esercizio 2 del 21/06/2022</i> | 157 |
| <i>Esercizio 1 del 29/03/2019</i> | 158 |
| <i>Traccia B del 16/06/2021</i> | 158 |
| <i>Esercizio 3 del 29/03/2019</i> | 160 |
| <i>Esercizio 1 del 25/03/2022</i> | 161 |

1. Introduzione

Programma del corso

1) Definizione e analisi di algoritmi

- Correttezza
- Tempo di Esecuzione
 - Notazione asintotica
 - Tecniche di calcolo del tempo di esecuzione
esempi: algoritmi di ordinamento

2) Strutture dati per rappresentazione di insiemi di dati

- Lista/Array (non) ordinati
- Alberi binari
- Alberi binari di ricerca
- Alberi bilanciati

3) Grafi

- Tipi di grafi
- Rappresentazione di grafi
- Algoritmi sui grafi

Su docenti.unina.it trovate le lezioni frontali in “materiale didattico” nella cartella relativa ad [ASD1](#).

Su wpage.unina.it/benerece trovate informazioni sul corso, sugli esami e i lucidi delle lezioni.

Prerequisiti

- 1) **Analisi 1:** sommatorie, limiti, derivate e studio di funzioni
- 2) **Algebra:** funzione, relazione e loro proprietà, induzione (equivalente a ricorsione) e strutture algebriche (strutture dati)
- 3) **Programmazione 1:** Programma/Computazione, Strutture di controllo e Iterazione/Ricorsione

Sommatorie notevoli

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

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

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

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

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

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

Un metodo di risoluzione:

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

Nozioni di base: nozione di algoritmo; correttezza di un algoritmo

L'algoritmo è una descrizione non ambigua di una procedura. Sia A un algoritmo del problema P , con problema si intende l'insieme delle istanze $P = \{(I_1, O_1), \dots, (I_n, O_n)\}$ (un problema potrebbe essere la differenza tra numeri interi ed una sua istanza la differenza $5 - 3$); il nostro scopo sarà quello di definire una funzione $T_A: I \rightarrow \mathbb{R}^T$, ovvero che associa all'insieme delle istanze $I = (I_1, \dots, I_n)$ un valore numerico che

dovrebbe rappresentare quanto tempo impiega l'algoritmo a risolvere il problema. Questo tempo chiaramente cresce con la complessità delle istanze (ad esempio un conto è contare 10 coppie, un altro è contarne 10.000).

Cominciamo col dare una definizione più formale 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**. Indipendentemente dal fatto che le computazioni (la sequenza di passi) possano essere infinite (ad esempio, l'intervallo $[0,1]$ è una descrizione finita di un insieme infinito).

Alcune delle proprietà che un algoritmo deve avere sono:

- **Non ambiguità**: tutti i passi che definiscono l'algoritmo devono essere non ambigui e chiaramente comprensibili all'esecutore.
- **Generalità**: la sequenza di passi da eseguire dipende esclusivamente dal problema generale da risolvere, non dai dati che ne definiscono un'istanza specifica.
- **Correttezza**: un algoritmo è corretto se produce il risultato corretto a fronte di qualsiasi istanza del problema ricevuta in ingresso. Può essere stabilita, ad esempio, tramite:
 - dimostrazione formale (matematica);
 - ispezione informale.

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

- **Efficienza**: misura delle 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 permette di dire quante risorse deve usare la macchina per risolvere quella istanza).

Alcuni esempi sono:

- tempo di esecuzione;
 - memoria impiegata (che ovviamente è limitata a differenza del tempo);
 - altre risorse: banda di comunicazione.
- **Semplicità**: l'algoritmo deve essere facile da capire, modificare e mantenere.

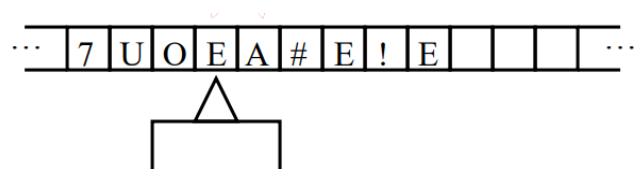
Macchina di Turing

Un noto modello computazionale è il modello della **Macchina di Turing**, esso, 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 calcolare tutto ciò che è calcolabile.

La Macchina di Turing è composta da:

- **Un nastro di lunghezza infinita**
 - In ogni cella può essere contenuta una quantità di informazione finita (un simbolo)
- **Una testina + un processore + programma**

Le operazioni che può compiere in **una unità di tempo** sono leggere o scrivere nella cella di nastro corrente e muoversi di 1 cella a sinistra, oppure di 1 cella a destra, oppure restare ferma.



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

Complessità degli algoritmi

La misura della complessità di una istanza è data dalla memoria utilizzata da quella istanza; ad esempio, per ordinare una sequenza di elementi, la memoria è chiaramente proporzionale al numero di elementi della sequenza. Quindi, più elementi ci sono e più tempo impiega l'algoritmo ad essere eseguito.

Di nostro interesse è analizzare asintoticamente quanto cresce la curva della funzione $T_A: I \rightarrow \mathbb{R}^T$ al crescere della complessità delle istanze (in altri termini, quanto velocemente arriva ad infinito).

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. Questo metodo è ovviamente poco preciso, poiché dipendente dal calcolatore e dal linguaggio di programmazione scelto.
- **Analisi asintotica:** questo tipo di analisi matematica trascende gli aspetti concreti (non misura direttamente il tempo) e assume che il calcolatore su cui misuriamo l'algoritmo possa effettuare esclusivamente operazioni elementari che impiegano la stessa unità di tempo (suddetta unità sarà la nostra misura di tempo).

Il nostro scopo è di dare delle misure precise per il comportamento di un algoritmo, anche se per alcuni di essi (atti a risolvere lo stesso problema) non basta conoscere la lunghezza dell'input ma è rilevante anche come la sequenza è impostata (ad esempio, per il problema dell'ordinamento, in alcuni algoritmi è rilevante anche se la sequenza è già ordinata, poco ordinata o completamente disordinata); in questi casi, va analizzato il caso migliore (ad esempio una sequenza ordinata per l'insertion sort) ed il caso peggiore. Da questi poi si può estrapolare il caso medio in base alla frequenza in cui si registrano i casi migliori e peggiori (questo lavoro ha senso solo dove il caso migliore ed il caso peggiore hanno un gap di prestazioni rilevante).

Tempo di esecuzione e modello RAM

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

- hardware su cui viene eseguito (velocità del calcolatore);
- compilatore/interprete utilizzato: compilatori differenti producono codice più o meno ottimizzato;
- altri fattori: casualità, etc.;
- **tipo e dimensione dell'input:** questa è anche l'unica dipendenza per noi ragionevole poiché è parte della definizione del problema.

Questo rende evidente che una misura di tempo (secondi) è inadatta alla misura del tempo di esecuzione (sulla bontà dell'algoritmo). 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 (e quello che useremo per la nostra analisi) è rappresentato dal modello computazionale RAM che, a differenza della Macchina di Turing, ha accesso diretto (la MT ha accesso sequenziale). Il modello RAM è una semplificazione dei moderni computer.

Il Modello RAM (Random Access Memory) 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 ad ogni cella di memoria (più efficiente della MT).
- **Singolo processore + programma**
 - In 1 unità di tempo: operazioni di lettura, passo di computazione elementare, scrittura;
 - Passi di computazione: addizione, moltiplicazione, assegnamento, confronto, accesso a puntatore, etc.

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, j) tali che $i, j \in \mathbb{N}$ e $1 \leq i \leq j \leq N$.

Esempio:

- **Input:** $N = 4$
 - $(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4)$
- **Output:** 10

Soluzione 1

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

La correttezza di questo algoritmo è ovvia, però è anche abbastanza intuitivo che questo sia il modo peggiore di poter risolvere il precedente problema, poiché genera tante coppie che non servono ai fini del conteggio delle coppie “esatte”.

Analizziamo la complessità di ogni linea contandone le operazioni elementari:

- 2) Operazione di lettura: 1 operazione elementare
- 3) Assegnamento di i e confronto con N : 2 operazioni elementari
- 4) Assegnamento di j e confronto con N : 2 operazioni elementari
- 5) Due letture ed un confronto: 3 operazioni elementari
- 6) Una somma: 1 operazione elementare
- 7) Scrittura: 1 operazione elementare

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

L'analisi precedente non basta, ovviamente, a descrivere il comportamento dell'algoritmo, poiché 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, sarà $1 \cdot 1 = 1$). Analizziamo meglio le linee:

- 3) Si ripete $n + 1$ ($\sum_{i=1}^{n+1} 1$) volte (l'ultimo confronto è quello usato per uscire dal ciclo), quindi il contributo totale della linea è $2(n + 1)$
- 4) Si ripete $n + 1$ volte per ogni volta che si ripete il for precedente, quindi $\sum_{i=1}^n (\sum_{j=1}^{n+1} 1)$, ed il contributo totale sarà $2(n + 1)n$
- 5) Si ripete $\sum_{i=1}^n (\sum_{j=1}^n 1) = n^2$ volte, dunque: $3n^2$
- 6) Sicuramente si ripete un numero di volte compreso tra 0 e n^2 , quindi il contributo totale della linea è $0 \leq x \leq n^2$

Ora non ci resta che analizzare la complessità totale: $T_1(N) = 1 + 2(n + 1) + 2n(n + 1) + 3n^2 + x + 1 = 1 + 2n + 2 + 2n^2 + 2n + 3n^2 + x + 1 = 5n^2 + 4n + 4 + x$. Si noti 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.

Soluzione 2

Poiché una coppia è utile ai fini del nostro problema 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 = i 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 | $\sum_{i=1}^n \left(\sum_{j=i}^{n+1} 1 \right) = \sum_{i=1}^n (n - i + 2) = \sum_{i=1}^n n - \sum_{i=1}^n i + \sum_{i=1}^n 2$ $= n^2 - \frac{n(n+1)}{2} + 2n = \frac{n^2}{2} + \frac{3}{2}n$ | $n^2 + 3n$ |
| 5 | 1 | $\sum_{i=1}^n \left(\sum_{j=i}^n 1 \right) = \sum_{i=1}^n (n - i + 1) = n^2 - \frac{n(n+1)}{2} + n = \frac{n^2}{2} + \frac{n}{2}$ | $\frac{n(n+1)}{2}$ |

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

Come la soluzione precedente, anche questo algoritmo ha tempo di esecuzione quadratico, dunque possiamo considerarli equivalenti dal punto di vista asintotico. Ciò non significa che impieghino lo stesso tempo (ovviamente $1,5n^2 < 5n^2$) ma nel lungo periodo (al crescere dell'istanza) il loro "peggioramento" segue lo stesso andamento. Poiché dal punto di vista algoritmico non ci sono differenze, non abbiamo apportato nessun reale miglioramento.

Soluzione 3

Dalla precedente analisi abbiamo osservato che, fissato i , eseguiamo l'istruzione di incremento del risultato tante volte quanto il valore totale (che abbiamo pure calcolato) da aggiungere a RIS; ma quindi risulta logico aggiungere direttamente il risultato totale, scrivendo il seguente algoritmo:

```

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

```

Diagramma di annotazione per la complessità:

- Linea 2: 1
- Linea 3: $2n + 2$
- Linea 4: $\sum_{i=1}^n 5 = 5n$
- Linea 5: 1

$T_3(N) = 7n + 4$; questo tipo di funzione cresce linearmente e quindi è nettamente migliore rispetto ai precedenti algoritmi poiché, per la stessa crescita di N , quest'ultima soluzione cresce molto più lentamente.

Soluzione 4

In realtà, il problema descritto è talmente semplice da poter ridurre ulteriormente il tempo di esecuzione. Infatti, sappiamo che per $i = 1$ il risultato è $N - 1 + 1$, per $i = 2$ bisogna aggiungere $N - 2 + 1$, e così via fino ad N a cui dobbiamo sommare 1 ai precedenti risultati; ma allora $RIS = \sum_{i=1}^N i = \frac{n(n+1)}{2}$.

Dunque, il nostro algoritmo si riduce alle seguenti poche righe di codice

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

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

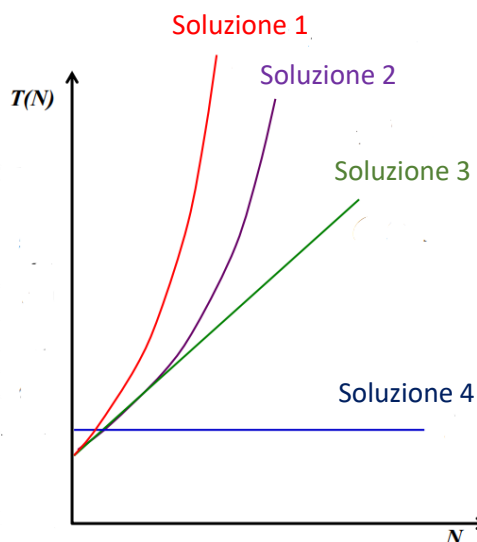
Era possibile arrivare a questa soluzione anche geometricamente: infatti, le coppie possono essere

considerate come celle di una matrice quadrata $\begin{pmatrix} 1 & \dots & N \\ \vdots & \ddots & \vdots \\ N & \dots & N \end{pmatrix}$ dove le righe rappresentano il primo valore della coppia e le colonne il secondo. Le celle totali (e dunque tutte le coppie possibili) sono $N \times N$ ma quelle che ci interessano sono tutte le celle sopra la diagonale principale (diagonale compresa).

Quindi essendo N elementi nella diagonale, sopra si trovano $\frac{N^2 - N}{2}$ elementi: ne segue che il numero di coppie che rispettano i requisiti del problema sono $\frac{N^2 - N}{2} + N = \frac{N(N+1)}{2}$.

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 gior. | 32 anni | 32 millenni |

2. Notazione asintotica

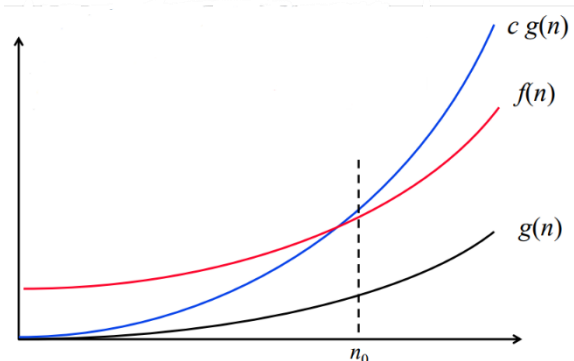
Limite superiore asintotico

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: $f(n) = O(g(n))$; quindi f o cresce più lentamente oppure cresce allo stesso modo di g (la si può considerare una versione più permissiva dell'"o-piccolo" vista in analisi 1).

$$f(n) = O(g(n)) \equiv$$

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

$g(n)$ è detto un limite superiore asintotico di $f(n)$.



Si noti come il confronto asintotico ignora gli intervalli limitati (da 0 a n_0), considerando rilevanti solo gli intervalli illimitati (a partire da n_0 fino a ∞).

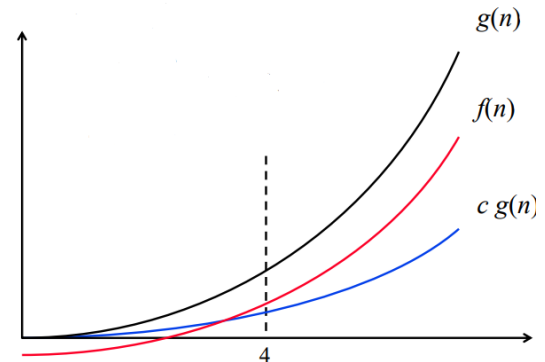
Di base, più è piccolo n_0 e più dovrà essere grande la costante c , che nel caso del limite superiore asintotico sarà un valore maggiore di 1, poiché il suo scopo è quello di far “superare” la funzione f dalla funzione g .

Limite inferiore asintotico

Diremo che una funzione $f(n)$ è in relazione “Omega grande” con una funzione $g(n)$ (oppure $g(n)$ è detto limite inferiore asintotico di $f(n)$) se $f(n)$ non cresce meno velocemente di $g(n)$, in simboli: $f(n) = \Omega(g(n))$.

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

La costante c è usualmente un valore compreso in un intervallo tra 0 e 1.



Inoltre, è fondamentale che le costanti abbiano valore maggiore di 0 (come esprime la definizione), altrimenti significa che quel limite asintotico non è valido.

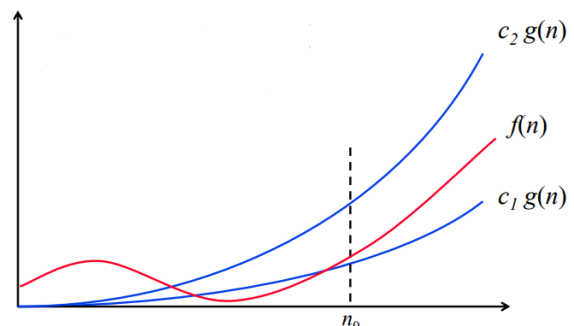
Limite asintotico stretto

Diremo che una funzione $f(n)$ è “Theta” di $g(n)$ se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ (si dice anche che $g(n)$ è un limite asintotico stretto di $f(n)$).

Quindi, se valgono entrambe (le due funzioni crescono allo stesso modo), si avrà la seguente relazione asintotica:

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

Si noti che n_0 è il valore maggiore tra gli n che soddisfano la relazione di Omega grande e O grande, in quanto è evidente che la condizione continua ad essere soddisfatta per un valore maggiore di n (infatti in ogni 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 due costanti con g).



Interpretazione dei limiti asintotici

Usiamo la notazione asintotica per dare un limite ad una funzione $f(n)$, a meno di un fattore costante c .

Ma 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 due costanti per cui, a partire da un n_0 , continua a valere la precedente relazione.

Dal punto di vista computazionale significa che a partire da un certo numero di istanze (n_0 , e da qui è evidente perché bisogna scegliere un valore positivo: non ha senso un algoritmo con nessun input), se si fanno girare i due algoritmi (ovvero le funzioni) su calcolatori con potenza differente (la differenza di velocità dei calcolatori è rappresentata dalle costanti c_1 e c_2), essi termineranno allo stesso momento.

Esempi sulla notazione asintotica

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

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

Ne segue che $f(n) = \Theta(g(n))$. Si noti come in quest'esempio abbiamo usato il buon senso per scegliere le costanti, ma come vedremo in seguito ci sono metodi ben precisi per tale scopo.

- 2) Sia $f(n) = n^2$ e $g(n) = 2n$. È facile notare che le due funzioni non sono in relazione Theta tra di loro, e quindi che non crescono allo stesso modo. Infatti, si dovrebbe avere $c_1 2n \leq n^2 \leq c_2 2n$; la prima relazione è banale da rendere vera e quindi risulta $f(n) = \Omega(g(n))$.

Supponiamo ora che esista un c_2 che soddisfi $n^2 \leq c_2 2n$, ma ciò significa che $c_2 \geq \frac{n^2}{2n}$ e quindi $c_2 \geq \frac{n}{2}$, ed è assurdo che esista una costante maggiore o uguale ad una funzione crescente che tende ad infinito per ogni n (è possibile solo trovare una costante che fino ad un certo punto sia maggiore di tale funzione).

Proprietà dei limiti asintotici

- **Transitività:** $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

Dimostrazione: Per ipotesi abbiamo $\exists n_1 > 0, \exists c_1 > 0 : \forall n > n_1, f(n) \leq c_1 g(n)$ e, analogamente, $\exists n_2 > 0, \exists c_2 > 0 : \forall n > n_2, g(n) \leq c_2 h(n)$, ma allora, scegliendo $n_0 = \max\{n_1, n_2\}$ risulta che $\exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n > n_0, f(n) \leq c_1 g(n) \wedge g(n) \leq c_2 h(n)$. Dunque, si ha, evidentemente, che $c_1 g(n) \leq c_1 c_2 h(n)$; la tesi si raggiunge scegliendo proprio $c = c_1 c_2$, ovviamente positivo; infatti, si ha: $\exists n_0 > 0, \exists c > 0 : \forall n > n_0, f(n) \leq c h(n)$.

Questa proprietà vale anche per il limite inferiore asintotico e il limite asintotico stretto.

- **Simmetria:** $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Dimostrazione: $\exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n > n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ipotesi. 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 $\frac{1}{c_2} f(n) \leq g(n)$. Ma allora, siano $c'_1 = 1/c_2$ e $c'_2 = 1/c_1$ (entrambi sono ovviamente positivi), possiamo scrivere come segue: $\exists n_0 > 0, \exists c'_1, c'_2 > 0 : \forall n > n_0, c'_1 f(n) \leq g(n) \leq c'_2 f(n)$. Come volevasi dimostrare.

Questa proprietà è valida solo per la relazione Theta.

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

Si lascia la dimostrazione di questa proprietà come esercizio.

Esempio d'uso della proprietà transitiva

La proprietà transitiva risulta essere utile quando bisogna confrontare due 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 sono arrivato a nulla).

Ad esempio, consideriamo $f(n) = n^2 - 3n + 4$ e $h(n) = 2n^2 + 7n - 10$, lo scopo è scegliere una funzione $g(n)$ affinché $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n))$ così da avere per transitività $f(n) = \Theta(h(n))$. Supponiamo $g(n) = n^2$ e dimostriamo che $c_1 n^2 \leq n^2 - 3n + 4 \leq c_2 n^2$: scegliendo $c_2 = 1$ abbiamo $n^2 - 3n + 4 \leq n^2 \Leftrightarrow -3n + 4 \leq 0$ che è verificata per ogni $n \geq \frac{4}{3}$; mentre, per $c_1 = \frac{1}{2}$ si ha $\frac{n^2}{2} \leq n^2 - 3n + 4 \Leftrightarrow 0 \leq \frac{n^2}{2} - 3n + 4 \Leftrightarrow 3n - 4 \leq \frac{n^2}{2}$, verificata per ogni $n \geq 1$. Analogamente si tratta il caso $g(n) = \Theta(h(n))$ e di conseguenza si ha anche $f(n) = \Theta(h(n))$.

Questo esempio è banale poiché era evidente già dall'inizio che le due funzioni crescevano allo stesso modo, ma per funzioni esponenziali o logaritmiche potrebbe essere non tanto banale e quindi, probabilmente, risulterà utile applicare la proprietà transitiva come in questo esempio.

Teorema sulla notazione asintotica

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

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

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

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

Il precedente teorema è un utile metodo per dimostrare i limiti asintotici senza dover preoccuparsi di calcolare le costanti, inoltre si potrebbe sfruttare anche la proprietà transitiva così da semplificare il limite, ad esempio $\lim_{n \rightarrow \infty} \frac{n^2-8n}{n^2-7}$ diventa $\lim_{n \rightarrow \infty} \frac{n^2-8n}{n^2} \wedge \lim_{n \rightarrow \infty} \frac{n^2}{n^2-7}$.

Dimostrazione della terza implicazione

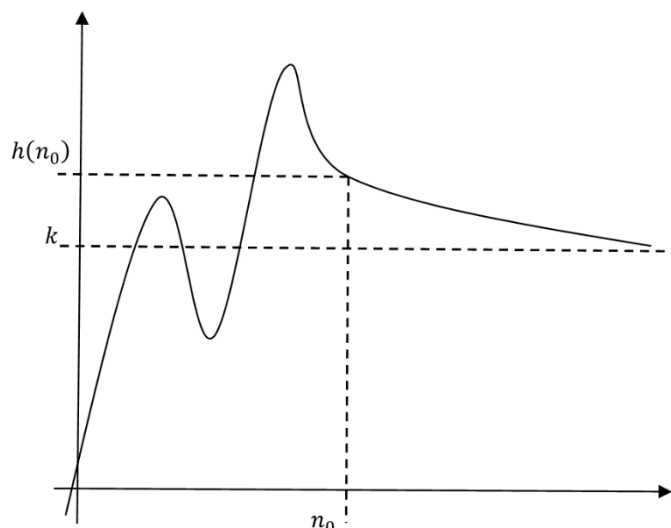
Se il $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$ con k costante, è intuitivo che nessuna funzione cresce più o meno dell'altra ma si mantengono ad una distanza proporzionale a k (se una cresce più velocemente dell'altra allora il limite del rapporto è 0 oppure ∞). Partiamo dalla definizione di Θ : $\exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$; e supponiamo che $g(n)$ ad un certo punto sia sempre positivo (supposizione più che realistica poiché è 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 $c_1 \leq h(n) \leq c_2$, ne segue che, essendo $h(n)$ sempre compreso tra due costanti positive, $\lim_{n \rightarrow \infty} h(n) = k$ con $c_1 \leq k \leq c_2$ (in modo del tutto analogo si trattano le altre due implicazioni).

A questo punto ci resta da 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 quindi che k sia un asintoto orizzontale per il rapporto $h(n) = \frac{f(n)}{g(n)}$. Per tale asintoto esistono soli due casi: $h(n)$ tende a k dall'alto, oppure tende a k dal basso (esiste anche un terzo caso, quello oscillatorio, che va trattato in maniera leggermente diversa).

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 decresca). 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 $\underbrace{k}_{c_1} \leq h(n) \leq \underbrace{h(n_0)}_{c_2}$.

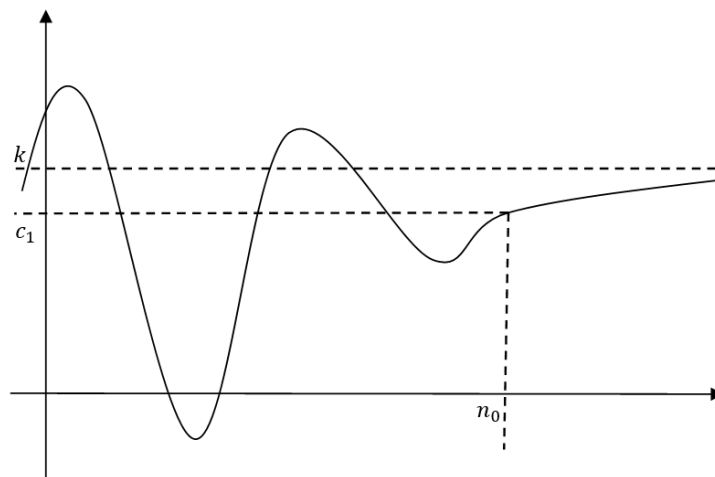
Le precedenti costanti sono state trovate avendo supposto che il rapporto $\frac{f(n)}{g(n)}$ esista e che la funzione (dopo lo studio della derivata prima di tale rapporto) sia decrescente per un intervallo $[n_0, +\infty)$.



Caso $h(n)$ dal basso:

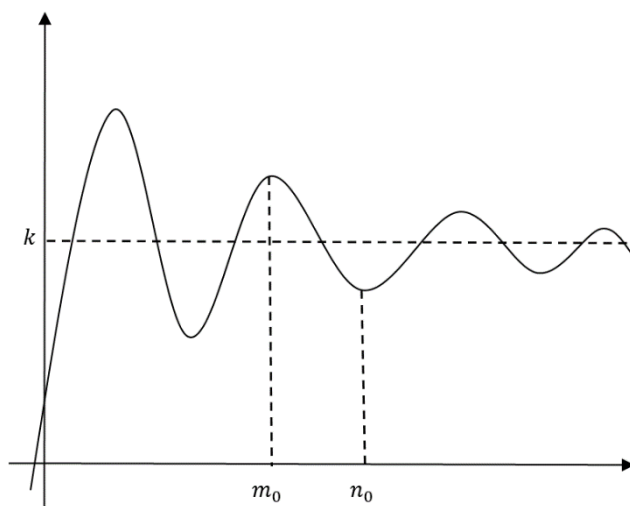
Supposto che lo studio della derivata prima abbia portato alla conferma dell'esistenza di un intervallo per cui la funzione è sempre crescente abbiamo praticamente già trovato le nostre costanti. Infatti, per c_2 è ovvio che basti prendere l'asintoto k , mentre per c_1 bisogna scegliere un n_0 per cui 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 alla nostra definizione di Θ). Una volta determinato n_0 allora $c_1 = h(n_0)$.



Caso oscillatorio:

Se la situazione è quella rappresentata 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 un po' più complicato. Infatti, bisogna scegliere un punto di minimo locale (n_0) ed uno di massimo locale (m_0) in modo 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 due. Una volta scelti questi due punti le costanti saranno $c_1 = h(n_0)$ e $c_2 = h(m_0)$.



Questo tipo di funzioni sono molto rare, nei nostri studi ci imbattemmo sempre e solo nei primi due casi.

Esempi di applicazione del teorema

Notare che $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Leftrightarrow \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$ e $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0 \Leftrightarrow \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{k} > 0$; 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 $g(n) = 4n - 10$. Avremo $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} 4 - \frac{10}{n} = 4 \Rightarrow g(n) = \Theta(f(n)) \Rightarrow f(n) = \Theta(g(n))$. Ma andiamo a stabilire anche le costanti per tale relazione: sia $h(n) = 4 - \frac{10}{n}$ andiamo a farne la derivata: $\frac{d}{dn} \left(4 - \frac{10}{n} \right) = \frac{d}{dn} 4 - \frac{d}{dn} \frac{10}{n} = -10 \frac{d}{dn} n^{-1} = \frac{10}{n^2}$; quindi questa funzione è sempre crescente (essendo la derivata positiva); scegliamo ora un n_0 per cui la funzione sia sempre crescente e positiva (se ad esempio scegliessimo 1 avremmo $h(1) < 0$): sia $n_0 = 3$ da cui ricaviamo $c_1 = h(3) = \frac{2}{3}$ mentre $c_2 = 4$. Dunque, $\forall n \geq 3$, $4f(n) \leq g(n) \leq \frac{2}{3}f(n)$ (si ricorda che abbiamo studiato il rapporto $\frac{g(n)}{f(n)}$), ovvero, $\frac{3}{2}g(n) \leq f(n) \leq \frac{1}{4}g(n)$.

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

N.B.: in questi semplici esempi la funzione è stata sempre crescente o sempre decrescente in tutto l'intervallo $(-\infty, +\infty)$, ma nella maggior parte dei casi non è così; quindi, bisogna sempre specificarne l'intervallo. Esempio: la funzione è crescente nell'intervallo $[7, +\infty)$.

Esercizi

Date le seguenti funzioni, verificarne la relazione ed estrarre le costanti:

- $f(n) = 10n + 25n^2 \wedge g(n) = 8n^2$
- $f(n) = n \log n + n - 1 \wedge g(n) = n \log n$
- $f(n) = n \log n \wedge g(n) = n + 1$

I precedenti esercizi sono una topologia di esercizio presente nella prova d'esame (ovviamente non saranno così semplici). Inoltre, se la base del logaritmo non viene esplicitamente specificata significa che la base è 2.

Metodologia di risoluzione di una tipologia di esercizi

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

Per dimostrare la precedente implicazione sfrutteremo la proprietà di monotonicità dei logaritmi, ovvero $x \leq y \Rightarrow \log x \leq \log y$.

Da $f(n) = \Theta(g(n))$ per definizione $\exists n_0 > 0, \exists c_1, c_2 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$, ma quindi $\log(c_1 g(n)) \leq \log(f(n)) \leq \log(c_2 g(n))$. A questo punto sfruttiamo un'altra proprietà dei logaritmi ($\log(xy) = \log x + \log y$): **$\log c_1 + \log(g(n)) \leq \log(f(n)) \leq \log c_2 + \log(g(n))$** (ciò ovviamente non basta; inoltre, se c_1 e/o c_2 sono compresi tra 0 e 1 il logaritmo sarà addirittura negativo).

Ma esiste la seguente proprietà: $\forall h(n), \log h(n) + k = \Theta(\log h(n))$ (ovviamente le costanti non sono le stesse per qualsiasi k , ma a noi interessa solo avere la certezza che tali costanti esistano $\forall k \in \mathbb{Z}$).

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 precedenti definizioni:

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

Da cui, per la transitività del minore ed uguale:

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

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

Confutare una implicazione

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

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

Ciò è 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 $\forall n \geq 1, \frac{1}{2} 2n \leq n \leq 1 \cdot 2n$. Ora sia la nostra funzione $h(n) = 2^n$. Risulta $2^{f(n)} = 2^n$ e $2^{g(n)} = 2^{2n}$, ma calcolando il limite di tale rapporto:

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

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

3. Tempo di esecuzione e ricorrenze

Somma massima di una sottosequenza contigua

Descriviamo un algoritmo che accetta come input una certa sequenza di numeri $A = (a_1, a_2, \dots, a_n)$, 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 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 $V = \sum_{k=i}^j a_k$ 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: $8 + 3 - 5 + 4 + 6 = 16$

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 proprietà vista per il [CONTACOPPIE](#): ne consegue che il numero di sottosequenze contigue, compresa quella vuota, è pari a $\frac{N(N+1)}{2} + 1$.

Quindi abbiamo decomposto il problema in:

- Generare tutte le sottosequenze (risolto sopra)
- 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 = i TO N DO
5              SUM = 0
6              FOR k = i TO j DO
7                  SUM = SUM + A[k]
8              IF SUM > V THEN
9                  V = SUM
10     RETURN V
```

Questo è l'unico blocco che differisce dalla [soluzione 2](#) svolta per il CONTACOPPIE.

Ci si aspetta che questo algoritmo sia almeno un $\Theta(n^2)$ vista la similitudine con la [soluzione 2](#) di CONTACOPPIE; più precisamente se il blocco evidenziato risulta essere costante allora $T(N) = \Theta(n^2)$, altrimenti sarà ovviamente maggiore.

Possiamo sfruttare 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 solo le linee da 5 a 9:

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

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

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

Restato le linee 6 e 7, ma si può notare che una volta fissato i e j la differenza tra la testa del for ed il corpo è 1 (praticamente il for viene eseguito una volta in più del corpo dovendo eseguire la condizione di uscita), e tale differenza avviene ogni volta che genero una coppia (i, j) ; in altri termini, 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 ci cambia nulla dal punto di vista asintotico. Dunque, possiamo analizzare solo la linea 7 (che ha un contributo singolo di 4) e viene eseguita il seguente numero di volte:

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

$$\begin{aligned} & \sum_{i=1}^N \left(\frac{\sum_{j=i}^N j}{2} - \frac{(i-1)i}{2} - \frac{\sum_{j=i}^N i}{i(N-i+1)} + \frac{\sum_{j=i}^N 1}{N-i+1} \right) = \\ & = \sum_{i=1}^N \frac{N(N+1)}{2} - \sum_{i=1}^N \frac{i^2}{2} + \sum_{i=1}^N \frac{i}{2} - \sum_{i=1}^N iN + \sum_{i=1}^N i^2 - \sum_{i=1}^N i + \sum_{i=1}^N (N-i+1) = \\ & \frac{N^2(N+1)}{2} - \frac{1}{2} \cdot \frac{N(N+1)(2N+1)}{6} + \frac{1}{2} \cdot \frac{N(N+1)}{2} - \frac{N^2(N+1)}{2} + \frac{N(N+1)(2N+1)}{6} - \frac{N(N+1)}{2} + \frac{N(N+1)}{2} = \\ & = \frac{1}{2} \left(\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} \right) + \frac{1}{2} \left(\frac{N^2}{2} + \frac{N}{2} \right) = \frac{1}{6} N^3 + \frac{1}{2} N^2 + \frac{1}{3} N \end{aligned}$$

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

Soluzione 2

Un notevole miglioramento al precedente algoritmo può essere attuato notando che molti calcoli sono fatti più volte, sprecando tempo. Ed il problema è legato al calcolo di una sottosequenza data (l'ultimo for della soluzione precedente): infatti, si può osservare che $\sum_{k=i}^j A[k] = \sum_{k=i}^{j-1} A[k] + A[j]$.

Ma $\sum_{k=i}^{j-1} A[k]$ è praticamente la quantità già presente in SUM ed è quindi già stata calcolata; dunque, $\sum_{k=i}^j A[k] = \text{SUM} + A[j]$ ed è evidente che una linea di questo tipo ha contribuito costante (abbiamo tratto vantaggio delle iterazioni precedenti). Ovviamente la suddetta osservazione non può essere usata così com'è, poiché bisogna prima capire quando azzerare SUM e sotto quali condizioni si può sfruttare il precedente algoritmo.

Fintanto che l'indice i rimane sempre lo stesso si vuole aggiornare il precedente valore di SUM, mentre, quando cambia i allora SUM va azzerato poiché siamo in una nuova sottosequenza. Questa idea può essere formalizzata dal seguente algoritmo:

```

int MAXSUM(A,N) -----  $\Theta(1)$ 
  V = 0
  FOR i = 1 TO N DO } -----  $\Theta(n)$ 
    SUM = 0
    FOR j = i TO N DO } -----  $\Theta(n^2)$ 
      SUM = SUM + A[i]
      IF SUM > V THEN
        V = SUM
  RETURN V -----  $\Theta(1)$ 

```

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

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, in quanto il numero di queste ultime in una sequenza è di per sé quadratico.

Ma fortuna vuole che il nostro algoritmo non richieda il calcolo di tutti i valori: è stata una nostra scelta usare quest'approccio (l'algoritmo ottimale di un problema non è detto che sia ottimale anche se usato all'interno di un altro problema). Infatti, si noti che non serve valutare tutte le sottosequenze e di conseguenza non è necessario esplorare del tutto la nostra istanza tramite un algoritmo brute force, come il precedente.

Con algoritmo di forza bruta si intende un algoritmo che offre una soluzione esaustiva, ovvero, che sonda tutti gli elementi possibili per quell'istanza (nel nostro caso tutte le sottosequenze).

Tramite un'attenta analisi della nostra sequenza possiamo arrivare alla seguente intuizione:

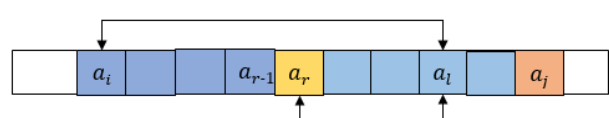
Assumiamo che $\text{SUM}(i, j-1) \geq 0$ (N.B.: $\text{SUM}(i, j-1) \Leftrightarrow \sum_{k=i}^{j-1} a_k$), ciò significa anche che tutte le sottosequenze che vanno da i a $j-1$ sono non negative, ovvero che $\text{SUM}(i, j-1) \geq 0 \forall k : i \leq k < j$.



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

$$1) \text{SUM}(i, j) \geq 0 \Rightarrow \text{SUM}(r, l) \leq \text{SUM}(i, l) \\ \forall i \leq r \leq l \leq j$$

Ovvero, nessuna sottosequenza di una sequenza non negativa può essere migliore



di quella di partenza. Infatti:

$$\sum_{k=i}^j a_k = \sum_{k=i}^{r-1} a_k + \sum_{k=r}^j a_k \Rightarrow \sum_{k=r}^j a_k = \sum_{k=i}^j a_k - \sum_{k=i}^{r-1} a_k \Rightarrow \sum_{k=r}^j a_k \leq \sum_{k=i}^j a_k$$

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

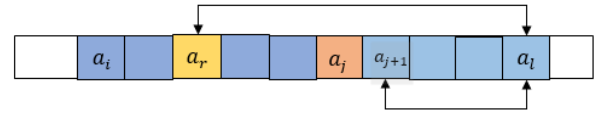
2) $\text{SUM}(i, j) \leq 0 \Rightarrow \text{SUM}(r, l) < \text{SUM}(j+1, l)$
 $\forall i \leq r \leq j \wedge \forall j+1 \leq l \leq n$

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:

$$\sum_{k=j+1}^l a_k = \sum_{k=i}^l a_k - \sum_{k=i}^j a_k \Rightarrow \sum_{k=j+1}^l a_k \geq \sum_{k=i}^l a_k$$



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 mi 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 (ciò che è scritto tra $/* */$ è un commento):

```
int MAXSUM(A,N)
  V = 0 /*i=1*/
  j = 1 /*j=i*/
  SUM = 0
  WHILE j <= N DO
    SUM = SUM + A[j]
    /*caso 2 (V>SUM):*/
    IF SUM <= 0 THEN
      /*i=j+1*/
      SUM = 0 /*simula il salto*/
    /*caso 1*/
    ELSE IF SUM > V THEN
      V = SUM
    j = j + 1
  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 V.

Ed è ovvio che questo algoritmo con $T(N) = \Theta(n)$ sia ottimale, poiché supponendo di avere un input con valori tutti positivi è evidente che MAXSUM sia la somma di tutti i positivi; tuttavia, anche nel caso in cui sapessi a priori che sono tutti positivi, non ho altra scelta se non quella di scorrere comunque tutta la sequenza per poterne calcolare la somma.

Approccio incrementale ed approccio ricorsivo

L'algoritmo precedente è intrinsecamente iterativo, ma se il mio problema avesse un comportamento ricorsivo come potrei fare l'analisi del tempo di esecuzione?

[Insertion sort](#), ad esempio, è una tecnica di soluzione incrementale (ovvero ottenuta "pezzo per pezzo", appunto incrementale); infatti, parte da un array ordinato di dimensione uno e ad ogni iterazione incrementa la porzione ordinata fino ad n .

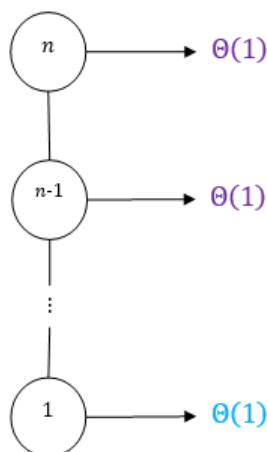
Di tutti gli algoritmi visti finora, siamo arrivati alla soluzione decomponendo il problema in sottoproblemi più semplici da risolvere, i quali erano diversi tra loro (come generare coppie e sommare una sequenza). Ma se ho una istanza di problema che può essere decomposta in soluzioni di istanze dello stesso problema?

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 (poiché mi aspetto che sia più semplice ordinare una sequenza di due elementi che ordinarne una di cento). Questo tipo di approccio ricorsivo prende il nome di divide et impera (ne è un esempio il [merge sort](#)).

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 capire il numero di chiamate ricorsive che la suddetta funzione compie bisogna analizzarne la struttura tramite l'albero di ricorrenza, ovvero un albero dove i nodi rappresentano le chiamate ricorsive e ad ogni nodo sono associati il numero di ricorrenza ed il contributo locale di tale nodo (e quindi abbiamo anche il contributo di ogni livello dell'albero). Nel nostro caso avremmo il seguente albero di ricorrenza:



C'è una corrispondenza tra il livello dell'albero e l'output del programma; infatti, al livello i avremmo 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), quindi il livello è foglia se $n - i = 1 \Rightarrow i = n - 1$

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

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

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

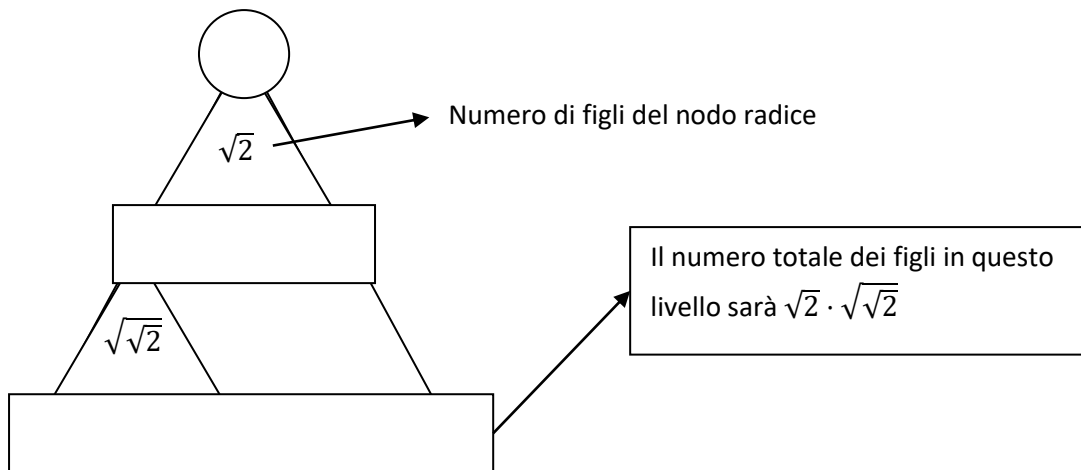
Forniamo una generalizzazione della struttura dell'equazione di ricorrenza:

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

- Il caso base, in linea di massima, è una costante (anche se ci potrebbero essere dei casi in cui il caso base ha un numero di operazioni, ad esempio, lineare, e quindi invece di $\Theta(1)$ si avrà un $\Theta(n)$);
- $g(n)$ è il contributo delle operazioni nelle chiamate ricorsive;

- $z(n)$ è il numero di chiamate ricorsive; l'abbiamo definita come funzione e non come costante poiché il numero di chiamate potrebbe dipendere dal valore in input. Infatti, non è detto che una chiamata debba fare lo stesso numero di chiamate ricorsive di un'altra chiamata nello stesso algoritmo;
- $f_i(n)$ è l'input che prende ogni chiamata ricorsiva (non è detto che sia la stessa per ogni chiamata) ed ovviamente, affinché l'algoritmo sia corretto deve risultare $f_i(n) < n$

Prima di passare a degli esempi, si noti che per calcolare il numero di nodi di un livello basta moltiplicare tra loro le espressioni dei livelli precedenti, esempio:



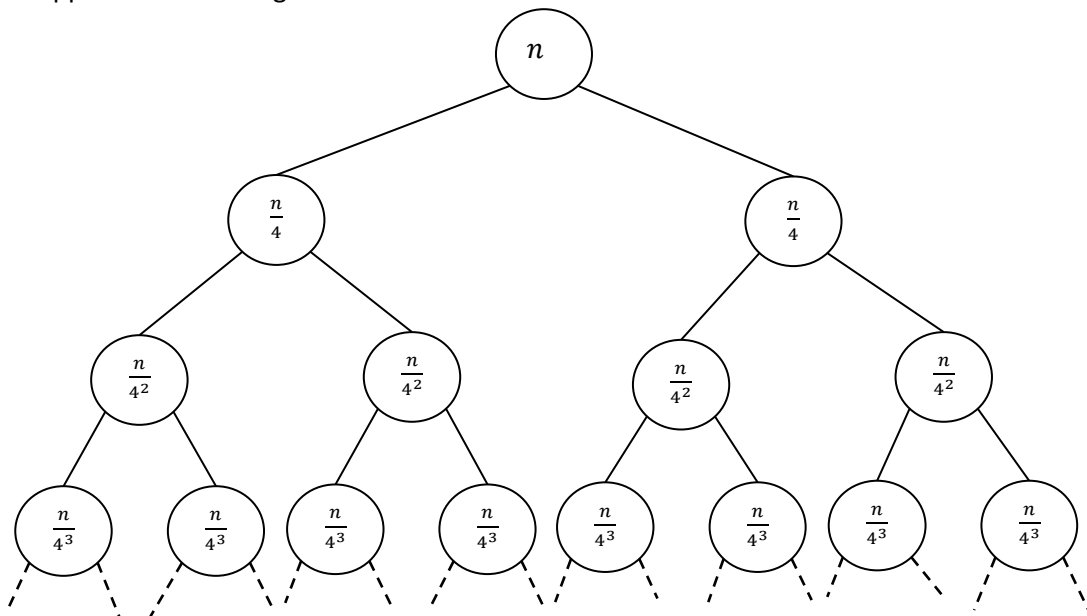
P.S.: la precedente rappresentazione è solo per questo specifico esempio, non si disegnano così gli alberi!

Esempio 1: equazione di ricorrenza

Sia $z(n) = 2$ (quindi si hanno due chiamate ricorsive per ogni chiamata, che si traduce in albero binario) e siano $f_i(n) = \frac{n}{4}$ (suddivisione input), $g(n) = n^2$ (tempo di esecuzione delle altre istruzioni). Allora avremo la seguente equazione asintotica:

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

La quale è rappresentata dal seguente albero di ricorrenza:



N.B.: abbiamo usato la forma $\frac{n}{4^i}$ così da rendere più facile capire 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 servirà per calcolare l'altezza dell'albero).

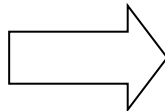
Si noti che se vale la seguente proprietà: $f_i(n) = f_j(n), \forall 1 \leq i, j \leq z(n)$; significa che i nodi di uno stesso livello ricevono lo stesso input, e ciò vale per ogni livello dell'albero.

Quindi ogni nodo di ciascun livello ha lo stesso input, ma allora il livello 0 avrà contribuito $g(n) = n^2$, il livello 1 invece sarà $g\left(\frac{n}{4}\right) = \left(\frac{n}{4}\right)^2$, livello 2 $g\left(\frac{n}{4^2}\right) = \left(\frac{n}{4^2}\right)^2$, etc...

P.S.: conviene sempre mantenere i risultati nella forma più generale possibile (senza semplificare) così da non perdere le relazioni tra i livelli e poterne ricavare più facilmente la somma.

A questo punto, per il calcolo totale possiamo isolare i livelli (quindi calcolare il contributo di ogni livello) e poi farne un'unica somma in verticale (ovviamente serve conoscere l'altezza dell'albero). Dunque:

| | |
|-----------|---|
| 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 $2^0\left(\frac{n}{4^0}\right)^2$, ma allora è evidente che per il livello i -esimo avremo:

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

Abbiamo così trovato il termine generale.

Per quanto riguarda l'altezza dell'albero bisogna ragionare sul contributo delle foglie, che sappiamo essere 1; 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:

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

Quindi poiché l'altezza dell'albero è $h = \frac{1}{2} \log n$, il numero delle foglie sarà semplicemente il numero dei figli per nodo elevato all'altezza dell'albero; nel nostro caso:

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

Ora abbiamo tutte le informazioni per calcolare il tempo di esecuzione della nostra funzione:

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

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

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

Ma allora, poiché la nostra serie tende ad una costante k avremo

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

Per scrupolo andiamo a calcolare la serie geometrica senza sfruttarne la proprietà di convergenza ma andando ad usare la sua forma chiusa:

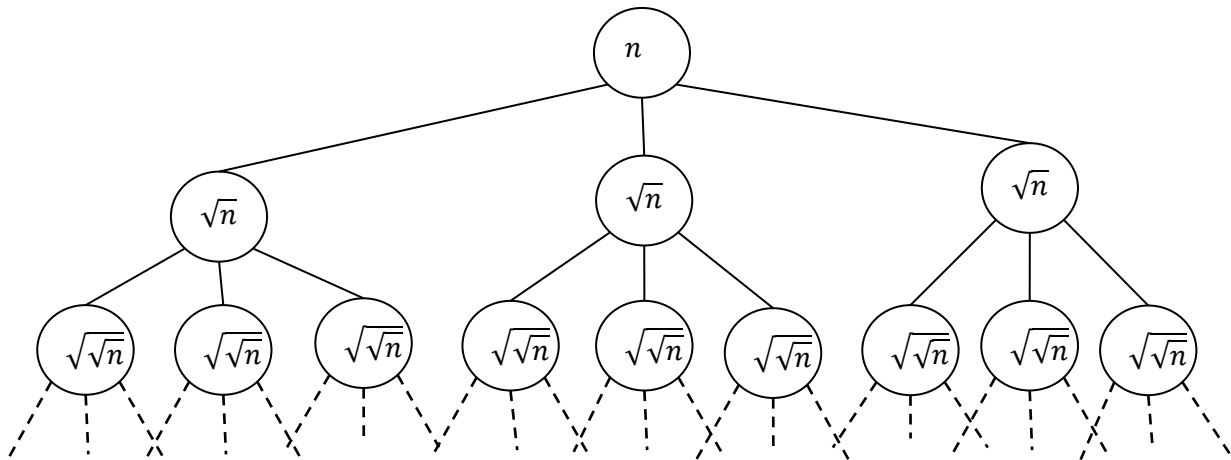
$$\begin{aligned} \sum_{i=0}^{\frac{\log n}{2}-1} \left(\frac{1}{8}\right)^i &= \frac{\left(\frac{1}{8}\right)^{\frac{\log n}{2}} - 1}{\frac{1}{8} - 1} = \frac{-\left(\frac{1}{8}\right)^{\frac{\log n}{2}} + 1}{-\frac{1}{8} + 1} = \frac{1 - \left(\frac{1}{8}\right)^{\frac{\log n}{2}}}{\frac{7}{8}} = \frac{8}{7}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

Ma la funzione $f(n) = 1 - 1/\sqrt{n^3}$ tenderà a 1 per $n \rightarrow \infty$ dunque $\frac{8}{7}$ sarà il limite superiore della nostra sommatoria, mentre 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, come già dimostrato precedentemente, $T(n) = \sqrt{n} + cn^2 = \Theta(n^2)$.

Esempio 2: equazione di ricorrenza con input esponenziale

$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ 3T(\sqrt{n}) + 1 & \text{se } n > 2 \end{cases}$; da questa funzione, poiché cresce più lentamente di $\frac{n}{4}$ mi aspetto che

l'altezza dell'albero sia minore rispetto a quella dell'esempio precedente (il numero di figli non influisce sull'altezza dell'albero ma solo sulla sua ampiezza) e ciò significa che arriverò 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 $n^{1/2^i}$

Calcoliamo 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 $1 \cdot 2$):

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

N.B.: $\log n > \log(\log n)$; quindi la supposizione fatta all'inizio è ora evidentemente vera; infatti $\log(\log n)$ è esponenzialmente minore di $\log n$.

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

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

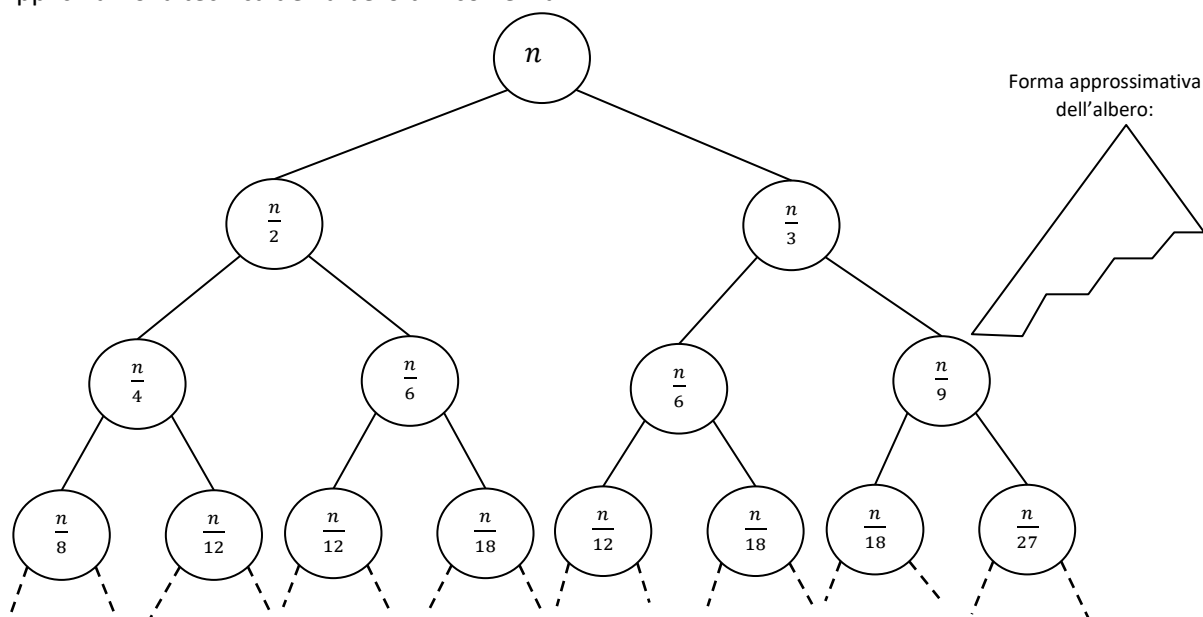
N.B.: $\log 3$ è una costante, quindi non va approssimata in nessun modo.

Esempio 3: chiamate ricorsive con input diversi

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

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

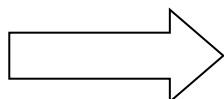
Applichiamo la tecnica dell'albero di ricorrenza:



Come si evince anche dalla forma approssimata dell'albero, è chiaro che ci saranno dei rami (sequenze discendenti di nodi) di questo albero che arrivano prima alle foglie e percorsi che vi arrivano dopo. Nello specifico il ramo più a sinistra decresce più lentamente del ramo più a destra; infatti, già al livello 3 abbiamo un 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 | $\frac{n}{8} + \frac{3n}{12} + \frac{3n}{18} + \frac{n}{27} = \frac{5^3}{6^3}n$ |
| ... | |

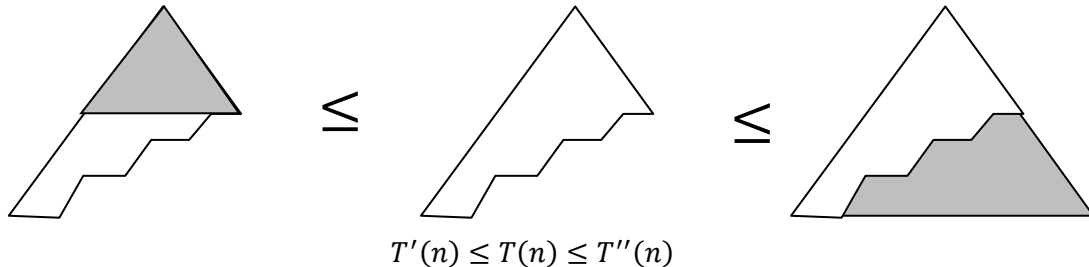


Da cui segue che per un livello i avremmo $\left(\frac{5}{6}\right)^i n$ come termine generale

A questo punto però non possiamo procedere come negli esempi precedenti; infatti, si noti che la relazione calcolata precedentemente vale solo per i livelli **pieni**, ovvero per quei livelli che hanno il numero massimo

di nodi possibile. Ma come abbiamo già illustrato, l'albero non è pieno, e per tutti quei livelli non pieni il termine generale non è corretto.

Però, prendendo solo i livelli pieni, è evidente che si ottiene un nuovo albero con tempo di esecuzione certamente minore di quello che dobbiamo calcolare, e quindi sarà un limite inferiore asintotico per $T(n)$. Analogamente, otterremo un limite superiore asintotico se, approssimando per eccesso, "fingiamo" che tutti i livelli del nostro albero siano pieni e che quindi valga il termine generale:



I due diversi alberi così generati 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 fino ad ora, dunque:

- L'altezza del percorso lungo, poiché si divide sempre per due, si otterrà con la seguente relazione:

$$\frac{n}{2^i} = 1 \Rightarrow \log n = 2^i \Rightarrow i = \log n$$
- L'altezza del percorso breve invece sarà $\log n = 3^i \Rightarrow i = \log_3 n$

A questo punto è possibile calcolare i tempi di esecuzione delle due approssimazioni. N.B.: non useremo la forma completa poiché con buona approssimazione possiamo considerare il contributo dell'ultimo livello come se fosse un livello interno (quindi andiamo ad approssimare per eccesso il contributo delle foglie):

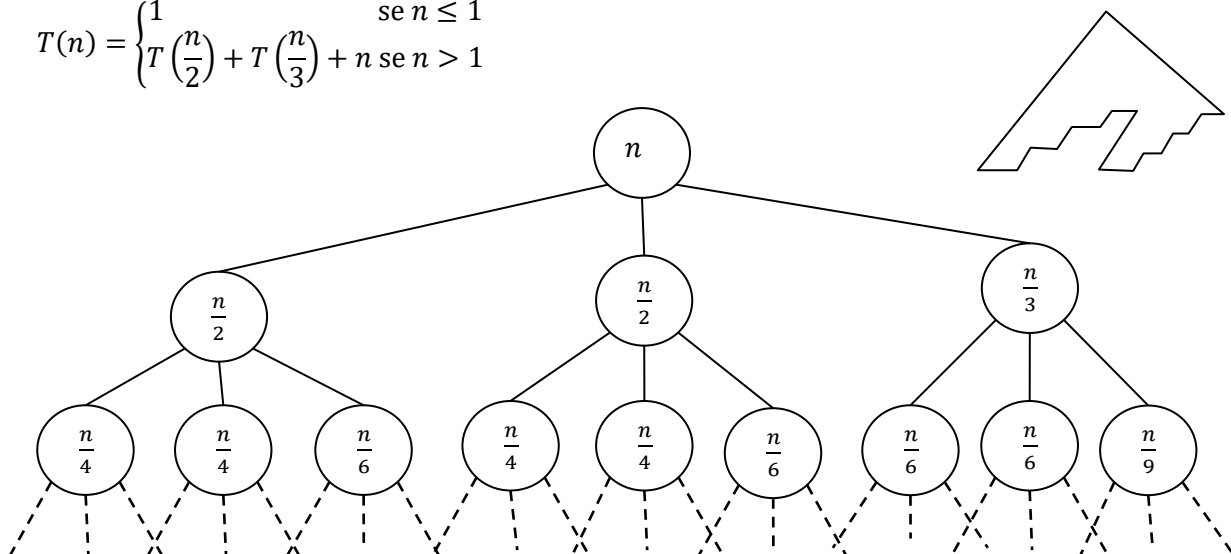
$$T'(n) = \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i n = n \sum_{i=0}^{\log_3 n} \left(\frac{5}{6}\right)^i \xrightarrow{\text{serie geometrica con ragione } < 1} T'(n) = \Theta(n)$$

$$T''(n) = \sum_{i=0}^{\log n} \left(\frac{5}{6}\right)^i n = n \sum_{i=0}^{\log n} \left(\frac{5}{6}\right)^i = \Theta(n) \quad \text{Entrambe sono comprese tra } 1 \text{ e } \frac{1}{1-\frac{5}{6}} = 6$$

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

Esempio 4: limite superiore ed 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 a quello dell'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:

$$T'(n) = \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i n \quad \wedge \quad T''(n) = \sum_{i=0}^{\log n} \left(\frac{4}{3}\right)^i n$$

A differenza dell'esempio precedente, stavolta la serie geometrica non ha ragione compresa tra 0 e 1; di conseguenza, bisognerà usare la forma chiusa della serie geometrica:

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

Ma per quanto siano molto vicini sono comunque due funzioni con **esponente diverso**; quindi, dalla relazione $T'(n) \leq T(n) \leq T''(n)$ posso solo dire che $T(n) = \Omega \left(n^{\log_3 \left(\frac{4}{3} + 1 \right)} \right)$ e $T(n) = O \left(n^{\log \left(\frac{4}{3} + 1 \right)} \right)$, ma nient'altro.

4. Algoritmi di ordinamento

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 $1 \leq i \leq n$
Nota: andrebbe bene anche un qualsiasi insieme (non per forza i numeri relativi), l'importante è che abbia una relazione d'ordine totale (altrimenti non potremmo ordinare gli elementi)
- **Output:** A' : una permutazione ordinata di A
N.B.: ci potrebbero essere più permutazioni ordinate, ad esempio per $A = (2, 6, 2)$ abbiamo le seguenti permutazioni ordinate: (a_1, a_3, a_2) e (a_3, a_1, a_2)

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

- Proprietà della **permutazione**: $\exists f: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ **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' posizione $f(i)$
- Proprietà dell'**ordinamento**: $a'_i \leq a'_{i+1}$ con $1 \leq i \leq n$

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

$f_1: (1, 1), (2, 3), (3, 2)$ 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 $j_k = f^{-1}(k)$. Esempio: $A = 42 \wedge A' = 24$ allora $j_2 = f^{-1}(2) = 1$.

Dalle definizioni di input/output si capisce che il nostro problema è quello di trovare, data una sequenza, una sua permutazione con determinate proprietà.

Approccio 1

Sappiamo che il numero di permutazioni è finito per insiemi finiti, più precisamente se $|A| = n$ allora il numero di permutazioni è $\prod_{i=1}^n i = n!$, ed 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 e, tra queste, prelevarne una ordinata.

Questa soluzione è banalmente corretta, ma questa volta, attuare un algoritmo di tipo brute force è drammaticamente inefficiente poiché dovremmo generare ben $n!$ permutazioni (e alla peggio la permutazione ordinata potrebbe essere l'ultima generata). Si noti che $n \leq n! \leq n^n$.

N.B.: 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.

Questo ci fa anche capire che per qualsiasi algoritmo brute force esiste almeno un input che costringe a esplorare tutti gli elementi dell'insieme prima di arrivare alla soluzione. Tornando al nostro approccio, anche supponendo per assurdo che si possa generare una permutazione in tempo costante, per verificare che la permutazione sia ordinata dobbiamo inevitabilmente controllare tutti gli elementi (tempo lineare) ed inoltre, nel caso peggiore dobbiamo generare $n! = \Omega(n^n)$ permutazioni.

Da questa analisi è evidente che un algoritmo brute force non è una soluzione applicabile.

Approccio 2

Bisogna cercare un algoritmo che da una sequenza ordinata cerchi di passare ad una sequenza un po' più ordinata fino ad arrivare alla sequenza completamente ordinata, ignorando dunque molte 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 prenderne tutte le coppie e confrontare a due a due gli elementi. Ad esempio, per (a_1, a_2, a_3) arrivo alla sequenza ordinata tramite i seguenti confronti:

$$a_1 \leq a_2 \begin{cases} a_1 \leq a_3 \begin{cases} a_2 \leq a_3 \rightarrow (a_1, a_2, a_3) \\ a_3 < a_2 \rightarrow (a_1, a_3, a_2) \end{cases} \\ a_3 < a_1 \rightarrow (a_3, a_1, a_2) \end{cases}$$

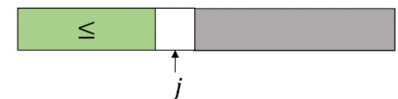
Quindi alla peggio devo confrontare tutte le coppie della sequenza e questo sappiamo che impiega tempo quadratico, altrimenti, nel caso migliore i confronti impiegheranno $T(n) = O(n)$.

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

Insertion Sort

L'idea di questo algoritmo è quella di dividere la sequenza in una parte ordinata ed una disordinata:

All'inizio possiamo supporre che la sequenza ordinata sia composta solo dal primo elemento, mentre tutti gli altri 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.



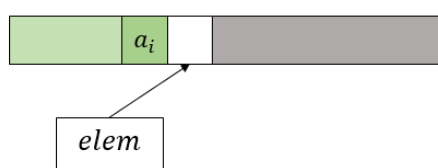
Quindi se $a_{j-1} \leq a_j$ allora lo lascio dov'è; altrimenti a_{j-1} andrà in posizione j e dovrò confrontare a_j con a_{j-2} ; anche in questo caso se $a_{j-2} \leq a_j$ allora a_j andrà nella posizione liberata precedentemente ($j - 1$), altrimenti è a_{j-2} ad andare nella posizione $j - 1$. Iterando questo processo posso arrivare a confrontare a_j con a_1 : ora se $a_1 \leq a_j$ posizionerò a_j nella posizione 2 (che sarà libera) altrimenti andrà in posizione 1 ed a_1 finirà in posizione 2 (questi spostamenti avranno ovviamente un loro costo).

L'algoritmo sarà dunque il seguente:

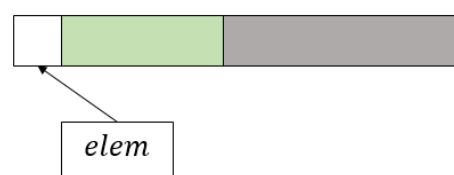
```
1 InsertionSort(A, N)
2     /*la prima posizione è già ordinata*/
3     FOR j = 2 TO N DO
4         i = j - 1 /*ultimo indice della posizione ordinata*/
5         elem = A[j]
6         WHILE i >= 1 AND elem < A[i] DO
7             A[i + 1] = A[i] /*nella cella libera metto A[i]*/
8             i = i - 1
9         A[i + 1] = elem
```

A prescindere dalla condizione che faccia terminare il while, la linea 9 è sempre la stessa; infatti:

Caso $elem \geq A[i]$



Caso $i = 0$



Analisi

Studiamo solo il ciclo while poiché sarà lui a determinare il comportamento asintotico dell'algoritmo. Tale ciclo può essere diverso per istanze con stessa dimensione a causa del confronto $\text{elem} \geq A[i]$ (che è anche il motivo per cui abbiamo usato il while e non il 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 esempio, per (1,2,3,4) non verrà mai eseguito il corpo del while (la testa 3 volte), mentre per (4,3,2,1) le istruzioni nel corpo saranno eseguite 6 volte (la testa del while invece 9).

Questa irregolarità ci costringe a fare i seguenti ragionamenti:

- Se l'algoritmo è sfortunato posso assumere un unico valore per $T(N)$? Vedremo che sarà così. Più precisamente, dirò che $T(N) = O(T_{\max}(N))$ con $T_{\max}(N)$ 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 dovrò analizzare il caso medio.

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

```
InsertionSort(A, N)
  FOR j = 2 TO N DO
    i = j - 1
    elem = A[j]
    WHILE (i >= 1 AND elem < A[i]) DO
      A[i+1] = A[i]
      i = i - 1
    A[i+1] = elem
```

Diagram showing complexity annotations for the InsertionSort code:

- $2(N + 1 - 1)$ points to the FOR loop header.
- $2(N - 1)$ points to the `i = j - 1` line.
- $2(N - 1)$ points to the `elem = A[j]` line.
- $3(N - 1)$ points to the `A[i+1] = elem` line.

Per il ciclo while, invece, possiamo descrivere per ora solo il costo delle singole istruzioni, che sono 4,3,1, rispettivamente.

Per quanto riguarda il ciclo while bisogna innanzitutto notare che 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 -esima, ad esempio (1,3,2,4), (10,25,17,60), (7,29,12,30) ...

Quindi ci sono tante classi di equivalenza quante permutazioni di una sequenza; ne consegue che per lo studio del while bisogna formalizzare la suddetta dipendenza. È chiaro che l'input ha impatto solo sul numero di esecuzioni del while (le linee analizzate precedentemente mantengono lo stesso costo).

Fissato j , il while viene ripetuto un tot di volte partendo da 0 e, ovviamente, per j diverso il numero di volte che viene ripetuto il blocco è diverso. Possiamo introdurre quindi un insieme 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, poiché varia da 2 a N , avrà 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 (ovviamente tale sommatoria non ha una forma chiusa), mentre, $\sum_{j=2}^N (t_j - 1)$ volte per il corpo.

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

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

Caso peggiore

Se la seconda condizione del while è sempre verificata, è evidente che $i = 0$ (poiché all'inizio $i = j - 1$) si verifica dopo j iterazioni (e quindi si esce dal while); dunque, più di j iterazioni non può fare.

Ora bisogna solo verificare quanto sia realistico tale limite superiore; ovvero, esiste un input tale per cui $\forall j, t_j = j$? La risposta è sì, ed è la sequenza decrescente; ad esempio, per (4,3,2,1) avremo 2 iterazioni per ordinare il secondo elemento, 3 per il terzo e 4 per l'ultimo. Inoltre, poiché più di j iterazioni non si possono fare è evidente che questo è il caso peggiore per il nostro algoritmo.

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

Caso migliore

Con sequenze ordinate accade che $\forall j : 2 \leq j \leq N$ risulta $t_j = 1$; ad esempio, per (3,7,8,10) la seconda condizione del while risulterà falsa per ciascun j , quindi ci saranno 4 esecuzioni.

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

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 su N).

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

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

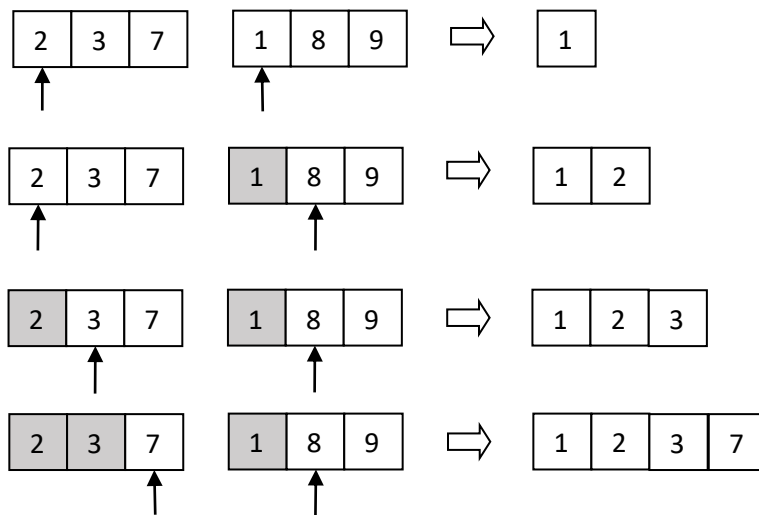
Essendo il caso dominante quello peggiore, possiamo assumere che insertion sort è un algoritmo che ha tempo di esecuzione quasi sempre quadratico.

Merge Sort

Prendiamo una sequenza e dividiamola in due parti e supponiamo che, per una strana magia, le due sequenze mi ritornino ordinate ed io devo solo preoccuparmi di metterle insieme.

Prendiamo ad esempio la sequenza (3,7,2,1,8,9), ed una volta divisa in due io devo solo preoccuparmi di mettere insieme la sottosequenza (2,3,7) con l'altra sottosequenza (1,8,9).

Possiamo sfruttare il fatto che le sequenze sono già ordinate e scorrerle entrambe partendo dal primo elemento; poiché i primi elementi sono i più piccoli delle sottosequenze è evidente che il primo elemento della sequenza totale sarà il più piccolo tra i due. Quello rimasto sarà confrontato con il secondo elemento dell'altra sequenza e il minimo tra i due andrà in seconda posizione, e così via fino al completamento di una sottosequenza:



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



Ma queste due sequenze di certo non verranno ordinate per magia; infatti, andranno divise in altre due sequenze e così via, fino ad arrivare ad una sequenza di un solo elemento che è ordinata per definizione.

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

N.B.: affinché la soluzione funzioni, deve essere possibile raggiungere il caso base da **ogni** possibile input del problema.

Si poteva pensare di usare la sequenza vuota come caso base ma così si ha 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, ma quest'ultima dovrà nuovamente essere divisa, e così via). Quindi, bisogna sempre 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 $1 \leq \left\lfloor \frac{N}{2} \right\rfloor < N \wedge 1 \leq N - \left\lfloor \frac{N}{2} \right\rfloor < N$ mentre per $N = 1$ ho già la sequenza ordinata. A questo punto possiamo scrivere l'algoritmo:

```
1 MergeSort(A, p/*indice d'inizio*/, r/*indice di fine*/)
2   IF p < r THEN /*ho almeno due elementi*/
3     q = (p + r)/2 /*base della divisione*/
4     MergeSort(A, p, q)
5     MergeSort(A, q+1, r)
6     Merge(A, p, q, r) /*unisce le due sequenze*/
```


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 $q = \left\lfloor \frac{p+r}{2} \right\rfloor$, e sappiamo che $\left\lfloor \frac{p+r}{2} \right\rfloor \leq \frac{p+r}{2}$; ma allora risulta

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

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

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

Resta ora solo da dimostrare che, per qualsiasi input, l'algoritmo faccia un numero finito di chiamate ricorsive e che quindi, prima o poi, raggiungerà il caso base; ma 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 poiché la condizione dell'if sarà alla prima chiamata $p < p$ ed alla seconda $p + 1 < r$ (entrambe ovviamente false).

Algoritmo Merge e occupazione in memoria

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

```
Merge(A, p, q, r)
    k = p /*indice che scorre la posizione di B*/
    i = p
    j = q + 1
    WHILE i <= q AND j <= r DO
        IF A[i] <= A[j] DO
            B[k] = A[i]
            i = i + 1
        ELSE
            B[k] = A[j]
            j = j + 1
        k = k + 1
    IF i < q THEN
        j = i /*poiché restano da copiare gli elementi della sequenza sinistra*/
        /*altrimenti non cambio j poiché l'indice è già corretto*/
        WHILE k <= r DO
            B[k] = A[j]
            j = j + 1
            k = k + 1
    /*qui andrebbero copiati gli elementi di B nuovamente in A*/
```

Il tempo di merge è ovviamente lineare poiché vado ad effettuare almeno n scritture in memoria. Ma si noti che il fatto di dover utilizzare un altro vettore 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, il solo MergeSort ha bisogno di spazio aggiuntivo a causa delle chiamate ricorsive; infatti, per ogni chiamata ricorsiva, prima di passare alla chiamata figlia, c'è bisogno di salvare dei dati sullo stack di attivazione.

Tempo di esecuzione

```

MergeSort(A, p, r)
  IF p < r THEN
    q = (p + r) / 2
    MergeSort(A, p, q)
    MergeSort(A, q+1, r)
    Merge(A, p, q, r)
  
```

Annotations:

- $\theta(1)$ points to the `IF` statement.
- $\theta(?)$ points to the recursive calls `MergeSort(A, p, q)` and `MergeSort(A, q+1, r)`.
- $\theta(n)$ points to the `Merge(A, p, q, r)` statement.

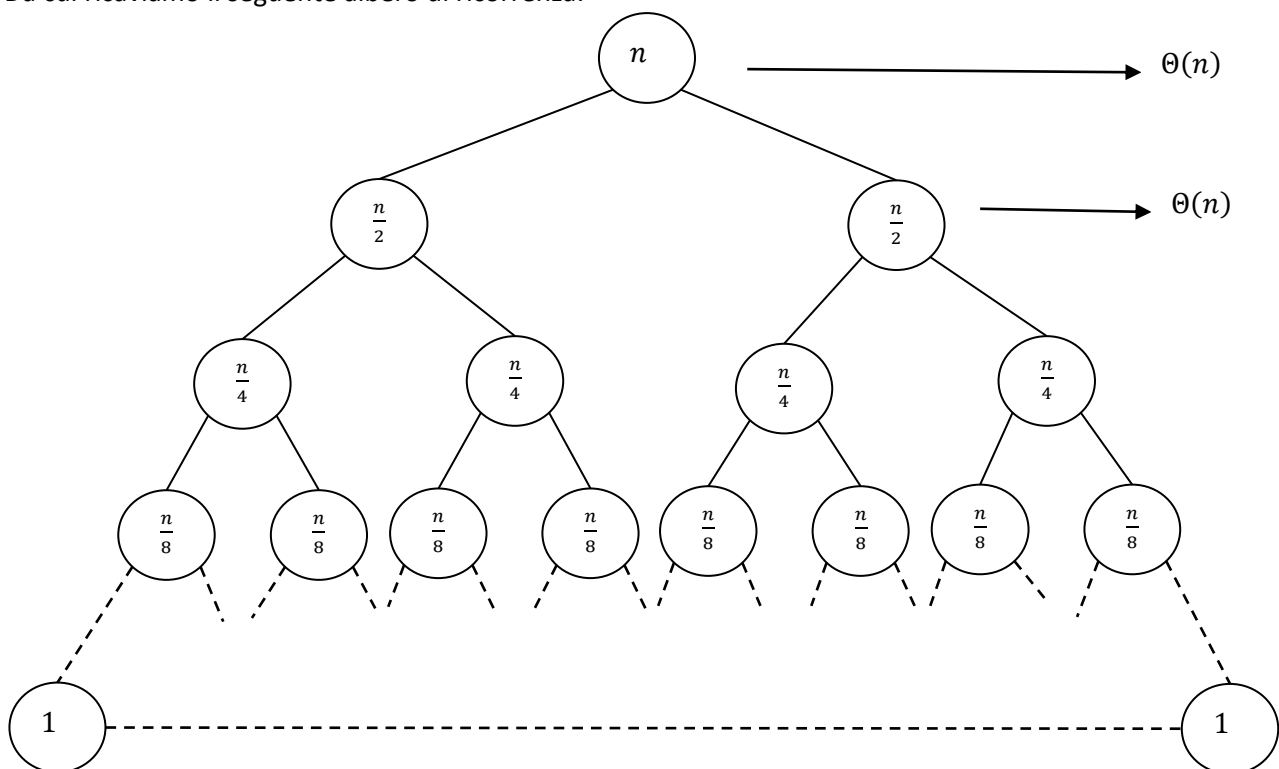
Sarebbe assurdo dover calcolare il valore del tempo di esecuzione della funzione se per farlo devo conoscere il tempo di esecuzione della funzione stessa. Ma, poiché la funzione è definita in modo induttivo, è possibile calcolarla su input più piccoli; dunque, è possibile sfruttare la definizione induttiva e definire in maniera induttiva anche la funzione tempo.

Con buona approssimazione avremo: $T_{MS}(n) = \begin{cases} \theta(1) & \text{se } n \leq 1 \\ \theta(1) + T_{MS}\left(\frac{n}{2}\right) + T_{MS}\left(\frac{n}{2}\right) + T_{Merge}(n) & \text{altrimenti} \end{cases}$

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

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

Da cui ricaviamo il seguente albero di ricorrenza:



Quindi il nodo del livello 0 contribuisce per un $\theta(n)$, ciascun nodo del livello 1 produce $\theta(n)$ poiché fa $\frac{n}{2}$ operazioni (da non confondere con il numero di elementi, infatti il nostro $\theta(n)$ deriva dal contributo $2T_{MS}\left(\frac{n}{2}\right) + \theta(n)$), analogamente si comportano tutti i livelli (eccetto l'ultimo). Praticamente, tutti i livelli hanno contributo lineare, eccetto quello delle foglie.

Di quest'ultimo posso solo dire che ogni nodo foglia contribuisce per un $\theta(1)$, ma per conoscere il contributo totale del livello devo prima capire quante foglie ci sono, così da poter calcolare il tempo di esecuzione con la seguente equazione:

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

- **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 $1 = \frac{n}{2^i} \Rightarrow 2^i = n \Rightarrow \log n = \log 2^i \Rightarrow i = \log n$. Quindi il numero di livelli interni è **$\log n - 1$**
- **Numero foglie:** sappiamo che le foglie sono a profondità $\log n$, ma sappiamo anche che un livello i ha 2^i nodi; quindi, il numero delle foglie è $2^{\log n} = n$

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

Selection Sort

L'idea del selection sort può essere vista come il duale dell'insertion sort; infatti, seleziona una posizione e poi trova l'elemento da inserire all'interno. Sia la posizione che l'elemento vengono ovviamente scelti con cognizione 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 1 allora cercheremo il minimo); a questo punto, si prosegue con la posizione $n - 1$ scambiando l'elemento in quella posizione con il massimo della sequenza rimanente (da 1 a $n - 1$) e si procede in questo modo sino alla posizione 2 (ovviamente l'elemento in posizione 1 sarà già ordinato, poiché dopo gli eventuali $n - 1$ scambi è ovvio che in posizione 1 rimanga il minimo).

| | | |
|--|---|---|
| <pre> 1 SelectionSort(A, n) 2 FOR i = n DOWNTO 2 DO 3 j = FindMax(A, i) 4 Swap(A, i, j) </pre> | { | <pre> /*ritorna la posizione del massimo*/ int FindMax(A, i) max = 1 FOR j = 2 TO i DO IF A[max] < A[j] THEN max = j RETURN max </pre> |
|--|---|---|

Poiché sappiamo che la scelta del massimo non può essere fatta con un algoritmo meno che lineare (poiché devo per forza controllare tutti gli elementi) sembrerebbe che questa soluzione sia la migliore. Ma questo algoritmo è un altro buon esempio di come usare soluzioni ottime per dei sottoproblemi non rende l'algoritmo così definito ottimale. Tuttavia, ciò non significa che l'idea descritta sopra sia pessima; infatti, a seconda di come viene implementato, quest'algoritmo avrà tempi di esecuzione molto diversi (vedremo come arrivare ad un algoritmo che abbia $T(n) = \Theta(n \log n)$).

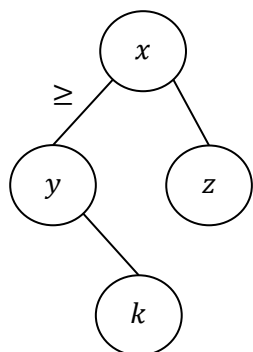
Prima di passare ad una migliore soluzione andiamo a calcolare il tempo di esecuzione del precedente algoritmo nonostante il risultato sia evidente:

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

Si nota facilmente che il problema del nostro algoritmo sia nella funzione FindMax: infatti, non viene sfruttato il fatto che durante lo scorrimento per la ricerca del massimo vengono eseguiti già dei confronti con alcuni elementi, però FindMax non ne tiene traccia e li "dimentica" alla successiva iterazione.

L'idea per migliorare il nostro algoritmo è proprio quella 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 dunque 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 usare 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)

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

Quindi, organizzando i valori in un albero dove l'arco rappresenterà la relazione di \geq possiamo migliorare il tempo di esecuzione dell'algoritmo FindMax e, conseguentemente, anche di SelectionSort.

Alberi binari pieni

Prima di passare all'algoritmo, è necessario anticipare delle proprietà sugli alberi. In particolare, noi utilizzeremo le proprietà degli alberi binari pieni (sono gli alberi più bassi possibili a parità di numero di nodi), ovvero:

- tutte le foglie sono sullo stesso livello;
- tutti i nodi interni hanno grado due;
- per ogni altezza h esiste uno, ed un solo, albero binario pieno;
- il numero di nodi è esattamente $\sum_{i=0}^h 2^i = 2^{h+1} - 1$

Quindi, in un albero pieno ho bisogno di un numero preciso di nodi, ovvero un valore pari ad uno in meno di una potenza di due; ciò significa anche che se ho un numero di elementi diverso da $2^x - 1$ allora non posso costruire un albero pieno.

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

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

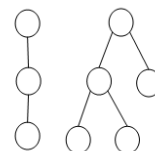
usiamo la base di tale logaritmo poiché dovendo lavorare con interi è evidente che $\log(2^{h+1} - 1) \notin \mathbb{N}$ essendo l'argomento diverso da una potenza della base. Inoltre, essendo $2^h \leq 2^{h+1} - 1 < 2^{h+1}$, risulta $\log 2^h \leq \log(2^{h+1} - 1) < \log 2^{h+1} \Rightarrow h \leq \log(2^{h+1} - 1) < h + 1$ e quindi $\lfloor \log(2^{h+1} - 1) \rfloor = h$. Allora:

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

Il fatto che un albero pieno sia quello con altezza minore è abbastanza evidente; infatti, basti pensare che l'unico modo per spostare un nodo è quello di metterlo nell'unico spazio disponibile, ovvero come figlio di una foglia, incrementando così l'altezza di uno.

Può essere utile anche sapere che il numero di nodi interni è $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ che è circa la metà del numero totale dei nodi; quindi, i nodi interni sono precisamente uno in più del numero delle foglie.

Si fa presente che l'albero pieno è l'unico tipo di albero in cui il numero di nodi è univocamente determinato dall'altezza e viceversa, poiché in altri tipi di alberi non è possibile determinare il numero di nodi avendo a disposizione come unica informazione l'altezza (un banale esempio è mostrato nella figura a destra).



Alberi binari completi

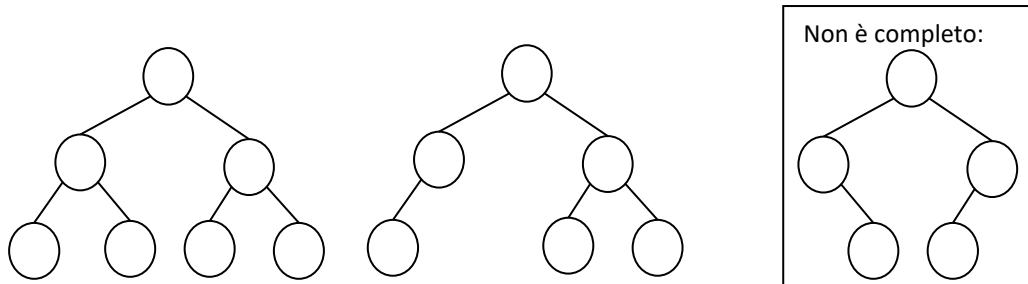
Per il nostro scopo, è necessario rilassare alcuni dei vincoli dell'albero pieno, poiché è evidente che non possiamo avere un algoritmo solo per determinati input. In particolare i vincoli da indebolire sono:

- Tutte le foglie sono sullo stesso livello
- Tutti i nodi interni hanno grado due

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

- Tutte le foglie sono al livello h oppure al livello $h - 1$
- Tutti i nodi interni hanno grado due tranne al più uno

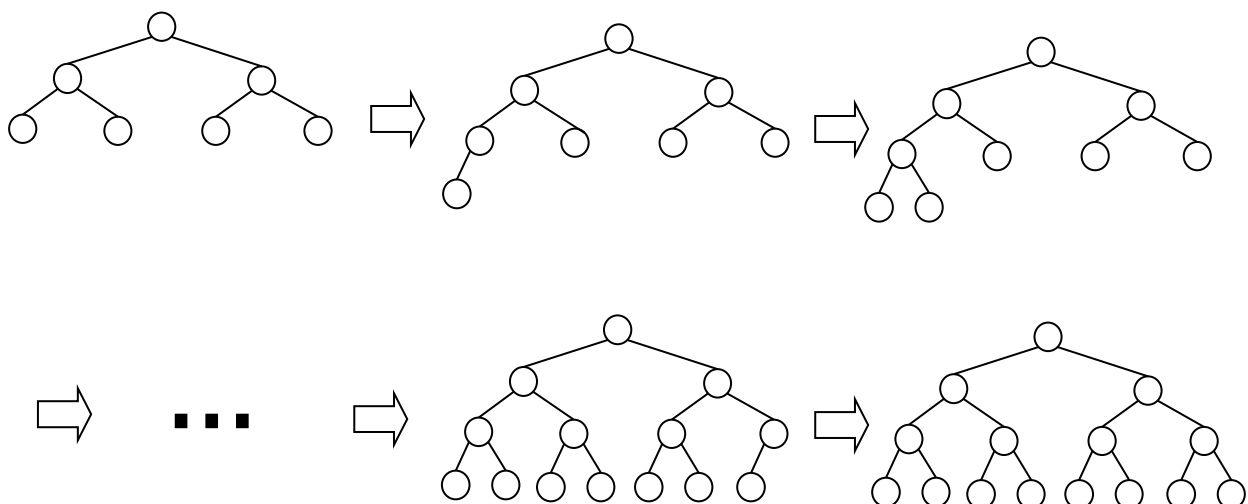
Esempi di alberi ammessi dalla precedente definizione sono i seguenti:



N.B.: ogni albero pieno è anche completo, ma il viceversa non sussiste.

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, e dunque esiste anche un albero pieno di $2^{h+2} - 1$, ma nessun albero pieno con numero di nodi compreso tra i precedenti due. Ora poiché 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. Dunque, ogni albero pieno è completo; resta da dimostrare che è possibile costruire un albero completo per ogni nodo nell'intervallo. Vediamolo con un esempio pratico per $7 \leq x \leq 15$:



Abbiamo risolto il problema della cardinalità. Ora, 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 ancora scrivere l'altezza in corrispondenza del numero di nodi: $h = \lfloor \log n \rfloor$.

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; ma è vero anche che $2^h \leq n \leq 2^{h+1}$ e quindi il nostro numero di nodi o è una potenza di due oppure si trova tra due potenze di due.

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

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

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

$$\log(2^{h+1} - 1) \leq \log n \leq \log(2^{h+2} - 1) \Rightarrow \underbrace{\lfloor \log(2^{h+1} - 1) \rfloor}_h \leq \lfloor \log n \rfloor \leq \underbrace{\lfloor \log(2^{h+2} - 1) \rfloor}_{h+1}$$

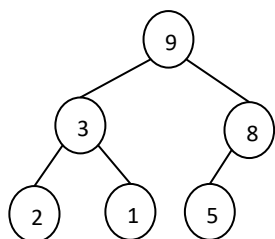
Ma quindi $h \leq \lfloor \log n \rfloor < h + 1$, ed essendo la base compresa tra h e $h + 1$ e, inoltre, $\lfloor \log n \rfloor \neq h + 1$ è evidente che $\lfloor \log n \rfloor = h$ come volevasi dimostrare.

Alberi heap

Con alberi heap ("mucchio") intendiamo 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 un'unica 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 due proprietà descritte precedentemente possiamo essere certi che nella radice dell'albero è situato il massimo valore della sequenza, poiché dalla radice discendono tutti i nodi dell'albero; infatti, la proprietà 2 specifica chiaramente che un nodo non può essere più grande di un suo antenato.

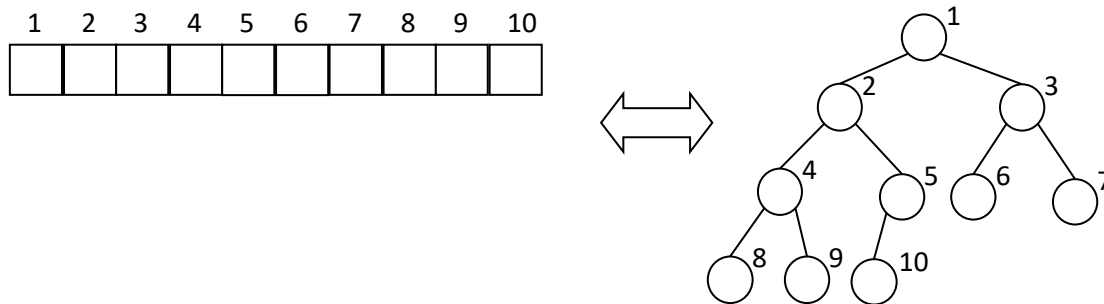
HeapSort

Dai precedenti ragionamenti abbiamo dedotto che la struttura più adatta a risolvere il nostro problema è l'albero heap. Nascono ora dei 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

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. Quindi si potrebbe pensare di sostituire il valore della radice (il nostro massimo) con quello di una foglia e poi cancellarlo; ovviamente la proprietà dell'ordinamento dopo questa operazione non è detto che sia mantenuta e quindi va ripristinata.

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. Forniamo un esempio di tale corrispondenza:



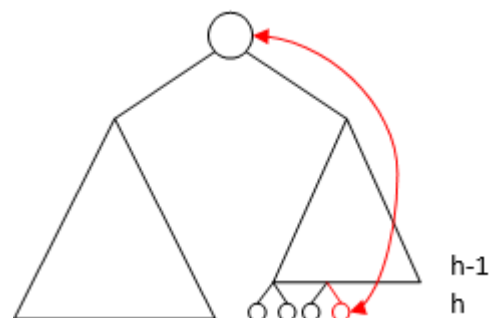
Quindi gli indici dell'array vanno da sinistra a destra livello per livello. Così facendo, un qualsiasi elemento in posizione i nell'albero avrà figlio sinistro in posizione $2i$ e figlio destro in posizione $2i + 1$ (ovviamente devono essere indici esistenti, ovvero non superiore alla lunghezza dell'array).

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 $1 \leq i \leq \lfloor n/2 \rfloor$, mentre le foglie $\lfloor n/2 \rfloor < i \leq n$.

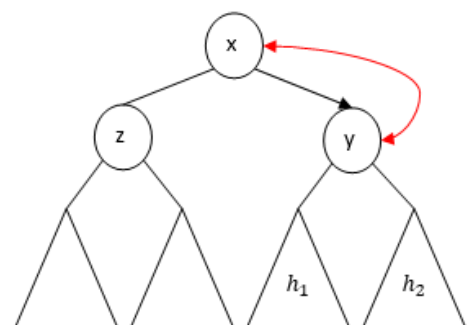
Persino la stratificazione dei livelli può essere ottenuta facilmente: basta dividere per due; ad esempio, l'ultimo livello va da $\lfloor n/2 \rfloor$ fino a n , il penultimo livello (i padri delle foglie) va da $\lfloor \frac{\lfloor n/2 \rfloor}{2} \rfloor$ fino a $\lfloor n/2 \rfloor$, etc...

Prendiamo un generico albero heap; poiché le foglie sono contigue è evidente che se l'albero non è pieno esistono solo due possibilità: o il sottoalbero sinistro è un albero pieno (come in figura) di altezza $h - 1$ (bisogna togliere il livello della radice) oppure è il sottoalbero a destra ad 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 cancellare quest'ultima. Così facendo è evidente che i due sottoalberi mantengono la proprietà di essere heap e l'unico nodo a poter introdurre una violazione sia la radice.

Tale proprietà può essere sfruttata; infatti, essendo i due sottoalberi heap, il massimo del nuovo albero sarà o nella radice del sottoalbero destro oppure in quella del sinistro. Supponiamo di avere il caso in figura e quindi scambiamo x con y , così facendo ci troveremo nella stessa situazione di prima dove 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 che l'algoritmo può fare un numero di chiamate ricorsive al più pari all'altezza dell'albero, ovvero $\lfloor \log n \rfloor$. Dunque, possiamo affermare che quest'algoritmo abbia tempo di esecuzione pari a $O(\log n)$.

Ma non possiamo ancora dire di aver migliorato l'algoritmo, infatti se la costruzione dello heap richiedesse tempo quadratico allora 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 e quindi l'unico nodo a poter violare la condizione sia la radice (se l'input iniziale non ha questa caratteristica l'algoritmo non funziona):

```

1  Heapify(A, i/*indice radice del sottoalbero*/)
2      /*definiamo i figli della radice*/
3      sx = 2i
4      dx = 2i + 1
5      IF 2i <= heapsize AND A[i] < A[sx] THEN
6          max = sx
7      ELSE
8          max = i
9      IF dx <= heapsize AND A[max] < A[dx] THEN
10         max = dx
11     IF max != i THEN
12         Swap(A, i, max)
13         Heapify(A, max)

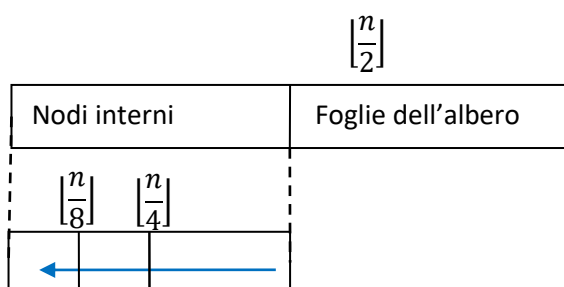
```

Supponiamo sia una variabile globale che tiene traccia della dimensione corrente dell'albero (nell'algoritmo principale abbiamo visto che dobbiamo cancellare i nodi)

Il fatto che quest'algoritmo non funzioni per sequenze che non siano quasi heap è dimostrabile facilmente: se funzionasse significherebbe che potremmo utilizzare Heapify così descritto per cercare il massimo di una sequenza (prenderemo la radice) ma ciò è assurdo poiché non è possibile cercare il massimo in un tempo minore di $\Theta(n)$ (si ricordi che $T_H(n) = O(\log n)$).

Dunque, 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 heap, allora se prendiamo i padri delle foglie possiamo applicare ad essi Heapify perché tutti i sottoalberi così definiti rispettano la proprietà necessaria al nostro algoritmo (di essere quasi heap); ma 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 ad arrivare alla radice della sequenza.

A questo punto l'unico problema rimasto da risolvere per l'algoritmo di costruzione dello heap è quello di come tradurre il precedente ragionamento in indici dell'array. Abbiamo già discusso di come sono suddivisi i livelli dell'array:



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

Di seguito si riporta l'algoritmo che costruisce l'heap:

```

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

```

Possiamo sicuramente dire che ogni iterazione di Heapify è $\frac{n}{2} O(\log n)$ e che quindi sia nel caso peggiore un $O(n \log n)$. Già 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 esempio) 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. Ma 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 caso 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 $O(n \log n)$. Ma dobbiamo ragionare anche sulle altre altezze.

Per tutti gli alberi di stessa altezza i , il contributo di una singola chiamata Heapify su ognuno di questi alberi sarà una certa costante moltiplicata per l'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 0 per $i \rightarrow \infty$ ed è esattamente ciò che ci aspettiamo poiché 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:

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

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:

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

Dunque, $\sum_{i=1}^n (i \cdot x^i)$ si riduce alla risoluzione di una derivata di una serie geometrica; nel nostro caso abbiamo anche che la serie ha ragione minore di 1 e quindi non serve utilizzare la forma chiusa, infatti:

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

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

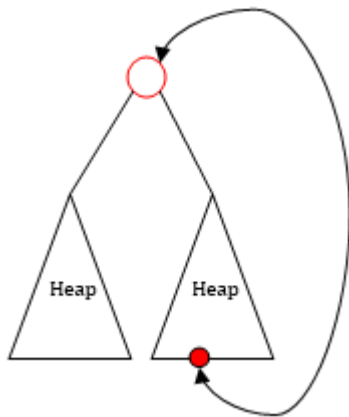
Ma quindi la nostra sommatoria cresce come una costante (essendo limitata sia superiormente che inferiormente da una costante), o più precisamente:

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

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

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. Dopodiché 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

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

Anche se abbiamo approssimato la sommatoria, questa volta 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 $n \log n$ (lo dice un teorema che dimostreremo in seguito) e quindi $\Omega(n \log n) \leq T_{HS}(n) \leq O(n \log n) \Rightarrow T_{HS}(n) = \Theta(n \log n)$

```

1  HeapSort(A, n)
2      CostruisciHeap(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 e mi trovo un "quasi" heap*/
7          heapsize = heapsize - 1
8          Heapify(A, 1)
```

N.B.: 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 fino ad ora; infatti, anche se impiega $\Theta(n \log n)$ come Merge Sort, a differenza di quest'ultimo non utilizza memoria aggiuntiva.

Quick Sort

L'idea è molto simile a quella del Merge Sort poiché sfrutta la tecnica del "Divide et Impera", ma la decomposizione avviene a livello di istanza, ovvero non dividerò banalmente la sequenza a metà ma lo farò in base al valore degli elementi di tale sequenza.

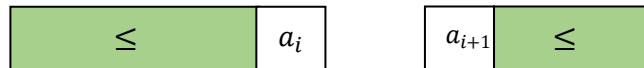
Praticamente la tecnica del divide et impera può essere così rappresentata:

➤ **RISOLVI**

- Decomponi in sottoproblemi
- **RISOLVI** i sottoproblemi
- Fondi soluzioni

Mentre il Merge Sort ha una decomposizione in sottoproblemi semplice 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 avrò le seguenti sequenze:



con $a_i \leq a_{i+1}$, e quindi poiché entrambe le sottosequenze sono già ordinate e l'ultimo elemento della prima sequenza è minore o uguale al primo elemento della seconda, la fusione dei due sarà già ordinata.

Il metodo usato da Quick Sort non riesce però a garantire che le due sottosequenze contengano più o meno lo stesso numero di elementi e ciò va ad impattare sul tempo di esecuzione; infatti, qualsiasi livello di ricorsione avrà da una parte una sequenza di q elementi e dall'altra $n - q$ con tutti gli elementi della sottosequenza sinistra minori o uguali rispetto a quelli dell'altra sottosequenza.

```
1 QuickSort(A, p, r)
2   IF p < r THEN
3       q = Partiziona(A, p, r)
4       QuickSort(A, p, q)
5       QuickSort(A, q + 1, r)
```

Siamo in una caso base? Sequenza vuota o di un solo elemento

Il punto in cui dividere la sequenza è più complicato da calcolare; Partiziona si occuperà di restituire l'indice che dividerà la sequenza e farà in modo che tutti gli elementi della partizione da p a q siano minori od uguali di quelli da $q + 1$ a 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, il "cervello" del nostro algoritmo", 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:

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

ossia, 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 due proprietà) e sia per il fatto di non dover fare nessun merge alla fine (l'ultima).

Vediamo di dimostrare le prime due 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, distruggerebbe l'algoritmo.

Supponiamo che $q = p - 1$. Avremmo una chiamata ricorsiva come segue: QuickSort($A, p, p - 1$), ovvero una istanza di 0 elementi; quindi, questa chiamata terminerebbe senza aver fatto nulla (è un caso base), ma ciò significa 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 $q = r$: QuickSort(A, p, r) sarà la chiamata che manderà in loop l'algoritmo e QuickSort($A, r + 1, r$) restituirà la sequenza vuota.

Quindi Partiziona deve garantire le due proprietà:

- 1) $p \leq q < r$ (per il motivo descritto precedentemente)
- 2) $\forall i : p \leq i \leq q \wedge \forall j : q + 1 \leq j \leq r, A[i] \leq A[j]$ (ovvio, altrimenti non si può avere la certezza che la sequenza totale sia già ordinata)

Partiziona

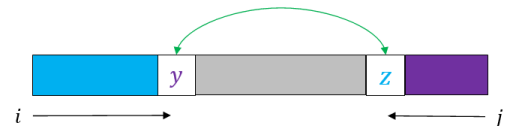
Quando viene chiamato Partiziona si ha la certezza che $p < r$, dunque possiamo scegliere un elemento $x \in A$, detto anche pivot, e metteremo nella parte sinistra tutti gli elementi minori o uguali di x , mentre a destra quelli maggiori o uguali.

N.B.: le condizioni di uguaglianza sono necessarie sia a destra che a sinistra per garantire che nessuna delle due sottosequenze sia vuota (altrimenti avremmo il problema del loop descritto precedentemente). Se ad esempio avrei 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.

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 due indici, un indice i che partirà da sinistra e un indice j che partirà da destra. L'indice i si sposterà a destra fino a quando incontrerà un valore $y \geq x$ (i valori minori sono già nella posizione giusta), mentre j verrà decrementato fino a quando non vi 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 $i < j$, scambierò i valori e ripeterò il processo fino a quando gli indici si incroceranno ($i \geq j$).



Nello scrivere l'algoritmo useremo il ciclo REPEAT ... UNTIL (condizione): questo ripeterà il blocco interno (facendo almeno una iterazione) fino a quando la condizione non sarà 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          REPEAT
10             i = i + 1
11          UNTIL A[i] >= x
12          IF i < j THEN
13              Swap(A, i, j)
14      UNTIL i >= j
15      RETURN j
```

Per confermare la correttezza di questo algoritmo vediamo 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 $p \leq q < r$. A tal proposito, dobbiamo dimostrare che i casi $j < p$ e $j \geq r$ siano impossibili; per far 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, questo viene per forza decrementato almeno una volta).

- **Caso $j = r$:** sappiamo che j viene decrementato almeno una volta e non riceve mai incrementi. Ora affinché sia $j = r$ significa che siamo al **primo** ciclo di iterazione del REPEAT esterno; quindi, bisogna dimostrare che i non arrivi mai a r , poiché è l'unico modo che ha per uscire dal ciclo con $j = r$. Ma se siamo alla prima iterazione allora $i = p - 1$ e al primo incremento di i (linea 10) non abbiamo ancora fatto alcuno scambio; dunque, x (il nostro 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 ho $i < j$ e quindi eseguo la linea 13 e 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 $j < r$ certamente.
- **Caso $j = p - 1$:** per dimostrare che questo caso non si verifichi basta garantire che il primo REPEAT interno (linea 6-8) non si ripeta all'infinito poiché il fatto che $p < r$ (sicuramente vero altrimenti non verrebbe eseguito Partiziona) garantisce che j si trovi tra p ed r . Poiché non possiamo garantire di essere alla prima iterazione non è detto che x sia ancora nella prima posizione della sequenza poiché potrebbero esserci stati degli scambi, ma se questi sono avvenuti significa certamente che in $A[p]$ c'è un valore minore od uguale a x e ciò garantisce che j si fermerà sicuramente in p (questo è un ragionamento molto 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). Ora poiché 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: $\forall z : p \leq z \leq j, A[z] \leq x$ e $\forall t : j + 1 \leq t \leq r, A[t] \geq x$.

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

È facile osservare che se richiedessi il $<$ invece del \leq (o, 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.

Analisi asintotica

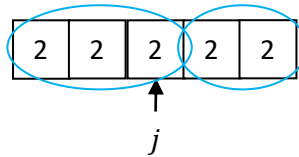
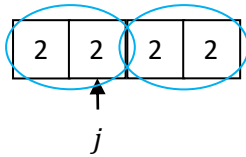
Per il QuickSort non bastano le tecniche viste sinora poiché non è tanto facile sapere quante volte le istruzioni di Partiziona vengono eseguite; in particolare per i "repeat until" interni posso solo dire che la somma delle loro iterazioni sia $n + 1$ o $n + 2$ (poiché l'algoritmo praticamente termina o con $i = j$ se l'input è dispari oppure con $i = j + 1$ se l'input è pari), essendo l'unica condizione di uscita $i \geq j$, e quindi che avranno un 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).

```
QuickSort(A, p, r)
  if p < r THEN
    q = Partiziona(a, p, r)
    QuickSort(A, p, q)
    QuickSort(A, q+1, r)
```

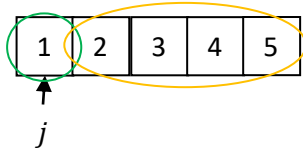
$$T_{QS}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{QS}(q) + T_{QS}(n - q) + \Theta(n) & \text{se } n > 1 \end{cases}$$

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

- Se tutti gli elementi sono uguali mi ritroverò con una divisione della sequenza in due parti quasi uguali e questo varrà per ogni chiamata ricorsiva:



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



Questi due casi li abbiamo già studiati in precedenza: infatti, se $q = \left\lfloor \frac{n}{2} \right\rfloor$ 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 $T_{QS}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{QS}(1) + T_{QS}(n-1) + \Theta(n) & \text{se } n > 1 \end{cases}$ e quindi $T_{QS}(n) = \Theta(n^2)$ (simile

all'[esempio sul fattoriale](#)). Più precisamente, per le sequenze ordinate avremo un albero degenerare dove un livello i -esimo (eccetto l'ultimo che è costante) ha un contributo di $n - i$, dunque:

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

Stranamente Quick Sort si comporta nel modo peggiore per sequenze dove tecnicamente non ci sarebbe nulla da dover fare.

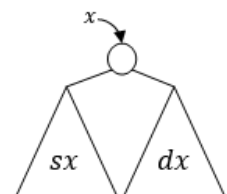
Si fa presente che a seconda del tipo di sequenza di dimensione n l'albero di ricorrenza potrebbe essere molto diverso; il nostro scopo è capire che tipo di forma potrebbero avere gli alberi generati dalla

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

Ovviamente vale $1 \leq q \leq n - 1$ altrimenti l'algoritmo non potrebbe terminare (abbiamo già discusso i casi per $q = 0$ e $q = n$), e da ciò ricaviamo che tutti i nodi interni hanno grado due poiché se non ci troviamo in un caso base l'algoritmo fa esattamente due chiamate ricorsive. Inoltre, poiché i casi base si hanno con sottosequenze di un elemento, è evidente che ci saranno tante foglie quanti sono gli elementi della sequenza in input; ciò rende facile dimostrare che se il numero di foglie n_f è k allora tutti i nodi interni n_i sono $k - 1$. Dunque, 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 due e quindi sia il sottoalbero destro che sinistro saranno non vuoti e con altezza $h < i$ e dunque $n_{f_{sx}} = n_{i_{sx}} + 1$ e $n_{f_{dx}} = n_{i_{dx}} + 1$, mentre il numero totale delle foglie dell'albero sarà semplicemente $n_f = n_{f_{sx}} + n_{f_{dx}}$ poiché 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 tutto ciò, risulta:

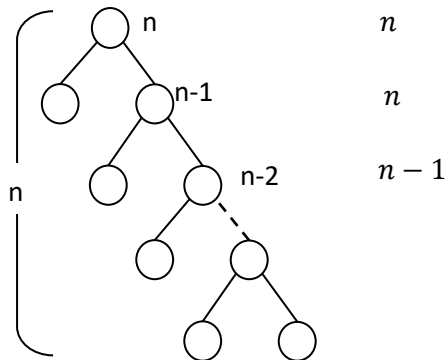


$$n_f = \underbrace{n_{isx} + 1}_{n_{f_{sx}}} + \underbrace{n_{idx} + 1}_{n_{f_{dx}}} = \underbrace{n_{isx} + n_{idx} + 1}_{n_i} + 1$$

ovvero che $n_f = n_i + 1$ come volevasi dimostrare.

Caso medio

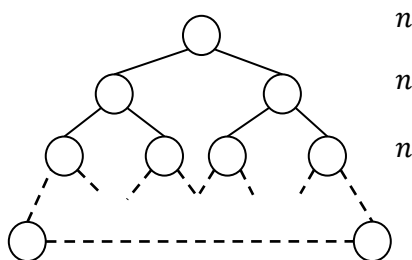
Prendiamo l'albero di ricorrenza del **caso peggiore** e andiamo a calcolare il contributo di ogni livello:



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

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

Per quanto riguarda il **caso migliore**, avremo:



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

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

È facile notare che, in qualunque albero io 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. Quindi risulta che il tempo di esecuzione dipende dall'altezza e poiché non posso 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 (poiché l'albero completo è quello 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 fatto per Insertion Sort.

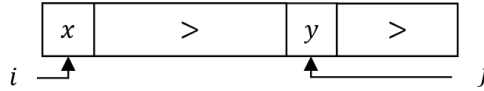
Decomponiamo il problema sapendo che ad ogni livello di ricorsione l'algoritmo si occupa di scegliere, tramite Partiziona, su una certa istanza il valore q e quindi ogni nodo sceglie localmente un valore di q ; fissato quest'ultimo verranno generati i due sottoproblemi con dimensioni q e $n - q$.

Calcoliamo il tempo medio di tutte le istanze in cui partiziona fa la stessa scelta, più precisamente dividiamo 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.

Dopo aver decomposto in classi calcoleremo la media di ogni classe e poi faremo una media aritmetica di queste. Vediamo come fare il suddetto partizionamento: prendiamo istanze di dimensioni arbitrarie in cui tutti gli elementi sono differenti; tale assunzione è una approssimazione per eccesso poiché più elementi uguali ci sono e meno sproporzionata sarà la partizione (si veda il 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 minore od uguale ad x nella sequenza. Essendo il pivot presente nella sequenza, avremo $r(x) \geq 1$ poiché almeno il pivot sarà minore o uguale di sé stesso. Più precisamente, se come pivot scelgo il minimo della sequenza, avrò esattamente $r(x) = 1$, mentre con il massimo sarà $r(x) = n$.

In realtà la scelta di q di cui parlavamo precedentemente è proprio il pivot, ma allora se il rango è uno avremo il pivot a sinistra e tutto il resto a destra; dunque $r(x) = 1 \Rightarrow q = 1$. Anche per $r(x) = 2 \Rightarrow q = 1$, poiché con due valori minori o uguali al pivot avremo il caso seguente (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 $r(x) = 3 \Rightarrow q = 2$, $r(x) = 4 \Rightarrow q = 3$, e così via (da due elementi in poi q crescerà sempre); quindi, per $r(x) \geq 2 \Rightarrow q = r(x) - 1$.

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 equivalenza saranno esattamente n :

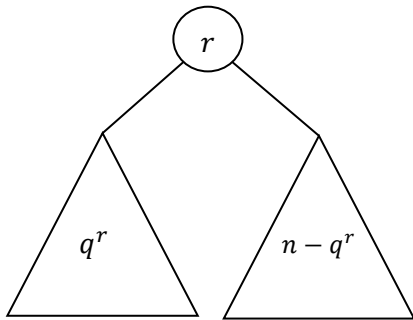
| | | | | | | |
|--------|---------|---------|--|-----|--|---------|
| $r(x)$ | 1 | 2 | | | | n |
| | T_M^1 | T_M^2 | | ... | | T_M^n |

Intuitivamente $T_M^r(n)$ rappresenta la funzione che dato il rango r dà 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:



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

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

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

che non sappiamo risolvere, ma possiamo semplificare secondo l'equivalenza $q^r = r - 1$. Infatti, scorporando il caso del rango $r = 1$, abbiamo una corrispondenza univoca per gli altri e quindi è possibile scrivere la precedente equazione come segue:

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

Sappiamo che $T_M(1) = \Theta(1)$ e che $T_M(n-1) = O(n^2)$ poiché non può essere peggio del caso peggiore (questa approssimazione vedremo che sarà influente); inoltre $r = q + 1$, ma allora:

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

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:

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

Ma questa equazione di ricorrenza non è così semplice da risolvere. A tal scopo introdurremo una tecnica di risoluzione che può essere usata anche per validare un'equazione di ricorrenza.

Tecnica per validare un'equazione di ricorrenza

Dimostriamo che $T_M(n) = O(n \log n)$ così da confermare $T_M(n) = \Theta(n \log n)$, e quindi bisogna verificare (useremo 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; infatti, poiché $\log 1 = 0$ risulterebbe $T_M(n) \leq 0$ e ciò è assurdo; dunque, il caso base è $n = 2$.

Grazie al fatto che $1 \leq q \leq n-1$ possiamo scrivere $T_M(q) \leq c(q \log q)$ che sarà la nostra ipotesi induttiva, ma allora, 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 ora vera la seguente proprietà che dimostreremo 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 $T_M(n) \leq c(n \log n)$. Ma sappiamo che il $\Theta(n)$ è assimilabile ad un kn , allora risulta $kn \leq \frac{cn}{4}$; la costante k è fissata dalla relazione Theta, ma la costante c può essere scelta arbitrariamente. Dunque, basta scegliere $c \geq 4k$ per concludere che $T_M(n) = O(n \log n)$ per $n \geq 2$.

Non abbiamo ancora verificato il caso base $n = 2$:

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

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

Ma allora 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 avere un tempo di esecuzione minore di $n \log n$, segue $T_M(n) = \Theta(n \log n)$.

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 $\sum_{q=1}^{n-1} (q \log q)$ è sfruttare il fatto che $q < n$ (banalmente ricavato da $1 \leq q \leq n-1$) e quindi $\log q \leq \log n \Rightarrow q \log q \leq q \log n$; dunque:

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

Anche se ci siamo avvicinati alla tesi, non abbiamo concluso nulla poiché $\frac{n \log n}{2} \leq \frac{n^2}{8}$, quindi significa che la nostra maggiorazione è stata troppo eccessiva. Quello che si può fare è “spezzare” la sommatoria in due parti così da fare delle approssimazioni più precise:

$$\begin{aligned} \sum_{q=1}^{n-1} (q \log q) &= \underbrace{\sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} (q \log q)}_{\substack{q < \lfloor \frac{n}{2} \rfloor \text{ quindi} \\ q \log q \leq q \log \frac{n}{2}}} + \underbrace{\sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} (q \log q)}_{\substack{\text{approssimiamo} \\ \text{come prima}}} \leq \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} \left(q \log \frac{n}{2} \right) + \sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} (q \log n) = \\ &= \log \frac{n}{2} \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} q + \log n \sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} q = (\log n - 1) \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} q + \log n \sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} q = \\ &= \underbrace{\log n \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} q + \log n \sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} q}_{\text{uniamo le due sommatorie}} - \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} q = \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} q \end{aligned}$$

Ora $\log n \sum_{q=1}^{n-1} q$ l'abbiamo già risolta, mentre poiché stiamo maggiorando non c'è problema a sottrarre qualcosa di più piccolo e quindi sfruttiamo il fatto che $\lfloor \frac{n}{2} \rfloor \geq \frac{n}{2}$:

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

Notiamo che $\frac{n \log n}{2} \geq \frac{n}{4} \Rightarrow \frac{n}{4} - \frac{n \log n}{2} \leq 0$, dunque, togliendo un valore negativo, la maggiorazione è valida.

Conclusioni su QuickSort

Abbiamo dimostrato che nel caso medio Quick Sort ha un comportamento ottimo: in realtà il tempo di esecuzione è quasi sempre $\Theta(n \log n)$ rendendolo nella pratica uno dei migliori algoritmi di ordinamento.

Problema generale sull'ordinamento

Abbiamo più volte detto che non è possibile ordinare una sequenza **arbitraria** di lunghezza n in meno di $n \log n$. In questo paragrafo ci occuperemo di dimostrare tale enunciato.

Nota bene che esistono algoritmi in grado di risolvere l'ordinamento per una specifica sequenza in maniera lineare, ma queste tecniche si basano appunto sul fatto che l'input non sia arbitrario ma abbia determinate proprietà (il Counting Sort ne è un buon esempio).

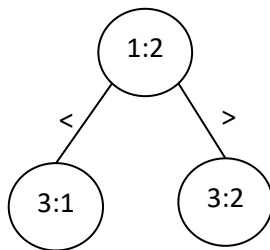
Se parliamo di sequenze arbitrarie è invece possibile dimostrare che caso medio e peggiore (non vale per il caso migliore) richiedano un tempo minimo di $n \log n$. Fondamentale è definire lo spazio delle possibili operazioni per un generale algoritmo di ordinamento.

Una cosa certa è che il confronto degli elementi è un'operazione essenziale per l'ordinamento e quindi il problema può essere risolto solo tramite confronti; lo scambio invece non è essenziale, lo usiamo solo per tener conto dei confronti. Quindi, da un punto di vista astratto possiamo tener traccia solo dei confronti ignorando gli scambi.

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

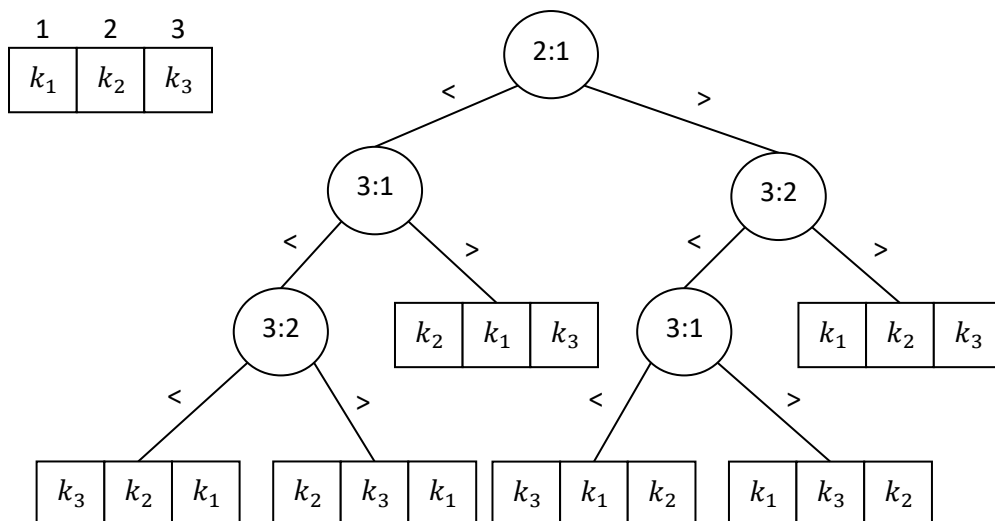
Alberi di decisione

La natura di un confronto è semplicemente una funzione booleana, ragion per cui un confronto può essere visto come un nodo con all'interno due 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 prossimo elemento da confrontare sia lo stesso a prescindere dall'esito, potrei ad esempio confrontare 3: 2 se $k_1 > k_2$ (quindi mi sposto sul figlio destro) oppure 3: 1 se $k_1 < k_2$,

Mostriamo un esempio più concreto ordinando una sequenza di 3 elementi sfruttando gli esiti dell'albero di decisione:



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 e le foglie le soluzioni dovute a tali scelte.

Si noti 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.

Algoritmi di ordinamento e alberi di decisione

Implicitamente ogni algoritmo di ordinamento che si basa su confronti crea un albero di decisione. Tuttavia, poiché un albero di decisione ci permette di ordinare una qualsiasi sequenza di dimensione fissata n , mentre un algoritmo di ordinamento vale per qualsiasi n è evidente che non c'è una corrispondenza univoca tra i due.

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

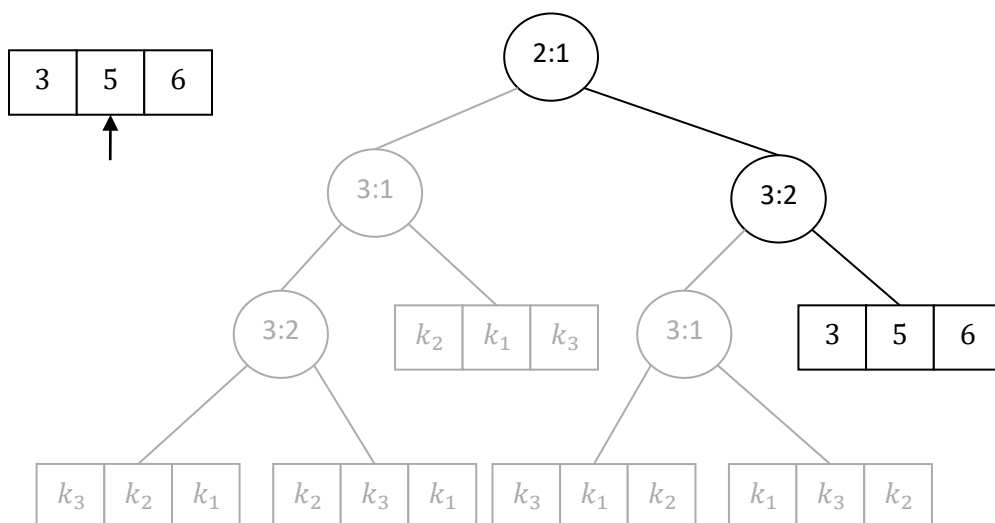
- `AlgoritmoOrdinamento(1)` è associato ad un albero di decisione di ordine 1
- `AlgoritmoOrdinamento(2)` avrà un albero di ordine 2
- ...

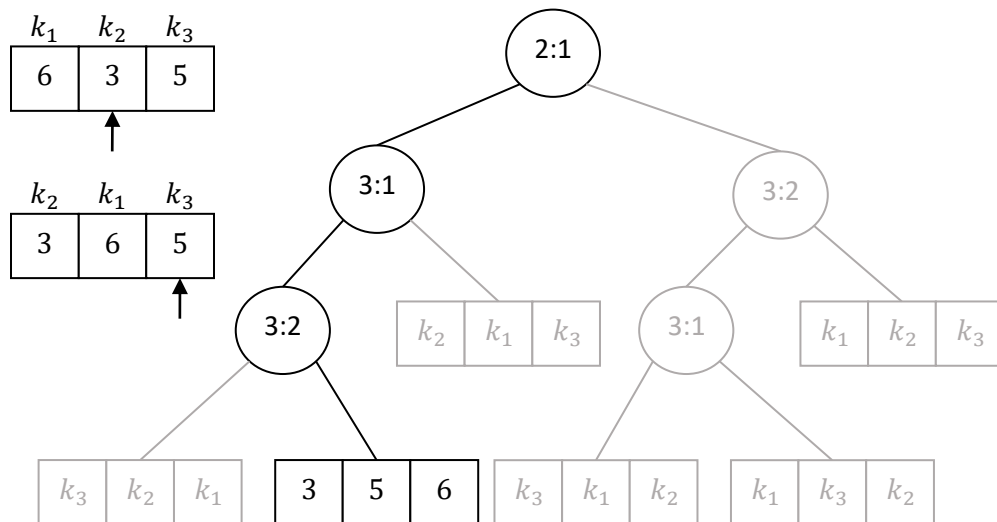
Ma ad algoritmi di ordinamento diversi possono esserci due alberi di decisione differenti (basti pensare che non esiste un solo albero di decisione per una sequenza di ordine 3, dipende dall'ordine dei confronti).

Definiamo le proprietà che accomunano tutti gli 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 in input, quindi $n!$
- Il percorso dalla radice a qualsiasi foglia confronta come minimo tutti gli elementi adiacenti.
Es.: 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 (o 3:2)
- Non posso avere meno di $n - 1$ nodi interni, poiché il numero di coppie adiacenti in una sequenza è proprio $n - 1$ (è una conseguenza della precedente proprietà)

Negli alberi di decisione noi andiamo a rappresentare solo i confronti ma non gli scambi poiché supponiamo di avere memoria di tutti i confronti, ed è quindi possibile ordinare la sequenza direttamente alla fine senza bisogno di passaggi intermedi. Tenendo conto di ciò presentiamo un esempio di albero di decisione per l'algoritmo [InsertionSort\(4,3\)](#):





Quindi, fissato n , se prendo l'albero di decisione di ordine n ed una sua istanza, il numero di operazioni elementari che un algoritmo di ordinamento fa 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 significa anche che il numero di confronti che un algoritmo può fare è relazionato all'altezza dell'albero, che nel caso di Insertion Sort sarà quadratico (per numeri piccoli non si nota).

Dimostrazione del teorema sull'ordinamento

Fissato n ho un numero finito di alberi di decisione; Il nostro scopo sarà vedere, tra tutti gli alberi, quello con altezza minima, così da trovare il numero minimo di confronti che un algoritmo fa nel caso peggiore (praticamente 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 poiché sappiamo che in un albero binario il numero di foglie è al massimo 2^h posso scrivere $n! \leq 2^h \Rightarrow h \geq \log(n!)$; ne consegue che l'altezza di un albero di decisione non può essere meno di $\log(n!)$. Ma cerchiamo di tradurla in una forma più utile:

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

Poiché a noi interessa una stima di quella sommatoria, proviamo ad approssimarla (faremo qualcosa di simile a quanto visto in "[Maggiorazione di una sommatoria](#)"). Cominciamo cercando qualcosa di più piccolo così da avere $h \geq \sum_{i=1}^n \log i \geq ()$. Sappiamo che

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

ma ciò non è molto utile poiché il nostro scopo è garantire che h abbia un limite inferiore asintotico.

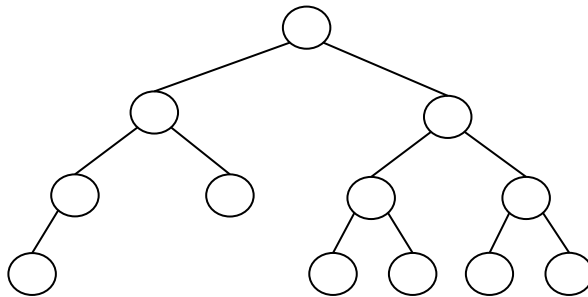
$$\sum_{i=1}^n \log i \geq \underbrace{\sum_{i=\lfloor \frac{n}{2} \rfloor}^n \log i}_{\text{sommò meno valori}} \geq \underbrace{\sum_{i=\lfloor \frac{n}{2} \rfloor}^n \log \frac{n}{2}}_{\text{sommò sempre il più piccolo } i} = \log \frac{n}{2} \cdot \sum_{i=\lfloor \frac{n}{2} \rfloor}^n 1 = \frac{n}{2}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

Ma allora abbiamo raggiunto il nostro scopo, infatti:

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

Quindi per il caso peggiore abbiamo dimostrato che 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 $\Theta(n \log n)$.

Il caso medio è un po' più complesso, infatti per poter calcolare il suo tempo di esecuzione bisognerà fare una media aritmetica tra la lunghezza del percorso esterno (la somma dei percorsi dalla radice a ciascuna foglia) e il numero delle foglie dell'albero. Esempio:



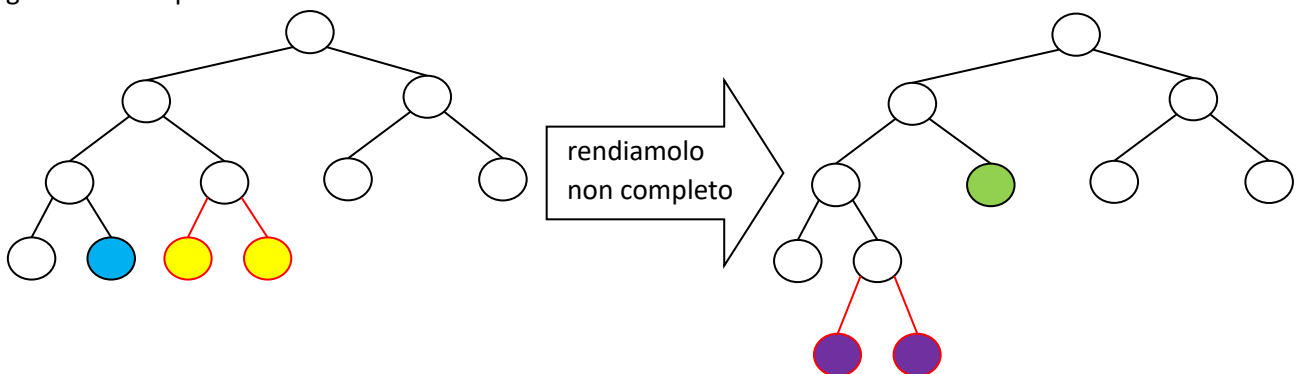
$$T_M(n) = \frac{\text{LPE}}{\# \text{foglie}}$$

Calcoliamo la LPE da sinistra a destra:

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

Ma come fatto nel caso peggiore, bisogna studiare il miglior caso medio e quindi innanzitutto cercare un tipo di albero che minimizzi il percorso esterno (poiché per minimizzare quel rapporto o minimizziamo il numeratore o massimizziamo il denominatore, ma ovviamente non possiamo cambiare il numero di foglie).

A questo punto dimostriamo che gli alberi binari che minimizzano la lunghezza del percorso esterno sono gli alberi completi:



Andiamo a calcolare entrambi i percorsi esterni mettendoli in relazione. Sia la lunghezza del percorso completo pari a k , andiamo a calcolare il percorso dell'albero non completo in funzione di k sapendo che h sia l'altezza dell'albero completo:

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

Dunque, il percorso esterno peggiora all'allontanarsi dalla completezza come abbiamo assunto.

Ora, per un albero completo sappiamo che le foglie sono ad altezza h o ad altezza $h - 1$; siano 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 è 2^h , che in relazione alle foglie del nostro albero risulta: $2N_{h-1} + N_h = 2^h$ (ogni foglia ad altezza $h - 1$ avrà due figli). A questo punto abbiamo un semplice sistema a due equazioni in due incognite:

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

Ora abbiamo tutte le informazioni necessarie per calcolare il tempo medio:

$$T_M(n) = \frac{\text{LPE}}{n!} = \frac{h \cdot N_h + (h - 1)N_{h-1}}{n!} = \frac{2hn! - h2^h + h2^h - hn! - 2^h + n!}{n!} =$$

$$= \frac{hn! - 2^h + n!}{n!} = h - \frac{2^h}{n!} + 1 \xrightarrow{h=\log n!} T_M(n) = \log n! - \frac{2^{\log n!}}{n!} + 1 = \log n!$$

Ma abbiamo già 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 nel caso peggiore, migliore di $\Theta(n \log n)$.

5. Strutture dati elementari

Struttura dati concreta e struttura dati astratta

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

Ad esempio, 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). Ma ora vediamo come rappresentare un insieme di dati dinamico S (con insieme dinamico si intende una collezione di elementi variabile nel tempo, quindi è possibile aggiungere o rimuovere elementi); considereremo un elemento un oggetto atomico e quindi la sua struttura sarà irrilevante poiché il concetto di struttura dati è sempre lo stesso indipendentemente da come è fatto il dato, ma si noti 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 (per intenderci, confrontare un array è molto diverso da confrontare un intero).

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) . Dunque, per tale insieme possiamo definire le seguenti operazioni:

- Ricerca(S, k): restituisce un elemento di S oppure NIL se $k \notin S$
- Inserimento(S, k): restituisce un nuovo insieme $S' = S \cup \{k\}$
- Cancellazione(S, k): restituisce un nuovo insieme $S' = S \setminus \{k\}$

Si noti che la presenza o meno dell'elemento k per le operazioni di inserimento e cancellazione (che sono simili tra loro) non cambia la loro definizione: 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 esempio, 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 due, più vincoli ho e più complesso sarà definire le operazioni.

L'operazione di ricerca, solitamente, 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 esempio, per $S = \{3, 6, 8\}$ avremo Successore($S, 6$) = 8, Predecessore($S, 6$) = 3 e Successore($S, 9$) = 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.

Array (non) ordinato

Abbiamo implicitamente utilizzato gli array negli algoritmi di ordinamento visti nei capitoli precedenti; vediamo ora come si comporta un insieme S rappresentato come array. Banalmente, se volessimo implementare un algoritmo del tipo RicercaMax(S), esso richiederebbe tempo $\Theta(n)$, ma se lo rappresentassimo tramite 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 $S[n]$).

Poiché ogni array ordinato è anche un array, un algoritmo di ricerca definito per un normale array funzionerà anche su un array ordinato, ma il viceversa non è vero; infatti, un array non è detto che sia ordinato quindi un algoritmo definito sfruttando le proprietà tipiche di un array ordinato non è detto che sia corretto su un qualsiasi array. In generale:

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

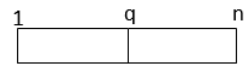
Se invece siamo nell'operazione di inserimento, per un array non ordinato è un $\Theta(1)$ (basta aggiungere alla fine) ma se l'array è ordinato, poiché dobbiamo mantenere tale proprietà, l'algoritmo di inserimento avrà tempo $\Theta(n)$ (in quanto una volta inserito il dato nella giusta posizione bisognerà traslare tutti gli elementi successivi).

Da questo ragionamento consegue che scegliere i vincoli della struttura dati ha un impatto rilevante sulla complessità; quindi, se ad esempio ci troviamo di fronte ad insiemi "poco dinamici" conviene usare un array ordinato, se invece abbiamo un insieme in cui bisogna fare poche ricerche e molti inserimenti/cancellazioni evidentemente useremo un array non ordinato.

Ricerca binaria

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

Ricerca(S, k): vado a dividere la sequenza in 2, quindi $q = \left\lfloor \frac{n}{2} \right\rfloor$



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

Mi fermerò o nel momento in cui ho trovato l'elemento oppure quando mi trovo in una sequenza di un solo elemento (in tal caso, se la cella non contiene l'elemento, allora $k \notin S$)

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 di dimensione $\frac{n}{2^2}$, dopo la terza in una di $\frac{n}{2^3}$, etc...

Dunque dopo i operazioni avrò una sequenza di grandezza $\frac{n}{2^i}$, ma allora raggiungerò la soluzione quando dopo i operazioni avrò una sequenza di un elemento; quindi $\frac{n}{2^i} = 1 \Rightarrow i = \log n$.

Avere una sequenza ordinata permette, come appena visto, di ottenere un algoritmo di ricerca di tempo logaritmo: 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
9              ret = RicercaBinaria(S, q+1, r, k)
10     ELSE
11         ret = p
12     RETURN ret

```

Restituiamo p poiché r non è detto che appartenga all'array essendo che $q + 1$ potrebbe essere un indice al di fuori della sequenza, infatti $p \leq q < r \Rightarrow q + 1 \leq r$ (dopo i operazioni potrebbe esserci la chiamata $RicercaBinaria(S, n, n + 1, k)$ che restituirebbe $n + 1$); quindi con p sono sicuro 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

```

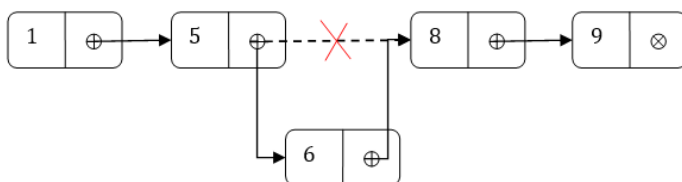
Tutto quello di cui abbiamo discusso finora ci dice che il problema della ricerca è risolvibile in un tempo **esponenzialmente** minore di lineare (ovvero logaritmico). Ma array ordinati hanno anche argomenti a sfavore poiché operazioni di inserimento e/o cancellazione sono lineari (anche se in realtà 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 ordinati con operazioni di modifica molto meno dispendiose.

Liste

La lista rappresenta una struttura dati dinamica dove le operazioni di inserimento e cancellazione sono meno dispendiose, a differenza di quanto accade negli array (che sono implementati come struttura statica, il che rende problematiche le suddette operazioni).

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 gli elementi in una qualsiasi area di memoria rendendo le operazioni di inserimento e cancellazione eseguibili in tempo costante; infatti, la sua struttura è composta da nodi, i quali contengono sia un certo dato, sia un'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 necessarie per creare un nuovo nodo e modificare i puntatori.

Il problema delle liste è però la ricerca, infatti per la loro struttura non è possibile accedere direttamente ad un qualsiasi elemento come negli array, ma bisogna obbligatoriamente partire dall'unico nodo di cui si conosce l'indirizzo (tipicamente il primo) e scorrere la lista fino al raggiungimento dell'elemento desiderato. Questo significa che accedere ad un elemento in mezzo alla lista richiederà tempo lineare poiché bisognerà fare $\frac{n}{2}$ letture in memoria.

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

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

Definizione formale di lista e algoritmo di ricerca

Una lista L è un oggetto con le seguenti proprietà:

- 1) L 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 sulle liste; 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 = Ricerca(L->next, k)
7      ELSE
8          ret = L /* se arriviamo qui significa che L = NIL*/
9      RETURN ret

```

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, come già visto, la lista è una struttura dati ad accesso **strettamente sequenziale** e questo comporta un tempo lineare per tutti gli algoritmi che comportano una ricerca.

Le liste che considereremo nei nostri algoritmi sono del tipo descritto sopra e quindi con un solo puntatore, ma diamo di seguito le definizioni di altri tipi di lista:

- Una **lista doppia puntata** è un insieme dinamico in cui ogni elemento ha:
 - una chiave (key);
 - un riferimento (next) all'elemento successivo dell'insieme;
 - un riferimento (prev) all'elemento precedente dell'insieme.
- Una **lista circolare puntata** è un insieme dinamico in cui:
 - ogni elemento ha una chiave (key) ed un riferimento (next) all'elemento successivo dell'insieme;
 - l'ultimo elemento ha un riferimento alla testa della lista.

- Una **lista circolare doppiamente puntata** è un insieme dinamico in cui:
 - ogni elemento ha 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 nella memoria, l'inserimento di un nuovo nodo in testa alla lista è banale:

```

1  Insert(L, k)
2      tmp = creanodo()
3      tmp->key = k
4      tmp->next = L
5      L = tmp
6      RETURN L

```

Ovviamente il precedente algoritmo è costante, ma se volessi inserire il nodo in una determinata posizione (e quindi con annessa ricerca), ad esempio in una lista ordinata, l'algoritmo sarebbe comunque lineare anche se il nodo venisse aggiunto e creato in tempo costante:

```

1  OrdInsert(L, k)
2      IF L != NIL AND L->key < k THEN
3          L->next = OrdInsert(L->next, k)
4      ELSE /*chiave k è la più piccola in L*/
5          tmp = creanodo()
6          tmp->key = k
7          tmp->next = L
8          L = tmp
9      RETURN L

```

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

```

1  InsertUnica(L, k)
2      IF L = NIL THEN
3          tmp = creanodo()
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      RETURN L

```

Per la cancellazione di un elemento ovviamente non si può far a meno della ricerca, ed inoltre i casi ricorsivi da gestire sono due (ci si basa sempre sulla definizione della lista): o la lista è vuota, oppure contiene un nodo con una chiave ed un riferimento ad un'altra lista. Dunque, avremo il seguente algoritmo lineare:

```

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

```

Di seguito riportiamo altri esempi di algoritmi di cancellazione ($x\%y$ è la funzione x modulo y):

```
1  /*cancella elementi pari*/
2  CancellaPari(L)
3      IF L != NIL THEN
4          L->next = CancellaPari(L->next)
5          IF (L->key)%2 = 0 THEN
6              tmp = L
7              L = L->next
8              dealloca(tmp)
9      RETURN L
```

- Poiché la chiamata ricorsiva è fatta prima dell'eventuale cancellazione dell'elemento, l'algoritmo andrà praticamente a cancellare gli elementi partendo dall'ultimo.

Se volessimo restituire il numero degli elementi cancellati potremmo scrivere il seguente algoritmo:

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

- Poiché l'algoritmo ricorsivo non restituirà più una lista ma un intero che rappresenta il conteggio, abbiamo bisogno di, oltre al nodo da cancellare, anche il suo precedente, altrimenti non avremmo modo di ricollegare la lista.

Ne consegue che la testa va trattata in maniera differente; dunque, una funzione che cancelli gli elementi pari di una lista stampando i numeri degli elementi cancellati può essere implementata nel seguente modo:

```
1  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 = counter + 1
8      print(counter)
9      RETURN L
```

Alberi binari

Un albero è un insieme dinamico che è

- **vuoto** oppure
- composto da $k + 1$ **insiemi disgiunti** di nodi:
 - un insieme di cardinalità uno, detto **nodo radice**
 - k alberi, ciascuno dei quali è detto **sottoalbero i -esimo** della radice (dove $1 \leq i \leq k$)

Un tale albero si dice di **grado k** ; quando $k = 2$, l'albero si dice **binario**

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

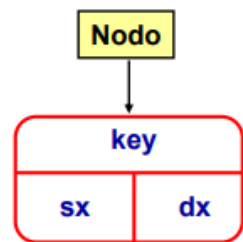
- **Visita in profondità** (verticale): si visitano tutti i nodi lungo un percorso, poi quelli lungo un altro percorso, etc...
- **Visita in ampiezza** (orizzontale): si visita l'albero per livelli, ossia prima si visitano tutti i nodi a livello 0, poi quelli a livello 1, ..., poi quelli a livello h

Visite in profondità

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

- **Visita pre-order:** prima si visita il nodo e poi i suoi sottoalberi (quindi radice, sottoalbero sinistro e poi sottoalbero destro)
- **Visita post-order:** prima si visitano i sottoalberi, poi il nodo (quindi prima il sottoalbero sinistro, poi il sottoalbero destro ed infine la radice)
- **Visita in ordine:** prima si visita il sottoalbero sinistro, poi il nodo ed infine il sottoalbero destro.

Quest'ultimo tipo di visita è possibile solo per alberi binari poiché si perderebbe la sua proprietà per alberi con cardinalità maggiore di due (per intenderci: se ho tre figli, visito il mio nodo prima o dopo il sottoalbero centrale?)

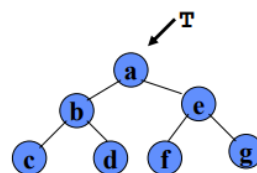


Implementiamo di seguito gli algoritmi di visita in profondità (con visita(*x*) intendiamo un qualsiasi algoritmo che operi sul nodo; inoltre, assumiamo che tale operazione venga fatta in tempo costante):

```

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

```

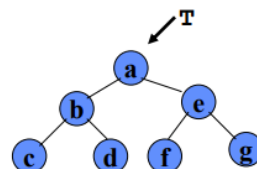


Ordine di visita:
a b c d e f g

```

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

```

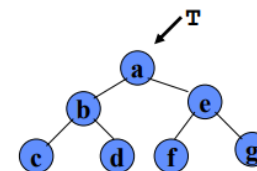


Ordine di visita:
c d b f g e a

```

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

```



Ordine di visita:
c b d a f e g

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 solo avere una differenza dal punto di vista asintotico) ma ci sono alcuni tipi di algoritmi che possono essere risolti solo con un determinato tipo di visita.

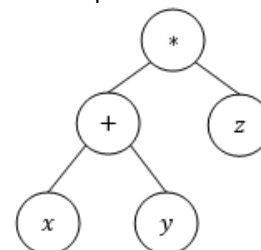
Un esempio potrebbe essere un algoritmo che risolve un'espressione aritmetica:

```

1  Eval(T, A)
2      IF T->key è una variabile THEN
3          RETURN A[T->key]
4      ELSE /*visito 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



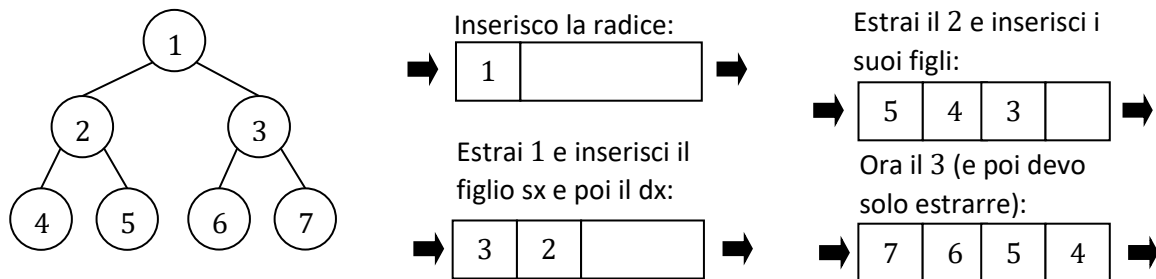
A:

| | | |
|----------------------|----------------------|----------------------|
| <i>x</i> | <i>y</i> | <i>z</i> |
| <i>k_x</i> | <i>k_y</i> | <i>k_z</i> |

Visita in ampiezza (BFS)

La difficoltà che si incontra in questo tipo di visita è dovuta al fatto che i nodi di ogni livello non sono collegati tra di loro (abbiamo collegamenti solo per le discendenze dirette); quindi, 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 la risoluzione di questo problema è la **coda** (inserisco gli elementi da un lato e li estraggo dall'altro). Praticamente inserisco in coda tutti i figli del nodo che sto visitando ed il prossimo nodo da visitare sarà proprio il primo inserito nella coda:



A questo punto l'implementazione dell'algoritmo per una visita in ampiezza è molto semplice (questo algoritmo viene conosciuto come BFS, breadth-first search):

```

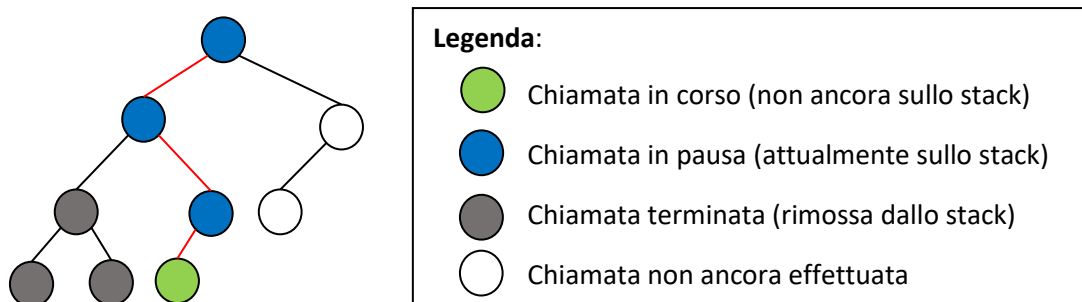
1  BFS(T)
2      Q = {T}
3      WHILE Q != {} DO
4          x = Testa(Q)
5          visita(x)
6          Q = Accoda(Q, x->sx)
7          Q = Accoda(Q, x->dx)
8          Q = Decoda(Q)

```

Il tempo di esecuzione sarà identico a quello delle visite in profondità; infatti, supponendo che l'algoritmo visita sia costante, il tempo di esecuzione sarà pari al numero di nodi e quindi è lineare. La coda non incrementa il tempo asintotico poiché (supposto che sia implementata decentemente, ad esempio come una lista con un puntatore in testa ed uno in coda) le operazioni su di essa avvengono in tempo costante.

Memoria aggiuntiva

Vediamo quanta memoria aggiuntiva richiedono gli algoritmi di visita, partendo da quelli in profondità: per una visita in profondità si richiede una quantità di memoria aggiuntiva pari alla lunghezza del percorso più lungo poiché l'algoritmo, quando visita un nodo, fa una chiamata ricorsiva al figlio (è proprio la chiamata ricorsiva che richiede memoria aggiuntiva per essere gestita), e quindi viene 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 esempio:



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

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

Questa analisi ci fa comprendere come non abbia senso modificare la struttura dell'albero aggiungendo altri puntatori (ad esempio da figlio a padre) poiché, non solo richiederebbe uno spreco di memoria maggiore rispetto alla ricorsione, ma renderebbe anche gli algoritmi più complicati da implementare.

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 poiché 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 . Ma allora la massima memoria aggiuntiva possibile è all'inizio dell'ultimo livello di un albero pieno (quindi tanta memoria quante sono le foglie), mentre ho uno spreco minimo di memoria per un albero degenerare (opposto alla visita in profondità):

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

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.

6. Alberi Binari di Ricerca (ABR)

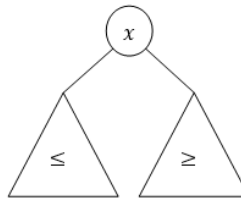
Definizione di ABR

Gli alberi binari di ricerca, o alberi binari ordinati, hanno un opportuno vincolo di ordinamento che darà dei progressi sul punto di vista asintotico.

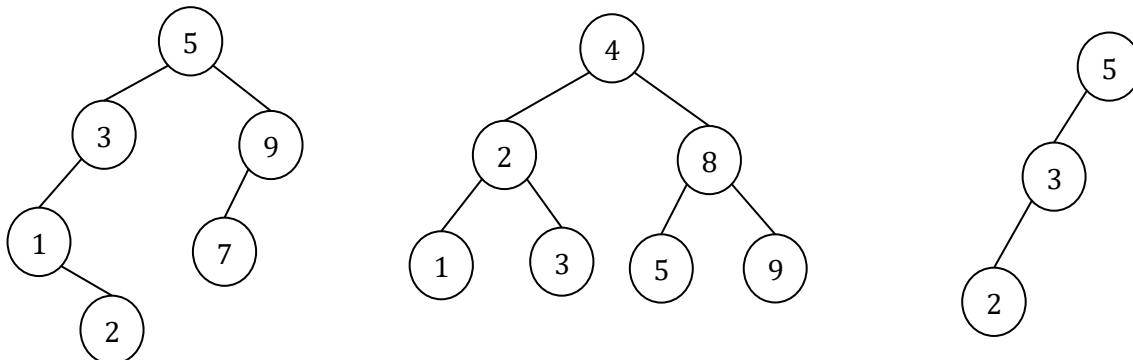
Un albero binario di ricerca T :

- È un albero binario
 - vuoto oppure composto da 3 insiemi disgiunti di nodi: un insieme di cardinalità uno, detto nodo radice e due sottoalberi, detti sottoalbero sinistro e destro, rispettivamente.
- $\forall x \in T$
 - $\forall y \in x \rightarrow sx \quad \text{val}(y) \leq \text{val}(x)$
 - $\forall y \in x \rightarrow dx \quad \text{val}(x) \leq \text{val}(y)$

(ogni nodo è maggiore o uguale di tutti gli elementi del suo sottoalbero sinistro e minore o uguale di tutti gli elementi del sottoalbero destro)



Si noti che a differenza degli alberi heap negli alberi binari di ricerca si ha un ordinamento **totale** tra gli elementi. Di seguito alcuni esempi di alberi binari di ricerca:

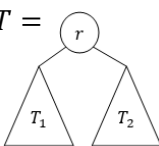


La prima conseguenza di questa definizione è che se ho un albero T allora sia il sottoalbero sinistro che quello destro sono a loro volta alberi binari di ricerca.

Definizione ricorsiva

T è ABR se

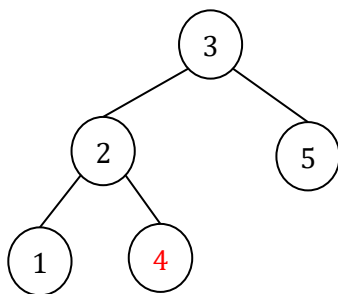
- 1) $T = \emptyset$
- 2) $T =$



T_1 è ABR e $\forall y \in T_1, \text{val}(y) \leq \text{val}(r)$
 T_2 è ABR e $\forall y \in T_2, \text{val}(r) \leq \text{val}(y)$

N.B.: non confondere la definizione di ABR con quella di [albero heap](#), infatti anche se un ABR rispetta sempre le regole di albero heap, il viceversa non è vero.

Il seguente albero è uno heap ma non un ABR:



State attenti alla precisione della definizione, infatti se volessimo scrivere un algoritmo che verifica se un determinato albero è un ABR non basta verificare che ogni nodo abbia un figlio sinistro con dato minore ad esso ed un figlio destro con dato maggiore. Infatti, è l'intero sottoalbero sinistro/destro a dover rispettare la relazione d'ordine.

Esercizio: implementare un algoritmo che in tempo lineare sia in grado di dire se l'albero passato in input sia o meno un ABR (suggerimento: applicare un tipo di visita in profondità)

Mia soluzione:

```

isABR(T)
    min = SearchMin(T)
    max = SearchMax(T)
    RETURN isABR(T, min, max)

1  isABR(T, min, max)
2      IF T != NIL THEN
3          IF T->key < min OR T->key > max THEN
4              RETURN false
5          sxABR = isABR(T->sx, min, T->key)
6          dxABR = isABR(T->dx, T->key, max)
7          IF !sxABR OR !dxABR THEN
8              RETURN false
9      RETURN true

```

Operazioni su alberi binari

Ricerca

Supponiamo di avere un ABR e vediamo di implementare un algoritmo che verifichi se un elemento $k \in T$. L'idea è abbastanza semplice, infatti avendo accesso al nodo radice x basta confrontarlo con k :

- $x = k \Rightarrow$ ho trovato l'elemento
- $k < x \Rightarrow$ il dato si trova nel sottoalbero sinistro
- $k > x \Rightarrow$ il dato si trova nel sottoalbero destro

Ma questo ragionamento è lo stesso che abbiamo applicato alla [ricerca binaria](#) in un vettore; dunque, è chiaro che un ABR ci permette di definire in modo naturale (non è più l'algoritmo a decidere la partizione

bensi 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          ELSE IF k > T->key THEN
6              RETURN Search(T->dx, k)
7      RETURN T

```

Si noti che il numero massimo di confronti nel caso peggiore sarà pari alla lunghezza del percorso più lungo; quindi, il tempo di esecuzione è pari all'altezza dell'albero. Ne consegue:

$$\underbrace{\log n}_{\text{albero completo}} \leq T(n) \leq \underbrace{n}_{\text{albero degenerare}}$$

Quindi se la forma dell'albero è buona abbiamo un significativo miglioramento rispetto ad un comune albero binario; inoltre, si potrebbe dimostrare (non lo faremo) che in realtà il tempo medio di questo algoritmo (quindi considerando tutte le forme) è un $\Theta(\log n)$.

Inserimento

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.

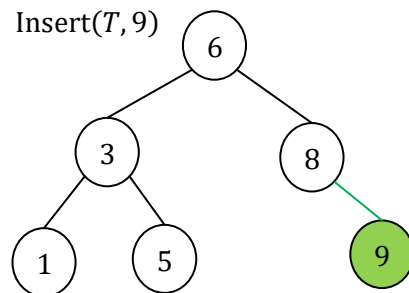
Supponiamo di voler inserire un elemento in un albero T , ovvero, di volere un albero $T' = T \cup \{k\}$ (assumiamo che l'albero non debba avere chiavi duplicate). Ci sono solo due possibili casi: l'albero iniziale $T = \emptyset \Rightarrow T' = \{k\}$ (questa è la situazione ideale), oppure T è non vuoto e quindi k va inserito nel giusto sottoalbero vuoto. Infatti, se $k \notin T$ allora raggiungerò un nodo con un sottoalbero vuoto (alla peggio raggiungo una foglia che li ha entrambi vuoti) dove va inserito k .

Si noti che tale algoritmo è una variante della ricerca, quindi avrà la sua complessità.

```

1  Insert(T, k)
2      IF T = NIL THEN
3          x = allocanodo()
4          x->key = k
5          x->sx = x->dx = NIL
6          RETURN x
7      ELSE IF k < T->key THEN
8          T->sx = Insert(T->sx, k)
9      ELSE IF k > T->key THEN
10         T->dx = Insert(T->dx, k)
11         /*se nessuna condizione è verificata*/
12         /*significa che k appartiene a T*/
13     RETURN T

```



Come con la ricerca, anche la complessità di Insert è $T(h) = O(h)$, quindi, nella complessità degli alberi binari non è più rilevante la cardinalità dell'insieme (il numero di nodi) bensì l'altezza dell'albero.

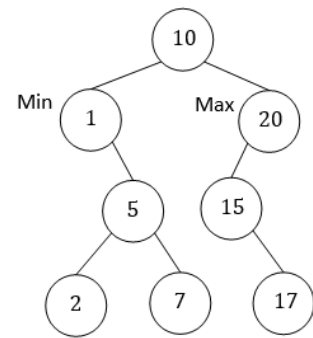
Abbiamo visto che mantenere intatta la struttura di ABR in un inserimento è abbastanza semplice, mentre per quanto riguarda la cancellazione bisogna fare più attenzione. Come è facile intuire, mentre l'inserimento viene sempre fatto in una situazione a noi vantaggiosa (sottoalbero vuoto), l'algoritmo di cancellazione è più complesso e richiede l'ausilio di altri algoritmi per facilitarne l'implementazione.

Ricerca del minimo e del massimo

La ricerca del minimo e del massimo sono banali in un ABR grazie proprio al vincolo di ordinamento che questi offrono: infatti, si trovano al nodo più a sinistra ed al nodo più a destra, rispettivamente.

Ne consegue che la ricerca sarà lineare sull'altezza poiché dobbiamo soltanto 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 un figlio destro). Dualmente, il massimo sarà il primo nodo del percorso estremo destro che non avrà figlio destro.



Ragionando sul minimo per induzione, posso avere o un albero vuoto (e quindi restituirò NIL), oppure avrò almeno un nodo che o è il minimo (se non ha figlio sinistro) oppure significa che il minimo si trova nel suo sottoalbero sinistro, dove ripeterò il precedente ragionamento. Dunque, avrò 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 ret
```

Volendo si potrebbe implementare in maniera puramente iterativa così da non dover usare memoria aggiuntiva:

```
1 SearchMinIter(T)
2   node = T /*conviene sempre non modificare l'input*/
3           /*per non perdere il riferimento*/
4   IF T != NIL THEN
5       WHILE node->sx != NIL DO
6           node = node->sx
7   RETURN node
```

Anche se il massimo è esattamente il duale riportiamo per completezza gli algoritmi:

```
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
```

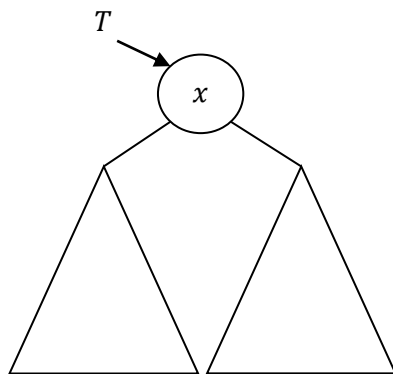
```
1 SearchMaxIter(T)
2   node = T
3   IF T != NIL THEN
4       WHILE node->dx != NIL DO
5           node = node->dx
6   RETURN node
```

Ricerca del successore e del predecessore

Prima di passare all'idea dell'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 tre possibili casi:



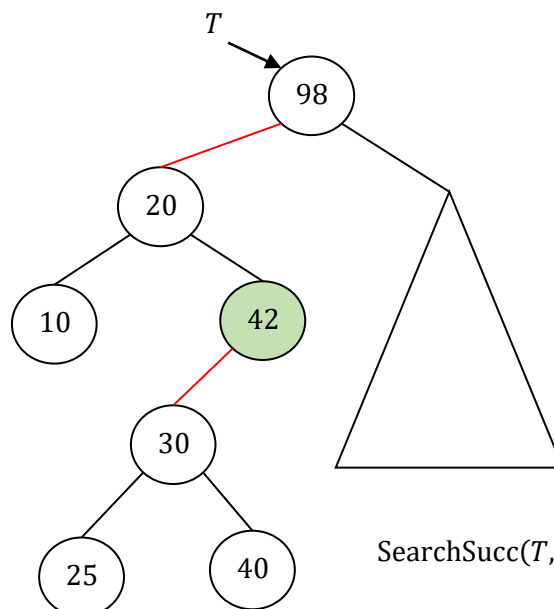
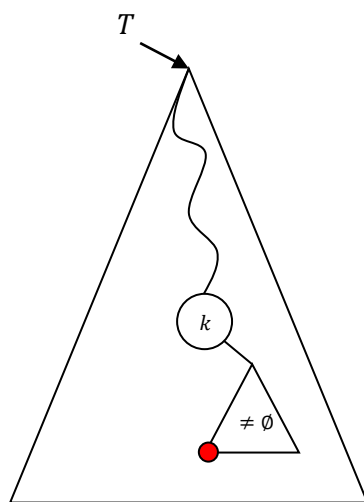
- $x < k$: la radice contiene un valore minore della chiave, quindi se il successore esiste sarà a destra di x (ovvero proprio il risultato che darà la chiamata $\text{Successore}(T \rightarrow dx, k)$)
- $x > k$: il successore sarà il risultato della chiamata $\text{Successore}(T \rightarrow sx, k)$; se $T \rightarrow sx = \emptyset$ allora il successore è x (a destra avrò solo valori maggiori di x quindi già so che il miglior candidato è x stesso)
- $x = k$: è evidente che collassa al caso $x < k$, poiché 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, ma se tale sottoalbero non esiste significa che ho già passato il successore; infatti, esso sarà proprio l'ultimo nodo da cui sono sceso a sinistra lungo il percorso:



$\text{SearchSucc}(T, 40) = 42$

Si noti di come la versione ricorsiva svolga in maniera del tutto implicita questo ragionamento. Inoltre, visto che dobbiamo ricordare un antenato, è necessario utilizzare una variabile ausiliaria che conservi l'ultimo nodo da cui scendere a sinistra man mano che si percorre 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 /*c->key = k e c->dx != NIL*/
13      RETURN SearchMinIter(c->dx)

```

Il ragionamento per trovare il predecessore è duale, quindi ne riporteremo solo gli algoritmi:

```

1 SearchPred(T, k)
2   IF T != NIL THEN
3       IF T->key >= k THEN
4           RETURN SearchPred(T->sx, k)
5       ELSE
6           x = SearchPred(T->dx, k)
7           IF x != NIL
8               RETURN x
9   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)

```

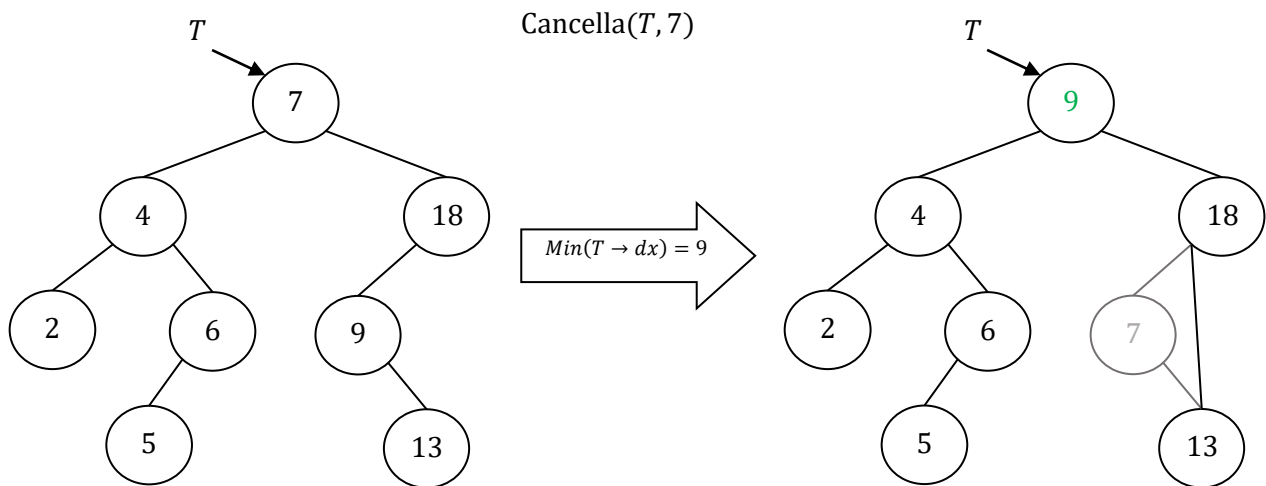
Cancellazione

Supponiamo di voler cancellare k da T e sempre per semplicità supponiamo 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 e quindi devo preoccuparmi solo di aggiornare il nodo padre mettendo 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 (ossia, $dad \rightarrow sx(o\ dx) = son \rightarrow sx(o\ dx)$, dipende da quale puntatore è diverso da NIL).

Il problema lo si ha quando il nodo contenente k ha entrambi i sottoalberi non vuoti poiché dovrei collegare i due nodi del figlio all'unico riferimento libero del padre. Tuttavia, il nostro scopo non è necessariamente cancellare l'intero nodo contenente il dato ma solo rimuovere quest'ultimo.

La soluzione è quella di sostituire il dato con quello di un altro nodo facile da cancellare; ovviamente devo scegliere un nodo che non mi distrugga la proprietà dell'ABR, ma tale nodo non è nient'altro che il minimo del sottoalbero destro di k (o analogamente il massimo del sottoalbero sinistro); infatti, non solo tale nodo

rende la sostituzione corretta ma per come abbiamo definito il 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; quindi bisogna definire anche altri algoritmi per poter implementare il seguente algoritmo:

```

1  Cancella(T, k)
2      IF T != NIL THEN
3          IF k < T->key THEN
4              T->sx = Cancella(T->sx, k)
5          ELSE IF k > T->key THEN
6              T->dx = Cancella(T->dx, k)
7          ELSE /*T->key = k*/
8              T = CancellaDatoRoot(T)
9      RETURN T

```

Qui si noti che la linea due non è necessaria poiché andiamo a richiamare il successivo algoritmo sicuramente su un input non vuoto, l'abbiamo inserita solo per robustezza:

```

1  CancellaDatoRoot(T)
2      IF T != NIL THEN
3          IF T->sx = NIL OR T->dx = NIL THEN
4              IF T->sx != NIL THEN
5                  x = T->sx
6              ELSE
7                  x = T->dx
8              dealloca(T)
9              RETURN x
10         ELSE /*ho due 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
6              k = T->key
7              IF P != NIL THEN
8                  IF T = P->sx THEN
9                      P->sx = T->sx
10                 ELSE
11                     P->dx = T->dx
12                 dealloca(T)
13      RETURN k

```

Questo caso non possiamo escluderlo poiché chiamiamo la funzione passandogli gli input $(T \rightarrow dx, T)$; quindi se T contiene il minimo allora dovrò fare esattamente ciò che abbiamo scritto alla linea 11

Ovviamente questo algoritmo è nel caso peggiore lineare sull'altezza; quindi, se l'albero è completo, allora ottengo un tempo logaritmico come sperato (questo vale per tutti gli algoritmi descritti in questo capitolo). A questo punto il problema che ci resta da risolvere è forzare un albero ad avere altezza logaritmica mantenendo al minimo il tempo di esecuzione necessario.

Alberi Bilanciati di Ricerca

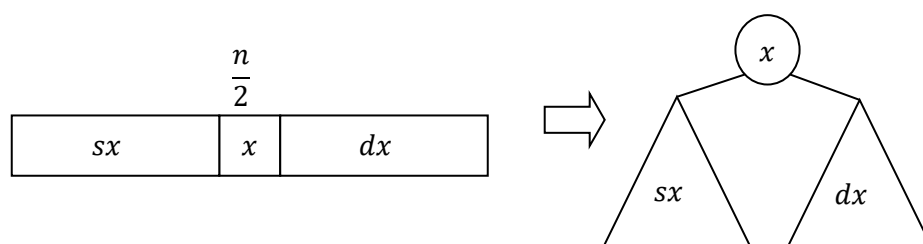
Definiamo dei vincoli aggiuntivi all'ABR in modo da garantire che l'altezza sia limitata superiormente da una funzione logaritmica, in quanto abbiamo visto che gli ABR sono semplici da gestire (inserimenti e cancellazioni facili da implementare) ma hanno prestazioni poco prevedibili e potenzialmente basse.

La famiglia di 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 $n = |T|$

Alberi Perfettamente Bilanciati (APB)

Un albero T si dice perfettamente bilanciato se $\forall x \in T, ||x \rightarrow sx| - |x \rightarrow dx|| \leq 1$ (la differenza in valore assoluto tra la cardinalità del sottoalbero sinistro e quello destro è di al più uno).

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:



Ed è abbastanza evidente che suddetta costruzione rispetti la proprietà $||x \rightarrow sx| - |x \rightarrow dx|| \leq 1$; inoltre, se tale proprietà è rispettata, è facile vedere che anche $H(t) = O(\log n)$ è verificata. Infatti, 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 dei nodi radicati al livello superiore; dunque, ad un livello i si hanno $\frac{n}{2^i}$ nodi, da cui segue $i = \log n$.

Si noti che $||x \rightarrow sx| - |x \rightarrow dx|| \leq 1 \Rightarrow H(t) = O(\log n)$ ma il viceversa non è detto che sia vero, poiché $H(t) = O(\log n)$ rappresenta un insieme con un numero maggiore di alberi.

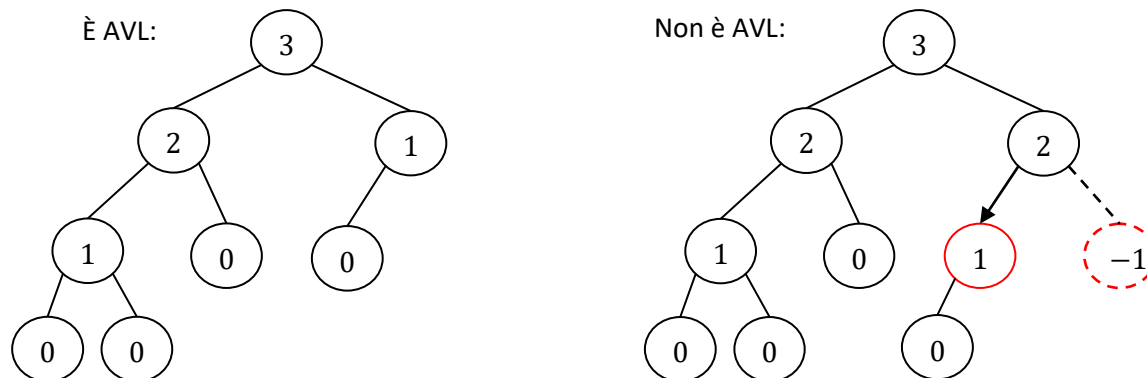
Dunque, gli APB hanno prestazioni ottimali ($\log n$ garantito) ma le operazioni di inserimento e cancellazione sono complesse (necessitano di ribilanciamenti) proprio per la restrittività della proprietà.

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 che garantisce una gestione relativamente semplice. È praticamente una versione più permissiva di APB (rilassa la sua proprietà).

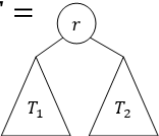
Definizione: T è AVL se T è ABR e $\forall x \in T, |H(x \rightarrow sx) - H(x \rightarrow dx)| \leq 1$

Riportiamo due esempi scrivendo in ogni nodo l'altezza di quel sottoalbero ($T = \emptyset \Rightarrow H(T) = -1$):



Questa proprietà universale (vale per ogni nodo) è facile da descrivere in maniera ricorsiva:

T è AVL se, e solo se, T è ABR e

- 1) $T = \emptyset$
- 2) $T =$

 $|H(T_1) - H(T_2)| \leq 1$
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, il quale rispetta la proprietà di AVL per definizione. Quindi, una volta aumentata la tolleranza (gli AVL sono più permissivi degli APB), ci 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 possibile, ed è intuitivo che se per gli alberi peggiori vale $O(\log n)$, allora anche per tutti gli altri la relazione è verificata.

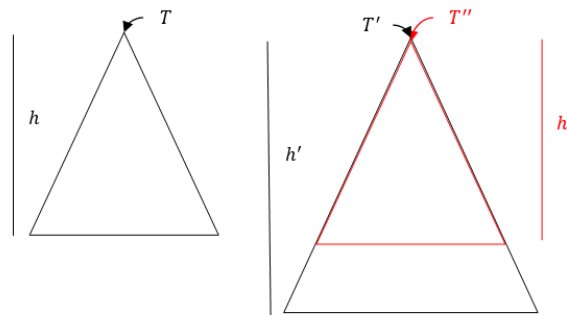
AVL minimi

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).

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: se $T \in \text{AVL minimo con } |T| = n, \forall T' \in \text{AVL con } |T'| = n \Rightarrow H(T') \leq H(T)$.

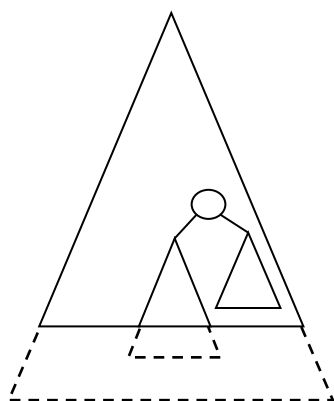
Ciò significa che gli AVL minimi sono quelli con altezza peggiore per stesso numero di nodi, e quindi una relazione tra altezza e numero di nodi in questa classe farà da limite superiore per tutti gli AVL (che è proprio quello che volevamo).

Dimostriamo tale proprietà per assurdo: supponiamo di avere un AVL minimo T e un AVL T' con altezza maggiore di T e stesso numero di nodi. Ma 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 .

Ma il fatto che T'' è ancora AVL è abbastanza evidente graficamente:



Supponiamo di aver fatto il “taglio” descritto in figura: ora poiché l'albero T' era AVL per definizione e non ho toccato il sottoalbero destro durante il taglio (quindi solo 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:

$$\underbrace{sx''}_{< sx'} - dx' < sx' - dx' \Rightarrow sx'' - dx' < 1$$

Quindi ho ottenuto un AVL con altezza uguale a quella di T (che era AVL minimo) ma con meno nodi, il che è assurdo.

Tutto questo ragionamento ci permette di concentrarci solo sugli alberi AVL minimi, poiché una proprietà valida per tale sottoclasse risulterà vera per un qualsiasi AVL.

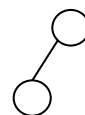
Relazione tra altezza e numero di nodi

Vediamo come sono strutturati gli AVL minimi in base all'altezza (ovviamente non sono gli unici AVL minimi con quell'altezza, solo per $h = 0$ si ha un unico AVL minimo):

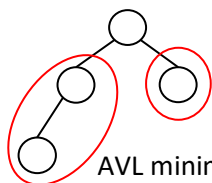
$h = 0$



$h = 1$



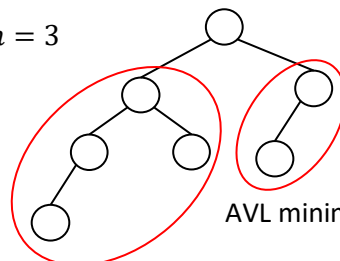
$h = 2$



AVL minimo di $h = 0$

AVL minimo di $h = 1$

$h = 3$



AVL minimo di $h = 1$

AVL minimo di $h = 2$

Ma allora c'è una regolarità tra gli AVL minimi: infatti un AVL minimo di altezza h è composto dalla radice e da due sottoalberi, in particolare un sottoalbero è un AVL minimo 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, come ad esempio 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 (il quale ricordiamo essere già AVL minimo) 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 $NN(T) = 1 + NN(T \rightarrow sx) + NN(T \rightarrow dx)$.

Ma 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 solo dire che $h \leq NN(h) \leq 2^{h+1} - 1$). Dunque per un AVL minimo possiamo definire la seguente equazione di ricorrenza:

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

Suddetta equazione è molto simile alla funzione 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 due equazioni possiamo sfruttare il fatto che la forma chiusa della sequenza di Fibonacci è nota in matematica.

| | | | | | | | | | | | |
|------|---|---|---|---|----|----|----|----|----|-----|-----|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| F | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | ... |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| NN | 1 | 2 | 4 | 7 | 12 | 20 | 33 | 54 | 88 | 143 | ... |

Differenziano di una unità!

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 $NN(h-1) = F(h-1+3) - 1 = F(h+2) - 1$ e $NN(h-2) = F(h-2+3) - 1 = F(h+3) - 1$ siano verificate. Allora:

$$\begin{aligned} NN(h) &= 1 + NN(h-1) + NN(h-2) = 1 + F(h+2) - 1 + F(h+3) - 1 = \underbrace{F(h+2) + F(h+3)}_{\text{è per definizione } F(h+3)} - 1 = \\ &= F(h+3) - 1 \text{ come volevasi dimostrare} \end{aligned}$$

La forma chiusa di Fibonacci è la seguente:

$$\begin{aligned} F(h) &= \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \underbrace{\left(\frac{1-\sqrt{5}}{2} \right)^h}_{\substack{\text{compreso tra } -1 \leq x \leq 0 \\ \text{quindi per } h \rightarrow \infty, x^h \rightarrow 0}} \right] \xrightarrow{\text{per } n \text{ sufficientemente grande}} F(h) \cong \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h \\ \Rightarrow \sqrt{5}F(h) &= \left(\frac{1+\sqrt{5}}{2} \right)^h \Rightarrow h = \log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5}F(h)) \end{aligned}$$

Ma allora anche $NN(h)$ sarà un'equazione esponenziale rispetto ad h e di conseguenza h sarà logaritmica rispetto al numero di nodi $n = NN(h)$:

$$n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \Rightarrow \sqrt{5}(n+1) = \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} \Rightarrow h+3 = \log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5}(n+1)) \Rightarrow$$

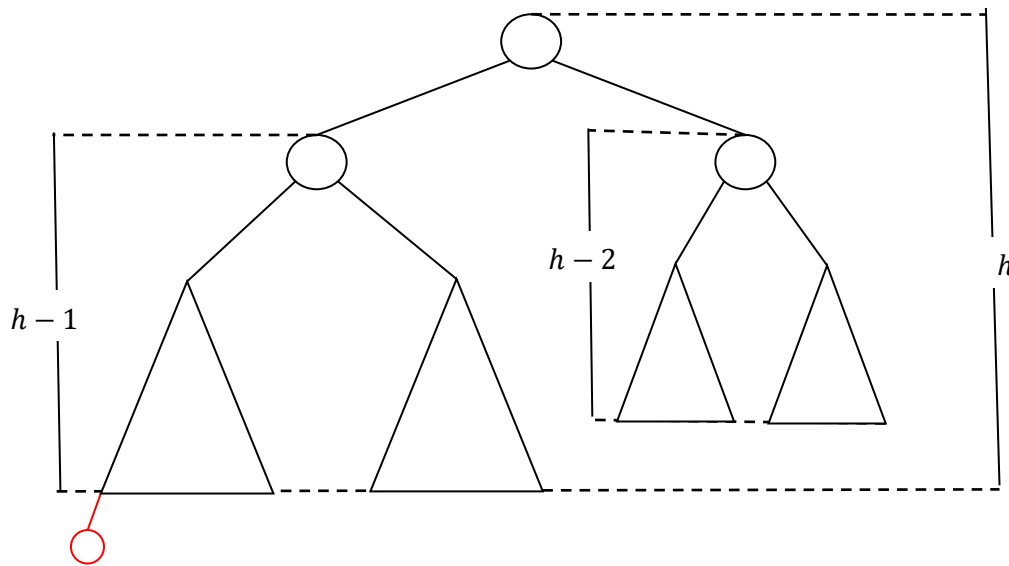
$$\Rightarrow h = \log_{\frac{1+\sqrt{5}}{2}}(\sqrt{5}(n+1)) - 3 = \frac{\log_2(\sqrt{5}(n+1))}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} - 3 = \Theta(\log n)$$

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.

Esercizio (per malati): Verificare che $NN(h) = \Theta(\log n)$ studiandone la funzione di ricorrenza (dovrebbe essere un procedimento quasi del tutto simile all'equazione di ricorrenza di Quick Sort).

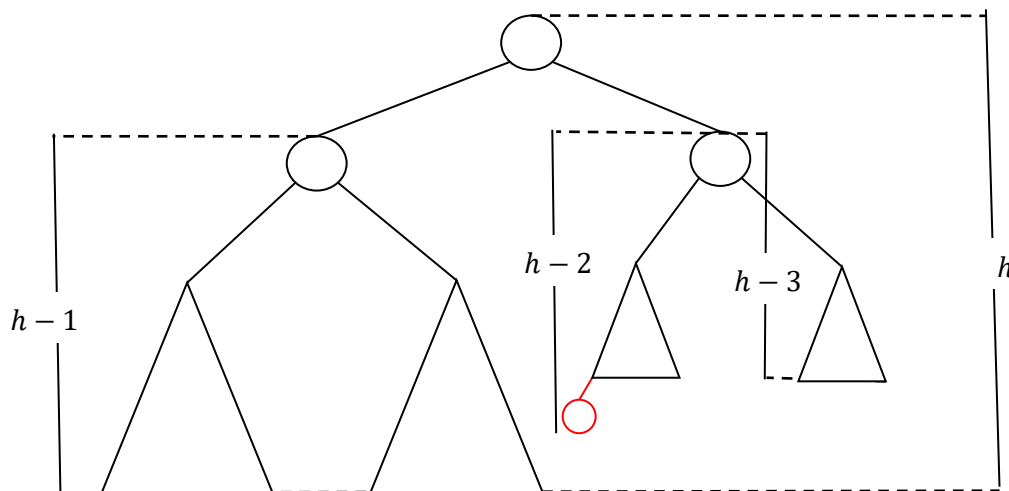
Operazione di inserimento in AVL

Prendiamo un generico AVL, che sappiamo essere anche binario di ricerca, oltre che completo. Esiste un unico punto dove è possibile inserire un nuovo dato senza distruggere la proprietà di ordinamento:



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 poiché ora la radice avrà sottoalbero sinistro di altezza h e quello destro ancora di $h - 2$.

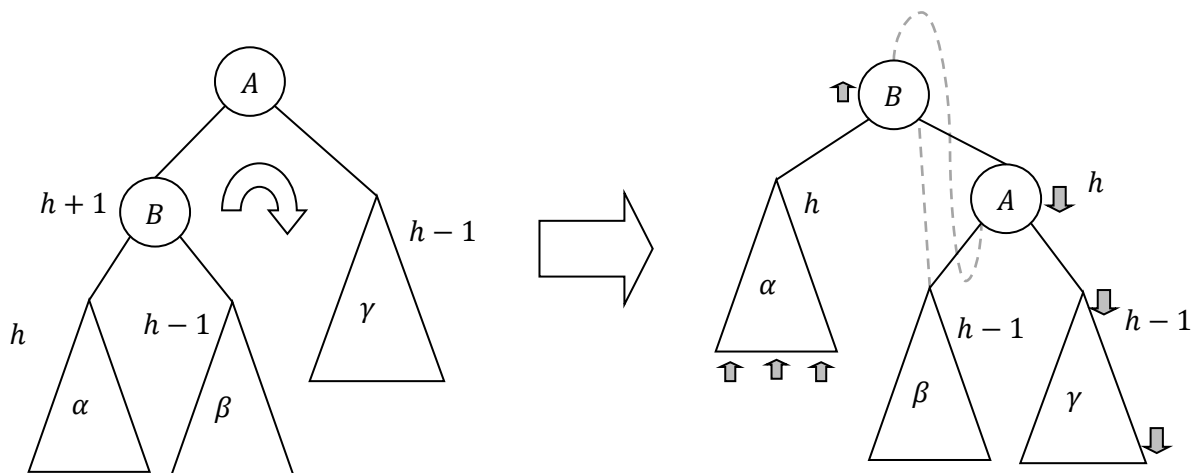
Ragionamento analogo si può fare con la cancellazione, supponendo infatti di rimuovere il nodo evidenziato nella seguente situazione genererei una violazione alla proprietà di AVL:



Dopo la cancellazione l'altezza del sottoalbero sarà $h - 3$ e quindi avremo una violazione alla radice (il sottoalbero sinistro ha altezza $h - 1$ e il destro $h - 3$, quindi $|h - 1 - h + 3| > 1$).

Di conseguenza è necessario un metodo che mantenga le proprietà della struttura; ovviamente, se i punti di violazione sono tanti allora è molto complicato dal punto di vista computazionale risistemare la struttura. Ma possiamo garantire che ci sia un unico punto di violazione semplicemente mettendo dopo ogni singola operazione di modifica (che sia cancellazione o inserimento) un controllo che nel momento in cui trova una violazione la va a sistemare. Suddivideremo dunque l'operazione in: $AVL \xrightarrow{\text{modifica}} \text{nonAVL} \xrightarrow{\text{bilancia}} AVL$

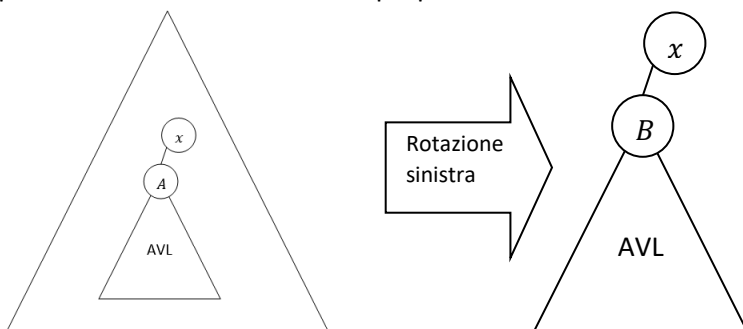
Supponiamo di essere di fronte ad un inserimento nel sottoalbero α che viola la proprietà di AVL. Per ribilanciare il peso dobbiamo "accorciare" di uno l'altezza del sottoalbero radicato in B aumentando quella del sottoalbero γ (non possiamo semplicemente cancellare dei nodi per ridurre il peso, quindi è ovvio che il peso tolto da un sottoalbero deve andare in un altro sottoalbero). Ovviamente le operazioni di bilanciamento devono preservare anche la proprietà di ordinamento e non solo quella delle altezze. L'operazione che risolve il nostro problema è detta **rotazione singola sinistra** (la chiamiamo così perché ruota l'albero partendo dal figlio sinistro):



In questo modo la proprietà di AVL è mantenuta, infatti α e γ non vengono toccati, e di conseguenza la proprietà di ordinamento è rispettata, mentre β prima della rotazione era a destra di B e dopo viene spostato a sinistra di A , ma per le proprietà di albero binario di ricerca sappiamo che prima della rotazione $\forall i \in \beta, i \geq B$ ma essendo B a sinistra di A abbiamo $i \leq A$. Di conseguenza è evidente che abbiamo mantenuto l'ordinamento poiché l'albero β si trova, dopo la rotazione, a sinistra di A ed a destra di B .

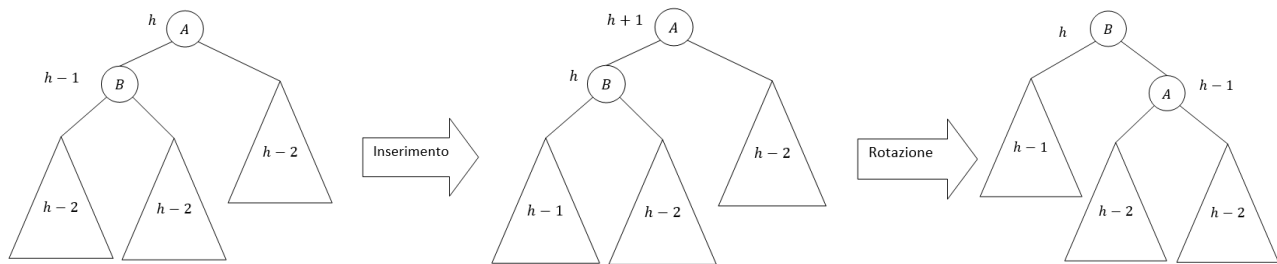
Per quanto riguarda le altezze, prima della rotazione era presente una violazione tra il sottoalbero sinistro di A (sottoalbero di altezza $h + 1$) a causa dell'inserimento e il suo sottoalbero destro (γ di altezza $h - 1$); dopo la rotazione andiamo ad alzare il sottoalbero α e ad abbassare γ . In altri termini, il sottoalbero sinistro della radice (che dopo la rotazione sarà B) avrà ora altezza h , così come il suo sottoalbero destro.

Cosa ancora più rilevante è che questa operazione non risolve il problema solo localmente; infatti, anche supponendo di aver fatto la precedente operazione in un sottoalbero, l'operazione di rotazione risolve il problema della violazione delle proprietà di AVL in maniera totale:

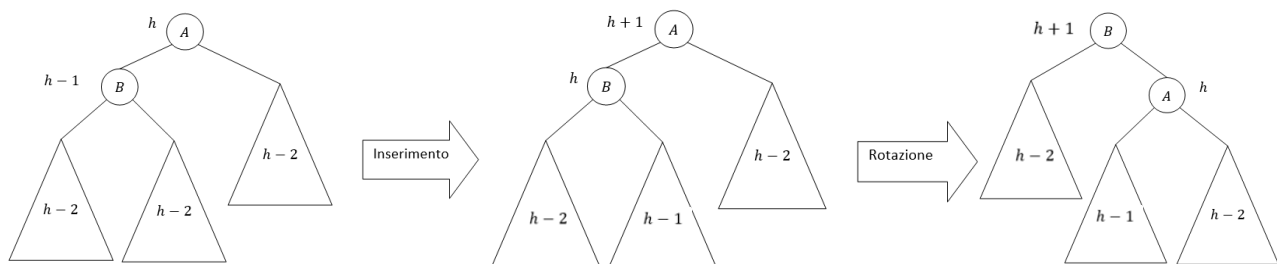


La proprietà di ordinamento viene ovviamente rispettata: prima della rotazione avevamo A a sinistra di x , quindi gli elementi di quell'albero erano tutti minori, ma allora anche dopo la rotazione tale proprietà è rispettata essendo $B < A < x$

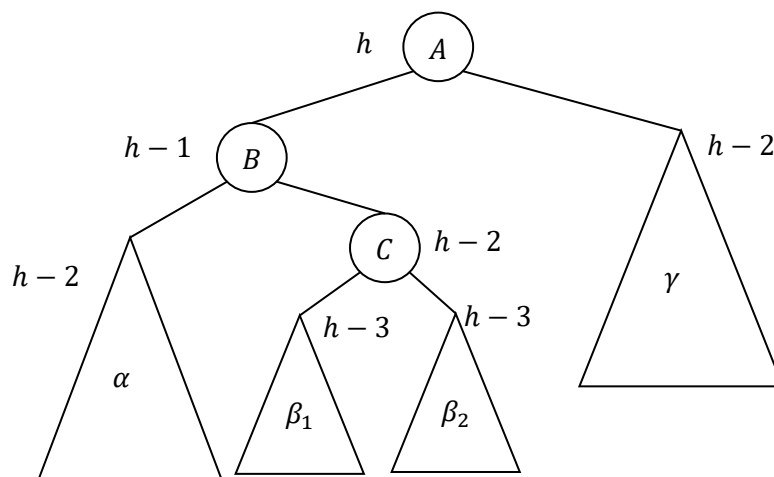
Inoltre, poiché l'altezza dell'albero dopo la rotazione rimane la stessa di quella che aveva prima dell'inserimento, è evidente che nemmeno la violazione sull'altezza viene propagata sugli antenati del sottoalbero ruotato. Andiamo a dimostrare tale proprietà graficamente:



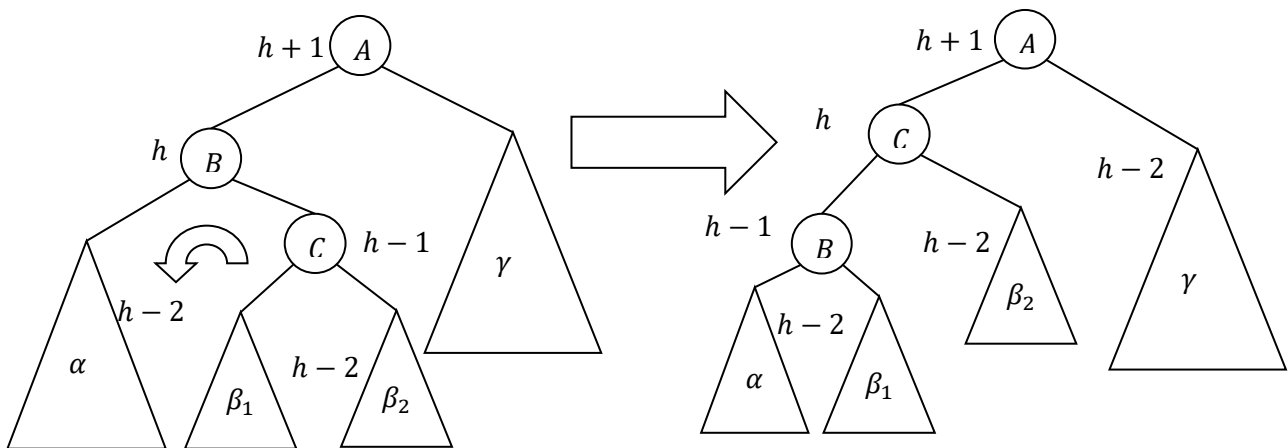
Abbiamo visto dunque che la rotazione singola sinistra risolve il problema della violazione nel caso in cui inserisco un elemento nel sottoalbero estremo sinistro (non è detto che un inserimento mi crei per forza una violazione); ma se invece di inserire in α inserisco in β ? Con la rotazione vista non risolvo nulla!



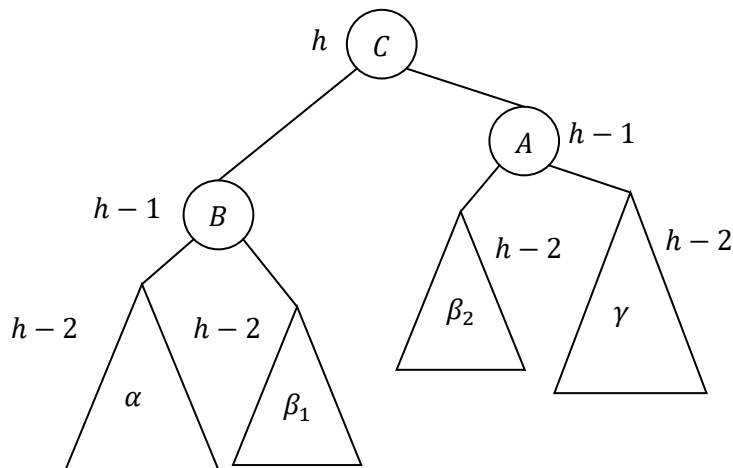
Ed il risultato era abbastanza evidente, poiché come abbiamo visto, la rotazione sinistra sposta il peso dal sottoalbero estremo sinistro al sottoalbero estremo destro, non toccando per nulla il peso di β . Ma se sappiamo spostare il peso da un estremo all'altro allora si potrebbe intuitivamente pensare che una rotazione destra sposterebbe il peso dall'estremo destro all'estremo sinistro. Andiamo allora a disegnare il nostro albero più in dettaglio:



Supponiamo che dopo l'inserimento l'altezza di β aumenti di uno in entrambi i sottoalberi β_1 e β_2 (non può succedere ovviamente con un solo inserimento ma vogliamo rendere il ragionamento più generale possibile così da mostrare che a prescindere dal sottoalbero di c in cui inseriamo la doppia rotazione risolve il nostro problema).



La violazione non è ancora stata risolta, ma adesso è il sottoalbero estremo sinistro ad avere peso troppo alto rispetto all'estremo destro. Tale situazione però è la stessa che abbiamo risolto con una rotazione sinistra! Infatti, se applichiamo adesso una rotazione singola sinistra, avremo il seguente albero:



N.B.: non confondere l'altezza (la distanza dal nodo alle foglie) con la profondità in cui si trova il nodo (il percorso dalla radice al nodo, ovvero il livello in cui si trova il nodo)

Ma allora anche in questo caso, con una doppia rotazione non solo manteniamo la proprietà dell'ordinamento, ma abbiamo anche la stessa altezza che avevamo prima dell'inserimento: ne consegue che tutto il ragionamento fatto per la singola rotazione è ancora valido. Quindi, così facendo, 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
- se si crea una violazione in β risolviamo con una rotazione destra seguita da una rotazione sinistra

Mentre, se l'inserimento crea una violazione in γ , il ragionamento è esattamente duale:

- la violazione è nel sottoalbero estremo destro, risolvo con una singola rotazione destra
- la violazione è nel sottoalbero sinistro di $\gamma \rightarrow dx$, risolvo con una rotazione sinistra ed una destra

Si ricorda infatti che la risoluzione locale nel sottoalbero risolve in maniera totale la violazione.

Abbiamo coperto tutti i casi possibili, ma 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 descrive l'altezza di quel nodo con la

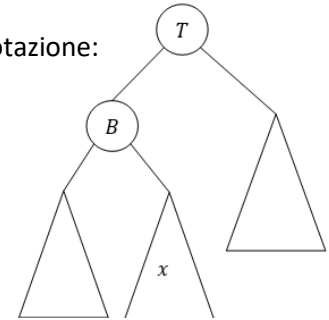
conseguenza di dover usare una memoria aggiuntiva pari a $n \log(\log n)$ (con n il numero di nodi, mentre $\log(\log n)$ è dovuto al fatto che usiamo una cella di memoria dove il valore massimo è un intero $h = \log n$).

Si potrebbe pensare di calcolare in maniera ricorsiva l'altezza evitando di dover usare tutta quella memoria aggiuntiva (tramite la linea $h(T) = 1 + \max(h(T \rightarrow sx), h(T \rightarrow dx))$), ma suddetta operazione ha tempo di esecuzione lineare sul numero di nodi (poiché ci serve conoscerla per ogni nodo), andando a vanificare tutti i nostri ragionamenti ed i nostri progressi (si ricorda che stiamo facendo tutto questo per rendere le operazioni sugli alberi binari eseguibili in tempo lineare sull'altezza).

RotazioneSingolaSx(T)

```
B = T->sx
x = B->dx
B->dx = T
T->sx = x
T->h = 1 + max(Altezza(T->sx), Altezza(T->dx))
B->h = 1 + max(Altezza(T->sx), Altezza(B->sx))
RETURN B
```

Prima della rotazione:



RotazioneDoppiaSx(T)

```
T->sx = RotazioneSingolaDx(T->sx)
RETURN RotazioneSingolaSx(T)
```

Altezza(T)

```
IF T = NIL THEN
    RETURN -1
ELSE
    RETURN T->h
```

Le operazioni di RotazioneSingolaDx e RotazioneDoppiaDx sono del tutto duali rispetto alle precedenti (basta scambiare *sx* con *dx* e viceversa), per cui riporteremo direttamente 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 allora potrà crearsi solo a sinistra*/
8         IF T->key > k THEN
9             T->sx = InsertAVL(T->sx, k)
10            T = BilanciaSx(T)
11     ELSE /*creo una foglia*/
12         T = AllocaNodoAVL()
13         T->key = k
14         T->sx = T->dx = NIL
15         T->h = 0
16     RETURN T
```

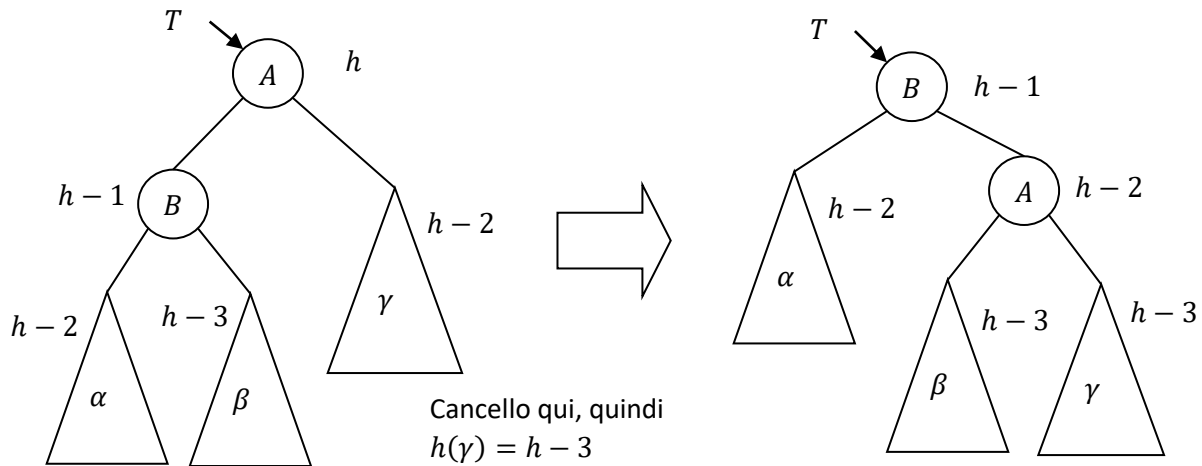
Abbiamo praticamente un inserimento in un albero binario di ricerca con l'aggiunta di 3 operazioni

Resta solo da descrivere l'operazione di bilanciamento: implementeremo l'algoritmo solo per il bilanciamento a sinistra (a destra sarà duale, bisogna invertire *sx* con *dx* e viceversa). Si noti che non serve il valore assoluto poiché abbiamo inserito a sinistra; per quanto detto, se si crea una violazione sarà perché l'altezza del sottoalbero sinistro è aumentata di uno ed è diventato troppo pesante rispetto il destro:

```
1 BilanciaSx(T)
2     IF Altezza(T->sx) - Altezza(T->dx) > 1 THEN
3         IF Altezza((T->sx)->sx) > Altezza((T->sx)->dx) THEN
4             RETURN RotazioneSingolaSx(T)
5         ELSE /*la violazione è in beta*/
6             RETURN RotazioneDoppiaSx(T)
7     ELSE /*se non c'è violazione devo solo aggiornare l'altezza*/
8         T->h = 1 + max(Altezza(T->sx), Altezza(T->dx))
9     RETURN T
```

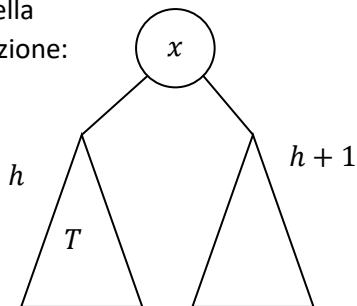
Operazione di cancellazione in AVL

Per l'inserimento abbiamo visto che valeva la proprietà di avere una risoluzione totale delle violazioni andando a eliminare la violazione localmente all'albero; sfortunatamente suddetta proprietà non vale per la cancellazione: infatti, per quest'ultima c'è la possibilità che la violazione venga propagata al nodo padre. Tutto ciò è causato dal fatto che non è possibile riportare l'altezza al valore precedente la cancellazione. Ad esempio:

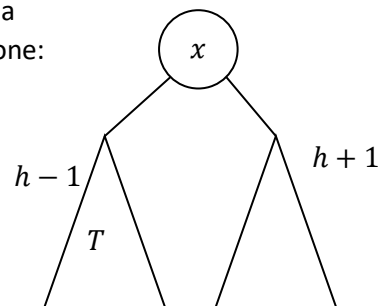


Dove T è un sottoalbero nella seguente situazione:

Prima della cancellazione:

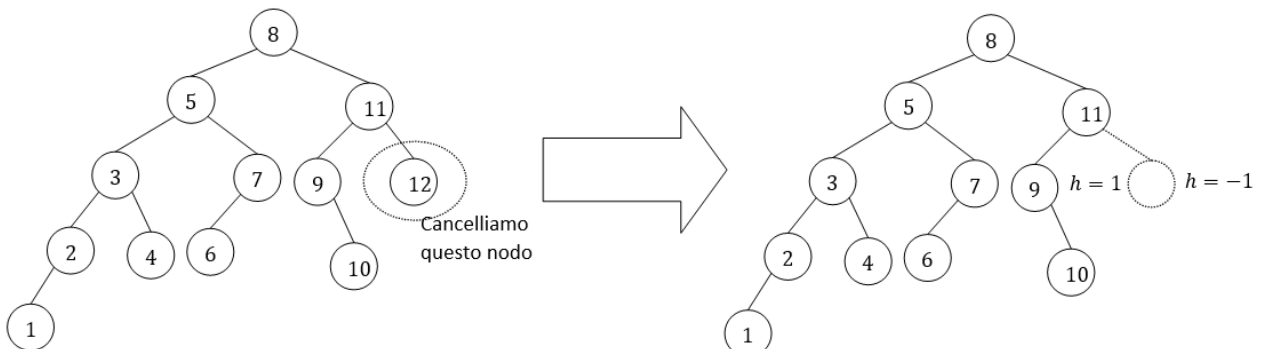


Dopo la rotazione:

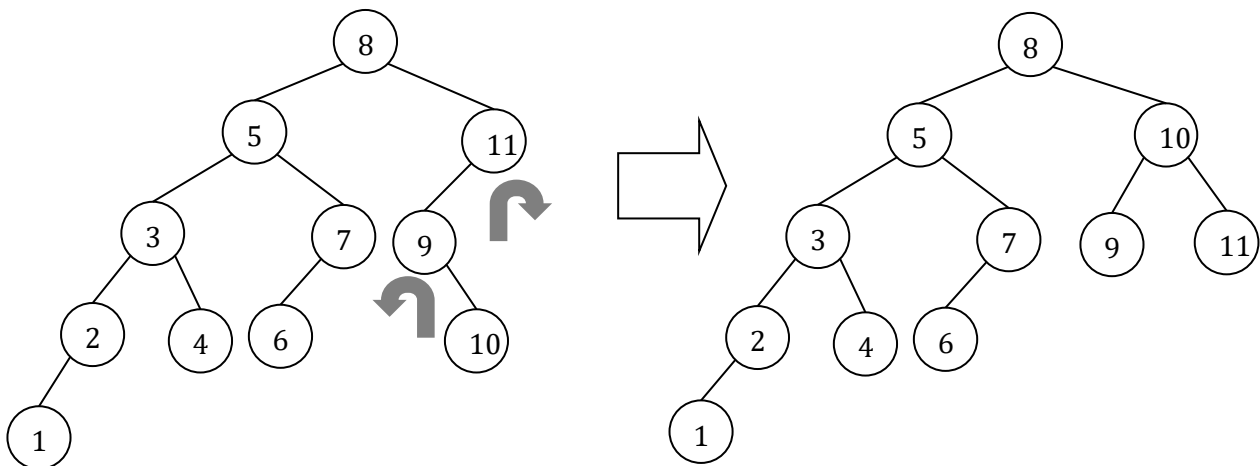


Dunque, nell'operazione di cancellazione, se c'è una violazione allora dopo il bilanciamento l'altezza sarà sempre decrementata di uno rispetto all'altezza precedente la cancellazione, e questo potrebbe propagare la violazione al padre (ciò accade se prima della cancellazione valeva $|h(P \rightarrow sx) - h(P \rightarrow dx)| = 1$ con P che rappresenta il padre della radice dell'albero appena bilanciato).

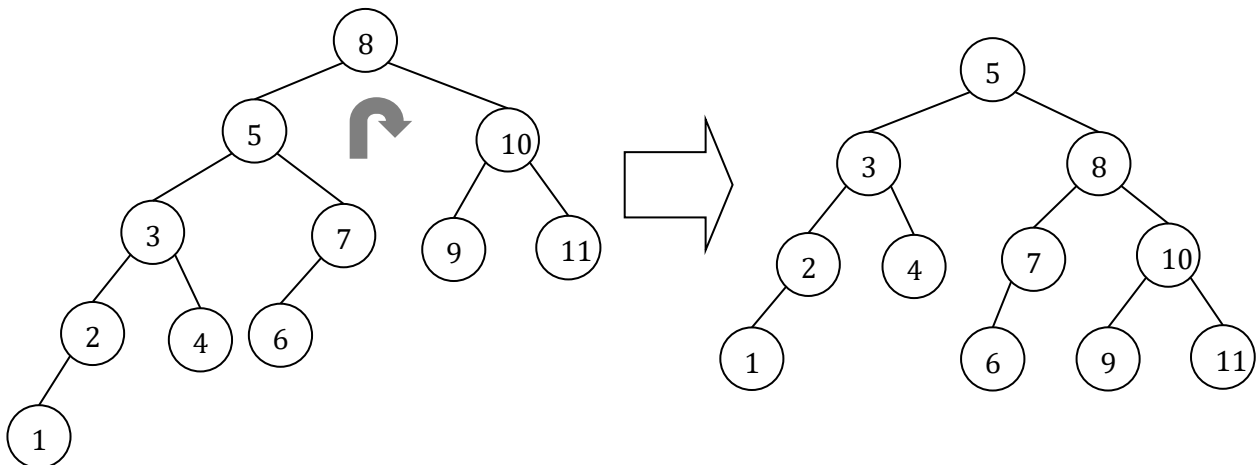
Vediamo a questo punto quante violazioni sequenziali si possano avere nel caso peggiore. Sorvoliamo sulla differenza tra operazioni singole e doppie: nel caso peggiore la violazione si potrebbe propagare in maniera lineare rispetto all'altezza (quindi $\log n$ nel caso siano solo operazioni singole e $2 \log n$ per sole operazioni doppie). Prendiamo ad esempio il seguente AVL:



Un primo bilanciamento è una doppia rotazione destra:



Ma questo albero ha ancora una violazione, poiché il nodo 5 ha altezza 3 mentre il nodo 10 altezza 1. Ne consegue che è necessario un altro bilanciamento tramite una rotazione singola sinistra:



Dunque, l'albero di partenza di altezza $h = 4$ diventa, dopo due 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.

Quindi, a differenza dell'inserimento, nella cancellazione dobbiamo ogni volta verificare se l'albero è bilanciato durante la risalita (questo lavoro lo svolge anche l'algoritmo di inserimento per come lo abbiamo definito, ma si potrebbe evitare tramite, ad esempio, un flag booleano).

```

1  CancellavaVL(T, k)
2      IF T != NIL THEN
3          IF T->key > k THEN
4              T->sx = CancellavaVL(T->sx, k)
5              T = BilanciaDx(T)
6          ELSE IF T->key < k THEN
7              T->dx = CancellavaVL(T->dx, k)
8              T = BilanciaSx(T)
9          ELSE
10             T = CancellavaVL_Root(T)
11     RETURN T

```

se cancelliamo a sinistra allora l'albero più pesante è il destro, quindi usiamo BilanciaDx

È praticamente l'algoritmo di cancellazione per ABR con alcune aggiunte; definiamo ora CancellaviL_Root:

```
1  CancellaviL_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     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 si ha bisogno di aggiornare le altezze poiché lo fa correttamente BilanciaDx:

```
1  StaccaAVL_Min(T)
2      IF T != NIL THEN
3          IF T->sx != NIL THEN
4              val = StaccaAVL_Min(T->sx, T)
5              tmp = BilanciaDx(T)
6          ELSE
7              val = T->key
8              tmp = T->dx
9              IF P != NIL THEN
10                 IF T = P->sx THEN
11                     P->sx = tmp
12                 ELSE
13                     P->dx = tmp
14             dealloca(T)
15     RETURN val
```

La radice di T potrebbe cambiare, per questo usiamo una variabile temporanea

Si noti che la linea 9 in realtà non è necessaria poiché sia in CancellaviL_Root che nelle chiamate ricorsive di CancellaviL_Min abbiamo la certezza che il padre non sia mai nullo; tale controllo rende però possibile usare questo algoritmo in maniera indipendente (lo rende più robusto).

Abbiamo terminato la discussione sugli AVL, ossia la nostra prima struttura che ammette tutte le operazioni in tempo logaritmico. Ha praticamente la stessa complessità computazionale degli array ordinati ma ne migliora notevolmente le operazioni di modifica.

Alberi Red-Black

Gli alberi Red-Black sono una struttura dati con proprietà simili agli AVL (anche tempo di esecuzione logaritmico) ma più tollerante: riduce il numero necessario di rotazioni per ribilanciare la struttura a fronte di modifiche.

Un albero RB è essenzialmente un albero binario **di ricerca** in cui:

- Le chiavi vengono mantenute solo nei nodi interni dell'albero
- Le foglie sono costituite da speciali nodi NIL, cioè nodi "sentinella" il cui contenuto è irrilevante e che evitano di trattare diversamente i puntatori ai nodi dai puntatori NIL.
 - In altre parole, al posto di un puntatore NIL si usa un puntatore ad un nodo NIL.

- Quando un nodo ha come figli nodi NIL, tale nodo corrisponde ad una foglia nell'albero binario di ricerca corrispondente.

La particolarità dei RB è proprio quella di avere le foglie dell'albero senza dati, quindi la cardinalità dell'albero è pari al numero di nodi interni (se parliamo dal punto di vista di dati utili). Il vincolo di bilanciamento è definito da un'etichettatura dei nodi: ad ogni nodo è associato un colore (basta un bit) rosso o nero.

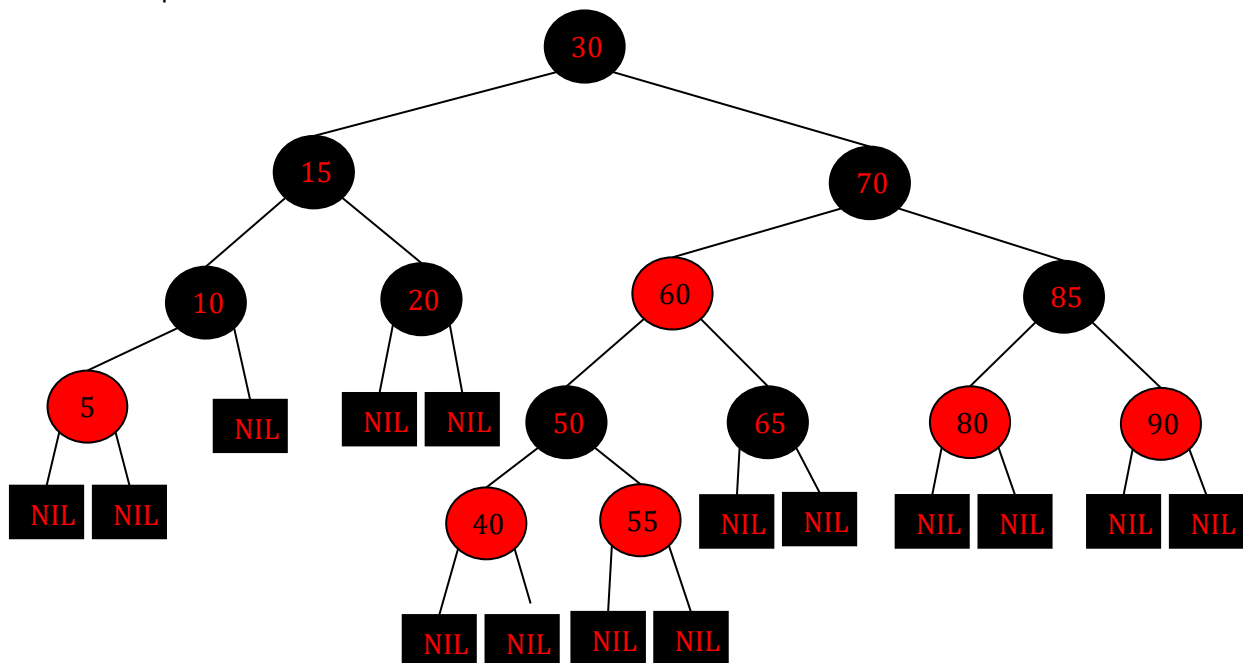
Il vincolo globale è espresso dalle seguenti proprietà:

- 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 dà la garanzia che i neri sono almeno la metà);
- 4) per ogni nodo x , ogni percorso da x ad un nodo NIL ha lo stesso numero di nodi neri.

Questi vincoli insieme (in particolare gli ultimi due) garantiscono un'altezza logaritmica sul numero di nodi.

Definiamo **altezza nera** dell'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 influente non viene calcolato nell'altezza. In altri termini, se ho due alberi con colore identico eccetto per la radice (uno ha radice nera e l'altro rossa) questi avranno la stessa altezza nera.

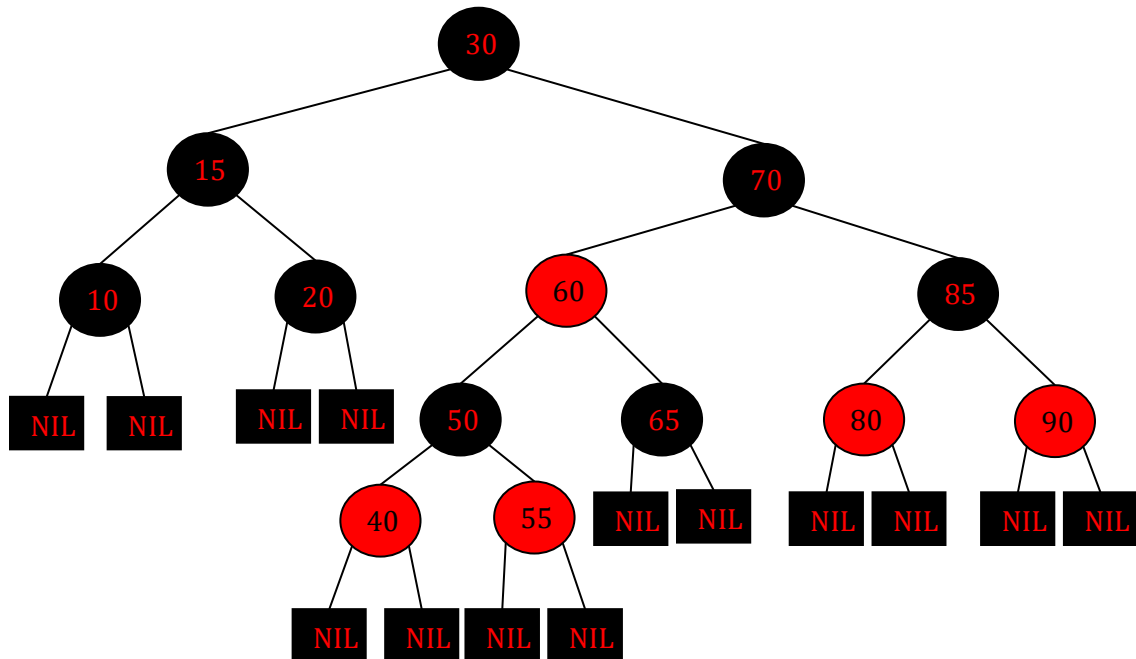
Diamo un esempio di albero RB:



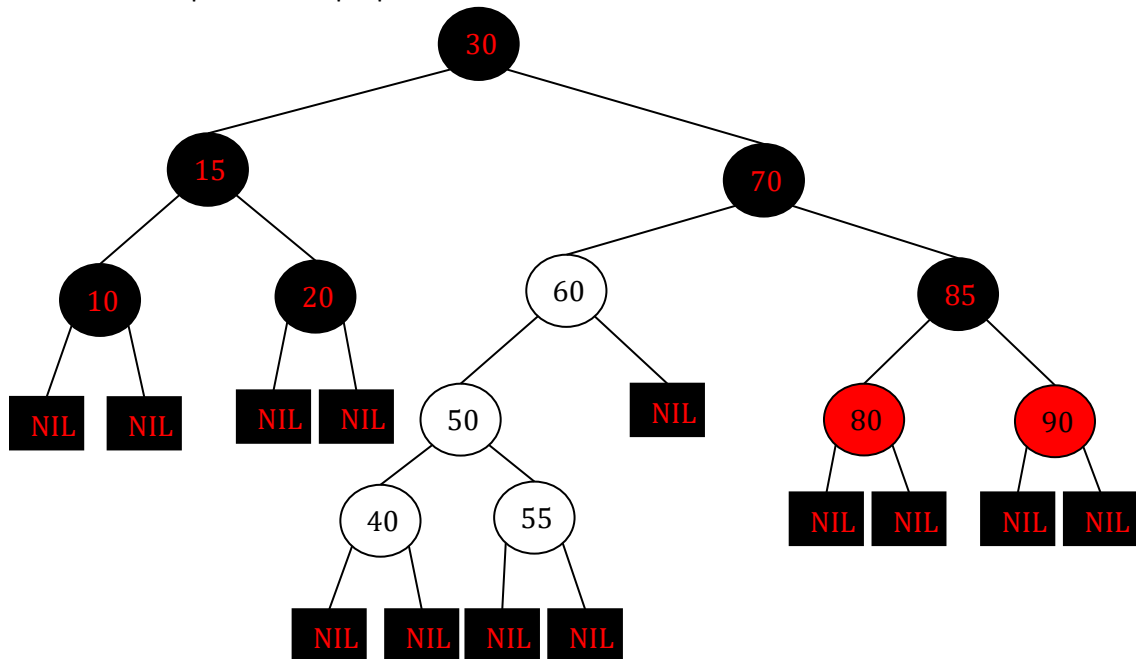
Il miglior modo per costruire un RB è partire dal percorso più breve: i nodi di tale percorso vanno colorati tutti di nero (poiché devo massimizzare il più possibile i nodi neri, se nel percorso più corto mettessi dei nodi rossi allora non è detto che riuscirei a garantire la proprietà 4); una volta fatto ciò e quindi dopo aver determinato i numeri di neri di ogni percorso, si incomincia 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 ottimo esempio è l'albero pieno, il quale ha il numero maggiore di differenti possibili colorazioni (posso farlo tutto nero, oppure un livello nero e uno rosso, o ancora un livello rosso per ogni due neri, etc...), e man mano che si sbilancia l'albero, le variazioni di colorazione diminuiscono fino ad arrivare ad uno sbilanciamento tale da non permettere più alcuna colorazione.

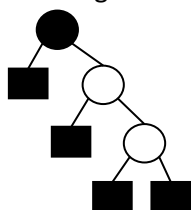
Ogni albero AVL è RB, ed un esempio è proprio l'albero precedente. In realtà si potrebbe dimostrare tramite induzione che è sempre possibile colorare un AVL (ce lo risparmieremo), ma noi abbiamo detto che gli alberi RB sono più permissivi degli AVL, allora vediamo di seguito un esempio di RB non AVL:



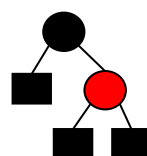
Fino a quando possiamo spingere il bilanciamento? Prendiamo come esempio il seguente albero: non è possibile colorarlo rispettando le proprietà di RB:



Gli alberi degeneri non banali non sono colorabili:



L'altezza nera è 1 e non posso usare due nodi rossi adiacenti!
Gli alberi degeneri più alti possono solo peggiorare la situazione



Questo è l'unico albero degenero RB (si noti che è anche un albero AVL)

Dimostrazione che i RB hanno altezza logaritmica sul numero di nodi

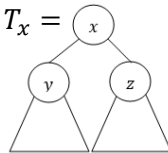
Sia n la cardinalità dell'insieme, che ricordiamo non coincide con il numero di nodi totali dell'albero, bensì ne rappresentano solo i nodi interni. Dimostriamo che vale la seguente relazione:

$$\forall T \in RB, \quad h(T) \leq 2 \log(n+1) \quad (\Theta(\log n))$$

Per dimostrare tale proprietà sfrutteremo una relazione tra il numero di nodi interni e l'altezza nera. Sia $n(x)$ il numero di nodi interni di un generico sottoalbero T_x , risulta che $n(x) \geq 2^{bh(x)} - 1$ con $bh(x)$ che rappresenta l'altezza nera. Si noti che tale relazione somiglia al numero di nodi di un albero pieno: $2^{h+1} - 1$. Ma se in un albero RB togliamo tutti i nodi rossi, otteniamo proprio un albero pieno con altezza $bh(x)$, e poiché non dobbiamo considerare il livello delle foglie, la relazione $n(x) \geq 2^{bh(x)} - 1$ è intuitivamente vera.

Svolgiamo una rappresentazione più formale della precedente relazione, procederemo per induzione sull'altezza. Il caso base è $h_x = 0$ e quindi T_x è rappresentato solo dal nodo NIL, ma allora $n(x) = 0$ e $bh(x) = 0 \Rightarrow 2^{bh(x)} - 1 = 2^0 - 1 = 0$ e la relazione $n(x) \geq 0$ è banalmente soddisfatta.

Sia ora $h_x = h > 0$, l'albero $T_x =$



Si noti che y e z sono certamente esistenti poiché devono essere almeno foglie (nodi NIL)

Ma se $h_x = h$ allora $h_y < h$ ed $h_z < h$ evidentemente, però se l'altezza dei sottoalberi è strettamente minore dell'altezza di x possiamo applicare il caso induttivo (il quale ci dice 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:

$$n(y) \geq 2^{bh(y)} - 1 \quad \wedge \quad n(z) \geq 2^{bh(z)} - 1$$

Dunque, risulta:

$$n(x) = 1 + n(y) + n(z) \Rightarrow n(x) \geq 1 + 2^{bh(y)} - 1 + 2^{bh(z)} - 1 \Rightarrow n(x) \geq 2^{bh(y)} + 2^{bh(z)} - 1$$

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 ignora quello di y , quindi a seconda del colore di y possiamo avere i due 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 $bh(x) - 1 \leq bh(y) \leq bh(x)$; allo stesso modo, per il nodo z avremo $bh(x) - 1 \leq bh(z) \leq bh(x)$. Sfruttando la monotonicità della funzione esponenziale è lecito scrivere $2^{bh(x)-1} \leq 2^{bh(y)}$ ed analogamente $2^{bh(x)-1} \leq 2^{bh(z)}$.

Ritornando all'equazione principale, con i seguenti passaggi abbiamo concluso la dimostrazione:

$$\begin{aligned} n(x) &\geq \underbrace{2^{bh(y)}}_{\geq 2^{bh(x)-1}} + \underbrace{2^{bh(z)}}_{\geq 2^{bh(x)-1}} - 1 \geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \Rightarrow \\ &\Rightarrow n(x) \geq 2 \cdot 2^{bh(x)-1} - 1 \Rightarrow n(x) \geq 2^{bh(x)} - 1 \end{aligned}$$

Questo discorso fatto per un generico sottoalbero varrà ovviamente anche per la radice dell'albero completo. Da ciò, ricaviamo che in un RB con altezza nera bh il numero di chiavi dell'albero è $n \geq 2^{bh} - 1$.

Ora resta solo da trovare una relazione tra altezza e altezza nera (poiché lo scopo principale è arrivare a $h = \Theta(\log n)$); sappiamo che in un percorso ci devono essere almeno tanti neri quanti rossi, altrimenti violeremo la proprietà 3 dei RB. Dunque:

$$\frac{h}{2} \leq bh \leq h \Rightarrow bh \geq \frac{h}{2} \Rightarrow 2^{bh} \geq 2^{\frac{h}{2}} \Rightarrow 2^{bh} - 1 \geq 2^{\frac{h}{2}} - 1$$

Ma allora, come volevasi dimostrare:

$$n \geq 2^{bh} - 1 \geq 2^{\frac{h}{2}} - 1 \Rightarrow n + 1 \geq 2^{\frac{h}{2}} \Rightarrow \log(n + 1) \geq \frac{h}{2} \log 2 \Rightarrow h \leq 2 \log(n + 1)$$

Inserimento in un RB

L'algoritmo di inserimento deve preservare, oltre ai vincoli di RB, anche quelli di ordinamento: deve sempre eseguire una ricerca binaria e ciò implica che esiste una sola (e precisa) foglia dove inserire un determinato dato. Il primo vincolo di un RB è che il nodo abbia un colore, ma è evidente che se inseriamo un non nero questo genera una violazione certa alla proprietà 4: di conseguenza, la miglior soluzione è quella di sostituire la foglia con un nodo rosso con due foglie.

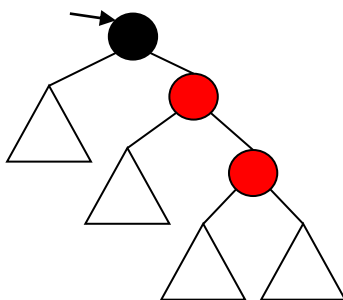
Questo ci consente di dover bilanciare l'albero solo se il padre della foglia (e quindi del nuovo nodo rosso aggiunto) è rosso; inoltre, tale metodo ci assicura che la proprietà 3 è l'unica che possa essere violata e la soluzione sarà quella di spostare verso l'alto i due nodi rossi fino alla radice, che andrà poi colorata di nero.

Questo processo funziona solo se la radice prima dell'inserimento è nera, in caso contrario avremmo due violazioni e non più una (tre nodi rossi in successione), ma possiamo assumere sempre vera tale proprietà poiché colorare la radice di nero non causa nessuna violazione alle proprietà di RB.

Diamo l'implementazione dell'inserimento (poi ci occuperemo del bilanciamento) supponendo che dopo l'algoritmo di inserimento ci sia un'istruzione che metta il colore della radice dell'albero a nero (così abbiamo la certezza di operare solo su alberi RB con radice nera):

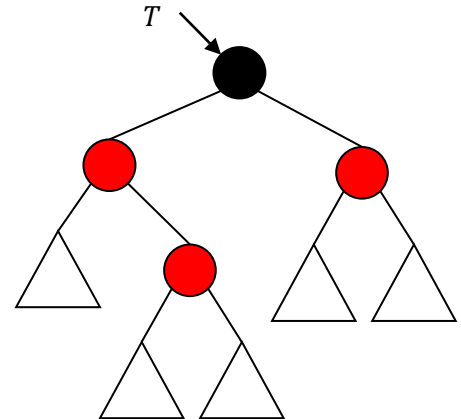
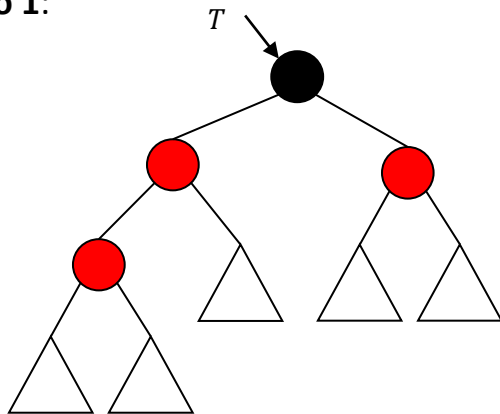
| | |
|---|--|
| <pre> 1 InsertRB(T, k) 2 IF !isNIL(T) THEN 3 IF T->key > k THEN 4 T->sx = InsertRB(T->sx, k) 5 T = BilanciaRB_Sx(T) 6 ELSE IF T->key < k THEN 7 T->dx = InsertRB(T->dx, k) 8 T = BilanciaRB_Dx(T) 9 ELSE /*sono in una foglia*/ 10 x = creaNodoRB() 11 x->key = k 12 x->c = R 13 RETURN x 14 RETURN T </pre> | <p>isNIL(T) è una funzione che restituisce vero se l'unico nodo presente è una foglia NIL (non possiamo utilizzare più T != NIL proprio a causa dei nodi foglia di un RB)</p> <p>La funzione creaNodoRB crea il nodo con già le foglie NIL</p> |
|---|--|

L'idea del bilanciamento è abbastanza semplice: sappiamo che la radice è di colore nero e sappiamo che l'unica violazione che può generarsi è della proprietà 3. Allora, per ogni nodo controllerò il figlio ed il figlio di quest'ultimo:

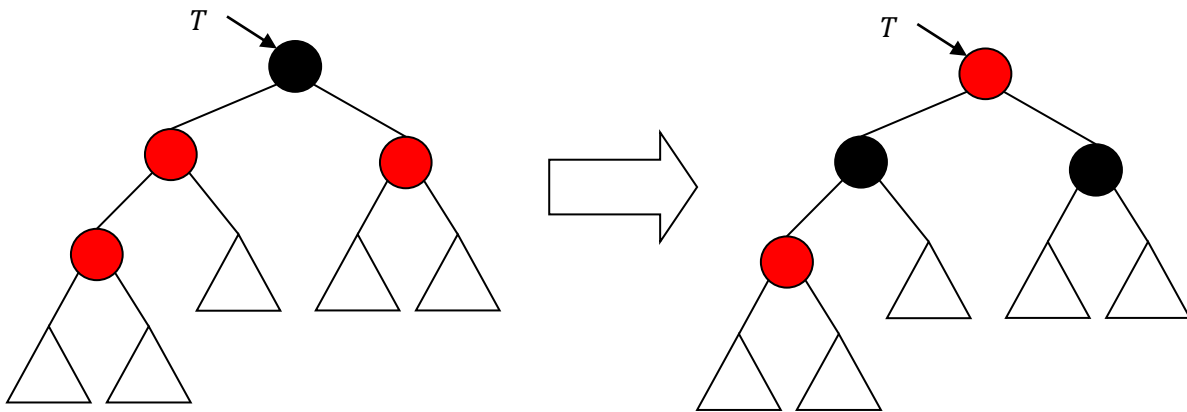


Il nodo che si accorge della violazione è evidentemente nero, perché altrimenti avremmo ben due violazioni e non una, ma un inserimento può generare una sola violazione.

Ci sono 3 possibili casi che si gestiscono in maniera differente. Di seguito vedremo la gestione dei casi per un inserimento a sinistra (che quindi genererà una violazione nel sottoalbero sinistro) ma l'inserimento a destra è del tutto duale (bisogna scambiare solo sinistra con destra).

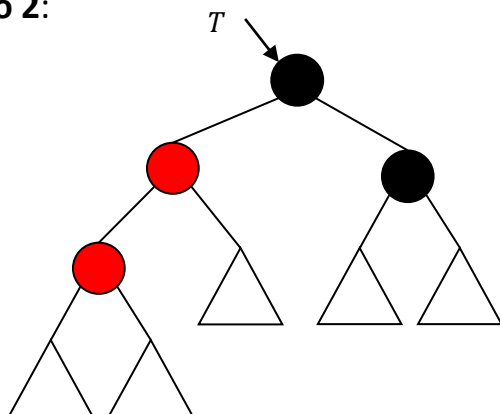
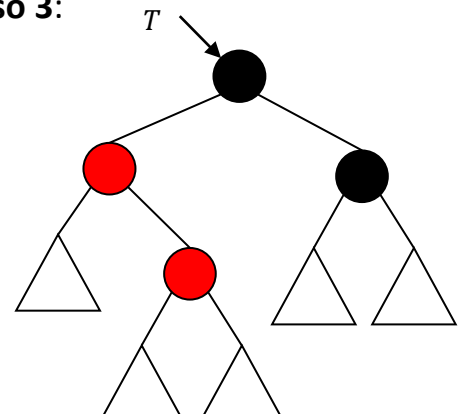
Caso 1:

Anche se le precedenti rappresentazioni sono differenti (una ha una violazione a sinistra con il figlio sinistro e l'altra a sinistra con il figlio destro) si attua la stessa soluzione e quindi lo considereremo come unico caso. Poiché 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:

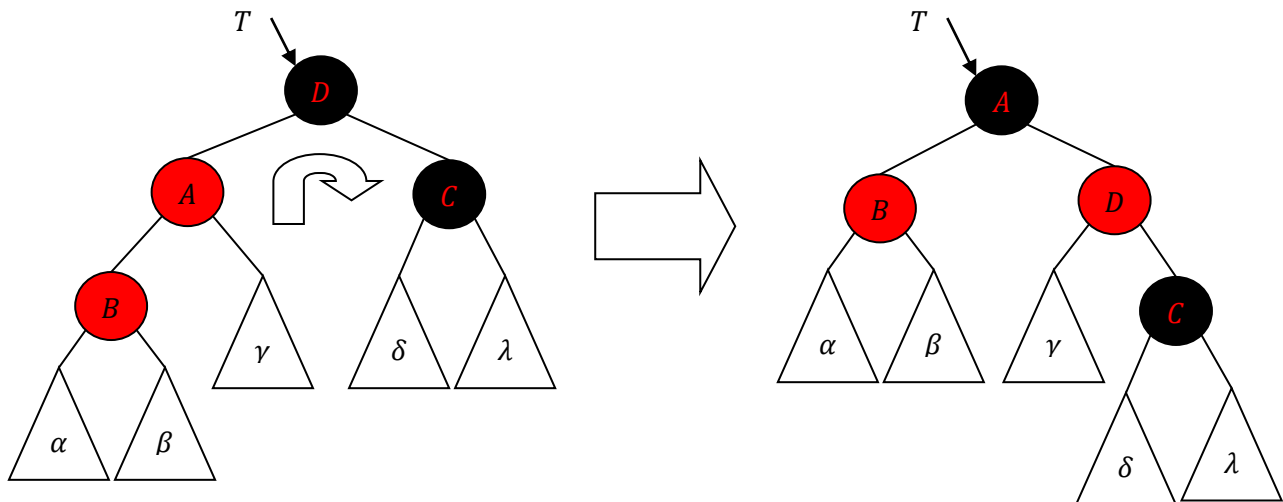


Il fatto che con questa operazione abbiamo risolto la violazione di tipo 3 è evidente, si noti inoltre che non è possibile violare in alcun modo le altre proprietà poiché l'altezza nera è rimasta invariata (per dimostrarlo basta banalmente contare le altezze nere di ogni percorso in T); le altre due sono banali.

A questo punto, se la radice di T è quella dell'albero totale, ho risolto (andrà solo colorata di nero dopo l'inserimento), altrimenti 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.

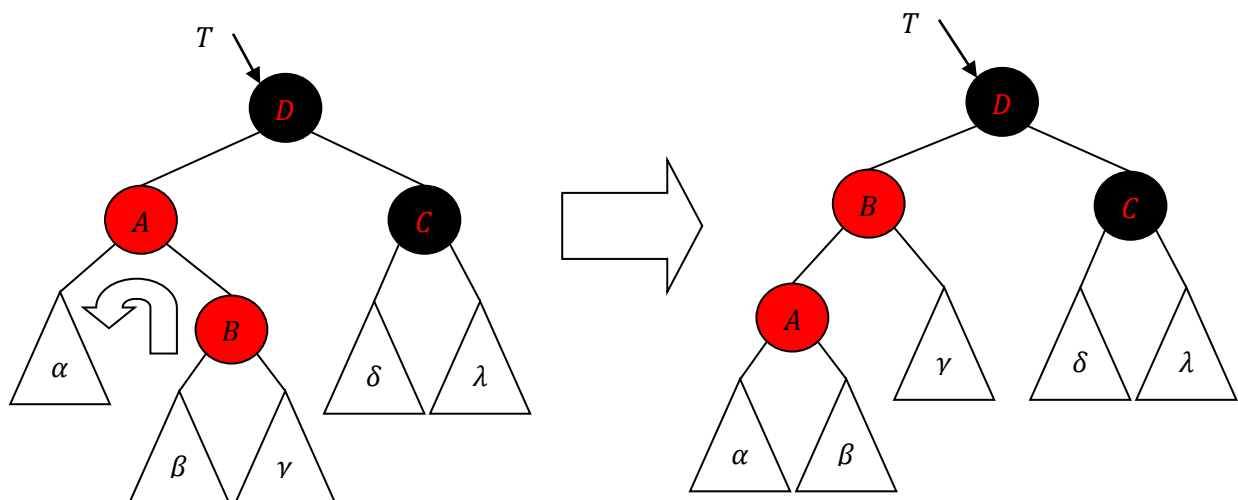
Caso 2:**Caso 3:**

Incominciamo con il caso 2: per risolvere tale violazione proviamo ad eseguire la rotazione singola sinistra già vista con gli AVL:



Dopo la rotazione, per risolvere la violazione è necessario invertire i colori tra A e D ; inoltre poiché l'albero γ era 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 (si faccia il conteggio dell'altezza nera per ogni nodo).

Ci resta solo il caso 3, a cui applicheremo una rotazione doppia sinistra:



Ma dopo la prima rotazione singola destra ci ritroviamo esattamente nella situazione del caso 2: siamo certi che applicando un'altra rotazione singola sinistra, e aver cambiando il colore del nodo radice e del suo figlio destro, abbiamo risolto la violazione anche per quest'ultimo caso.

Si noti che sia per il caso 2 che per il caso 3 la violazione viene risolta totalmente, senza il rischio che si propaghi verso l'alto (come nel caso 1), difatti la radice di T resta nera. Ma allora le operazioni di bilanciamento risolvono la violazione dopo al più due rotazioni (doppia rotazione del caso 3), perché anche se il caso uno può propagare la violazione fino alla radice dell'albero totale, esso non fa alcuna rotazione.

Vediamo gli algoritmi che mettono in pratica quanto appena detto:

```

1  BilanciaRB_Sx(T)
2      v = ViolazioneSx(T)
3      CASE v OF
4          1: T = Caso1Sx(T)
5          2: T = Caso2Sx(T)
6          3: T = Caso3Sx(T)
7      RETURN T

```

Dove $ViolazioneSx(T)$ restituisce il caso in cui ci troviamo e $CasoNSx(T)$ lo gestisce.

```

1  ViolazioneSx(T)
2      A = T->sx
3      IF A->c = R THEN
4          C = T->dx
5          IF C->c = R
6              IF (A->sx)->c = R OR (A->dx)->c = R THEN
7                  RETURN 1
8              ELSE /*C è nero*/
9                  IF (A->sx)->c = R THEN
10                     RETURN 2
11                 ELSE IF (A->dx)->c = R THEN
12                     RETURN 3
13             RETURN 0 /*non ho nessuna violazione*/

```

I seguenti algoritmi sono mie implementazioni:

```

Caso1Sx(T)
    T->c = R
    (T->sx)->c = N
    (T->dx)->c = N
    RETURN T

```

```

Caso2Sx(T)
    A = T->sx
    gamma = A->dx
    A->dx = T
    T->sx = gamma
    A->c = N
    (A->dx)->c = R
    RETURN A

```

```

Caso3Sx(T)
    A = T->sx
    B = A->dx
    beta = B->sx
    B->sx = A
    A->dx = beta
    T->sx = B
    RETURN Caso2sx(T)

```

Cancellazione nei RB

Sfortunatamente nella cancellazione non possiamo scegliere noi il colore del nodo da cancellare. Se cancelliamo un nodo rosso non ci saranno violazioni ai vincoli, ma se ci troviamo nella situazione di dover cancellare un nodo nero allora viene sicuramente creata una violazione di tipo 4: questo vincolo, essendo di tipo 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, e se questo era rosso allora diventa nero, ma se era già nero (per non perdere un nodo nero) diventerà "doppio nero" (praticamente questo nuovo colore contribuirà di 2 e non più di 1 all'altezza nera). In questo modo ho trasformato la violazione del vincolo 4 (globale) in una violazione di vincolo 1 (locale).

Nel caso in cui il doppio nero finisca in radice la soluzione è banalmente quella di colorare la radice di nero poiché abbiamo visto 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 ausiliarie. Supponiamo inoltre che dopo l'utilizzo del seguente algoritmo venga effettuata un'istruzione che colora la radice di nero (così da risolvere 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          ELSE 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 della dell'individuazione e risoluzione della violazione, nel caso poi 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).

| | |
|---|--|
| <pre> 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 cambiare 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 </pre> | <pre> PropagaNero(x, y) IF x->c = N THEN IF y->c = R THEN y->c = N ELSE y->c = DN </pre> |
|---|--|

L'algoritmo di StaccaMinRB(T, P) ha una struttura identica a quello visto per gli AVL (non aggiungeremo il controllo sulla variabile T poiché abbiamo la certezza che non sia mai un nodo foglia):

```

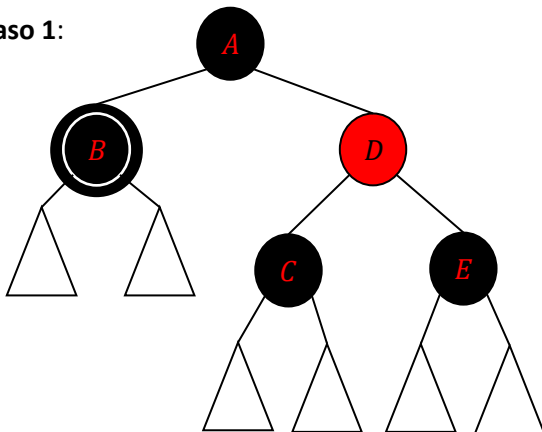
1  StaccaMinRB(T, P)
2      IF !isNIL(T->sx) THEN
3          val = StaccaMinRB(T->sx, T)
4          tmp = BilanciaCancSx(T)
5      ELSE
6          val = T->key
7          tmp = T->dx
8          IF T = P->sx THEN
9              P->sx = tmp
10         ELSE
11             P->dx = tmp
12         PropagaNero(T, tmp)
13         dealloca(T)
14     RETURN val

```

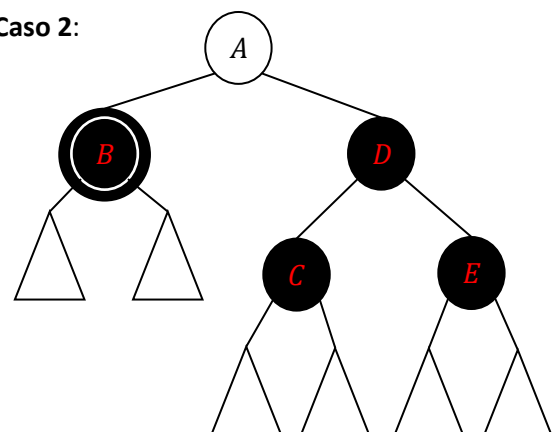
Passiamo ora a definire i vari casi del bilanciamento. Per i seguenti ragionamenti considereremo che la cancellazione sia avvenuta nel sottoalbero sinistro (quindi l'eventuale doppio nero sarà a sinistra), ma il ragionamento per BilanciaCancDx è duale (bisogna scambiare sinistra con destra e viceversa).

Per il bilanciamento ci sono 4 casistiche (se il nodo è bianco allora il comportamento è lo stesso a prescindere che tale nodo sia colorato di nero o di rosso):

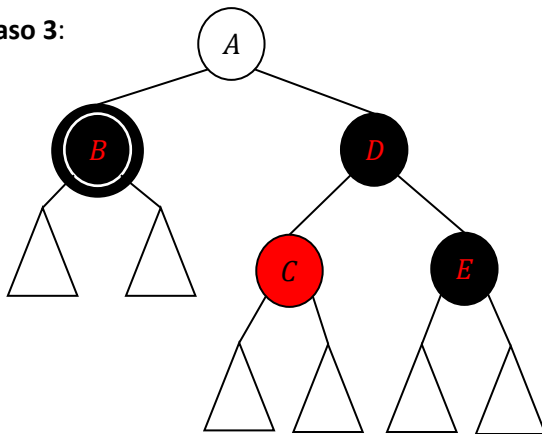
Caso 1:



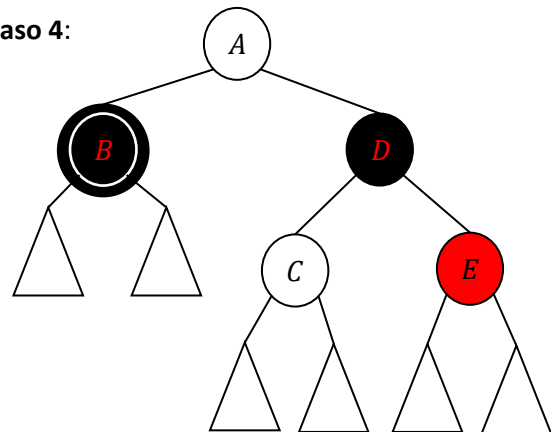
Caso 2:



Caso 3:



Caso 4:



Quindi il caso 2 e il caso 3 includono due possibilità (una per la radice rossa ed una per la radice nera) mentre il caso 4 ne include quattro (una per ogni combinazione di colore tra A e C).

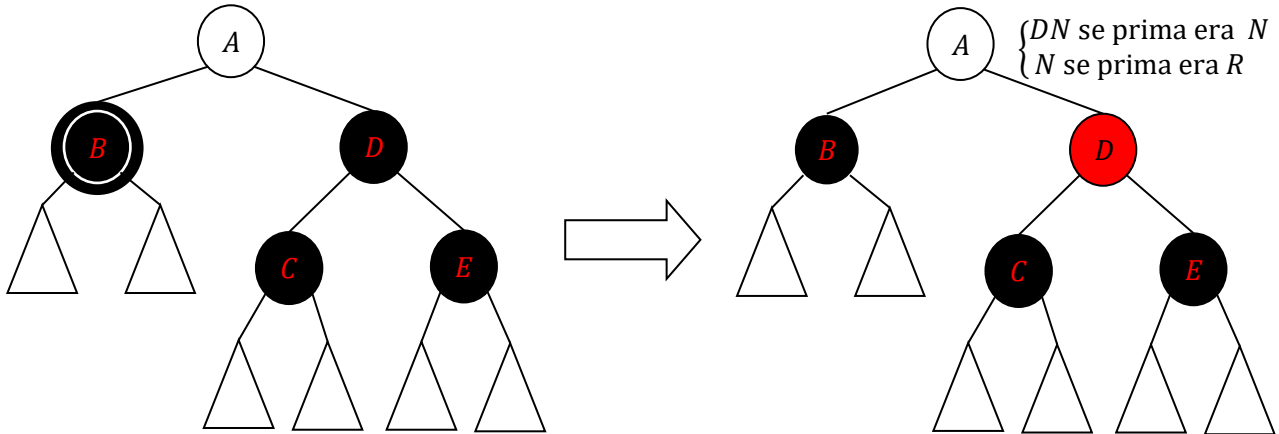
È evidente che per capire il caso in cui ci troviamo non serve analizzare la radice poiché sappiamo che se il figlio destro è rosso allora la radice sarà nera. Quindi se il sinistro è doppio nero allora basta controllare nel giusto ordine gli alberi del sottoalbero destro per capire il caso in cui ci troviamo:

```

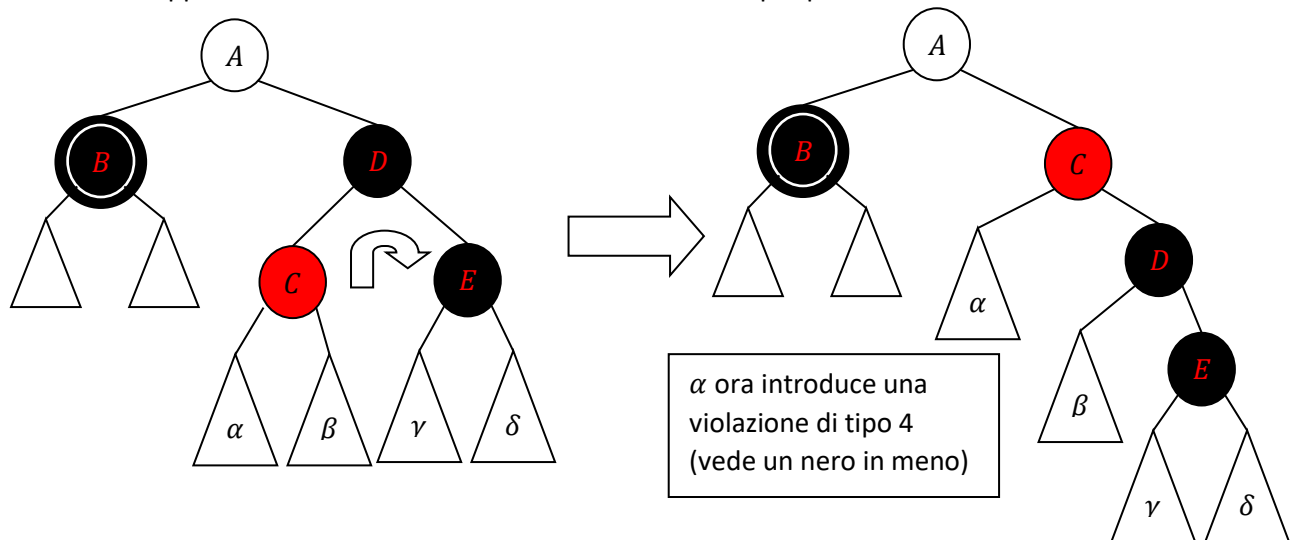
1  ViolazioneSx(S, D)
2      v = 0
3      IF S->c = DN THEN
4          IF D->c = R THEN
5              v = 1
6          ELSE IF (D->dx) ->c = N AND (D->sx) ->c = N THEN
7              v = 2
8          ELSE IF (D->dx) ->c = N THEN
9              v = 3
10         ELSE /*D->dx) ->c = R*/
11             v = 4
12     RETURN v

```

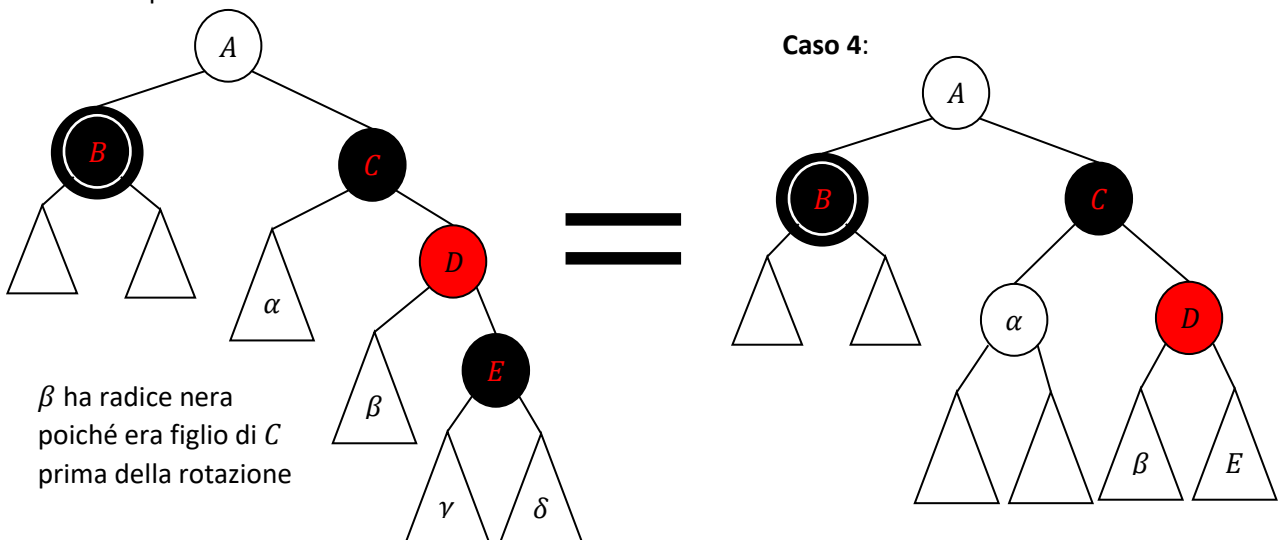
Analizziamo ora la risoluzione dei vari casi cominciando dal più semplice; ovvero, il **caso 2**:



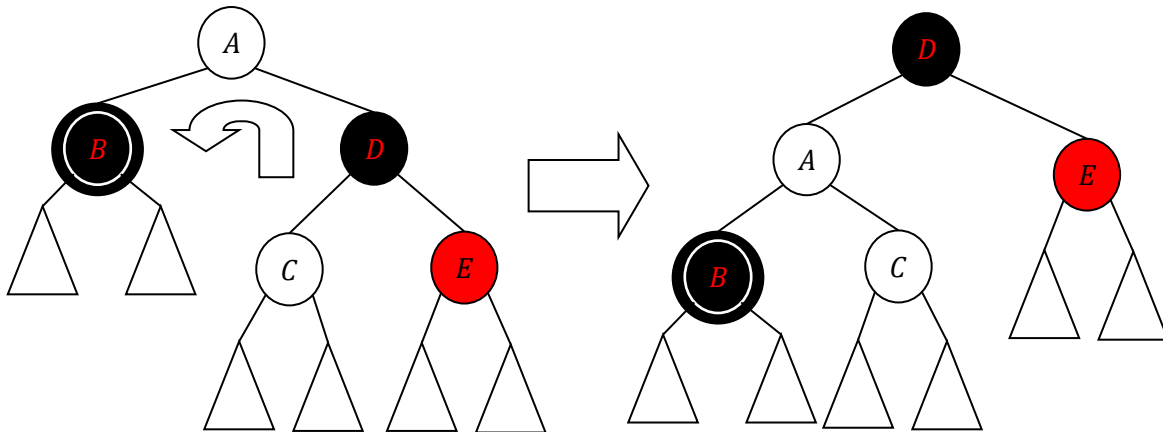
Per il **caso 3** applicheremo una rotazione e un cambio di colore per portarci al caso 4:



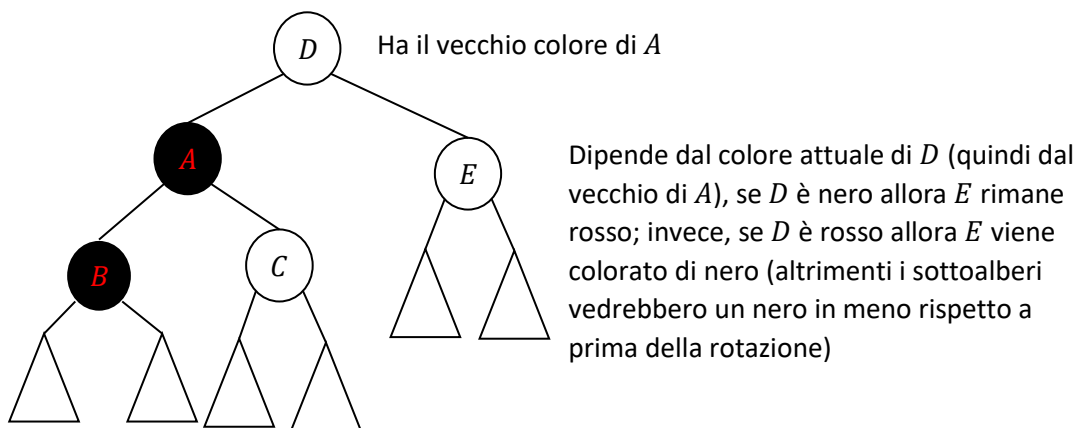
Risolviamo quindi la violazione ricolorando i nodi:



Il **caso 4** (che completa anche il caso 3) è risolvibile tramite una rotazione destra ed un cambio di colore:

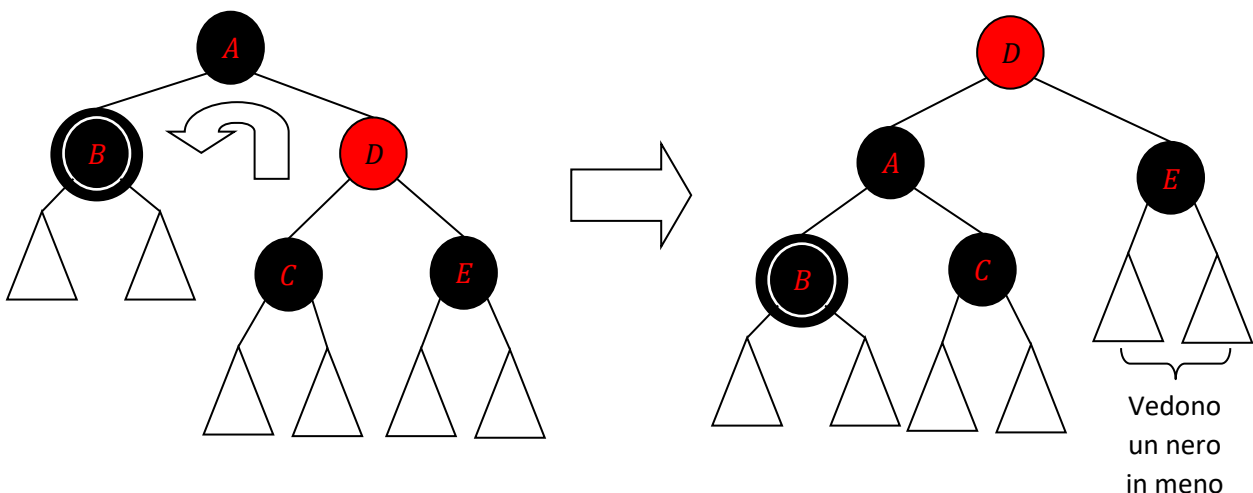


Dopo la rotazione il nodo *B* vede un nero in più, mentre i sottoalberi di *E* potrebbero vedere un nero in meno (dipende dal colore di *A*). Il vincolo quattro è risolvibile semplicemente ruotando i colori nel verso opposto alla rotazione effettuata (tenendo un nero in *B*):

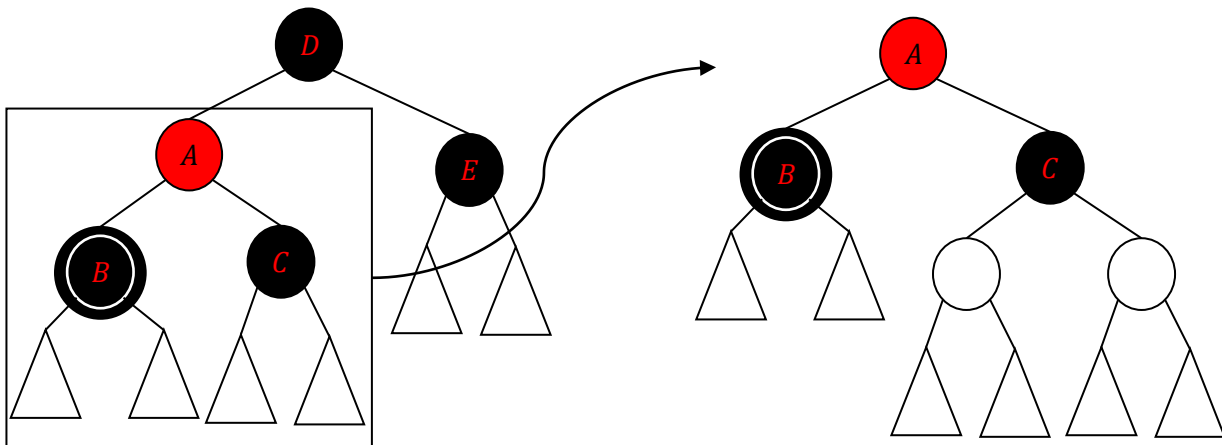


Una volta risolta la violazione del vincolo 4 (le altezze nere sono le stesse di quelle prima della rotazione) è del tutto risolta anche la violazione di tipo 3 e non vengono introdotte nuove violazioni. Dunque, il caso 3 e il caso 4 sono risolutivi (non propagano la violazione come nel caso 2).

Resta ora da dimostrare solo il caso 1, che è quello con l'idea più controintuitiva poiché spinge la propagazione verso il basso:



A questo punto per risolvere la violazione del sottoalbero destro basta colorare la radice D in nero, ma così introdurremmo una violazione nel sottoalbero sinistro (vedrebbe un nero in più); quest'ultima però si risolve facilmente colorando il nodo A di rosso:



Ma allora posso applicare $\text{BilanciaCancSx}(A)$ per risolvere il problema (possiamo farlo poiché siamo nel caso 2, o 3, oppure 4, ma mai nel caso 1 e quindi non ho rischio di loop). Inoltre, poiché ci possiamo trovare solo nel caso 2 con radice rossa abbiamo la certezza che la violazione viene risolta e non propagata; a ciò si aggiungono i casi 3 e 4 che abbiamo visto essere risolutivi. Dunque, il caso 1 risolve in maniera totale la violazione con due bilanciamenti.

Possiamo finalmente scrivere 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 la violazione nell'albero totale.

```

Caso1sx(T)
    T = RotazioneSingolaDx(T)
    T->c = N
    (T->sx)->c = R
    RETURN T

```

```

Caso2sx(T)
    (T->dx)->c = R
    (T->sx)->c = N
    PropagaNero(T->sx, T)
    RETURN T

```

```

Caso3sx(T)
    T->dx = RotazioneSingolaSx(T->dx)
    (T->dx)->c = N
    ((T->dx)->dx)->c = R
    T = Caso4sx(T)
    RETURN T

```

```

Caso4sx(T)
    T = RotazioneSingolaSx(T)
    (T->dx)->c = T->c
    T->c = (T->sx)->c
    (T->sx)->c = N
    ((T->sx)->sx)->c = N
    RETURN T

```

Si noti che a fronte di una cancellazione il numero massimo di rotazioni è tre (caso 1 seguito da caso 3); dunque, i RB non solo sono più tolleranti degli AVL, ma anche meno complessi (se consideriamo una rotazione come misura di complessità).

7. Conversione Algoritmo Ricorsivo in Iterativo

Introduzione

La ricorsione è un meccanismo astratto che in realtà non esiste nel calcolatore poiché quest'ultimo comprende solo il linguaggio macchina, dove non esistono concetti come la ricorsione o la funzione (è proprio come la macchina di Turing).

Di conseguenza, l'unico modo che ha un calcolatore di effettuare la ricorsione è iterando (volendo essere più precisi, nemmeno l'iterazione esiste nel linguaggio macchina, ma può essere facilmente simulata tramite salti ad altre istruzioni).

Esistono casi in cui la ricorsione non utilizza memoria aggiuntiva (ricorsione in coda); quindi una traduzione iterativa di questo tipo di ricorsione non necessita di stack. In tutti gli altri casi è sempre necessario uno stack in cui memorizzare delle variabili per simulare lo stack di attivazione delle chiamate ricorsive.

Vedremo che nei casi in cui l'albero di ricorrenza è almeno binario la richiesta di uno stack sarà obbligatoria.

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. Prendiamo un albero ricorsivo binario, è ovvio che per effettuare le chiamate ricorsive c'è bisogno di visitare l'albero; è proprio questa operazione di scendere e risalire attraverso l'albero che necessita la memorizzazione di determinate informazioni (se non salviamo i nodi del percorso mentre scendiamo come facciamo a risalire?).

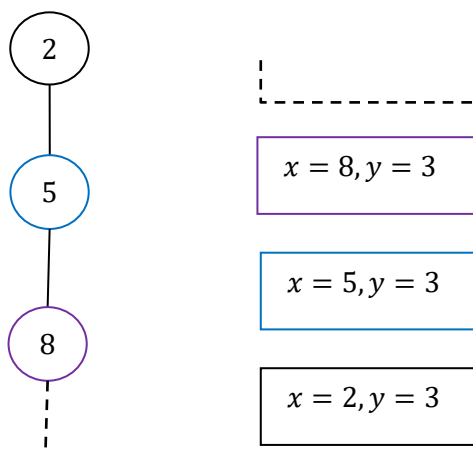
Sappiamo che durante una chiamata ricorsiva vengono memorizzate delle informazioni nello stack di attivazione, di cui le informazioni principali riguardano le variabili usate dall'algoritmo ricorsivo.

Sia la ricorsione che l'iterazione non fanno altro che ripetere blocchi di codice, ma la differenza fondamentale sta nel fatto che l'iterazione condivide le variabili, ovvero, ogni ripetizione usa e vede le stesse variabili (una modifica di una variabile in una ripetizione si ripercuote nella successiva). Ciò non accade nella ricorsione, dove ogni chiamata avrà una sua copia delle variabili in aree di memoria scollegate tra loro e quando la chiamata termina anche l'area di memoria utilizzata viene ripulita (le modifiche in una chiamata ricorsiva non vengono viste dalle altre).

Esempio:

```
Ric(x)
  y = 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 lo stesso valore sono tutte variabili allocate in aree di memoria diverse



Ed è proprio questo meccanismo che dobbiamo tradurre in algoritmo iterativo (ciò implica memoria aggiuntiva per ogni chiamata ricorsiva).

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, ragion per cui, prima di poter passare alla prossima 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 libero la memoria occupata e riprendo il controllo della chiamata precedente. Poiché chi riprende prima il controllo è l'ultima chiamata ad essere stata effettuata è evidente che siamo in una tipologia LIFO; dunque, per simulare questo meccanismo nell'iterazione è necessario utilizzare la struttura dati stack.

Il numero di ripetizioni del blocco di codice in una chiamata ricorsiva dipende dall'input, perciò, non possiamo sapere a priori quante iterazioni deve fare un determinato algoritmo. Ciò implica che non è possibile utilizzare un ciclo for bensì bisogna implementare un ciclo while.

Struttura dell'algoritmo

Una chiamata ricorsiva o è nel suo stato di inizio (deve cominciare a effettuare il suo blocco di istruzioni) oppure deve riprendere l'esecuzione dal punto in cui è stata sospesa. Ciò significa che il blocco while sarà suddiviso in due parti principali e la sua condizione sarà definita dalle seguenti variabili:

- Start: flag booleano che indica se dobbiamo iniziare una nuova chiamata ricorsiva
- (st $\neq \emptyset$): questo valore booleano determinerà se sono presenti o meno 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 è falso) o riprendere (stack vuoto).

Il blocco while sarà così strutturato (vedremo in seguito come vanno inizializzate le variabili):

```
WHILE start OR st != NIL DO
  IF start THEN
    /*simulo l'inizio della nuova chiamata ricorsiva*/
  ELSE
    /*ripristino il contesto della vecchia chiamata*/
    /*prendendolo dalla stack e riprendo l'esecuzione*/
```

La complessità della ripresa di una chiamata ricorsiva sta nel capire in quale punto riprendere l'esecuzione, poiché se un algoritmo fa diverse chiamate ricorsive vuol dire che ho più punti in cui le chiamate vengono sospese e altrettanti punti in cui vengono riprese.

Esempi di traduzione

PrintTree

Partiamo con un banale algoritmo che prende in input un albero binario e ne stampa i nodi in post order. La sua implementazione ricorsiva è estremamente semplice:

```
1 PrintTree(T)
2   IF T != NIL THEN
3     PrintTree(T->sx)
4     PrintTree(T->dx)
5     print(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 /*ci permette di cominciare 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              /*partiamo con la prima istruzione*/
9              IF cT != NIL THEN
10                 /*dobbiamo sospendere l'attuale chiamata ricorsiva*/
11                 st = Push(st, cT) /*dobbiamo sospendere l'attuale chiamata ricorsiva*/
12                 cT = cT->sx /*simulo il passaggio di parametri*/
13                 start = true /*devo cominciare una nuova chiamata ricorsiva*/
14             ELSE /*se siamo in un caso base dobbiamo terminare la chiamata*/
15                 start = false /*non devo cominciare una nuova chiamata*/
16                 last = cT /*mi memorizzo l'input dell'attuale chiamata ricorsiva*/
17                 /*così da capire quale chiamata ricorsiva va ripresa*/
18             ELSE /*ripristino il contesto*/
19                 cT = top(st)
20                 /*per capire se sono nella prima chiamata ricorsiva o nella seconda*/
21                 /*controllo l'input della chiamata figlia, poiché se T è foglia, è*/
22                 /*possibile che sia sx che dx siano NIL quindi devo differenziare*/
23                 IF last != cT->dx AND cT->dx != NIL THEN
24                     /*simulo la seconda chiamata*/
25                     cT = cT->dx
26                     start = true
27                 ELSE IF last != cT->dx THEN
28                     /*qui dovrei eventualmente simulare la chiamata ricorsiva*/
29                     /*PrintTree(T->dx = NIL) ma in questo algoritmo non serve*/
30                     /*poiché non devo far nulla, ma ciò non è sempre vero per*/
31                     /*altri algoritmi(serve per differenziare i due casi NIL)*/
32                 ELSE
33                     Print(cT->key)
34                     /*simulo la terminazione*/
35                     last = cT
36                     start = false
37                     pop(st)

```

Altezza

Proponiamo ora un esempio di algoritmo ricorsivo che prende in input un albero e ne restituisce l'altezza:

```

1  Altezza(T)
2      IF T != NIL THEN
3          sx = Altezza(T->sx)
4          dx = Altezza(T->dx)
5          h = 1 + max(sx, dx)
6          RETURN h
7      RETURN -1

1  AltezzaIter(T)
2      cT = T
3      st = NIL
4      st_sx = NIL
5      ret = -1 /*così da coprire anche la chiamata 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 qui devo pulire la memoria di sx corrente*/
44      RETURN ret

```

Si noti 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 abbiamo utilizzato `st_sx`).

QuickSort

```

1  QuickSort(A, p, r)
2      IF p < r THEN
3          q = Partiziona(A, p, r)
4          QuickSort(A, p, q)
5          QuickSort(A, q + 1, r)

```

Partiziona è già iterativo di suo, quindi possiamo usarla come chiamata a funzione (se fosse stato ricorsivo avremmo dovuto tradurlo per poterlo usare).

Sullo stack devo mettere tutti quei valori definiti (quindi scritti) prima di una chiamata ricorsiva e letti dalla chiamata successiva (se è solo scritta prima o solo letta dopo non è necessario lo stack, usare stack superflui è un inutile spreco di spazio che influenza negativamente su un eventuale esercizio d'esame). Nel nostro caso le uniche variabili necessarie da memorizzare su uno stack sono q e r ; infatti, p viene perso dopo la prima chiamata ricorsiva; ovvero, non viene usato 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 e r poiché abbiamo la certezza che $p \leq q < r$; tra questi useremo r perché se scegliessimo p non avremmo la possibilità di confrontarlo (p viene perso). Se gli r tra una chiamata ricorsiva figlia ed una padre sono diversi allora è sicuramente terminata la prima chiamata, mentre se sono uguali siamo alla fine della seconda.

Ricapitolando: nello stack abbiamo q e r e 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 p,q > 0*/
6   start = true
7   WHILE (start OR st_r != NIL) DO
8       IF start THEN
9           IF cp < cr THEN
10              q = Partiziona(A, cp, cr)
11              st_q = push(st_q, q)
12              st_r = push(st_r, cr)
13              cr = q
14           ELSE
15              last = cr
16              /*forziamo la risalita*/
17              start = false
18           ELSE /*ripristiniamo il contesto, si noti che abbiamo perso cp*/
19              cr = top(st_r)
20              q = top(st_q)
21              /*riconorso in quale punto siamo*/
22              IF cr != last THEN
23                  cp = q + 1
24                  /*cr non cambia*/
25                  start = true
26              ELSE /*sono tornato dalla seconda chiamata*/
27                  /*aggiorno lost e pulisco lo stack*/
28                  last = cr /*istruzione obsoleta*/
29                  start = false /*anche questa è ridondante*/
30                  st_q = pop(st_q)
31                  st_r = pop(st_r)

```

Questo controllo basta poiché i due stack sono sincronizzati quindi uno vale l'altro. Ciò ovviamente non è detto per qualsiasi algoritmo poiché in alcuni potrei avere degli stack che si riempiono in maniera differente.
N.B.: ci sarà sempre uno stack che si riempie in base alle chiamate ricorsive

Count

```

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 l'algoritmo poiché la traduzione in iterativo trascende il funzionamento in sé (è più legato alla sintassi del linguaggio). Infatti, lo scopo è **simulare** l'algoritmo ricorsivo e non creare un algoritmo iterativo che faccia la stessa cosa (anche se risulta più efficiente l'esercizio verrà considerato sbagliato).

Prestiamo attenzione al nostro algoritmo:

- $q < r$ sicuramente, e quindi anche $q - 1 < r$, allora possiamo benissimo usare r come *last*
- Per gli stessi motivi visti in [QuickSort](#), abbiamo bisogno di uno stack per r e di uno per q
- Anche x viene scritto e letto in entrambe le chiamate e quindi bisogna tenerne traccia (serve uno stack), inoltre bisogna tener conto anche del fatto che x cambia nel tempo (viene assegnato nella linea 6, letto dalla prima chiamata ricorsiva e riassegnato dopo la stessa; viene riassegnato anche nella linea 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                     /*qui perderei 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 che cambia nella chiamata successiva*/
18                 ELSE
19                     ret = x
20                     last = cr
21                     start = false
22             ELSE /*ripristino il contesto*/
23                 /*a prescindere da quale chiamata torno le variabili da ripristinare*/
24                 /*sono le stesse. N.B.: non è detto che valga per altri algoritmi*/
25                 cr = top(st_r)
26                 q = top(st_q)
27                 x = top(st_x)
28                 IF last != cr THEN
29                     x = x + ret /*simula l'istruzione di RETURN*/
30                     /*ora in st_x c'è un valore obsoleto e non quello aggiornato*/
31                     st_x = pop(st_x)
32                     st_x = push(st_x, x)
33                     /*assegno gli input che cambiano e comincio la nuova chiamata*/
34                     cp = q + 1
35                     start = true
36                 ELSE /*torno dalla seconda chiamata*/
37                     x = x + ret
38                     ret = x
39                     last = cr
40                     start = false
41                     st_r = pop(st_r)
42                     st_q = pop(st_q)
43                     st_x = pop(st_x)
44             RETURN x

```

Si noti che, come valore di ritorno della funzione (linea 44), avremmo potuto mettere anche *ret* poiché al termine del while avranno lo stesso valore, però conviene sempre separare il valore di ritorno della chiamata principale con i return tra chiamata padre e figlia onde evitare possibili errori.

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, L')
10         ELSE
11             L'->next = Algo(A, p, q - 1, L)
12             x = Algo(A, q + 1, r, L')
13     RETURN x

```

Ora è un po' più complesso discriminare le chiamate ricorsive ma fortunatamente le copie vengono suddivise dalla condizione $A[q] \% 2 = 0$, che insieme a *last*, ci dà tutte le informazioni per poter disambiguare le quattro 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 chiamata 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 ma viene letto solo nel valore di ritorno, quindi dopo tutte le chiamate ricorsive. Inoltre, poiché 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.

```

7      WHILE (start OR st_r != NIL) DO
8          IF start THEN
9              x = L
10             IF cp <= cr THEN
11                 q = (cp + cr) / 2
12                 L' = AllocaNodo()
13                 L'->key = A[q]
14                 /*devo distinguere quale coppia far partire*/
15                 /*ma nello stack devo mettere le stesse cose*/
16                 st_r = push(st_r, cr)
17                 st_p = push(st_p, cp)
18                 st_L' = push(st_L', L')
19                 IF A[q] % 2 = 0 THEN
20                     cp = q + 1
21                 ELSE
22                     cr = q - 1
23             ELSE /*caso base*/
24                 ret = x
25                 start = false
26                 last = cr

```

```

27 ELSE
28     cp = top(st_p)
29     cr = top(st_r)
30     L' = top(st_L')
31     q = (cp + cr)/2
32     IF A[q]%2 = 0 THEN
33         IF last != cr THEN
34             x = ret
35             /*ret = x*/
36             last = cr
37             start = false
38             st_p = pop(st_p)
39             st_r = pop(st_r)
40             st_L' = pop(st_L')
41         ELSE /*è terminata la prima*/
42             L'->next = ret
43             /*cambiano i parametri 3 e 4*/
44             cr = q - 1
45             cL = L'
46             start = true
47         ELSE /*siamo nella seconda coppia*/
48             IF last != cr THEN
49                 L'->next = ret
50                 /*cambiano i parametri 2 e 4*/
51                 cp = q + 1
52                 cL = L'
53                 start = true
54             ELSE /*è terminata la seconda*/
55                 x = ret
56                 /*ret = x*/
57                 last = cr
58                 start = false
59                 st_p = pop(st_p)
60                 st_r = pop(st_r)
61                 st_L' = pop(st_L')
62 RETURN x

```

N.B.: se nell'assurdo caso ci si ritrova a dover mettere un RETURN all'interno del while allora stiamo sicuramente sbagliando.

Esercizio svolto

Gli esercizi di questa tipologia sono facilmente generabili (basta creare un algoritmo ricorsivo a caso e tradurlo) ma di seguito ne proponiamo uno interessante. La soluzione è alla prossima pagina.

| | | |
|---|--------------------------------------|--|
| <pre> 1 Algo(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) </pre> | <pre> 10 11 12 13 14 15 16 17 </pre> | <pre> x = x + Algo(T->sx, k1, k2, T) IF P != NIL THEN IF T = P->sx THEN P->sx = cancellaRoot(T) ELSE P->dx = cancellaRoot(T) x = x + 1 RETURN x </pre> |
|---|--------------------------------------|--|

Suggerimenti: Nello stack dobbiamo salvare P e T ed anche x , poiché dopo la linea 9 ho bisogno di salvarlo prima di cominciare la seconda chiamata ricorsiva (ovviamente non ho bisogno di $ck1$ e $ck2$ perché non cambiano durante la loro esecuzione, uso direttamente $k1$ e $k2$).

Soluzione: Possiamo notare già dall'algoritmo ricorsivo che anche se P e T sono due variabili differenti, esse sono strettamente collegate tra loro. Dunque, possiamo usare un unico stack per entrambe, infatti mettendo P nello stack prima di T posso raggiungere P semplicemente risalendo una volta in più nello stack (sarà più chiaro guardando il codice), 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; infatti, ci obbliga a dover inizialmente (prima del while) mettere il parametro P già nello stack e quindi non possiamo più utilizzare il controllo $st \neq \emptyset$ per uscire dal while. Fortunatamente abbiamo la certezza che P sia esattamente il primo elemento inserito nello stack e determina in maniera univoca la prima chiamata ricorsiva (in tutte le altre avrò 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 cT != NIL THEN
11                 /*una chiamata ricorsiva la farò sicuramente*/
12                 st_T = push(st_T, T)
13                 cP = cT /*il quarto parametro è sempre lo stesso in ogni chiamata*/
14                 IF cT->key > k2 THEN
15                     cT = cT->sx
16                 ELSE
17                     cT = cT->dx
18             ELSE
19                 ret = x
20                 start = false
21                 last = cT
22         ELSE
23             cT = top(st_T)
24             /*se sono in una chiamata singola devo fare la stessa cosa*/
25             IF cT->key < k1 OR cT->key > k2 THEN
26                 x = ret
27                 st_T = pop(st_T)
28                 /*start = false*/
29                 last = cT
30             ELSE
31                 IF last != cT->sx AND cT->sx != NIL THEN
32                     /*torno da destra (prima chiamata ricorsiva) e sinistra non è NIL*/
33                     x = ret
34                     st_x = push(st_x, x)
35                     cP = cT
36                     cT = cT->sx
37                     start = true
38                 ELSE IF last != cT->sx THEN
39                     /*torno da destra e il sinistro è NIL*/
40                     x = ret /*basta per simulare il caso base, infatti x = x + 0*/
41                 ELSE /*se sono qui ho last = cT->sx ma questo potrebbe essere NIL*/
42                     IF cT->sx = NIL THEN
43                         x = ret
44                     ELSE
45                         x = top(st_x) + ret
46                         st_x = pop(st_x)
47                         cT = top(st_T)

```

Equivale a scrivere

```

IF cT->key < k1 THEN
    cT = cT->dx
ELSE IF cT->key > k2 THEN
    cT = cT->sx
ELSE
    cT = cT->dx

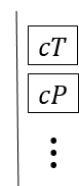
```

```

48      /*dovrò in ogni caso ripulire st_T*/
49      st_T = pop(st_T)
50      cP = top(st_T)
51      IF cP != NIL THEN
52          IF cT = cP->sx THEN
53              cP->sx = cancellaRoot(cT)
54          ELSE
55              cP->dx = cancellaRoot(cT)
56          ret = x
57          /*start = false*/
58      st_T = pop(st_T) /*dopo il while abbiamo ancora P in st_T*/
59      RETURN x

```

Alla linea 48
st_T avrà lo
stato seguente



Ricorsione in coda

Abbiamo già accennato a algoritmi ricorsivi per cui esiste una versione iterativa che non fa uso di stack, come già visto negli [algoritmi sulle liste](#).

Questo tipo di algoritmi sono possibili solo nei casi in cui si conosce esattamente la “strada” da seguire, ovvero, hanno 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 su un generico albero binario dove non si hanno proprietà di ordinamento totale 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  CancellateIter(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
12             IF cT = cP->sx THEN
13                 cP->sx = CancellateRoot(cT)
14             ELSE
15                 cP->dx = CancellateRoot(cT)
16         ELSE /*dobbiamo cancellare la radice*/
17             T = CancellateRoot(T) /*cT = T*/
18     RETURN T

```

L'algoritmo ricorsivo che più si avvicina all'algoritmo di cui sopra è il seguente:

```

1  CancellateRC(T, k, P)
2      IF T != NIL THEN
3          IF T->key > k THEN
4              CancellateRC(T->sx, k, T)
5          ELSE IF T->key < k THEN
6              CancellateRC(T->dx, k, T)
7          ELSE /*T contiene k*/
8              IF P != NIL THEN
9                  IF T = P->sx THEN
10                     P->sx = CancellateRoot(T)
11                  ELSE
12                     P->dx = CancellateRoot(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 precedente andrebbe usato dal seguente algoritmo:

```

1  Cancelli(T, k)
2      IF T->key != k THEN
3          CancelliRC(T, k, NIL)
4      ELSE
5          T = CancelliRoot(T)
6      RETURN T

```

In ogni caso, ritornando a CancelliRC si può notare che ogni chiamata ricorsiva viene effettuata prima del RETURN: tale proprietà è proprio quella che deve avere ogni algoritmo per essere definito ricorsivo in coda, oltre a quella di avere al più una sola chiamata ricorsiva per ogni chiamata.

Quindi se non ci sono altre istruzioni di lettura dopo una chiamata ricorsiva (non bisogna riprendere il contesto della chiamata precedente) e c'è al più una chiamata ricorsiva, allora l'algoritmo è detto ricorsivo in coda; un tale algoritmo, dunque, non ha bisogno di stack per gestire la sua esecuzione.

N.B.: se invece di cancellare solamente il nodo volessimo restituire la profondità in cui si trovava il nodo cancellato l'algoritmo ricorsivo sarebbe 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 = CancelliRoot(T)
12                 ELSE
13                     P->dx = CancelliRoot(T)
14             RETURN x

```

E questo è ben lontano dall'essere ricorsivo in coda. Ma poiché 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, l)
2      IF T != NIL THEN
3          IF T->key > k THEN
4              RETURN Canc(T->sx, k, T, l + 1)
5          ELSE IF T->key < k THEN
6              RETURN Canc(T->dx, k, T, l + 1)
7          ELSE
8              IF P != NIL THEN
9                  IF T = P->sx THEN
10                     P->sx = CancelliRoot(T)
11                  ELSE
12                     P->dx = CancelliRoot(T)
13              RETURN l
14      RETURN -1

```


8. Grafi

Definizioni sui grafi

I grafi sono uno strumento di rappresentazione (modellazione) di problemi molto importante; infatti, la soluzione di molti problemi reali può essere ricondotta alla soluzione di opportuni problemi su grafi.

Un grafo è definito matematicamente come una coppia di insiemi $G = (V, E)$, dove V è un insieme di nodi, chiamati **vertici**, ed $E \subseteq V \times V$ è una relazione binaria tra gli elementi di V (è praticamente un insieme di coppie di vertici, detto insieme degli **archi**).

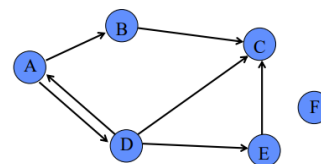
Sia $G = (V, E)$ un grafo, allora per l'insieme degli archi sono possibili $2^{|V|^2}$ combinazioni (o in altri termini, dato V è possibile ottenere $2^{|V|^2}$ grafi diversi); per quanto detto, è possibile definire il collegamento tra due vertici con un singolo bit (1 se c'è un arco, 0 altrimenti).

Tipi di grafi

I grafi possono esprimere sia relazioni simmetriche (grafo non orientato), come ad esempio la relazione "amico di", che antisimmetriche (grafo orientato), come ad esempio una qualsiasi relazione gerarchica.

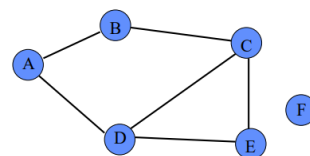
Un **grafo orientato** G è una coppia (V, E) dove:

- 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 che un grafo orientato, mentre se la relazione è antisimmetrica allora posso rappresentarla solo con un grafo orientato. Nonostante sia meno potente, il grafo non orientato è utile in ambito informatico poiché rende più semplice la soluzione di alcuni problemi (noi ci concentreremo su grafi orientati).

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 due vertici non esiste un arco, si dice che il costo è infinito.

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 entrate e grado uscente:
 - Il **grado entrate** di un vertice è il numero di archi **incidenti** in esso (l'arco (u, v) è incidente da u in v , ovvero la freccia parte da u 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

Sottografo

Sia $G = (V, E)$ un grafo, un grafo $G' = (V', E')$ è **sottografo** di G se, e solo se, $V' \subseteq V$ e $E' \subseteq E \cap (V' \times V')$ (quindi abbiamo che $E' \subseteq E$ e $E' \subseteq V' \times V'$ contemporaneamente, N.B.: scrivere solo una delle due non definisce necessariamente un grafo poiché potrebbero esserci archi inesistenti in G').

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 poiché ci permette di decomporre i problemi su grafi in sottoproblemi della stessa natura.

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 $(v_i, v_{i+1}) \in E, \forall 0 \leq i < k$ (dove $k = |V|$)

Si noti che tale definizione rende il percorso più breve il cosiddetto percorso vuoto, ovvero, un percorso composto da un singolo vertice. Infatti, $(v_i, v_{i+1}) \in E, \forall 0 \leq i < 0$ è banalmente vera (essendo il dominio di tale proprietà vuoto, non esiste alcun elemento che possa contraddire tale proprietà universale).

Escludendo suddetto 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 esempio, $v_0 v_3 v_2 v_4 v_0$ è semplice, mentre non lo è $v_0 v_1 v_2 v_3 v_2 v_6$)

La **lunghezza** di un percorso è pari al numero di vertici meno uno (ovvero il numero di archi di un percorso). La **distanza** tra due vertici è pari alla lunghezza del percorso minimo (quello più corto) tra questi due vertici (si noti che il concetto di livello per un albero non è altro che la distanza tra la radice ed un nodo situato in quel livello).

Raggiungibilità

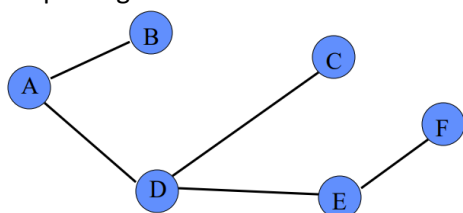
Siano u, v vertici diremo che u è **raggiungibile** da v in G se, e solo se, esiste un percorso in G che parte da v e termina in u . La relazione di raggiungibilità è transitiva (se x è raggiungibile da y e y è raggiungibile da z , allora x è raggiungibile da z) e riflessiva (basti pensare al percorso vuoto).

La verifica di raggiungibilità ci permette di risolvere molti problemi su grafi, inoltre 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: $R_E = \underbrace{\{(v, v) | v \in V\}}_{\text{riflessività}} \cup \underbrace{\{(v, u) | \exists w \in V : (w, u) \in E \wedge (v, w) \in R_E\}}_{\text{transitività}}$ (N.B.: questa è una definizione ricorsiva ben definita, con caso base il percorso vuoto).

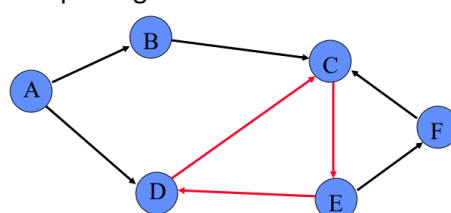
Grafi ciclici e aciclici

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**, mentre un grafo ciclico è un grafo con uno o più cicli.

Esempio di grafo aciclico:



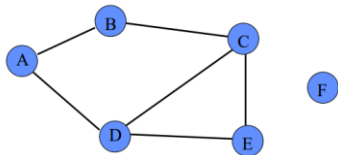
Esempio di grafo ciclico:



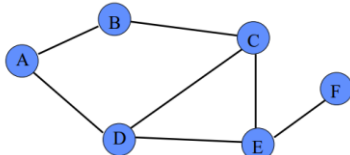
Grafi connessi

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.

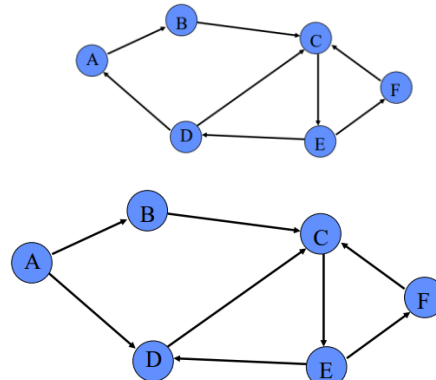
Non connesso (o **sconnesso**):



Connesso:



Fortemente connesso:



Se G è un grafo orientato non fortemente connesso, ma se il grafo non orientato sottostante (cioè senza la direzione degli archi) è connesso, diciamo che G è **debolmente connesso**.

Il grafo sulla destra non è fortemente connesso perché non esiste percorso da D ad A , ma è debolmente connesso.

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 (praticamente contiene più alberi).

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, fondamentalmente, in due rappresentazioni: rappresentazione a **matrice di adiacenza** e rappresentazione a **liste di adiacenza**.

Matrice di adiacenza

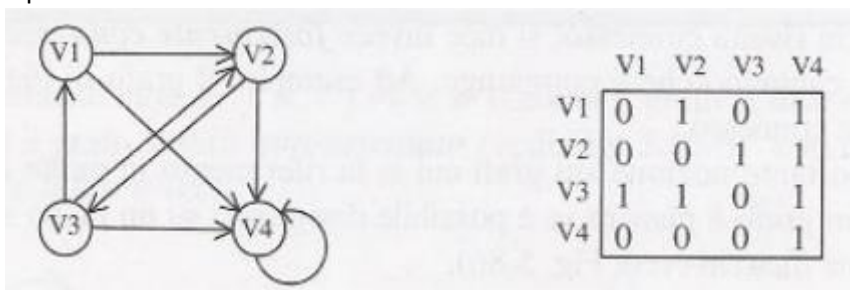
Una delle possibili implementazioni è quella di descrivere l'insieme E tramite la funzione caratteristica (tale funzione restituisce 1 se l'elemento in input appartiene all'insieme, 0 altrimenti).

Dunque, essendo $E \subseteq V \times V$, la funzione caratteristica avrà dominio $V \times V$, ma questo ragionamento vale solo per insiemi finiti; infatti, dovendo generare tutte le combinazioni possibili tra i vertici 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:

$$M(i, j) = \begin{cases} (i, j) \mapsto 1 & \text{se } (i, j) \in E \\ (i, j) \mapsto 0 & \text{se } (i, j) \notin E \end{cases}$$

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 rappresentazione è la più usata nei libri di teoria dei grafi (per la sua naturale traduzione in grafo) ma non è la più efficiente in ambito informatico, ed il motivo è abbastanza evidente: questa rappresentazione necessita di uno spazio in memoria pari a $|V|^2$ a prescindere dal numero di archi presenti.

Ma idealmente, essendo un grafo una coppia di insiemi ci aspettiamo che la dimensione per la rappresentazione sia $\Theta(|V| + |E|)$ e non $\Theta(|V|^2)$. Se ad esempio abbiamo il seguente grafo:

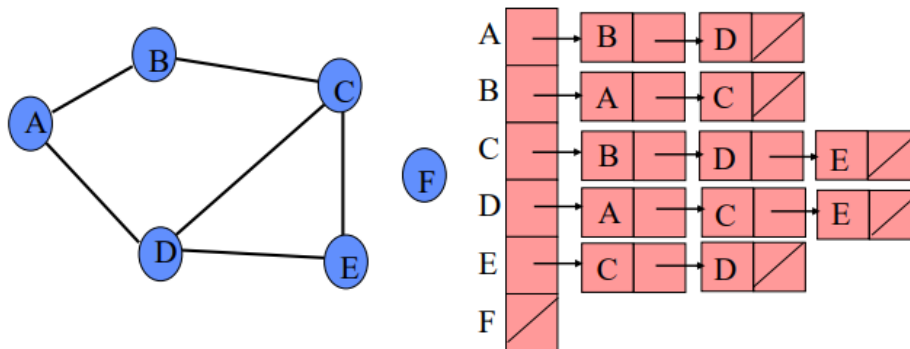
$\boxed{v_1} \rightarrow \boxed{v_2} \cdots \rightarrow \boxed{v_n}$ dove $|V| = n$ e $|E| = n - 1$, ci piacerebbe rappresentarlo in uno spazio in memoria pari a $2n - 1$ e non n^2 come avviene nella rappresentazione con matrice di adiacenza.

Questo spreco di memoria è dovuto al fatto che questa rappresentazione dà informazioni su ogni elemento del dominio (anche gli archi inesistenti), ma ha comunque dei vantaggi sul tempo di esecuzione di alcuni algoritmi; infatti, aggiungere o rimuovere un arco è un'operazione a tempo costante, così come l'accesso ad un elemento (verificare se due vertici sono adiacenti). Però aggiungere vertici è un'operazione molto costosa poiché richiede l'allocazione di una nuova matrice e la copia degli elementi presenti nell'altra (tempo quadratico).

Liste di adiacenza

A noi l'informazione che più interessa è sapere quali coppie sono in E , in quanto puntiamo a muoverci tra i vertici attraverso gli archi (quindi tramite percorsi); una rappresentazione con tale caratteristica e quindi molto più efficiente dal punto di vista della memoria è la rappresentazione a liste di adiacenza.

Ad ogni vertice associa i vari archi uscenti: abbiamo una struttura dati che rappresenta l'insieme degli archi (usualmente un array ma può anche essere una lista, con ovviamente i vantaggi e gli svantaggi di quella rappresentazione), nel quale, ad ogni elemento è associata una struttura dati che rappresenta gli adiacenti di quel vertice (di solito è una lista ma è anche possibile usare un albero, nel caso in cui si vogliano sfruttare altre utili caratteristiche). Formalmente una lista di adiacenza $L(v)$ con $v \in V$ è una lista di w , tale che $(v, w) \in E$. Ad esempio:



Tale rappresentazione ha una occupazione in memoria di $\Theta(|V| + |E|)$ che è proprio ciò che desideravamo; ovviamente ciò comporta un aumento del tempo di esecuzione per alcune operazioni ma una riduzione per altre. Per aggiungere un arco (i, j) viene impiegato un tempo lineare sul numero di adiacenti del vertice v_i , però l'aggiunta un vertice avviene in tempo lineare su $|V|$ e non più su $|V|^2$.

Dunque, 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 (ovviamente casi particolari possono avere rappresentazioni più efficienti che sfruttino le proprietà di quel particolare grafo). Nei nostri algoritmi assumeremo sempre e solo una delle due

rappresentazioni appena descritte, tra le quali la più efficiente è la seconda, poiché il nostro principale interesse sarà 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).

Visita in ampiezza

Il problema della 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, ma ciò rende necessario gestire gli eventuali cicli (altrimenti l'algoritmo non terminerebbe) e l'assicurarsi di visitare ogni vertice una sola volta (altrimenti avremmo un tempo di esecuzione esponenziale, pari al numero di percorsi in un grafo, ovvero $n!$).

N.B.: qualsiasi algoritmo che visiti tutti i percorsi semplici di un grafo avrà, nel caso peggiore, costo esponenziale, poiché il fattoriale è una funzione compresa 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 grafo 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 tre stati ad ogni vertice:

- Vertici non ancora incontrati (o non visti), codificati dal colore bianco.
- Vertici visti ma non ancora visitati, codificati con il grigio.
- Vertici già visitati, codificati con il colore nero.

Questo tipo di codifica è fondamentale per garantire che venga effettuata al più una visita per ogni vertice.

Algoritmo BFS

Scriviamo un algoritmo di visita in ampiezza per grafi, ma prima definiamo una funzione che inizializzi il colore di ogni vertice a bianco. Si noti che per codificare i tre colori bastano quattro bit (esempio: 00 = b , 01 = g , 10 = n): ci 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).

```
Init(G)
  FOR EACH v IN V DO
    Color[v] = b
```

La visita in ampiezza visiterà tutti i vertici raggiungibili da una sorgente s (l'algoritmo $\text{Adj}(v)$ restituisce la lista di adiacenti di v):

```
1  VisitaAmpiezza(G, s)
2    Init(G)
3    Q = Accoda(Q, s)
4    color[s] = g
5    WHILE Q != NIL DO
6      v = Testa(Q)
7      FOR EACH u IN Adj(v) DO
8        IF Color[u] = b THEN
9          Color[u] = g
10         Q = Accoda(Q, u)
11      Visita(v)
12      Q = Decoda(Q)
13      Color[v] = n
```

Aggiungo tutti gli adiacenti non ancora incontrati nella coda (mettendo il colore a grigio prima mi assicuro che non accoderò mai lo stesso elemento più di una volta).

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; dunque, un arco o non

viene mai attraversato, oppure viene attraversato una sola volta, ossia quando sto raggiungendo un nuovo vertice. In particolare, è grazie all'array di colori che il tempo richiesto è lineare: $T_{\text{BFS}}(|V| + |E|) = O(|V| + |E|)$.

Calcolo distanze e percorsi minimi

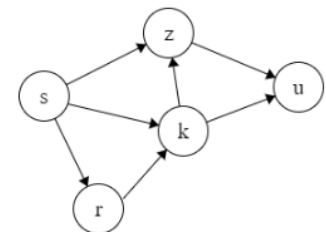
La BFS visita il grafo in ordine crescente di distanza da s , quindi sostanzialmente ogni vertice viene raggiunto tramite il percorso più corto (ovvero quello che determina la distanza); dunque, l'idea sottostante è che questo algoritmo segua i percorsi minimi.

Ne deriva che 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. Vediamo, a questo punto, una versione di BFS che calcoli anche le distanze, ma prima ne discutiamo l'idea.

Dato un qualsiasi grafo $G = (V, E)$ e due vertici v e u , chiameremo $\delta(v, u)$ la distanza di u da v ; quindi, la funzione δ restituisce la lunghezza del percorso più corto che parte da v ed arriva ad 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 esempio, per il grafo di destra abbiamo szu e sku 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. Riprendendo il nostro esempio: inizializzeremo una coda con $\{s\}$, poi supponiamo di scoprire da s i vertici r, k, z in quest'ordine; dunque, la coda sarà $\{r, k, z\}$ (s verrà rimosso dalla coda una volta visitato) ed a ciascuno di questi verrà associato il vertice s come colui che li ha scoperti. Poi si passa ad r (la 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; quindi, termineremo l'algoritmo con le seguenti informazioni: $s \rightarrow r$, $s \rightarrow k$, $s \rightarrow z$, $k \rightarrow u$ (\rightarrow = scopre); praticamente, 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 attraversati da BFS nello scoprire nuovi vertici per ottenere i percorsi minimi (e quindi le distanze) del grafo dalla sorgente. In sostanza, per risolvere il nostro problema basta raffinare l'algoritmo della BFS:

- $d[v] = \delta(s, u)$ dove la funzione $d: V \rightarrow \mathbb{N} \cup \{\infty\}$. Il valore ∞ è usato per convenzione per definire la distanza tra due vertici non raggiungibili.
- $p[u] = v$: vettore p che associa al vertice u il vertice v che ha permesso la sua scoperta; quindi definiamo la funzione $p: V \rightarrow \mathbb{N} \cup \{\text{NIL}\}$, con NIL che sarà il valore associato alla sorgente (la sorgente non viene scoperta da nessun vertice) o ai 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

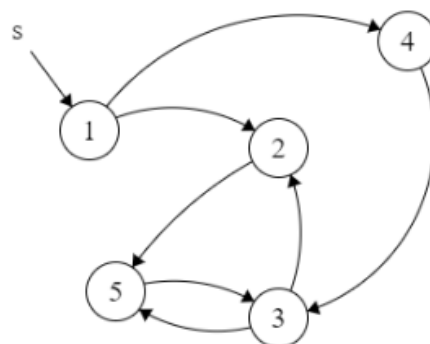
```

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'

```



Correttezza della BFS

Facciamo una prima verifica di correttezza con un esempio, ossia descriviamo il grafo a destra dell'algoritmo attraverso due 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, da 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 |

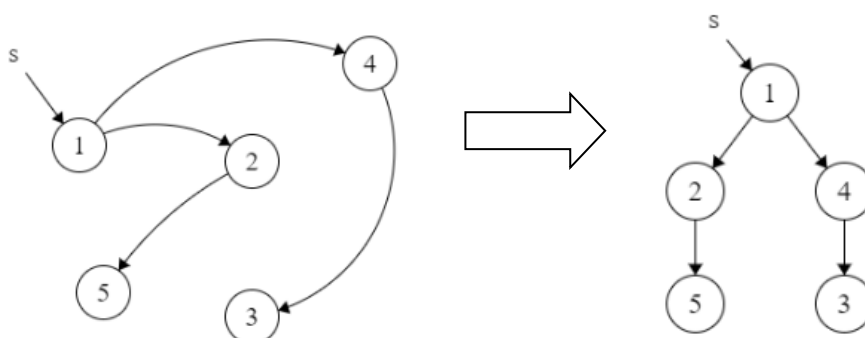
| | 0 | 1 | 2 | 3 |
|------|-----|-----|-----|-----|
| p[1] | NIL | NIL | NIL | NIL |
| p[2] | NIL | 1 | 1 | 1 |
| p[3] | NIL | NIL | NIL | 4 |
| p[4] | NIL | 1 | 1 | 1 |
| p[5] | NIL | NIL | 2 | 2 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|-------|-------|-------|-----|-----|
| F | (1) | (2,4) | (4,5) | (5,3) | (3) | () |

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), (2,5), (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 che ha come radice la sorgente s , e le distanze come percorsi. Esempio, il percorso che va dalla radice alla foglia 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 visita **tutti** i nodi raggiungibili dalla sorgente e che le distanze siano aggiornate correttamente (ovvero che la BFS scopra i vertici entro il “turno” della distanza del vertice).

Formalmente, dobbiamo dimostrare che al termine di $\text{BFS}(G, s)$ 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) $(u, v) \in E \Rightarrow \delta(s, v) \leq \delta(s, u) + 1$

Questa proprietà delle distanze è abbastanza ovvia, infatti, possiamo avere i seguenti casi:

- a) 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.
- b) 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 solo essere uguale o più corto; quindi, la proprietà $\delta(s, v) \leq \delta(s, u) + 1$ è verificata anche in questo caso.

2) In ogni momento, vale che $\forall v \in N, d[v] \geq \delta(s, v)$; ovvero, 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. Quindi basta verificare la suddetta proprietà solo nei momenti in cui il vertice viene inserito nella coda (gli altri vertici avranno la stessa stima):

- a) 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).
- b) 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: $\forall v \in V, d[v] \geq \delta(s, v)$. 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 proprietà 1 (l’arco v_1, v_{r+1} esiste altrimenti non avremmo potuto scoprire v_{r+1}), risulta $\delta(s, v_1) + 1 \geq \delta(s, v_{r+1}) \geq d[v_{r+1}]$ come volevasi dimostrare.

3) Se $F = (v_1, v_2, \dots, v_k)$, allora:

- a) $d[v_i] \leq d[v_{i+1}] \quad \forall 1 \leq i < k$ (i vertici in coda o hanno la stessa stima o sono più grandi)
- b) $d[v_k] \leq d[v_1] + 1$ (gli estremi hanno una stima che differisce al più di uno)

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 vero (a è una proprietà universale su un insieme vuoto, mentre per b abbiamo $0 \leq 1$). Nel 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 $F = (v_1, \dots, v_k, v_{k+1})$ 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 $d[v_{k+1}] = d[v_1] + 1$ e quindi la proprietà b è verificata. Per la proprietà a 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 durante il decodamento non cambia nessuna stima la proprietà a è banalmente verificata. Per la proprietà b dobbiamo invece verificare che $d[v_k] \leq d[v_2] + 1$: prima del decodamento si ha $d[v_1] \leq d[v_2] \Rightarrow d[v_1] + 1 \leq d[v_2] + 1$ e $d[v_k] \leq d[v_1] + 1$; dunque, per transitività risulta $d[v_k] \leq d[v_1] + 1 \leq d[v_2] + 1$, come volevasi dimostrare.

Possiamo ora dimostrare, finalmente, le tre 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: $V_0 = \{1\}$, $V_1 = \{2,4\}$, $V_2 = \{5,3\}$ e $V_\infty = \emptyset$). L'idea è quella di usare l'induzione sulle classi di equivalenza precedentemente descritte ($i \in \mathbb{N}$), poiché attraverso quest'ultime 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, ed 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 due 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 due. 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 $l \in V_r$, chiedendoci se tale l potrà mai scoprire v (il nostro scopo è assicurarci che $r = z - 1$):

- Se $r < z - 1$ allora l non potrà mai scoprire v poiché dovrebbe esistere un percorso minimo da s a l che attraverso un arco raggiungesse v , ma è assurdo perché $r + 1 < z - 1 + 1 \rightarrow r < z$.
- Il fatto che $r > z - 1$ viene escluso dalla proprietà di monocità della coda (proprietà 3); infatti, se l scopre v allora significa che l è in cima alla coda, ma ciò è assurdo perché i vertici quando vengono messi in coda la loro stima non potrà più decrescere (o restano uguali o crescono); quindi, prima che un vertice della classe V_i possa entrare in coda, tutti i vertici di V_{i-1} devono già essere entrati in coda (altrimenti non entreranno mai più in coda). Quindi prima che $l \in V_r$ possa entrare in coda, tutti i vertici V_{z-1} devono già essere in 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 poiché la BFS assegnerà a $p[v] = u$ e il percorso minimo sarà proprio il percorso minimo da s a $p[v]$ concatenato con l'arco $(p[v], v)$.

A questo punto rimangono solo i vertici non raggiungibili, ma di questi dovremmo solo garantire che le distanze siano corrette; tuttavia, la loro distanza sarà quella iniziale che è ∞ , esattamente il valore corretto.

Algoritmo del percorso minimo

Abbiamo dimostrato che l'algoritmo della BFS precedentemente descritto calcola le distanze corrette e i percorsi minimi sono ottenibili con il metodo 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 e 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; dunque, il costo dell'algoritmo principale PercorsoMinimo sarà semplicemente il costo della BFS ($O(|V| + |E|)$) più l'eventuale costo di StampaPercorsoMinimo: $T_{\text{PercorsoMinimo}}(|V| + |E|) = O(|V| + |E|) + O(|V|) = O(|V| + |E|)$.

Visita in profondità

La visita in profondità è definibile in maniera naturale grazie alla definizione di percorso; esso ha gli stessi problemi di terminazione e efficienza che abbiamo riscontrato nella visita in ampiezza e si risolvono, come quest'ultima, grazie alla colorazione.

Se per una certa tipologia di problemi è indifferente il tipo di visita utilizzato, ci sono dei problemi come il calcolo dei percorsi minimi o la scoperta di cicli in un grafo che possono essere risolti solo con un determinato tipo di visita. Infatti, per il primo l'unica soluzione corretta è la BFS, mentre per il secondo si può procedere solo tramite DFS.

L'idea di visita in profondità è quella di avere un algoritmo che in maniera ricorsiva si chiama sugli adiacenti; infatti, 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, dunque anche in questo tipo di visita bisognerà evitare i cicli per far sì che l'algoritmo termini; inoltre, allo stesso tempo, bisogna visitare ogni vertice una ed una sola volta così da rendere l'algoritmo efficiente (altrimenti nel caso peggiore avremmo tempo esponenziale).

Il tipo di applicazioni della visita in profondità in generale necessita la visita di tutto il grafo: definiremo dunque un algoritmo che non si limita ad una singola sorgente (come abbiamo visto per la visita in ampiezza, dove il tipo di applicazioni per quest'ultima è "sensata" solo se applicata su una singola sorgente).

Prima di passare all'algoritmo di DFS (Depth First Search) diamo una prima implementazione di algoritmo di visita in post order e pre order. Poi definiremo la DFS in post order (quella più utilizzata).

```

VisitaPostOrder(G, s)
  Color[s] = 'g'
  FOR EACH v IN Adj[s] DO
    IF Color[v] = 'b' THEN
      VisitaPostOrder(G, v)
  /*qui ho visitato tutti i vertici*/
  /*raggiungibili dalla sorgente s*/
  Visita(s)
  Color[s] = 'n'

```

```

VisitaPreOrder(G, s)
  Color[s] = 'g'
  Visita(s)
  Color[s] = 'n'
  FOR EACH v IN Adj[s] DO
    IF Color[v] = 'b' THEN
      VisitaPreOrder(G, v)

```

Algoritmo DFS

Come per la BFS, anche qui possiamo associare ad ogni vertice, tramite un array, delle informazioni aggiuntive:

- $p[v]$: associa al vertice v il suo predecessore. Quindi con $p[v] = s$ indico 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 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    /*qui faccio quello che devo fare con s*/
10   Color[s] = 'n'
11   tempo = tempo + 1
12   f[s] = tempo

```

Si noti che per come abbiamo gestito la variabile tempo non esisterà mai un valore uguale per i vettori d e f (né all'interno dello stesso array, né tra celle dei due array). Il precedente algoritmo verrà usato dal seguente, così 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 oppure sono neri; ciò 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).

Terminazione e complessità

Questo algoritmo termina per lo stesso motivo per cui la BFS terminava; infatti, il fatto che non possiamo visitare due 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 viene colorato in grigio per la prima volta e in seguito un secondo istante in cui rivisiteremo quel vertice. Ma ciò vuol dire che ci saranno due momenti in cui chiamo la DFS_Visit sullo stesso vertice, il che è assurdo poiché una volta che il vertice diventa grigio non c'è modo per quel vertice di ritornare bianco (al massimo diventa

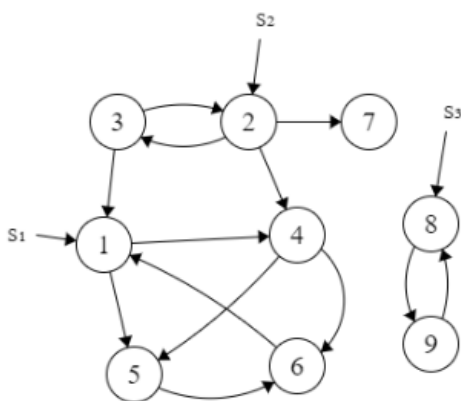
nero); dunque, quando il vertice viene incontrato per la seconda volta esso è già grigio. La DFS_Visit può essere chiamata in DFS_Visit oppure in DFS, ma in entrambi i casi ho come guardia un test sul colore del vertice (che sarà grigio dopo la prima visita), quindi non potrà mai essere rivisitato.

Ciò significa che il numero di chiamate ricorsive è pari al numero di vertici del grafo. L'albero di ricorrenza di DFS_Visit, infatti, al più genera una chiamata per ogni vertice, mentre 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).

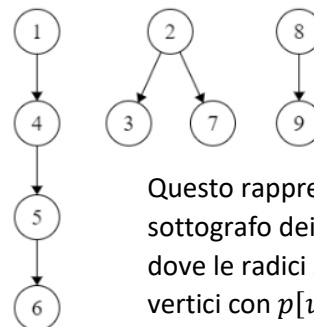
Poiché scopro tutti i vertici del grafo e ogni vertice lo visito una sola volta, allora anche un arco viene attraversato una sola volta. Quindi, anche se non posso dare un tempo di esecuzione ad un singolo albero di ricorrenza, posso dire che l'algoritmo di DFS ha tempo lineare sul numero di vertici e di archi:

$T_{DFS}(|V| + |E|) = \Theta(|V| + |E|)$ (è sicuramente un Theta poiché non lascio nessun vertice bianco).

Per una migliore comprensione forniamo un esempio degli alberi di ricorrenza della DFS per un grafo:



Il grafo a sinistra genererà la seguente foresta di alberi di ricorrenza:



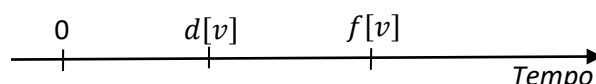
Questo rappresenta anche il sottografo dei predecessori dove le radici sono gli unici vertici con $p[v] = \text{NIL}$

Si noti che mentre la visita in ampiezza 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 ma vedremo che sarà utile per verificare se un grafo è ciclico o aciclico.

Teorema della struttura a parentesi

Il teorema che annunceremo a breve è conseguenza diretta delle proprietà della DFS che descriveremo in seguito. Tali proprietà anche se astratte hanno un impatto diretto sullo studio della correttezza di algoritmi che usano la DFS.

Prima di passare al teorema descriviamo il seguente asse temporale:



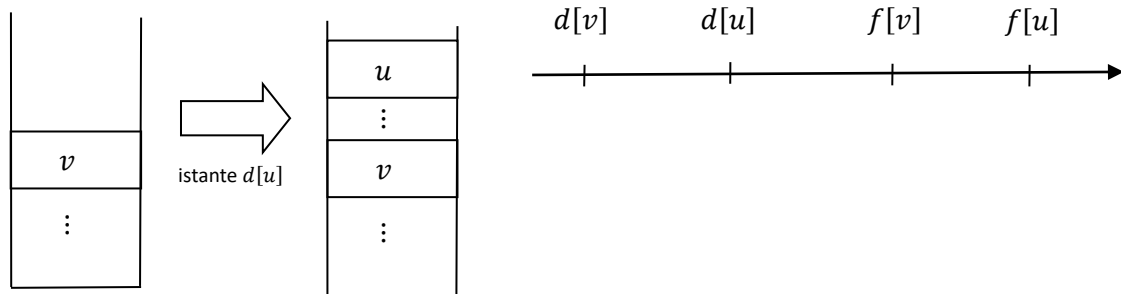
In quest'asse ad ogni vertice è associato un unico intervallo $[d[v], f[v]]$ che non potrà mai essere modificato dopo l'inizializzazione (per come è definito l'algoritmo). Tale intervallo viene definito come **intervallo di scoperta** e rappresenta la durata in cui il vertice è stato attivo (ossia, presente nello stack).

Teorema: $\forall u, v \in V$ con $u \neq v$, vale **una ed una sola** delle seguenti proprietà:

- $d[v] < d[u] < f[u] < f[v]$
- $d[u] < d[v] < f[v] < f[u]$
- $d[v] < f[v] < d[u] < f[u]$
- $d[u] < f[u] < d[v] < f[v]$

Praticamente, se i due intervalli non sono disgiunti, allora uno è dentro l'altro (proprio come le parentesi in matematica, infatti $[()]$ è lecito, ma $[(])$ no). Dimosteremo questo teorema concentrandoci sulle parentesizzazioni non possibili: supponiamo per assurdo che sia possibile $d[v] < d[u] < f[v] < f[u]$.

Consideriamo l'istante di tempo $d[v]$: in questo istante viene attivata la chiamata ricorsiva su v ($\text{DFS_Visit}(G, v)$). Ciò vuol dire che lo stack del record di attivazione alla chiamata ricorsiva su u cambia nel seguente modo:



Ma affinché possa eseguire l'istante $f[v]$ deve essere la chiamata di v ad avere il controllo; significa allora che in cima allo stack deve esserci v , e ciò può accadere solo in due modi:

- C'è una nuova chiamata $\text{DFS_Visit}(G, v)$. Questo però non è possibile per l'algoritmo di DFS descritto (non posso scoprire lo stesso vertice più di una volta)
- La chiamata ricorsiva su v rappresentata nello stack riprende il controllo: devono terminare tutte le chiamate in cima allo stack fino a v .

Dunque, l'unica possibilità è che le chiamate in cima allo stack terminino, con la conseguenza di dover prima terminare la chiamata ad u e quindi avere $d[v] < d[u] < f[u]$: ciò va contro la nostra supposizione.

La dimostrazione che anche la parentesizzazione $d[u] < d[v] < f[u] < f[v]$ non è possibile è analoga. Abbiamo così verificato che le uniche possibilità sono proprio le assunzioni descritte nel teorema.

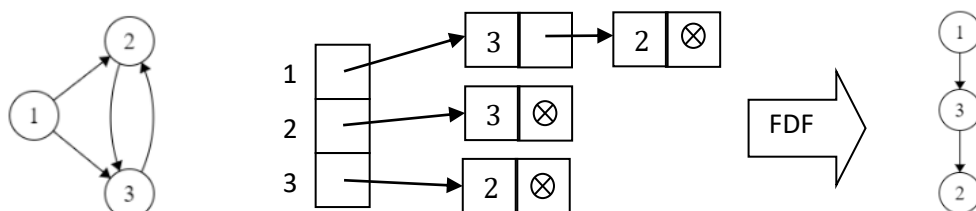
Una diretta conseguenza di tale teorema è che se un intervallo è dentro un altro, allora c'è una discendenza diretta nel sottografo dei predecessori. Si noti che se non c'è nessuna discendenza diretta non vuol dire che non ci siano archi tra i vertici, ma significa solo che la DFS non li ha seguiti.

Foresta DF

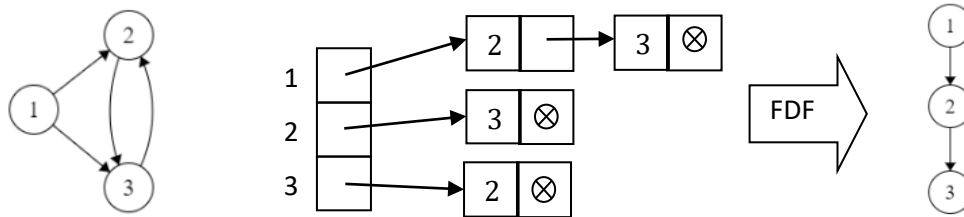
Consideriamo il grafo dei predecessori $G^P = (V, E^P)$ dove $E^P = \{(p[v], v) | v \in V \wedge p[v] \neq \text{NIL}\}$ (prendo tutti i vertici e solo gli archi scoperti dalla DFS). Allora G^P è una foresta poiché le sorgenti non hanno archi entranti (fungono da radice dell'albero) e ogni nodo può essere scoperto da un unico nodo (quindi ho esattamente un albero); G^P viene anche definito come Foresta Depth First, in breve 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 essa dipende da come viene rappresentato concretamente il grafo.

Forniamone un banale esempio prendendo un semplice grafo con la sua lista di adiacenza:



Se cambiamo la lista di adiacenza (il grafo resta lo stesso) cambierà anche la foresta:



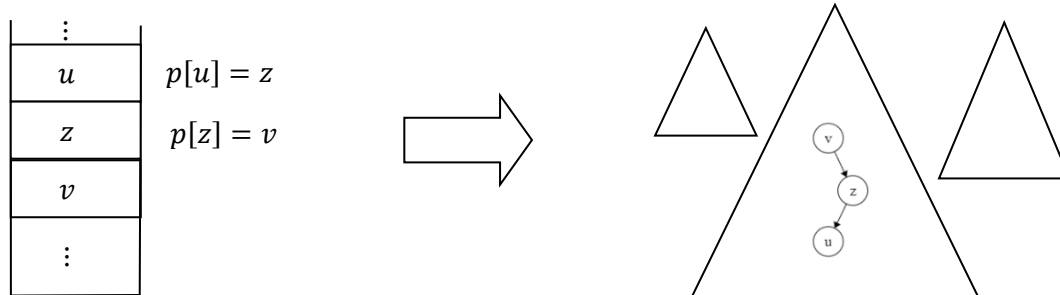
Quindi, data una rappresentazione concreta del grafo posso dedurre la sua foresta DF, ma se mi è dato solo un grafo astratto non mi è in alcun modo possibile prevedere quali percorsi la DFS esplorerà (a meno che non lo decido arbitrariamente) e quindi non posso conoscerne la FDF. L'unica certezza è che 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 $\Leftrightarrow d[v] < d[u] < f[u] < f[v]$

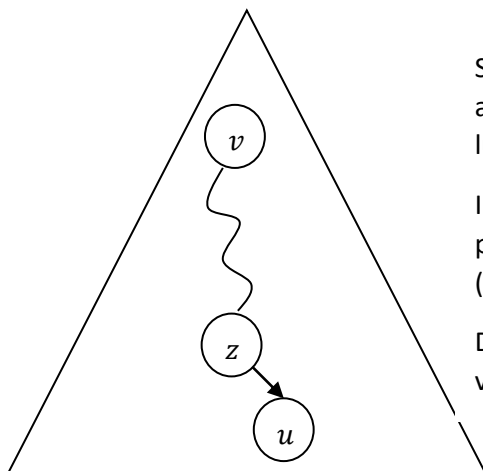
Questo significa che per il seguente stack di attivazione avremmo una specifica FDF:



(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 uno (essendo $v \neq u$); sia proprio $|\pi| = 1$ il nostro caso base, che è banalmente vero: per come 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 : $f[v] > f[u]$ (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 avremmo $d[v] < d[z] < f[z] < f[v]$, poiché 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.

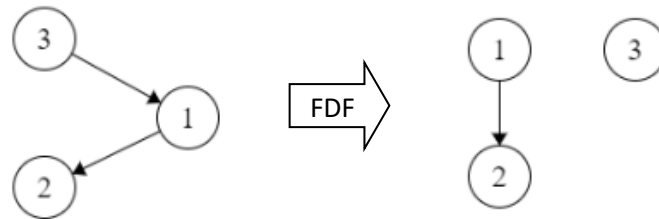
Teorema del percorso bianco

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 \Leftrightarrow al tempo $d[v]$, $\exists \pi$ in G da v a u fatto di vertici bianchi.

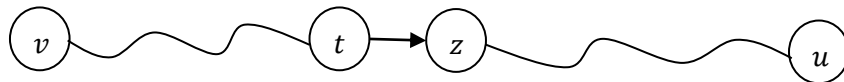
N.B.: si potrebbe pensare che u è raggiungibile da $v \Leftrightarrow$ esiste un percorso nella FDF. In realtà è vera solo l'implicazione \Rightarrow , infatti se non esiste un percorso in FDF non è detto che non esista un percorso nel grafo.

Di seguito un banale esempio:



Dimostriamo l'implicazione \Rightarrow 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 $d[v]$ esista un percorso π di vertici bianchi fino ad u .

Dimostriamo l'implicazione \Leftarrow : 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 assunzione è più forte di " u è discendente di v in FDF", ragion per cui sarà evidentemente vera anche quest'ultima). Supponiamo per assurdo che esista un vertice nel percorso π che non sia discendente di v , e supponiamo che il primo di questi sia z (con $z \neq v$ altrimenti sarebbe discendente). Dunque, esiste un predecessore di z , che chiameremo t , che discenderà da v (altrimenti z non sarebbe il primo vertice non discendente da v).



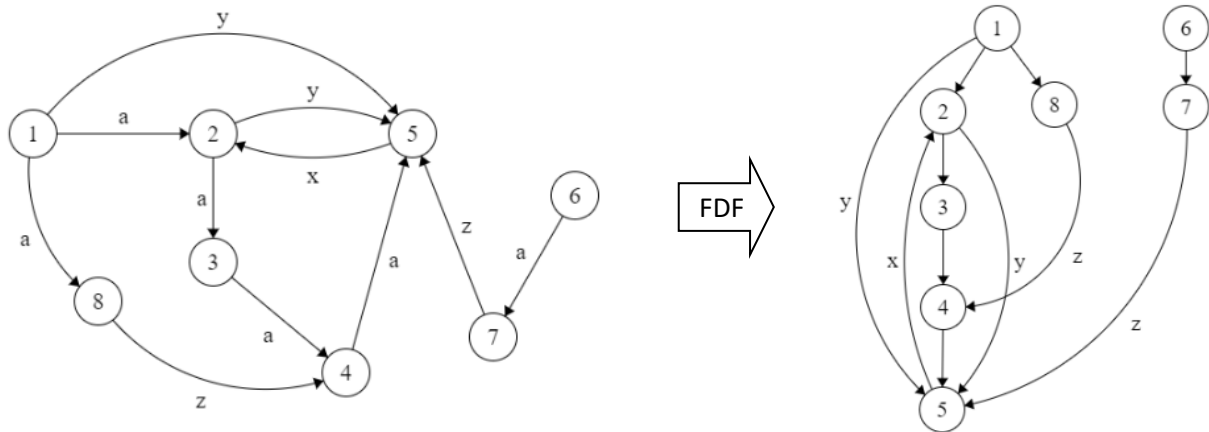
Abbiamo due casi possibili:

- $t = v$: siamo nella situazione $\boxed{v} \rightarrow \boxed{z}$ con z bianco e adiacente a v . Ma allora la visita DFS, prima di terminare v , guarderà tutti i suoi adiacenti. Se z è nero (sappiamo solo che all'istante $d[v]$ è bianco, quindi potrebbe essere stato colorato nel mentre) le uniche possibilità sono $d[v] < f[v] < d[z]$ e $d[v] < d[z] < f[z] < f[v]$, ma la prima contraddice il fatto che v e z siano adiacenti. Dunque, l'unica possibile è la seconda con la conseguenza che z è discendente di v .
- $t \neq v$: 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 ci garantisce che dalla prima sorgente saranno percorsi tutti i vertici raggiungibili da tale sorgente, mentre sulle sorgenti successive ciò non è detto (potremmo imbatterci in vertici già scoperti dalle sorgenti precedenti).

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 due nodi che non hanno nessuna relazione tra loro (sono in 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 suo 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** (tra due sottoalberi distinti)

Questi quattro tipi di archi sono riconoscibili dalla DFS, infatti, 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 da un nodo ad un altro, percorrendo un arco (v, u) :

- Se *u* è bianco allora 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 (sono in un percorso che da *u* mi ha riportato ad *u* attraverso *v*)
- Se *u* è nero posso avere due casi:
 - $d[v] < d[u] < f[u] < f[v]$: 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))
 - $d[u] < f[u] < d[v] < f[v]$: (v, u) è un arco di attraversamento

Si noti che possiamo distinguere i due 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[v] = 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*/

```



```

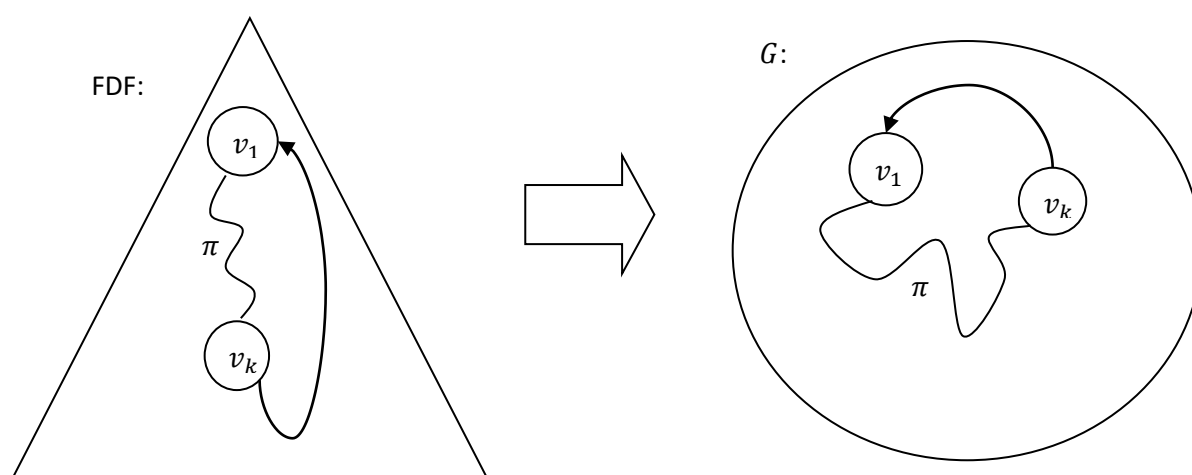
11      ELSE
12          IF d[v] < f[u] THEN
13              /*(v,u) è arco in avanti*/
14          ELSE
15              /*(v,u) è arco di attraversamento*/
16      Color[v] = 'n'
17      f[v] = tempo = tempo + 1

```

La distinzione di tali archi è molto importante in alcuni contesti, come ad esempio la verifica della presenza di cicli in un grafo. Infatti, 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 durante la DFS.

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, ma il nostro obiettivo è dimostrare che se esistono cicli nel grafo l'algoritmo se ne accorge (ciò significa che se non esistono cicli nel grafo, allora l'algoritmo sarà in grado di dire che tale grafo è aciclico).

Dimostriamo che se il grafo ha 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 , e qui dovrà visitare tutti gli adiacenti, compreso v_1 che sarà grigio poiché dovrà terminare successivamente rispetto a 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 ed 1 (true) se il grafo è aciclico:

```

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.

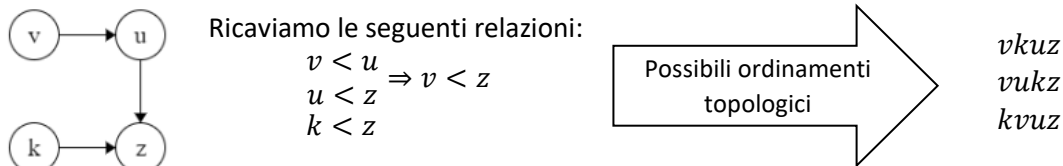
N.B.: con la BFS non è possibile tale verifica poiché il colore grigio, in quel caso, non implica aciclicità.

Ordinamento Topologico

Per ordinamento topologico intendiamo 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 ($v < u$)

Esempio:



Questi vari ordinamenti sono determinati dal fatto che non esiste nessuna relazione tra k ed i vertici u, v : l'importante è che k venga prima di z e che ci sia l'ordinamento relativo vuz (qualsiasi altra permutazione al di fuori delle tre 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 (grafo senza archi), ma è anche possibile avere grafi con un unico ordinamento topologico (grafo in sequenza: $\boxed{v} \rightarrow \boxed{u} \rightarrow \boxed{x} \rightarrow \boxed{z}$, unico ordinamento topologico: $vuxz$).

Esistono grafi che non hanno nessun ordinamento topologico:

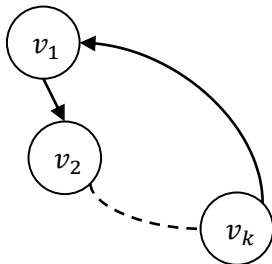


Infatti, vu non rispetta l'arco (u, v) , e simmetricamente uv non rispetta l'arco (v, u)

Abbiamo le seguenti proprietà:

- G è aciclico $\Leftrightarrow \exists$ ordinamento topologico
- G è ciclico $\Leftrightarrow \nexists$ ordinamento topologico (questa è praticamente la negazione della precedente)

Dimostriamo che con un grafo ciclico non può esistere nessun ordinamento topologico:



Ipotizziamo di avere il ciclo qui a sinistra, e 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 due punti distinti di una sequenza.

Di conseguenza, non può esistere alcun ordinamento topologico per un grafo ciclico.

Mostriamo ora che per ogni grafo aciclico esiste almeno un ordinamento topologico: sia $G = (V, E)$ orientato e aciclico, dimostriamo che esiste una permutazione π con π ordinamento topologico di G .

Procederemo con una dimostrazione costruttiva, ovvero, mostriamo un modo per costruire tale permutazione (uno dei modi per dimostrare che qualcosa esiste è far vedere come si costruisce). Notare che tale dimostrazione conterrà in sé anche l'idea di come implementare l'algoritmo.

Per suddetta dimostrazione sfrutteremo le seguenti proprietà:

- 1) $G = (V, E)$ è aciclico $\Rightarrow \exists v \in V$ che ha zero archi entranti

N.B.: esistono anche grafi ciclici che rispettano queste proprietà, ad esempio: $\boxed{1} \rightarrow \boxed{2} \rightleftharpoons \boxed{3}$.

Questa proprietà è universale per tutti i grafi, quindi possiamo procedere per assurdo (poiché non abbiamo una definizione induttiva). Supponiamo che $\forall v \in V$ ci sia almeno un arco entrante; sia $v_1 \in V$: poiché ha almeno un arco entrante è lecito supporre che ci sia una sorgente $v_2 \neq v_1$ che arrivi a v_1 (nel caso $v_2 = v_1$ siamo in presenza 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, poiché lavoriamo su insiemi finiti, possiamo iterare questo procedimento fino al vertice v_n con $n = |V|$, ma anche quest'ultimo 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 $G \Rightarrow G'$ 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' per definizione. Di conseguenza conterrà anche il ciclo formato dai vertici v_1, \dots, v_k e ciò è assurdo poiché G è aciclico.

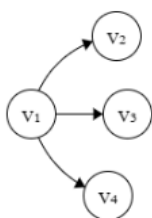
Possiamo ora procedere con la nostra dimostrazione: all'inizio della sequenza π ci dovrà essere un vertice v_1 che non ha archi entranti, altrimenti esisterebbe un vertice v con l'arco (v, v_1) che dovrebbe essere messo prima di v_1 in π . Tale vertice esiste per la proprietà 1, quindi per ora abbiamo $\pi = v_1$.

A questo punto, siccome v_1 è primo, tutti i suoi archi andranno nella direzione "giusta" (e i vertici adiacenti a v_1 saranno disposti dopo v_1 nella sequenza); quindi, posso costruire un nuovo grafo $G_1 = (V \setminus \{v_1\}, E \setminus \{(v_1, v) \in E \mid v \in V\})$ (praticamente vado a togliere il vertice v_1 e tutti gli archi che partono da esso). Tale G_1 sarà aciclico per la proprietà 2; dunque, 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 .

Algoritmo del grado 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: $v_1 = 0, v_2 = v_3 = v_4 = 1$), e un vertice lo si "cancella" semplicemente decrementando di uno il grado entrante dei suoi adiacenti ($v_2 = v_3 = v_4 = 0$: è come se avessimo creato un sottografo $G_1 = (V \setminus \{v_1\}, E \setminus \{(v_1, v) \in E \mid v \in V\})$). In questo modo possiamo evitare di dover generare i sottografi (risparmiano anche memoria).

- 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 dati (una coda fa proprio al caso nostro) tutti i vertici incontrati che non hanno archi entranti.

Senza queste ottimizzazioni avrei dovuto crearmi, ogni volta cancellato un vertice, un nuovo sottografo nel quale cercare un vertice con grado entrante 0 (praticamente implementare così come descritto il processo usato nella dimostrazione precedente); un algoritmo così fatto avrebbe tempo $\Omega(|V|^2 + |V||E|)$ (che potrebbe essere addirittura cubico nel caso in cui $E = V \times V$).

```

1  OrdinamentoTopologico(G)
2      Q = NIL
3      /*associamo ad ogni vertice un grado entrante*/
4      GradoEntrante(G, GE) /*GE è array*/
5      /*inizializziamo la coda con i vertici*/
6      /*che hanno grado entrante pari a 0*/
7      FOR EACH v IN V DO
8          IF GE[v] = 0 THEN
9              Q = Accoda(Q, v)
  
```

```

10     WHILE Q != NIL DO
11         v = Testa(G)
12         /*v sarà il prossimo vertice*/
13         /*nell'ordinamento topologico*/
14         print(v)
15         /*decodo il grado entrante negli adiacenti*/
16         FOR EACH u IN Adj[v] DO
17             GE[u] = GE[u] - 1
18             /*controllo se aggiungerlo in coda*/
19             IF GE[u] = 0 THEN
20                 Q = Accoda(Q, v)
21         Q = Decoda(Q)

```

Il while viene eseguito una sola volta per vertice, mentre il for al suo interno viene effettuato una sola volta per ogni vertice adiacente a v ; dunque, tutte le esecuzioni del for sono, come al solito, 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 vedere che la complessità di questo algoritmo sia lineare sulla dimensione del grafo: $\Theta(|V| + |E|)$. Dunque, possiamo concludere che il costo complessivo di OrdinamentoTopologico è proprio $\Theta(|V| + |E|)$.

Algoritmo con DFS

Esiste anche una soluzione che sfrutta la DFS: so che appena un vertice non ha più vincoli da soddisfare posso metterlo arbitrariamente alla fine. Praticamente utilizziamo un approccio simmetrico al precedente ove, grazie alla DFS, costruiamo l'ordinamento topologico dalla fine verso l'inizio, in quanto intuitivamente esiste sicuramente un vertice con grado entrante zero (non lo dimostreremo ma la verifica è praticamente speculare alla proprietà 1), quindi 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 via (si noti che quando annerisco un vertice esso può essere messo prima di tutti gli altri già anneriti nell'ordinamento).

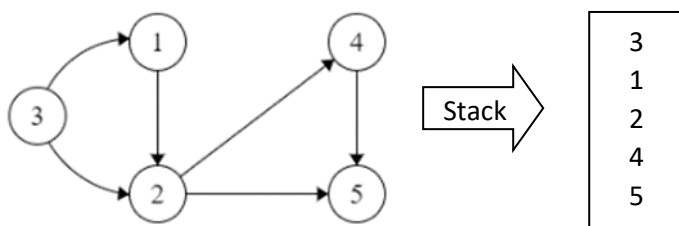
```

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, OT)
7      RETURN OT /*letto dalla testa darà il nostro ordinamento*/

1  OT_DFS_Visit(G, v, OT)
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, OT)
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:

$$\pi = 3 \ 1 \ 2 \ 4 \ 5$$

Verifichiamo che tale algoritmo sia corretto dimostrando che, dato $G = (V, E)$ al termine di $OT_DFS(G)$, 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 basso: "stare prima" \equiv "stare sopra").

Si noti che v sta sopra in $OT \Leftrightarrow f[v] > f[u]$; infatti, mettiamo in cima allo stack quando la DFS annerisce il vertice e quando 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 $OT_DFS(G)$.

Sappiamo che ogni arco verrà attraversato dalla DFS, quindi prendendo un arbitrario arco $(v, u) \in E$, quando la DFS lo attraversa posso avere tre diversi casi (v è sempre attualmente attivo, quindi grigio):

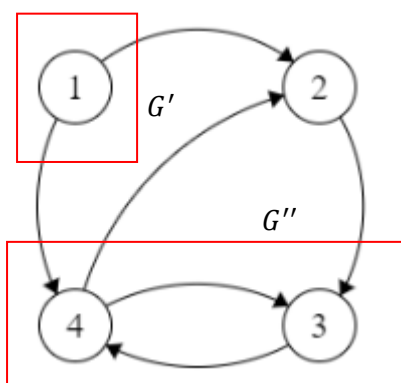
- 1) u è bianco: $d[v] < d[u] < f[u] < f[v]$ per il teorema della struttura a parentesi. Questo caso è quindi verificato essendo $f[v] > f[u]$
- 2) u è nero: vuol dire che u è già terminato prima di v , quindi anche in questo caso $f[v] > f[u]$, più precisamente $d[u] < f[u] < d[v] < f[v]$
- 3) u è grigio: questo significa che nel grafo c'è un ciclo; questo caso, quindi, non può verificarsi in un grafo aciclico (l'unico caso problematico dove $d[u] < d[v] < f[v] < f[u]$ non può avverarsi).

Componenti Fortemente Connesse

Il concetto di componente fortemente connessa (in breve CFC) è definibile dal concetto di grafo fortemente connesso.

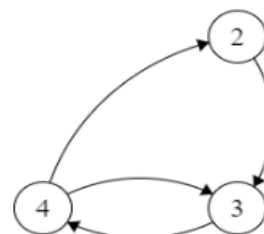
Un **grafo** $G = (V, E)$ è **fortemente connesso** $\Leftrightarrow \forall v, u \in V, v$ raggiunge u e u raggiunge v (un grafo è fortemente connesso se tutte le coppie di vertici sono reciprocamente raggiungibili).

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 più fortemente connesso). Prendiamo ad esempio il seguente grafo G (si noti che non è 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:

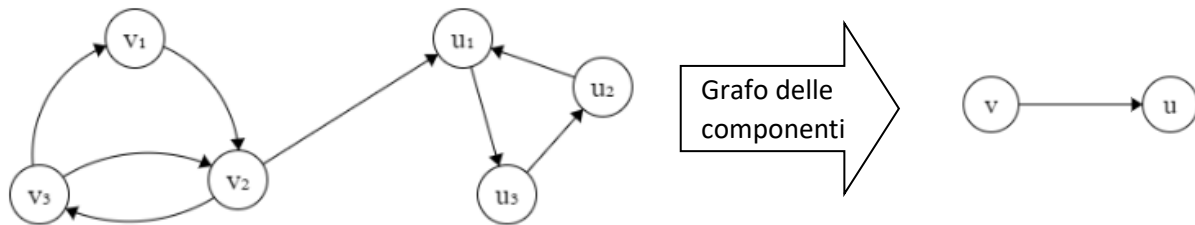


Ogni grafo contiene almeno una componente fortemente connessa; in particolare:

- un grafo aciclico ha tante componenti fortemente connesse quanti sono i vertici; infatti, poiché nessuna coppia di vertici è mutuamente 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 ($u \sim v \Rightarrow v \sim u$) e transitiva ($u \sim v \wedge v \sim t \Rightarrow u \sim t$).

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. In altri termini, 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 (ma occupa meno spazio e riduce la complessità di diverse operazioni!). Esempio:



N.B.: Il grafo delle componenti è sempre aciclico: se ad esempio ci fosse un ciclo tra due CFC v e u , queste collasserebbero in una singola CFC, poiché ogni vertice di v sarebbe mutuamente raggiungibile con ogni vertice di u .

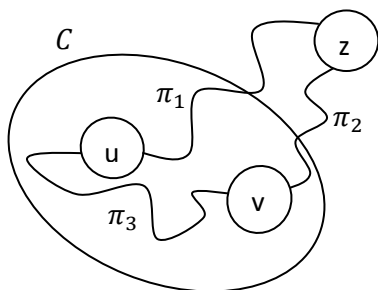
Dunque, dal concetto di CFC è possibile ridurre la complessità del grafo tramite il grafo delle componenti.

Proprietà delle 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 mai uscire da C .

(praticamente ogni percorso da u a v e viceversa è necessariamente contenuto nella componente)

La validità di tale proprietà è abbastanza naturale, poiché supposto un percorso da u a v che passi attraverso $z \notin C$ si trova facilmente un assurdo:



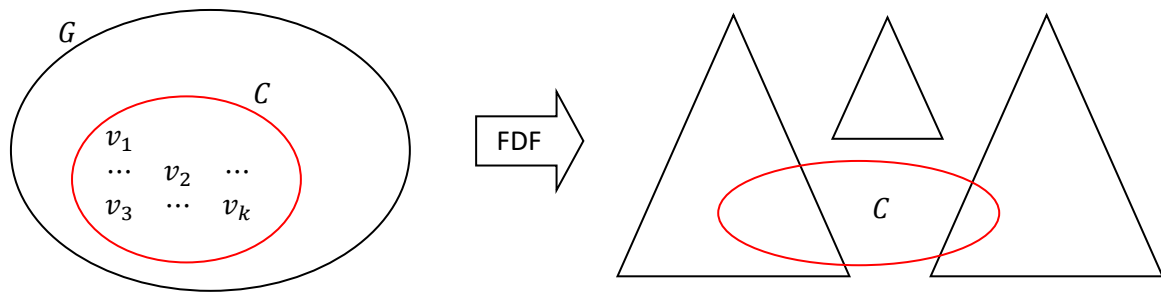
u e v sono mutuamente raggiungibili, dunque esiste un percorso π_3 che va da v a u ed un percorso $\pi_1\pi_2$ che va da u a v attraverso z . Però abbiamo anche che il percorso $\pi_3\pi_1$ va da v a z , mentre il percorso π_2 va da z a v ; ma allora v e z sono mutuamente raggiungibili e dunque z deve appartenere alla stessa componente per definizione.

Questa caratteristica ci permette di introdurre la seguente 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.

N.B.: 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 due vertici della stessa CFC C in due alberi distinti:

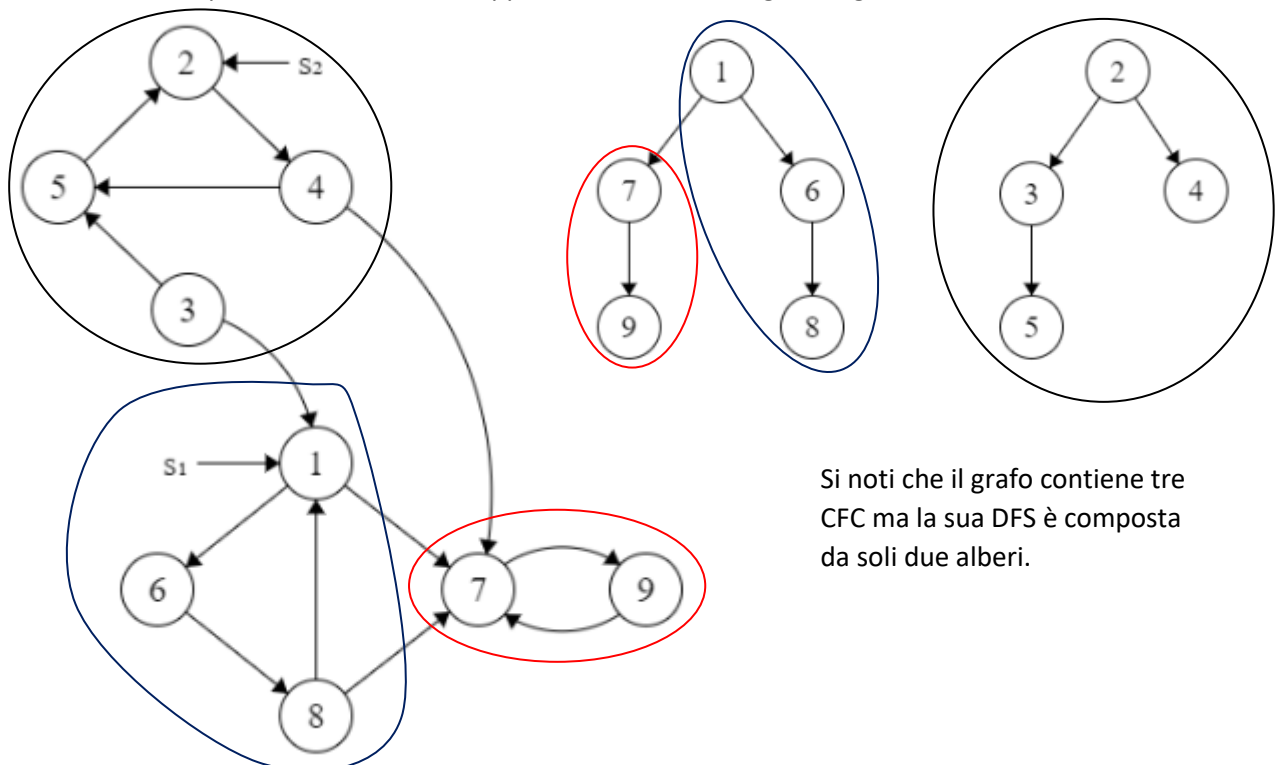


Per definizione sappiamo che ogni coppia di vertici in C è mutuamente raggiungibile (qualsiasi vertice può andare in ogni altro vertice). Ipotizziamo di eseguire una DFS su G : sappiamo che prima o poi visiterà il primo vertice in C . Possiamo dunque fare le seguenti osservazioni:

- Supponiamo che il primo vertice scoperto in C dalla DFS sia v_1
- Sappiamo che esiste un percorso da v_1 ad ogni altro vertice di C
- 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 poiché abbiamo supposto che v_1 sia il primo vertice scoperto di C . Da questa conclusione, unita alle osservazioni b e c, segue che al tempo $d[v_1]$ esiste un percorso tutto bianco da v_1 a ciascun vertice di C . Ciò vuol dire che possiamo applicare il [teorema del percorso bianco](#) e concludere che ogni vertice di C diventerà discendente di v_1 nella FDF; ovvero, tutti i vertici di C saranno nello stesso albero di derivazione (abbiamo così trovato l'assurdo).

La proprietà due ci garantisce che la DFS preserva l'integrità delle CFC. Per una migliore comprensione forniamo un esempio di cosa succede se applichiamo la DFS al seguente grafo:



Da questo esempio si evince che la DFS non può garantire che ogni albero rappresenti una singola componente, ragion per cui non possiamo risolvere il problema delle componenti in maniera ovvia.

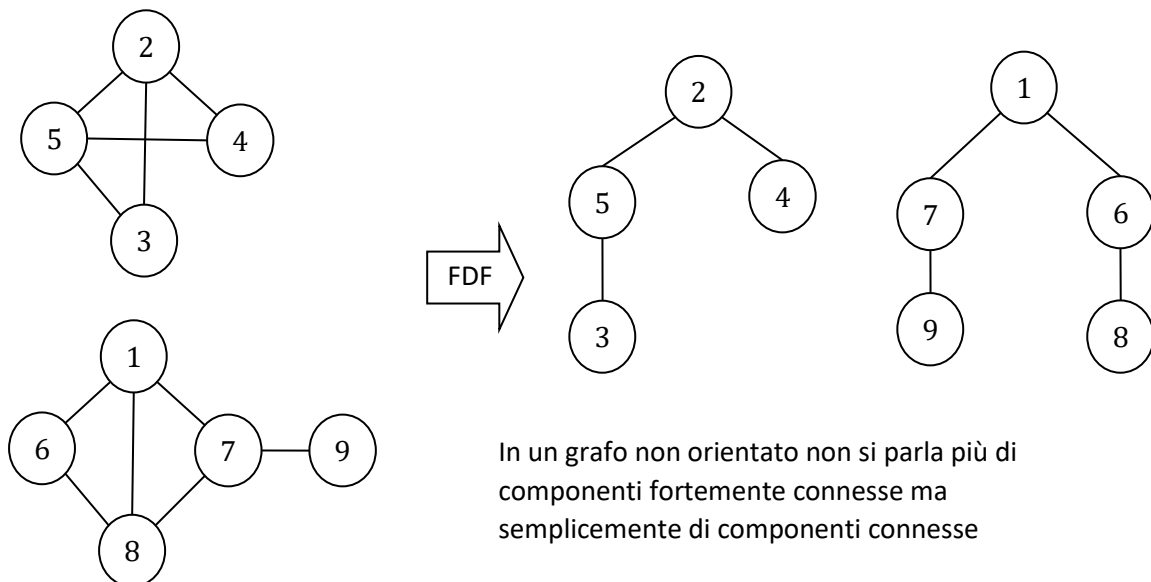
Calcolo delle CFC

Il problema da risolvere è quello di individuare le componenti fortemente connesse, o 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); ma la sola informazione della raggiungibilità non basta a poter dire che un nodo fa parte di una determinata componente (basti pensare alla CFC formata dai vertici 7 e 9 del precedente esempio). Ma 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 . Dunque, ci resta da capire come ottenere tale informazione.

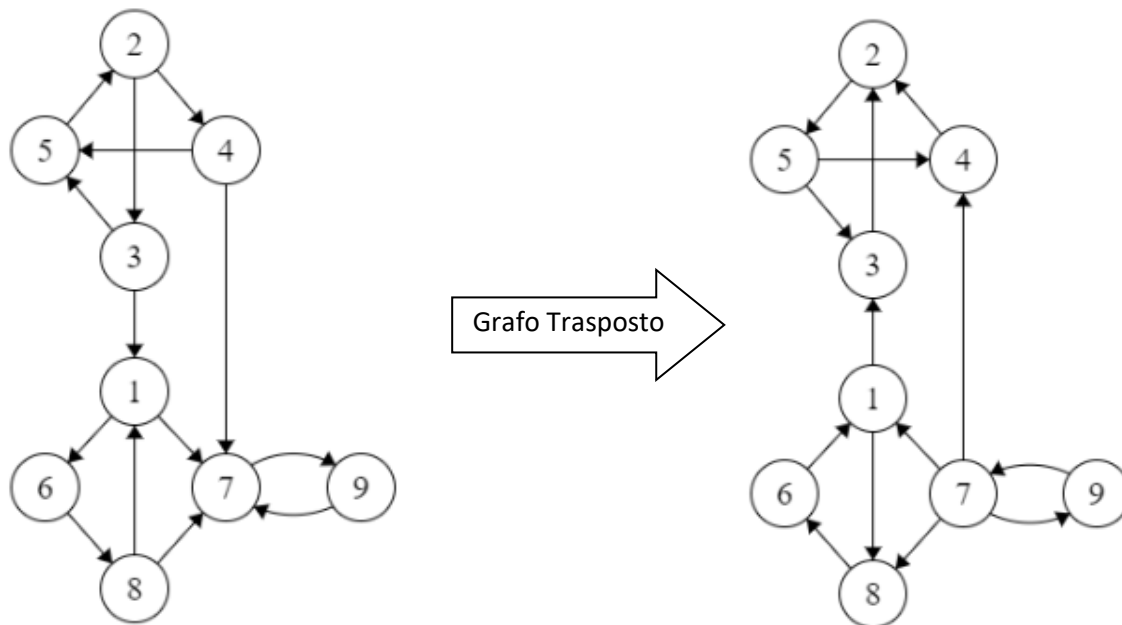
Tuttavia, esistono dei tipi di grafi dove la sola informazione della raggiungibilità basta a risolvere il problema del calcolo delle componenti. Tali grafi sono i grafi non orientati; infatti, se G è un grafo non orientato allora u raggiunge $v \Leftrightarrow v$ raggiunge u (la raggiungibilità è sinonimo di mutua raggiungibilità), di conseguenza l'informazione che una DFS fornisce basta a risolvere il problema.

Esempio (si noti che i vertici di una componente coincidono con tutti e soli i nodi di un albero):



Ritornando al grafo orientato, dobbiamo risolvere il problema della possibilità di avere più componenti in uno stesso albero. Quindi, come già accennato, dobbiamo recuperare l'informazione della raggiungibilità anche nell'altro senso (non solo dalla radice ai discendenti). Sapere quali vertici possono raggiungere la radice (che raggiunge tutti i nodi del suo albero) potrebbe essere molto costoso se usassimo un algoritmo brute force.

Possiamo risolvere il problema in tempo lineare grazie alla definizione di grafo trasposto, ossia, un grafo in cui tutti gli archi hanno direzione inversa rispetto a quelli del grafo originario:

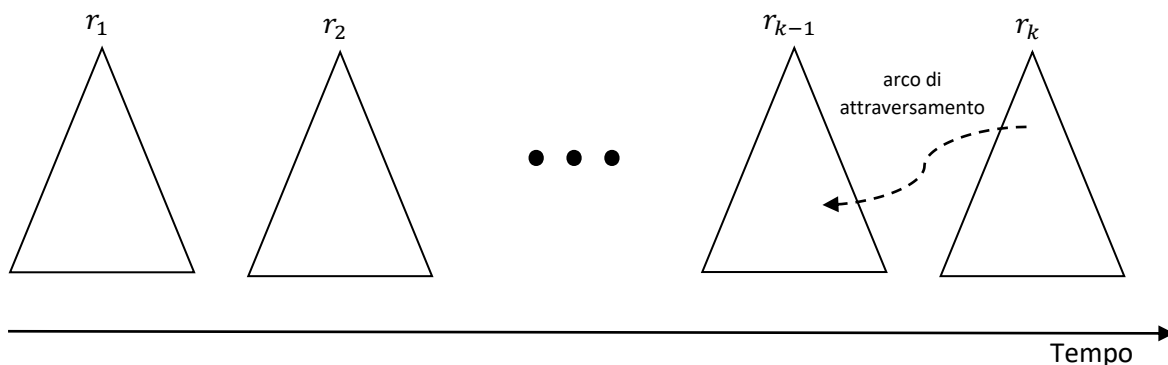


Adesso, se facessi una $\text{DFS_Visit}(G^T, 2)$ raggiungerei i vertici 5,3,4, ovvero, tutti i vertici della CFC di cui 2 fa parte; se a ciò segue una $\text{DFS_Visit}(G^T, 1)$ avremo anche i vertici della CFC di cui 1 fa parte (3 lo trova nero per la visita precedente).

Generalmente, sia $G = (V, E)$, essendo $G^T = (V^T, E^T)$ con $V^T = V$ e $E^T = \{(v, u) | (u, v) \in E\}$ è chiaro che se in G^T esiste un percorso da v ad u allora lo stesso percorso letto nella direzione inversa è un percorso in G da u a v (e viceversa).

Si noti però che non è detto che i vertici raggiungibili nel grafo traspuesto tramite una DFS appartengano a priori alla stessa componente (infatti se nell'esempio precedente partissimo da 1, l'albero generato conterrebbe anche la CFC di 2); dunque, bisogna trovare un ordine di visita per i vertici in maniera tale da poter discriminare le diverse CFC del grafo.

Riassumendo quanto detto, 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 così da evitare di mettere nello stesso albero vertici di componenti diverse. Ovviamente non possiamo eliminare gli archi di attraversamento in G^T , altrimenti non sarebbe più il grafo traspuesto di G (inoltre sarebbe dispendioso); dunque, l'unica soluzione è fare in modo che quando si esplora un arco di attraversamento esso sia "innocuo" (il vertice di arrivo è nero).

Assumiamo che un arco (v, u) sia di attraversamento: sappiamo che quando la DFS lo esplora, u è già nero e $d[u] < d[v]$ (per definizione di arco di attraversamento). Ma allora 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}); quindi, non esistono archi di attraversamento che vanno da r_i a r_j con $i < j$. Quindi, se nella DFS(G^T) parto dalla radice r_k , non potrò mai trovare un arco che mi porti ad un altro albero (si noti che nel grafo trasposto gli archi di attraversamento cambieranno di segno, quindi nell'esempio andrà da r_{k-1} a r_k).

Ma allora basta visitare i vertici in DFS(G^T) esattamente nell'ordine inverso di DFS(G) così da risolvere, non solo il problema degli archi di attraversamento, ma anche quello di avere più CFC in uno stesso albero (l'arco di attraversamento resta innocuo e le CFC si suddivideranno ognuna in un albero distinto).

Resta ora solo la stesura dell'algoritmo, che sarà evidentemente una sequenza di due DFS e la costruzione del grafo trasposto:

```

1  DFS1(G)
2      Init(G)
3      O = NIL /*stack dove 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

```

```

1  DFS1_Visit(G, v, O)
2      Color[v] = 'g'
3      FOR EACH u IN Adj[v] DO
4          IF Color[u] = 'b' THEN
5              O = DFS1_Visit(G, u, O)
6      Color[v] = 'n'
7      O = push(O, v)
8      RETURN O

```

Utilizzeremo un array CFC[v] che ad ogni vertice assocerà un valore intero (i vertici con lo stesso valore apparterranno alla stessa CFC):

```

1  DFS2(Gt, O)
2      Init(Gt)
3      c = 1
4      FOR EACH v IN O DO
5          IF Color[v] = 'b' THEN
6              DFS2_Visit(Gt, v, c)
7              c = c + 1

```

```

1  DFS2_Visit(Gt, v, c)
2      Color[v] = 'g'
3      CFC[v] = c
4      FOR EACH u IN Adj_t[v] DO
5          IF Color[u] = 'b' THEN
6              DFS2_Visit(Gt, u, c)
7      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
```

Poiché 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 della linea 4 di quest'ultimo algoritmo avremo: $CFC[v] = CFC[u] \Leftrightarrow v, u \in C$ (stessa CFC)

9. Esercizi svolti

Regole utili per gli esercizi di analisi asintotica

Derivate elementari e regole di derivazione

$$\frac{d}{dx}(k \cdot f(x)) = k \cdot f'(x)$$

$$\frac{d}{dx}(f(x) \pm g(x)) = f'(x) \pm g'(x)$$

$$\frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{[g(x)]^2}$$

$$\frac{d}{dx}(f(g(x))) = f'(g(x)) \cdot g'(x)$$

Proprietà logaritmi

$$\log_a b = \frac{\log_c b}{\log_c a}$$

$$\log_a b = \frac{\log_b b}{\log_b a} = \frac{1}{\log_b a}$$

$$n = n \log_a a = \log_a(a^n)$$

$$n = a^{\log_a n}$$

Sfruttando la prima e l'ultima proprietà:

$$a^{\log_b n} = a^{\frac{\log_a n}{\log_a b}} = (a^{\log_a n})^{\frac{1}{\log_a b}} = n^{\frac{1}{\log_a b}} = n^{\log_b a}$$

Sommatorie notevoli

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

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

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

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

Tutorato ASD

I seguenti esercizi sono svolti dal dottorando F. Altiero durante il corso di tutorato per la preparazione al compito scritto. Sono presenti le registrazioni nel canale telegram al seguente link:

<https://t.me/+EFvXYrUpt1AwNzc0>

Derivate elementari

$$D(x^n) = nx^{n-1}$$

$$D(\log_a x) = \frac{1}{x} \cdot \frac{1}{\ln a} = \frac{\log_a e}{x}$$

$$D(\ln x) = \frac{1}{x}$$

$$D(a^x) = a^x \ln a = a^x \cdot \frac{1}{\log_a e}$$

Teoremi

$$\log_a(b \cdot c) = \log_a b + \log_a c$$

$$\log_a\left(\frac{b}{c}\right) = \log_a b - \log_a c$$

$$\log_a(b^c) = c \log_a b$$

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

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

Un metodo di risoluzione:

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

Lezione 1 del 14/11/2022

- Si risolva la seguente equazione di ricorrenza, calcolandone l'**andamento asintotico**:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ \sqrt[3]{n^2} \cdot T(\sqrt[3]{n}) + n & \text{altrimenti} \end{cases}$$

Scriviamo l'equazione precedente in una notazione più facile da manipolare:

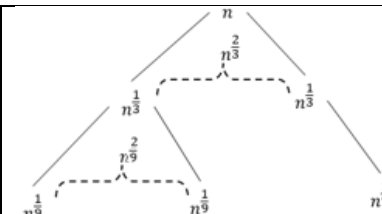
$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ n^{\frac{2}{3}} \cdot T\left(n^{\frac{1}{3}}\right) + n & \text{altrimenti} \end{cases}$$

Utilizziamo il metodo dell'albero delle ricorrenze. Attraverso il seguente schema:

- Per ogni livello ricaveremo la taglia dell'input, il contributo di un singolo nodo, il numero di rami di quel livello ed infine calcoleremo il contributo totale.
- Taglia dell'input:** è ricavata dall'input del livello precedente. Supponendo che la chiamata al livello $i - 1$ sia stata $T(x) = x^{\frac{2}{3}} \cdot T\left(x^{\frac{1}{3}}\right) + x$, allora la taglia dell'input del livello i è proprio $x^{\frac{1}{3}}$.
- Contributo di un singolo nodo:** si ricava dal caso ricorsivo. Se $T(n) = n^{\frac{2}{3}} \cdot T\left(n^{\frac{1}{3}}\right) + n$, il contributo sarà n , quindi se la taglia dell'input sarà x allora avremo $T(x) = x^{\frac{2}{3}} \cdot T\left(x^{\frac{1}{3}}\right) + x$, dunque il contributo sarà x .
- Numero di rami:** è dato dal numero di sottoproblemi che ogni nodo precedente genera, e quindi per il numero totale di rami bisogna moltiplicare il numero di sottoproblemi che genera un singolo sottoproblema del livello precedente per il numero di sottoproblemi generati dal livello precedente. Il numero di sottoproblemi è semplicemente il fattore moltiplicativo della chiamata ricorsiva: esempio per $T(x) = x^{\frac{2}{3}} \cdot T\left(x^{\frac{1}{3}}\right) + x$ il numero di sottoproblemi generato da ogni nodo sarà $x^{\frac{2}{3}}$, ne consegue che la successiva chiamata genererà un numero di sottoproblemi pari a $x^{\frac{2}{3}} \cdot y$

$$\text{con } y = \left(\underbrace{x^{\frac{1}{3}}}_{\substack{\text{taglia} \\ \text{input}}} \right)^{\frac{2}{3}} = x^{\frac{2}{9}}.$$

- Contributo totale:** semplicemente il prodotto tra contributo del nodo e i rami generati.

| Lvl | Input | | Contr. | Numero rami | Tot. |
|-----|--|---|-------------------|--|---|
| 0 | n |  | n | 1 | n |
| 1 | $n^{\frac{1}{3}}$ | | $n^{\frac{1}{3}}$ | $n^{\frac{2}{3}}$ | $n^{\frac{1}{3}} \cdot n^{\frac{2}{3}} = n$ |
| 2 | $\left(n^{\frac{1}{3}}\right)^{\frac{1}{3}} = n^{\frac{1}{9}}$ | | $n^{\frac{1}{9}}$ | $n^{\frac{2}{3}} \cdot \left(n^{\frac{1}{3}}\right)^{\frac{2}{3}} = n^{\frac{2}{3}} \cdot n^{\frac{2}{9}} = n^{\frac{8}{9}}$ | $n^{\frac{1}{3}} \cdot n^{\frac{8}{9}} = n$ |

Possiamo già fermarci a questo livello poiché è evidente che il contributo totale di ogni livello sarà n . Di solito lo studio di 3 livelli basta e avanza per capire il pattern, ma se non si è sicuri si consiglia di perderci un po' più di tempo procedendo all'analisi del livello successivo, che nel nostro caso sarà il livello 3 con contributo totale di $n^{\frac{1}{27}} \cdot \left(n^{\frac{8}{9}} \cdot n^{\frac{2}{27}}\right) = n^{\frac{1+26}{27}} = n$

A questo punto non ci resta che effettuare la somma di tutti i livelli, ma per questo dobbiamo prima analizzare il numero di chiamate ricorsive che viene fatto; in altri termini, bisogna analizzare quando si verifica il caso base.

| Livello | 0 | 1 | 2 | ... | i | ... |
|---------|-------------------------|---------------------------------------|---------------------------------------|-----|---------------------|-----|
| Input | $n = n^{\frac{1}{3^0}}$ | $n^{\frac{1}{3}} = n^{\frac{1}{3^1}}$ | $n^{\frac{1}{9}} = n^{\frac{1}{3^2}}$ | | $n^{\frac{1}{3^i}}$ | |

Il mio albero arriverà ad un certo punto fino al livello h che sarà l'altezza del nostro albero.

La ricorsione terminerà quando l'input che viene dato è 2 o inferiore; quindi, la taglia del sottoproblema si riduce fino ad avere $n^{\frac{1}{3^h}} \leq 2$. Semplifichiamo la disequazione calcolando h nel caso peggiore (input del caso base più alto):

$$n^{\frac{1}{3^h}} = 2 \Rightarrow \log_2 n^{\frac{1}{3^h}} = \log_2 2 \Rightarrow \frac{1}{3^h} \log_2 n = 1 \Rightarrow 3^h = \log_2 n \Rightarrow \log_3 3^h = \log_3(\log_2 n) \Rightarrow h = \log_3(\log_2 n)$$

Adesso abbiamo tutte le informazioni per poter risolvere la nostra equazione (si noti che abbiamo approssimato per eccesso il contributo delle foglie incorporandolo nella sommatoria, ma è una approssimazione più che lecita, altrimenti avremmo dovuto scrivere $T(n) = \text{cntr.f} + \sum_{i=0}^{h-1} n$; inoltre, è consigliabile sostituire ad h il valore trovato solo alla fine, così da semplificare i calcoli):

$$T(n) = \sum_{i=0}^h n = n \sum_{i=0}^h 1 = n \left(\underbrace{1}_{i=0} + h \right) = n + nh = n + n(\log_3(\log_2 n)) = \Theta(\log_3(\log_2 n))$$

- Si individuino, nel caso esistano, le costanti moltiplicative atte a mostrare la seguente relazione asintotica: $\log_2(n^{2n}) + n - \log_2(n) = \Theta(\log_2(n^n))$. In caso contrario, mostrare la falsità della relazione.

Possiamo suddividere l'esercizio in due parti:

- Innanzitutto, verificare se tale relazione è vera studiando il limite del rapporto.
- Nel caso sia vera sviscerare la relazione in una forma più esplicita in maniera tale da trovare le costanti, altrimenti confutarla.

Sfrutteremo il seguente teorema: $f(n) = \Theta(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k \in \mathbb{R} \setminus \{0\}$:

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log n^{2n} + n - \log n}{\log n^n} &= \lim_{n \rightarrow \infty} \frac{2n \log n + n - \log n}{n \log n} = \lim_{n \rightarrow \infty} \frac{n \left(2 \log n + 1 - \frac{\log n}{n} \right)}{n \log n} \\ &= \lim_{n \rightarrow \infty} \frac{2 \log n + 1 - \frac{\log n}{n}}{\log n} = \lim_{n \rightarrow \infty} \frac{\log n \left(2 + \frac{1}{\log n} - \frac{1}{n} \right)}{\log n} = 2 \end{aligned}$$

A questo punto, dato che la relazione è verificata, dobbiamo trovare delle costanti tali che

$$\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, c_1 \cdot n \log n \leq 2n \log n + n - \log n \leq c_2 \cdot n \log n$$

Suddividiamo tale ricerca in due parti, prima cerchiamo la costante c_1 e poi la c_2 :

$c_1 \cdot n \log n \leq 2n \log n + n - \log n$. Questa c_1 deve essere costante, quindi praticamente dobbiamo far "sparire" la n , procederemo nel seguente modo: la nostra disequazione $x \leq a$ la andremo a semplificare maggiorando la seconda parte (si potrebbe procedere anche minorando la prima); ovvero, cercheremo un $b \leq a$ tale che $x \leq b \leq a$ così che le costanti che rispettano $x \leq b$ siano valide anche per $x \leq a$ (ovviamente la b dovrà essere più facile da manipolare altrimenti non avrebbe senso).

Cerchiamo dunque di trovare dei valori minori di $2n \log n + n - \log n$ che ci rendano banale la successiva divisione per $n \log n$ (in pratica non vogliamo frazioni):

$$\begin{aligned} c_1 \cdot n \log n &\leq 2n \log n + n - \log n \xrightarrow{\log n \leq n \quad \forall n \geq 1 \quad \text{quindi } -\log n \geq -n} c_1 \cdot n \log n \leq 2n \log n + n - n \Rightarrow \\ &\Rightarrow c_1 \cdot n \log n \leq 2n \log n \Rightarrow c_1 \leq \frac{2n \log n}{n \log n} \Rightarrow c_1 \leq 2 \end{aligned}$$

Quindi abbiamo trovato che $\forall n \geq 1$ (la condizione usata per la nostra semplificazione) $c_1 \leq 2$, allora possiamo scegliere la costante $c_1 = 2$ e $n_0 = 1$.

Resta ora da risolvere la disequazione $2n \log n + n - \log n \leq c_2 \cdot n \log n$, ma possiamo procedere in maniera duale alla precedente: andremo a maggiorare la prima sfruttando il fatto che $n - \log n \leq n \log n$ $\forall n \geq 4$ (infatti per $n = 1$ o $n = 2$ la suddetta relazione non è valida, si consiglia sempre di verificare gli input; in ogni caso ci basta sapere che da un certo punto in poi la relazione è vera).

$$2n \log n + n - \log n \leq c_2 \cdot n \log n \Rightarrow 2n \log n + n \log n \leq c_2 \Rightarrow c_2 \geq 3$$

Ma allora per $c_1 = 2, c_2 = 3$ e $n_0 = \max\{1, 4\} = 4$ la relazione $\log_2(n^{2n}) + n - \log_2(n) = \Theta(\log_2(n^n))$ è valida ed abbiamo concluso (un altro metodo di soluzione è descritto nella [lezione 6](#)).

Lezione 2 del 21/11/2022

- Si scriva un **algoritmo ricorsivo** che, dati in ingresso un albero binario di ricerca su interi T e due valori $k_1, k_2 \in \mathbb{N}$, cancelli da T le chiavi k comprese tra k_1 e k_2 ($k_1 \leq k \leq k_2$). Tale algoritmo dovrà essere efficiente e non far uso **né di variabili globali né di parametri passati per riferimento**.

Consiglio: cercare il più possibile di modularizzare il codice; infatti, suddividere le implementazioni rende il codice più facile da gestire (si evitano errori) e da leggere.

Dalla traccia si evince che è necessaria una procedura ausiliaria che cancella il nodo, i cui casi principali sono i seguenti (per la teoria approfondita si rimanda al capitolo sulla [cancellazione in ABR](#)):

- 1) Rimozione di una foglia: cancello il nodo e rimuovo il puntatore del padre (lo metto a NIL)
- 2) Rimozione nodo con un solo figlio:
 - a) Collego l'unico figlio del nodo da cancellare al padre di quest'ultimo (uso lo stesso puntatore che collegava il padre al nodo che devo cancellare)
 - b) Cannello il nodo
- 3) Rimozione nodo con due figli
 - a) Cerco il minimo dei maggioranti (o il massimo dei minoranti) del nodo
 - b) Copio la chiave del minimo nel nodo da cancellare
 - c) Rimuovo il nodo sostituito (in questo istante sarò o al caso 1 o al caso 2)

La seguente funzione ausiliaria verrà certamente chiamata da un nodo $T \neq \text{NIL}$:

```

1  RimuoviNodo(T)
2      /*Caso 1 o Caso 2*/
3      IF T->sx = NIL OR T->dx = NIL THEN
4          IF T->sx != NIL THEN
5              newroot = T->sx
6          ELSE
7              /*caso1 sarà newroot = NIL*/
8              newroot = T->dx
9          dealloca(T)
10         RETURN newroot
11     ELSE /* Caso 3*/
12         k = StaccaMin(T->dx, T)
13         T->key = k
14         RETURN T

```

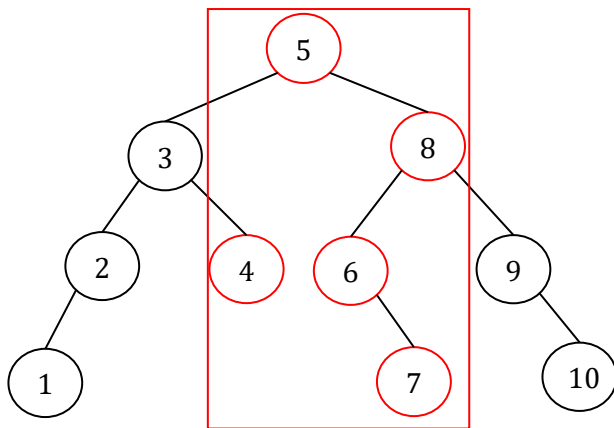
T e P sono sempre diversi da NIL

```

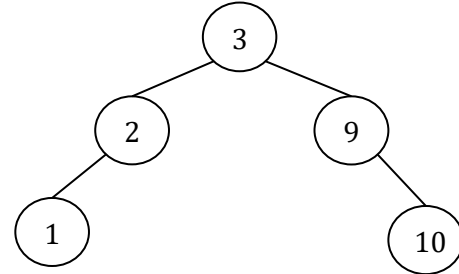
StaccaMin(T, P)
IF T->sx != NIL THEN
    RETURN StaccaMin(T->sx, T)
ELSE
    k = T->key
    IF T = P->sx THEN
        P->sx = T->sx
    ELSE
        P->dx = T->dx
    dealloca(T)
    RETURN k

```

Ora dobbiamo creare l'algoritmo principale, quindi bisogna rimuovere da un ABR T tutti i nodi con chiave k tale che $k_1 \leq k \leq k_2$. Vediamo come comportarci su un concreto ABR:



Se $k_1 = 4$ e $k_2 = 8$ dovrei cancellare i nodi nel riquadro rosso ed avere come risultato il seguente albero:



Si noti che i nodi da cancellare potrebbero non avere una discendenza diretta ma è evidente che sono racchiusi in un intervallo. In seguito a queste considerazioni, bisogna capire il tipo di visita da fare.

Supponiamo di scendere ed arrivare al nodo 3: poiché $3 < k_1$ siamo certi che a sinistra non abbiamo nulla da cancellare; in maniera analoga se arrivo al nodo 8 so che quel nodo va cancellato ma alla sua destra non ho nulla da cancellare poiché $9 > k_2$.

Passando al nostro algoritmo, tolto il caso base dove $T = \text{NIL}$ e quindi non ho nulla da fare, posso imbattermi nei seguenti casi:

- $T \rightarrow \text{key} < k_1$: scendo a destra
- $T \rightarrow \text{key} > k_2$: scendo a sinistra
- $k_1 \leq T \rightarrow \text{key} \leq k_2$:
 - 1) Scendo a sinistra
 - 2) Scendo a destra
 - 3) Cancello il nodo

Stiamo praticamente eseguendo una visita in **post order**, in quanto prima visito i sottoalberi e poi elaboro il nodo. Generalmente la visita in post order è quella più adatta per le operazioni di cancellazione poiché se cancellassimo prima i nodi perderemmo i collegamenti agli altri nodi (potremmo sviare questo problema con delle variabili ausiliare ma ciò renderebbe l'algoritmo più difficile da gestire).

```

1  CancellaIntervallo(T, k1, k2)
2      IF T != NIL THEN
3          IF T->key < k1
4              T->dx = CancellaIntervallo(T->dx, k1, k2)
5          ELSE IF T->key > k2 THEN
6              T->sx = CancellaIntervallo(T->sx, k1, k2)
7          ELSE
8              T->sx = CancellaIntervallo(T->sx, k1, k2)
9              T->dx = CancellaIntervallo(T->dx, k1, k2)
10             T = CancellaNodo(T)
11     RETURN T

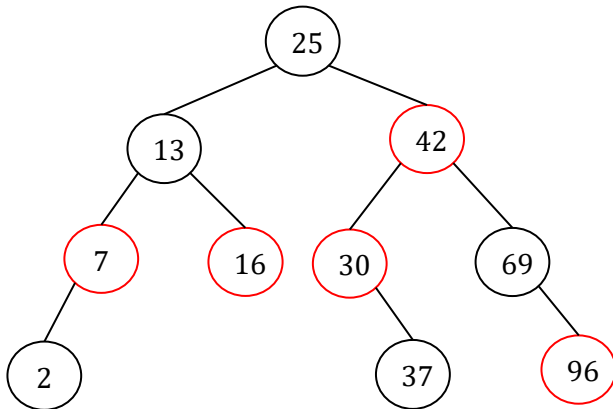
```

Lezione 3 del 28/11/2022

- Scrivere un algoritmo ricorsivo che, dato in ingresso un albero binario di ricerca T ed un numero intero k , cancelli dall'albero tutti i nodi che si trovano in posizione multipla di k nell'ordinamento totale dell'albero.

Tale algoritmo dovrà essere efficiente e non far uso **né di variabili globali né di parametri passati per riferimento**.

Per ordinamento totale si intende sostanzialmente l'ordine delle chiavi in maniera crescente (è praticamente il risultato della visita InOrder).



Se $k = 2$ allora bisogna rimuovere tutte le chiavi che sono in posizione pari, poiché l'ordinamento totale dell'albero a sinistra è:

2,7,13,16,25,30,37,42,69,96

Bisogna dunque cancellare i nodi rossi.

Il miglior modo di procedere per una soluzione efficiente è lavorare direttamente sull'albero senza utilizzare strutture di supporto (sarebbe uno spreco di memoria).

Riflettiamo su come capire quali nodi rimuovere tenendo in considerazione l'albero di cui sopra e $k = 2$:

- Posizioniamoci in radice, che sappiamo essere in posizione 5: quello che posso dedurre è che la posizione relativa della mia radice è data dal numero di nodi del sottoalbero sinistro più uno. Allo stesso modo il nodo 13 sarà in posizione 3.
- La posizione relativa però non basta, infatti il nodo 42 avrà posizione relativa 3, ma sappiamo che nell'albero totale esso occupa l'ottava posizione nell'ordinamento totale. Da ciò deduciamo che per capire la posizione del nodo non posso basarmi solo sulle proprietà locali dei sottoalberi.

Abbiamo però già notato che una visita in order processa i nodi esattamente nell'ordinamento totale che a noi interessa; dunque, abbiamo necessità di "portarci dietro" un parametro ulteriore a T e k che ci dice il numero di nodi visitati prima di andare a controllare il sottoalbero radicato in T .

Ad esempio, per il nodo con chiave 42 passeremo al parametro v il valore 5, che sommato alla sua posizione relativa ci darà esattamente la sua posizione nell'ordinamento totale: $5 + (2 + 1)$.

Attenzione però: anche se la visita in order è l'unico tipo di visita che ci permette di risalire alla posizione totale non possiamo assolutamente cancellare i nodi durante suddetta visita. Infatti, se cancellassimo immediatamente il nodo prima di andare nel sottoalbero destro romperemmo l'ordinamento totale degli elementi (esempio, se cancelliamo 42, il nodo 96 sarà in posizione 9 e non più 10). Quindi, per la cancellazione useremo la visita in post order.

Ricapitolando, la posizione assoluta di un nodo è data dal numero di nodi del sottoalbero sinistro e la somma dei nodi visitati prima di chiamare la funzione su quel nodo più il nodo stesso. Poiché non è possibile ricavare i nodi già visitati da un sottoalbero bisogna aggiungere tale dato come parametro (in generale se durante un algoritmo ricorsivo non possiamo ricavarci un particolare dato allora questo va aggiunto come parametro).

Di base, la posizione di un nodo la calcoliamo come il numero di nodi a sinistra (quindi non solo quelli del sottoalbero sinistro ma tutti i nodi alla sua sinistra, ad esempio per il nostro nodo con chiave 42 saranno i seguenti: 2,7,13,16,25,30,37). Il sottoalbero destro non ci serve, ma ciò che ottengo mi servirà successivamente quando torno indietro poiché il chiamante vorrà sapere il numero di nodi a sinistra di esso (il nodo 13 non ha bisogno di sapere il numero di nodi che ha il sottoalbero con radice 16, ma quello con radice 25 sì).

Di seguito generiamo l'algoritmo che restituisce il numero di nodi a sinistra e che in più si occupa di cancellare in post order i nodi durante la discesa (non è necessario fare due tipi di visita):

```

1  Algo(T, P, k, visited)
2      /*il valore di visited non cambia*/
3      IF T = NIL THEN
4          RETURN visited /*RETURN 0 non avrei più il numero di nodi a sx*/
5      ELSE
6          /*calcolo la posizione del mio nodo*/
7          pos = 1 + Algo(T->sx, T, k, visited)
8
9          /*vado a destra e aggiorno visited prima di restituirlo
10         visited = Algo(T->dx, T, k, pos)
11
12         /*ora possiamo eventualmente cancellare il nostro nodo*/
13         IF pos%k = 0 THEN
14             IF P->sx = T THEN
15                 P->sx = RimuoviNodo(T) /*vedi lezione 2*/
16             ELSE
17                 P->dx = RimuoviNodo(T)
18         RETURN visited

```

Si noti che qui non trattiamo il caso in cui T sia la radice totale dell'albero (l'algoritmo crasherebbe nella linea 14; infatti, essendo $P = NIL$ non potrei leggere $P \rightarrow sx$). Per la radice useremo il seguente algoritmo:

```

1  AlgoRoot(T, k)
2      IF T != NIL THEN
3          pos = 1 + Algo(T->sx, T, k, 0)
4          Algo(T->dx, T, k, pos)
5          IF pos%k = 0 THEN
6              T = RimuoviNodo(T)
7      RETURN T

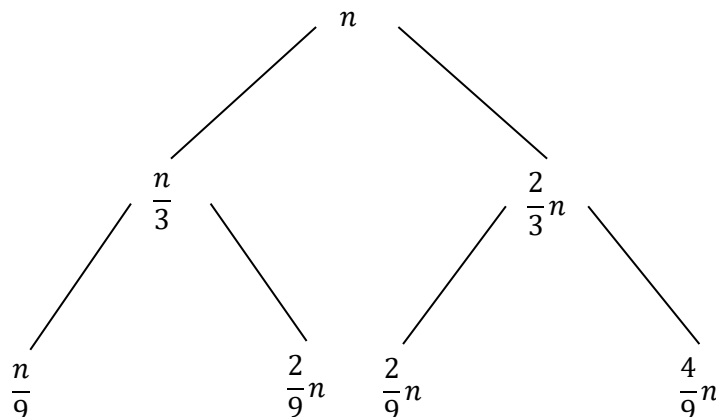
```

Lezione 4 del 05/12/2022

- Si risolva la seguente equazione di ricorrenza, calcolandone l'**andamento asintotico**:

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

Si noti che la chiamata ricorsiva ha taglie di input differenti, e poi effettua n passi (dove n è la taglia dell'input). Utilizziamo il metodo già visto nella [lezione 1](#) dell'albero di ricorrenza.



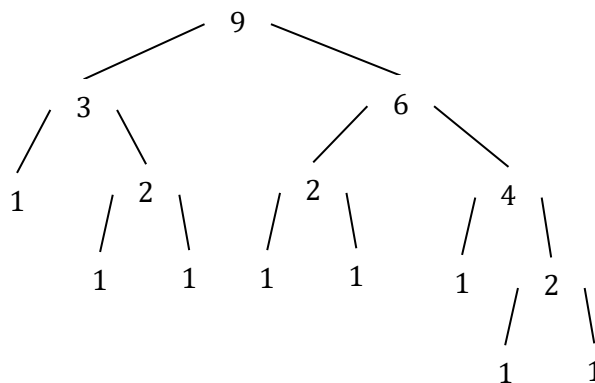
Nella tabella seguente non considereremo il numero di rami poiché il contributo totale sarà dato dalla somma dei contributi dei singoli nodi e non più dal prodotto tra il contributo di un nodo e il numero di rami.

Inoltre, non scriveremo nemmeno l'input poiché la taglia è differente per ogni nodo, ed al suo posto scriveremo l'equazione di ricorrenza associata a quel livello al fine di calcolarne il contributo:

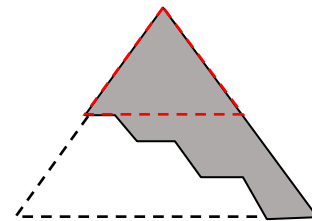
| Livello | | Contributi singoli | Contributo totale |
|---------|--|---|--|
| 0 | $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2}{3}n\right) + n$ | n | n |
| 1 | $T\left(\frac{n}{3}\right) = T\left(\frac{n}{9}\right) + T\left(\frac{2}{9}n\right) + \frac{n}{3}$ $T\left(\frac{2}{3}n\right) = T\left(\frac{2}{9}n\right) + T\left(\frac{4}{9}n\right) + \frac{2}{3}n$ | $\frac{n}{3}, \frac{2}{3}n$ | $\frac{n}{3} + \frac{2}{3}n = n$ |
| 2 | per $\frac{n}{3}$ $\left\{ \begin{array}{l} T\left(\frac{n}{9}\right) = T\left(\frac{n}{27}\right) + T\left(\frac{2}{27}n\right) + \frac{n}{9} \\ T\left(\frac{2}{9}n\right) = T\left(\frac{2}{27}n\right) + T\left(\frac{4}{27}n\right) + \frac{2}{9}n \end{array} \right.$ per $\frac{2}{3}n$ $\left\{ \begin{array}{l} T\left(\frac{2}{9}n\right) = T\left(\frac{2}{27}n\right) + T\left(\frac{4}{27}n\right) + \frac{2}{9}n \\ T\left(\frac{4}{9}n\right) = T\left(\frac{4}{27}n\right) + T\left(\frac{8}{27}n\right) + \frac{4}{9}n \end{array} \right.$ | $\frac{n}{9}, \frac{2}{9}n, \frac{2}{9}n, \frac{4}{9}n$ | $\frac{n}{9} + \frac{2}{9}n + \frac{2}{9}n + \frac{4}{9}n = n$ |

N.B.: il fatto che al livello due abbiamo la seconda equazione della prima parte uguale alla prima equazione della seconda è un caso (dunque se magari in un esercizio non ci capita non è detto che abbiamo sbagliato).

Possiamo assumere che ogni livello abbia contributo totale pari ad n . A questo punto dovremmo calcolare l'altezza ma è evidente che essendo le taglie dell'input differenti l'albero non sarà completo, bensì sbilanciato (avrà un "lato" più profondo dell'altro). Infatti, se ad esempio supponiamo di avere $n = 9$ risulterà un albero del seguente tipo (approssimiamo per difetto gli input non interi):



Un albero di taglia n avrà, con buona approssimazione, la seguente forma:



Dunque, se seguo il percorso estremo sinistro avrò l'altezza minima mentre se seguo il percorso estremo destro avrò l'altezza massima.

Nel percorso estremo sinistro la taglia decresce per un generico livello i di $\frac{n}{3^i}$, dunque:

$$\frac{n}{3^{h_{\min}}} = 1 \Rightarrow 3^{h_{\min}} = n \Rightarrow h_{\min} = \log_3 n$$

In maniera analoga, per il percorso destro avremo:

$$\left(\frac{2}{3}\right)^{h_{\max}} n = 1 \Rightarrow \frac{2^{h_{\max}}}{3^{h_{\max}}} n = 1 \Rightarrow 2^{h_{\max}} n = 3^{h_{\max}} \Rightarrow n = \frac{3^{h_{\max}}}{2^{h_{\max}}} \Rightarrow \left(\frac{3}{2}\right)^{h_{\max}} = n \Rightarrow h_{\max} = \log_{\frac{3}{2}} n = \frac{\log_3 n}{\log_3 \frac{3}{2}} \Rightarrow$$

$$\Rightarrow h_{\max} = \frac{1}{1 - \log_3 2} \log_3 n$$

Per avere una stima precisa dovremmo analizzare tutti i rami, ma poiché a noi interessa una stima asintotica possiamo semplificare ponendo il nostro albero in relazione con uno completo di altezza minima ed uno completo di altezza massima; infatti, $T_{\min}(n) \leq T(n) \leq T_{\max}(n)$.

$$\sum_{i=0}^{h_{\min}} n = n(h_{\min} + 1) = n \log_3 n + n \leq T(n) \Rightarrow T(n) = \Omega(n \log_3 n)$$

$$T(n) \leq \sum_{i=0}^{h_{\max}} n = n(h_{\max} + 1) = \frac{1}{1 - \log_3 2} n \log_3 n + n \Rightarrow T(n) = O(n \log_3 n)$$

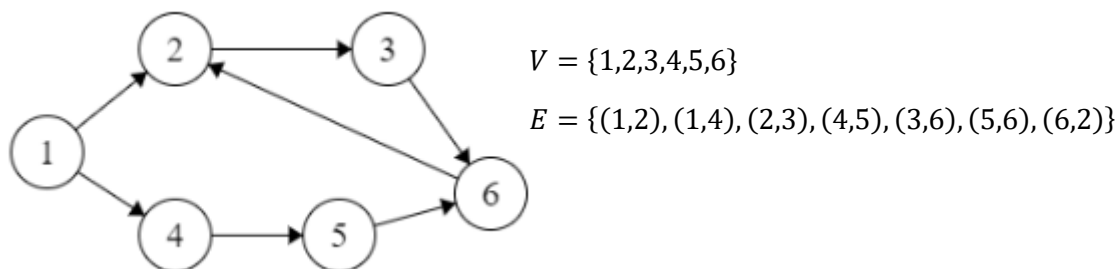
Dunque, essendo le due funzioni uguali, possiamo concludere che $T(n) = \Theta(n \log_3 n)$.

N.B.: se avessimo avuto, ad esempio, $T(n) = \Omega(\log_3 n)$ invece di $T(n) = \Omega(n \log_3 n)$, allora non avremmo avuto una relazione Theta. In questo caso si lasciano le due funzioni Omega e O-grande.

Lezione 5 del 12/12/2022

- Sia $G = (V, E)$, scrivere un algoritmo che verifichi la presenza di cicli in G .

Prendiamo il seguente grafo come esempio:



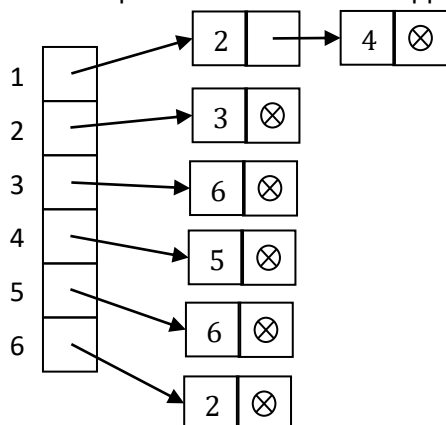
È evidente che in questa istanza del problema è presente un ciclo che riguarda i vertici 2, 3 e 6 (posso percorrere indefinitamente questi tre vertici).

Di base, per esercizi sui grafi bisogna supporre che non ci siano informazioni ulteriori memorizzate nei vertici; l'informazione nel vertice rappresenta proprio l'indice in cui è etichettato il nodo. Tale informazione ci consente di semplificare la struttura del nostro grafo.

In applicazioni pratiche bisogna definire le strutture utilizzate per memorizzare il grafo stesso.

Sostanzialmente, l'implementazione effettiva avviene tramite [liste di adiacenza](#) o [matrici di adiacenza](#); generalmente la prima è quella preferibile nella risoluzione degli esercizi.

La lista di adiacenza per la nostra istanza è rappresentata come segue:



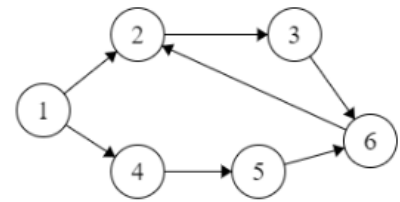
Supponiamo di non avere informazioni satelliti, ovvero informazioni al di là di un identificativo del vertice stesso. Quindi di base possiamo definire una struttura del seguente tipo:

```
struct Grafo {
    int n /*numero nodi*/
    Lista<int> Adj[]
    char Color[]
}
```

N.B.: definire sempre la struttura struct che si vuole utilizzare per il Grafo in ogni esercizio.

Nota: molti esercizi sui grafi vengono posti in maniera più complessa di quel che in realtà sono; infatti, buona parte degli esercizi può essere risolta andando semplicemente a comprendere cosa ci stanno chiedendo e a mappare ciò che ci viene chiesto in uno o più visite in base alla teoria studiata.

Di base potremmo utilizzare due tipologie di visite: la [BFS](#) (in ampiezza) o la [DFS](#) (in profondità). Ma per la soluzione del nostro problema solo la DFS ci permette di scrivere un algoritmo corretto; infatti, solo la DFS analizza i nodi in un ordine preciso e non si ferma come fa la BFS che analizza tutti gli adiacenti prima di passare al prossimo nodo (la BFS permette di ottenere percorsi minimi, cosa che non può fare la DFS).



Nella DFS, quando scopriamo un vertice lo coloriamo di grigio, e questo, poiché visitiamo in post order, resta grigio finché non abbiamo visitato tutti i nodi raggiungibili da quel nodo. Prendendo il grafo sovrastante, e supponendo di iniziare la visita da 1 seguendo il percorso 1,2,3,6, una volta a 6 si tenterà di visitare il 2 che però troverà già grigio. Si noti che i nodi 2,3,6 creano un ciclo, e ciò non è casuale ma è proprio il metodo che utilizzeremo per trovare un ciclo (se si trova un vertice di colore nero allora abbiamo già visitato tutti i percorsi di quel vertice).

Definiamo meglio le situazioni che possono capitarci durante la visita di un nodo. In particolare, i casi che possono avvenire seguendo un arco (v, u) (il nodo v da cui partiamo è sempre grigio):

- 1) Il colore di u è bianco: siamo in un percorso semplice.
- 2) Il colore di u è nero: abbiamo già visitato tutti i percorsi di quel nodo, con la certezza che esso non potrà mai tornare al nodo v (altrimenti avremmo dovuto scoprire v durante la visita di u). Siamo ancora in un percorso semplice.
- 3) Il colore di u è grigio: il nodo u non è ancora stato visitato completamente, ma è stato scoperto. Ciò significa che sto procedendo in un percorso che da u (non necessariamente è la sorgente della visita) mi ha riportato a u (attraverso v). Siamo in un ciclo che è proprio il percorso $u \cdots vu$ (i puntini rappresentano uno, nessuno o più vertici diversi da u e v). (v, u) è detto arco di ritorno.

Passiamo ora ad implementare l'algoritmo:

```

1  boolean DFS_Visit(G, s)
2      Color[s] = 'g'
3      FOR EACH v IN Adj[s] DO
4          IF Color[v] = 'g' THEN
5              RETURN true
6          ELSE IF Color[v] = 'b' THEN
7              ciclico = DFS_Visit(G, v)
8              IF ciclico = true THEN
9                  RETURN true
10     Color[s] = 'n'
11     RETURN false

```

```

1  haCicli(G)
2      FOR EACH v IN G DO
3          Color[v] = 'b'
4      FOR EACH v IN G DO
5          IF Color[v] = 'b' THEN
6              ciclico = DFS_Visit(G, v)
7              IF ciclico = true THEN
8                  RETURN true
9      RETURN false

```

Lezione 6 del 19/12/2022

- Trovare le costanti o confutare la relazione $\log_2(n^{2n}) + n - \log_2(n) = \Theta(\log_2(n^n))$

Applichiamo questa volta un metodo diverso da quello visto nella [lezione 1](#). Utilizzeremo il [metodo delle derivate](#) (più meccanico).

Per capire se $2n \log n + n - \log n = \Theta(n \log n)$ sia effettivamente una relazione Theta calcoliamo il limite del rapporto:

$$\lim_{n \rightarrow \infty} \frac{\log n^{2n} + n - \log n}{\log n^n} = \lim_{n \rightarrow \infty} \left(\frac{2n \log n}{n \log n} + \frac{n}{n \log n} - \frac{\log n}{n \log n} \right) = \lim_{n \rightarrow \infty} 2 + \lim_{n \rightarrow \infty} \frac{1}{\log n} - \lim_{n \rightarrow \infty} \frac{1}{n} = 2 > 0$$

Quindi, la relazione Theta è valida e il valore 2 sarà una delle nostre costanti (si noti che già dal limite possiamo capire che 2 è un limite asintotico inferiore poiché la funzione decresce al crescere di n).

Studiamo la monotonicità del rapporto $h(n) = \left(2 + \frac{1}{\log n} - \frac{1}{n} \right)$ attraverso la derivata prima:

$$\frac{d}{dn} \left(\frac{2n \log n + n - \log n}{n \log n} \right) = \frac{d}{dn} (2) + \frac{d}{dn} \left(\frac{1}{\log n} \right) - \frac{d}{dn} \left(\frac{1}{n} \right) = 0 + \frac{d}{dn} \log^{-1} n - \frac{d}{dn} n^{-1} = -\frac{1}{n \log^2 n} + \frac{1}{n^2}$$

N.B.: un logaritmo con una certa base può essere scritto in un logaritmo con un'altra base a meno di una costante e poiché tale costante viene assorbita dalla relazione asintotica possiamo supporre di essere di fronte ad un logaritmo naturale invece che ad uno in base due semplificando così le derivate.

Studiamone ora la positività:

$$\frac{1}{n^2} - \frac{1}{n \log^2 n} \geq 0 \Rightarrow \frac{\log^2 n - n}{n^2 \log^2 n} \geq 0$$

Numeratore: $\log^2 n - n \geq 0 \Rightarrow \forall n \geq 1 \quad \log^2 n - n < 0$ poiché n cresce sempre più velocemente di $\log n$

Denominatore: $n^2 \log^2 n > 0 \Rightarrow \begin{cases} n^2 > 0 \Rightarrow n \neq 0 \\ \log^2 n > 0 \Rightarrow n \neq 0 \wedge n \neq 1 \end{cases} \Rightarrow n^2 \log^2 n > 0 \quad \forall n \geq 2$

| | | |
|---|--|---|
| N | <div style="display: flex; justify-content: space-around; width: 100%;"> 0 1 </div> <div style="border-top: 1px solid black; width: 100%;"></div> <div style="border-top: 1px dashed black; width: 100%;"></div> | $\Rightarrow h(n) < 0 \quad \forall n \geq 2$ |
| D | <div style="display: flex; justify-content: space-around; width: 100%;"> $\left(\frac{1}{n^2} \right)$ $\left(\frac{1}{n \log^2 n} \right)$ </div> <div style="display: flex; justify-content: space-around; width: 100%;"> + - </div> | |

Dunque, la mia funzione sarà decrescente nell'intervallo $[2, +\infty)$, e questo significa che la costante 2 rappresenta il limite inferiore asintotico ($h(n)$ tende a 2 dall'alto).

L'altra costante è molto più semplice da calcolare; infatti, basta prendere un valore maggiore di 2 che sicuramente sarà sopra al limite inferiore asintotico (essendo $h(n)$ decrescente da $n \geq 2$). Sia 4 tale valore, l'altra costante sarà semplicemente il valore $h(4)$ (va preso il rapporto iniziale e non la derivata); ovvero:

$$\frac{2(4) \log 4 + 4 - \log 4}{4 \log 4} = \frac{16 + 4 - 2}{8} = \frac{9}{4}$$

Ma allora per $n_0 = 4, c_1 = 2, c_2 = \frac{9}{4}$ avremo:

$$\forall n \geq 4, \quad 2n \log n \leq 2n \log n + n - \log n \leq \frac{9}{4} n \log n \quad \text{C.V.D.}$$

- Trovare la costante o confutare la relazione $\frac{\sqrt{n}}{\log n} = O(\sqrt{n})$

La relazione è verificata, infatti:

$$\lim_{n \rightarrow \infty} \frac{\sqrt{n}}{\sqrt{n} \log n} = 0 \Rightarrow \exists c, n_0 > 0 : \forall n \geq n_0, c\sqrt{n} \geq \frac{\sqrt{n}}{\log n}$$

Studiamone la derivata (si noti che è stata già risolta in precedenza):

$$\frac{d}{dn} \left(\frac{1}{\log n} \right) = -\frac{1}{n \log^2 n} \Rightarrow -\frac{1}{n \log^2 n} < 0 \forall n \geq 2$$

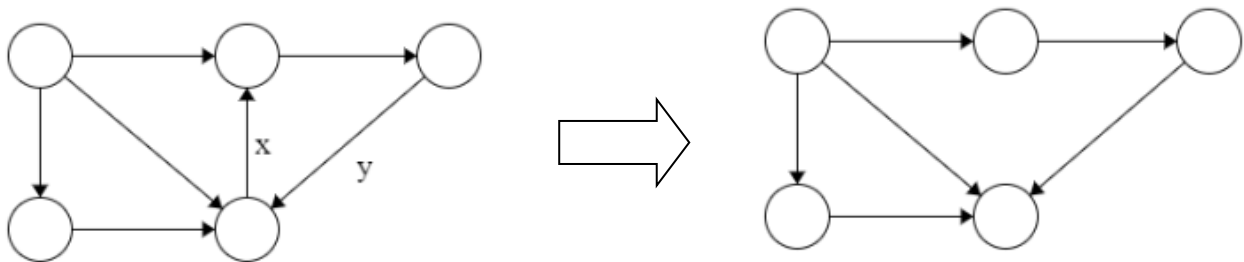
Scegliendo $n = 8$, avremo $\frac{\sqrt{8}}{\sqrt{8} \log 8} = \frac{1}{\log 8} = \frac{1}{3}$; ovvero:

$$\frac{\sqrt{n}}{\log n} \leq \frac{1}{3} \sqrt{n} \quad \forall n \geq 8 \quad \text{C.V.D.}$$

N.B.: se la relazione è O grande la funzione deve essere decrescente da un certo punto in poi (in caso contrario significa che abbiamo sbagliato alcuni calcoli); analogamente, se la relazione è un Ω la funzione deve essere crescente.

- Scrivere un algoritmo che dato un grafo G restituisca il grafo G' : sottografo massimale aciclico di G .

Dovendo essere massimale è evidente che dobbiamo avere lo stesso numero di vertici; per quanto riguarda gli archi, devo prenderne il più possibile e togliere solo quelli che mi causano dei cicli. Esempio:



Si noti che non è l'unico sottografo aciclico massimale: potremmo infatti lasciare l'arco x e togliere l'arco y .

La stesura dell'algoritmo è abbastanza facile poiché l'idea è molto simile a quella vista nella [lezione 5](#):

- Partiamo con un sottografo G' con tutti i vertici di G ma senza archi.
- Facciamo una DFS su G ed ogni volta che incontriamo un arco che non aggiunge cicli (quindi non incontriamo un nodo grigio) lo inseriamo in G' .

```

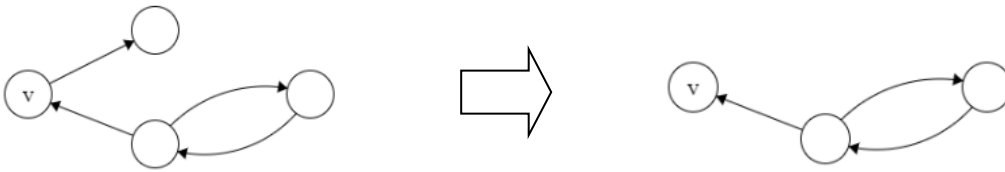
1 maxSubGraphAcyclic(G)
2   G' = NIL
3   FOR EACH v IN V DO
4     Color[v] = 'b'
5     V' = add(V', v)
6   FOR EACH v IN V DO
7     IF Color[v] = 'b' THEN
8       DFS_visit(G, v, G')
9   RETURN G'
```

```

DFS_visit(G, v, G')
  color[v] = 'g'
  FOR EACH u IN Adj[v] DO
    IF Color[u] != 'g' THEN
      E' = add(E', (v,u))
      IF Color[u] = 'b' THEN
        DFS_visit(G, u, G')
  color[v] = 'n'
```

Esercizio: Dato un grafo $G = (V, E)$, ed un vertice $v \in V$, costruire il sottografo massimale dei vertici di G che raggiungono v

Esempio:



Suggerimenti: questo problema può essere risolto attraverso il grafo trasposto e una visita DFS su quest'ultima. In particolare, bisogna creare G^T (usare una funzione ausiliaria) e attraverso una $\text{DFS_Visit}(G^T, v)$ aggiungere gli archi che percorre (al contrario) in un grafo G' , restituendo quest'ultimo.

Mia soluzione (non ne assicuro la correttezza):

```
1  maxSubGraphReachable(G, v)
2      Gt = GrafoTrasposto(G, v)
3      IF Gt = NIL THEN
4          RETURN NIL
5      /*inizializzo V' e E' A NIL*/
6      G' = newGraph(NIL, NIL)
7      /*v sicuramente esiste*/
8      G' = DFS_Visit(Gt, v, G')
9      RETURN G'
```



```
1  DFS_Visit(Gt, v, G)
2      Color_t[v] = 'g'
3      V = add(V, v)
4      FOR EACH u IN Adj_t DO
5          IF Color_t[u] = 'b' THEN
6              G = DFS_Visit(Gt, u, G)
7              /*ho sia il vertice u che v*/
8              E = add(E, (u,v))
9      Color_t[v] = 'n'
10     RETURN G
```

Costruisce il grafo trasposto e controlla la validità del parametro d'ingresso:

```
1  GrafoTrasposto(G, s)
2      Vt = NIL
3      Et = NIL
4      valid = false
5      /*aggiungo i vertici*/
6      FOR EACH v IN V DO
7          Vt = add(Vt, v)
8          Color_t[v] = 'b'
9          IF v = s THEN
10             valid = true
11      /*controllo la validità di s*/
12      IF !valid THEN
13          RETURN NIL
14      /*aggiungo gli archi*/
15      FOR EACH v IN V DO
16          FOR EACH u IN Adj[v] DO
17              Et = add(Et, (u,v))
18      Gt = newGraph(Vt, Et)
19      RETURN Gt
```


Lezione 7 del 16/01/2023

- Scrivere un algoritmo iterativo che simuli precisamente il comportamento ricorsivo dell'algoritmo sotto riportato. Si supponga data la funzione $BEST(a, b, k)$

```
1  Alg(T, k)
2      ret = NIL
3      IF T != NIL THEN
4          IF T->key > k THEN
5              b = T
6              ret = BEST(Alg(T->sx, k), b, k)
7          ELSE IF T->key < k THEN
8              a = T
9              ret = BEST(a, Alg(T->dx, k), k)
10         ELSE
11             a = Alg(T->sx, k)
12             b = Alg(T->dx, k)
13             ret = BEST(a, b, k)
14     RETURN ret
```

La prima cosa da fare è capire quali parametri di input o variabili locali bisogna memorizzare in uno stack:

- Cominciamo con i parametri locali:
 - T viene utilizzata per controllare la chiave, tra le linee 11 e 12 ho bisogno di recuperare il parametro T della chiamata precedente per poterlo riutilizzare nella successiva. Si noti anche che nella chiamata 6 e 9 restituiamo il risultato subito dopo la chiamata ricorsiva. Quindi se l'algoritmo terminasse con la linea 9 non avrei necessità di utilizzare lo stack di T , ma in questo caso st_T è necessario.
 - k viene letto in diversi punti ma si può notare che il suo valore non viene mai scritto; dunque, essendo costante lo stack è inutile così come la variabile ausiliaria ck
- Analizziamo adesso le variabili locali:
 - ret viene scritto in riga 2 e poi in riga 6, 9 e 13 (si noti che viene solo scritto e mai letto), si noti inoltre che potremmo benissimo sostituire ret con RETURN. Poiché la variabile ret non viene mai letta non necessita di uno stack
 - b viene scritta nella linea 5 ma si può notare che è un passaggio superfluo poiché coincide con il valore corrente del parametro T . Lo stesso vale per la b scritta in riga 13; infatti, essendo proprio il valore di ritorno della chiamata 12 quella linea sarebbe $b = ret$ ed è quindi superflua: potremmo scrivere direttamente $ret = BEST(a, Alg(T \rightarrow dx, k), k)$
 - a nella linea 8 e 9 funziona esattamente come b , ma il discorso cambia nell'ultimo blocco. Infatti, la a è assegnata prima della linea 12 ma utilizzata dopo. Quindi siamo sicuri che lo stack di a è necessario.

Per quanto riguarda il capire da quale chiamata ritorniamo possiamo sfruttare la chiave di T :

- $T \rightarrow key > k$: siamo di ritorno dalla linea 6
- $T \rightarrow key < k$: siamo di ritorno dalla linea 9
- $T \rightarrow key = k$: siamo nell'ultimo blocco e bisogna discriminare ulteriormente
 - Se l'ultimo nodo ($last$) è uguale al figlio sinistro torniamo da riga 11
 - Se l'ultimo nodo è uguale al figlio destro torniamo invece da riga 12

Ricapitolando:

- Stack: T ed a
- Backtracking; T per le condizioni, e $T \rightarrow sx$ o $T \rightarrow dx$ per discriminare l'ultimo blocco

```

1  AlgIter(T, k)
2      cT = T
3      st_T = st_a = NIL
4      last = NIL
5      start = true
6      WHILE start || st_T != NIL DO
7          IF start THEN
8              ret = NIL
9              IF cT != NIL THEN
10                 IF cT->key > k THEN
11                     /*b = cT superfluo: posso ricavarla da cT*/
12                     st_T = push(st_T, cT)
13                     cT = cT->sx
14                     /*start = true*/
15                 ELSE IF cT->key < k THEN
16                     /*a = cT superfluo*/
17                     st_T = push(st_T, cT)
18                     cT = cT->dx
19                     /*start = true*/
20                 ELSE
21                     st_T = push(st_T, cT)
22                     cT = cT->sx
23                     /*start = true*/
24             ELSE /*!start*/
25                 last = cT
26                 start = false
27         ELSE /*ripresa del contesto*/
28             cT = top(st_T)
29             IF cT->key > k THEN /*torno da riga 6*/
30                 ret = BEST(ret, cT, k)
31                 st_T = pop(st_T)
32                 last = cT
33                 /*start = false*/
34             ELSE IF cT->key < k THEN /*torno da riga 9*/
35                 ret = BEST(cT, ret, k)
36                 st_T = pop(st_T)
37                 last = cT
38                 /*start = false*/
39             ELSE /*torno da riga 11 o da riga 12*/
40                 IF cT->dx != last THEN /*riga 11*/
41                     IF cT->dx = NIL THEN /*ignoro la chiamata 12*/
42                         ret = BEST(ret, NIL, k)
43                         st_T = pop(st_T)
44                         last = cT
45                         /*start = false*/
46                     ELSE /*chiamata 12 da fare*/
47                         a = ret
48                         st_a = push(st_a, a)
49                         /*ho già T sullo stack*/
50                         cT = cT->dx
51                         start = true
52                     ELSE /*torno da riga 12*/

```

```

53         a = top(st_a)
54         ret = BEST(a, ret, k)
55         st_T = pop(st_T)
56         st_a = pop(st_a)
57         last = cT
58         /*start = false*/
59     RETURN ret

```

Si noti che *ret* ha sempre il valore di ritorno dell'ultima chiamata che ho fatto; quindi, non è necessario utilizzare una variabile ausiliaria per discriminare il *ret* dell'algoritmo ricorsivo con il valore di ritorno delle singole chiamate.

Esercizio: si vada a snellire il codice sopradescritto ottimizzandolo (di seguito una mia soluzione).

```

1  AlgIter(T, k)
2      cT = T
3      st_T = st_a = NIL
4      last = NIL
5      start = true
6      WHILE start || st_T != NIL DO
7          IF start THEN
8              ret = NIL
9              IF cT != NIL THEN
10                 st_T = push(st_T, cT)
11                 IF cT->key < k THEN
12                     cT = cT->dx
13                 ELSE
14                     cT = cT->sx
15             ELSE
16                 last = cT
17                 start = false
18         ELSE
19             cT = top(st_T)
20             IF cT->dx != last && cT->dx != NIL THEN
21                 a = ret
22                 st_a = push(st_a, a)
23                 cT = cT->dx
24                 start = true
25             ELSE
26                 IF cT->key > k THEN
27                     ret = BEST(ret, cT, k)
28                 ELSE IF cT->key < k THEN
29                     ret = BEST(cT, ret, k)
30                 ELSE IF cT->dx != last THEN
31                     a = top(st_a)
32                     ret = BEST(a, ret, k)
33                     st_a = pop(st_a)
34                 ELSE /*cT->dx = NIL*/
35                     ret = BEST(ret, NIL, k)
36                 last = cT
37                 st_T = pop(st_T)
38     RETURN ret

```

Lezione 8 del 23/01/2023

- Un percorso finito $\pi = v_1 v_2 \dots v_k$ in un grafo si dice *massimale* se ogni sua estensione $\pi' = \pi u$, con $u \in V$ arbitrario, non è più un percorso del grafo.

Siano dati due grafi orientati $G_1 = (V, E_1)$ e $G_2 = (V, E_2)$, rappresentati con liste di adiacenza, e due vertici $s \in V$ e $v \in V$.

Si scriva un algoritmo che verifichi in tempo lineare sui due grafi in ingresso se sono soddisfatte entrambe le seguenti condizioni:

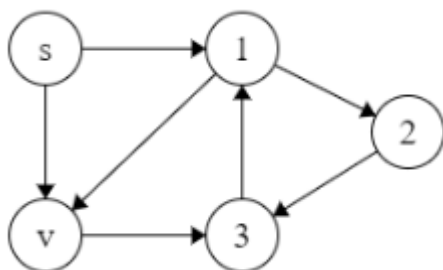
- Ogni percorso *finito massimale* in G_1 che parte da s non passa per v
- Tutti i percorsi *infiniti* in G_2 che partono da s passano per v

Si descriva prima l'idea ad alto livello, si dia poi l'algoritmo e, infine, se ne discuta la complessità.

Quando è richiesto che tutti i percorsi devono soddisfare una condizione conviene ragionare sull'opposto. Infatti, trovata la violazione che falsifichi la condizione possiamo restituire *false*.

Cerchiamo di capire come valutare il negato delle due condizioni:

- Non esiste un percorso finito massimale in G_1 che parte da s e passa per v .

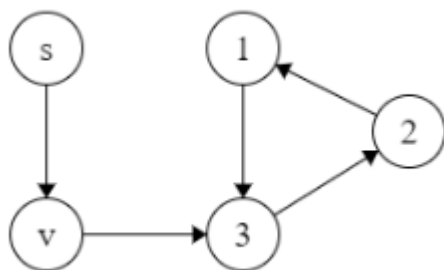


Un percorso massimale di tale grafo è $sv312$ (qualsiasi arco aggiungiamo non è più un percorso finito) N.B.: un percorso infinito non può mai essere massimale poiché è sempre possibile aggiungere vertici, esempio: $123123123123 \dots$

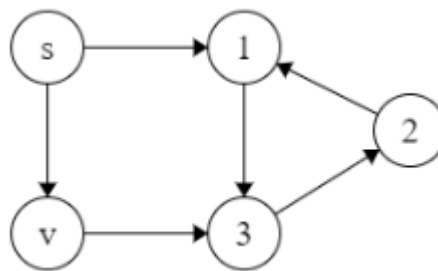
Si noti che se attraverso una visita si scopre il vertice v allora esisterà sicuramente un percorso massimale che passi per v ; dunque, il controllo di questa condizione è sostanzialmente capire se da s è possibile raggiungere v (possiamo benissimo ignorare per questa condizione la presenza di cicli).

- Non esiste un percorso infinito in G_2 che parte da s e **non** passa per v .

Soddisfa la 2:



Non soddisfa la 2:



Il nostro scopo è verificare che esiste un percorso infinito che parte da s e non passa per il nodo v . Banalmente posso colorare di nero il vertice v prima della visita, così facendo vado ad escludere tutti i percorsi che passano per quel vertice. Di conseguenza, se in tale modo troviamo un ciclo nel grafo esso sicuramente non passa per v e quindi possiamo restituire *false*.

Di seguito implementiamo gli algoritmi:

```
1  Check1(G1, s, v)
2      FOR EACH u IN G1 DO
3          Color[u] = 'b'
4      RETURN Check1_Visit(G1, s, v)
5
6  /*controlliamo la raggiungibilità*/
7  Check1_Visit(G1, u, v)
8      Color[u] = 'g'
9      FOR EACH w IN Adj[u] DO
10         IF w = v THEN
11             /*abbiamo trovato un percorso che passa per v*/
12             RETURN false
13         /*altrimenti proseguo con la visita*/
14         ELSE IF Color[w] = 'b' THEN
15             ret = Check1_Visit(G1, w, v)
16             IF !ret THEN
17                 RETURN false
18         Color[u] = 'n'
19     RETURN true

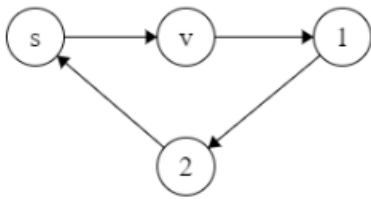
1  Check2(G2, s, v)
2      FOR EACH u IN G2 DO
3          Color[u] = 'b'
4          Color[v] = 'n'
5      RETURN Check2_Visit(G2, s, v)
6
7  /*cerchiamo un ciclo che non passi per s*/
8  Check1_Visit(G2, s)
9      Color[s] = 'g'
10     FOR EACH u IN Adj[s] DO
11         IF IF Color[u] = 'g' THEN
12             /*abbiamo trovato un arco di ritorno*/
13             RETURN false
14         ELSE IF Color[u] = 'b' THEN
15             ret = Check2_Visit(G2, u, v)
16             IF !ret THEN
17                 RETURN false
18     Color[s] = 'n'
19     RETURN true
```

L'algoritmo finale si riduce a:

```
1  Algo(G1, g2, s, v)
2      IF Check1(G1, s, v)
3          RETURN Check2(G2, s, v)
4      RETURN false
```

Ma se la condizione due fosse: “tutti i percorsi infiniti in G_2 che partono da s passano **infinite** volte per v ?” Ciò significa che v debba far parte del ciclo, con la ovvia conseguenza di non poter applicare la tecnica della colorazione in nero vista precedentemente. Per capire se un nodo appartenga ad un ciclo non basta la relazione di adiacenza, bisogna interpellare anche i tempi di inizio e fine visita. Infatti, posso capire se un

nodo fa parte di un ciclo confrontando i suoi tempi con il primo nodo grigio trovato:



- $d[s] = 1$
- $d[v] = 2$
- $d[1] = 3$
- $d[2] = 4$
- $f[s] = 8$
- $f[v] = 7$
- $f[1] = 6$
- $f[2] = 5$

Si potrebbero utilizzare ma definire un algoritmo di questo tipo potrebbe essere molto complesso e non è nemmeno sicuro sia possibile definirlo in tempo lineare.

Si noti che possiamo sfruttare le componenti fortemente connesse. Infatti, sappiamo che se v è in una CFC allora sicuramente esisterà un ciclo (in quella componente) che passerà infinite volte per v . Non abbiamo bisogno di sapere che v faccia parte di un ciclo, poiché possiamo verificarlo attraverso le CFC. Fatto ciò, dobbiamo verificare che s non raggiunga altre componenti connesse, il che si riduce a controllare nel grafo delle componenti fortemente connesse che s raggiunga solo la CFC di v (il che basti verificare che ci sia un altro nodo nero nel grafo delle componenti).

Consiglio: verificare l'appartenenza di un vertice in un ciclo è sempre risolvibile attraverso una CFC.

Esercizio: provare a definire l'algoritmo descritto sopra.

```

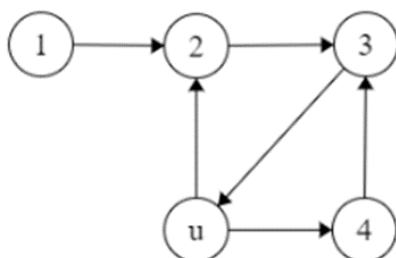
1  MiaSoluzione(G, s, v)
2      O = DFS1(G)
3      Gt = GrafoTrasposto(G)
4      DSF2(Gt, O)
5      /*Controllo che s raggiunga solo la CFC di v*/
6      Init(G)
7      DFS_Visit(G, s) /*semplice visita DFS*/
8      FOR EACH u IN G DO
9          IF Color[u] = 'n' && CFC[u] != CFC[v] THEN
10             RETURN false
11     RETURN true

```

} Vedi capitolo "[Calcolo delle CFC](#)"

- Sia dato un grafo orientato G , un array $VAL[\cdot]$ che associa a ogni vertice v un *numero naturale* $VAL[v]$, e un vertice u del grafo. Un percorso semplice in G si dice **massimale** se non può essere ulteriormente esteso mantenendo la proprietà di essere semplice.

Si definisca un algoritmo che, **in tempo lineare sulla dimensione del grafo** G , verifichi se tutti i percorsi semplici massimali che si dipartono dal vertice u passano necessariamente da due vertici cui sono associati numeri di differente parità.



| | | | | | |
|--------|---|---|---|---|-----|
| | 1 | 2 | 3 | 4 | u |
| $VAL:$ | 2 | 5 | 7 | 4 | 6 |

I valori di parità sono rispettivamente:

0 1 1 0 0

Ci viene chiesto di verificare se tutti i percorsi massimali che partono da u partono per almeno due vertici con parità differente (ad esempio con $u43$ trovo valori di parità differente 6,4,7; ovvero, due numeri pari ed uno dispari). Possiamo associare 1 a valori pari e 0 a quelli dispari (poiché a noi interessa la disparità).

Neghiamo la condizione: “Non esiste nessun percorso semplice massimale che si diparte da u e passa solo da vertici con stessa parità di u ”. Quindi; se troviamo un percorso massimale dove $\forall v \in \pi : \text{val}[v] \% 2 = \text{val}[u] \% 2$ possiamo restituire false.

Vediamo come procedere:

- 1) Capire i nodi terminali (grado entrante pari a 0) raggiungibili da u e marcarli (tenerli da parte). Così facendo posso poi partire da questi nodi andandomi a calcolare il percorso a retroso e confrontare ogni nodo di quel percorso con la parità di u
- 2) Costruire il grafo trasposto. Si noti che ora il nostro problema è diventato un problema di raggiungibilità. Dai nostri nodi terminali (che sappiamo già raggiungeranno u) poi posso controllare il loro valore nell'array VAL
- 3) Posso “bloccare” i nodi con parità diversa da quella di u semplicemente colorandoli di nero. Così se raggiunge u so che la condizione della traccia è falsa.
- 4) Per ogni nodo rosso (useremo questo colore per marcare i nodi terminali), vedere se raggiunge u nel trasposto.
- 5) Se trovo anche un solo percorso che da un nodo terminale arriva ad u , allora la condizione è falsificata e ritorno false. True altrimenti.

Esercizio: implementare l'idea ad alto livello descritta precedentemente.

```

1  MiaSoluzione(G, VAL, u)
2      FOR EACH v IN V DO
3          Color[v] = 'b'
4      MarcaTerminali(G, u)
5      Gt = GrafoTrasposto(G)
6      FOR EACH v IN Vt DO
7          IF VAL[v]%2 != VAL[u]%2 THEN
8              Color[v] = 'n'
9          ELSE
10             Color[v] = 'b'
11     FOR EACH v IN V DO
12         IF Color[v] = 'r' THEN
13             DFS_Visit(Gt, v)
14             IF Color[u] = 'n' THEN
15                 RETURN false
16     RETURN true

```

```

1  MarcaTerminali(G, v)
2      Color[v] = 'g'
3      FOR EACH u IN Adj[v] DO
4          IF Color[u] = 'b' THEN
5              MarcaTerminali(G, u)
6          ELSE IF Color[u] != 'r' THEN
7              Color[v] = 'r'
8  IF Color[v] != 'r' THEN
9      Color[v] = 'n'

```

Svolti da me

Di seguito ci sono dei miei tentativi di soluzione delle prove d'esame, non garantisco sulla loro correttezza.
P.S.: Da qui in poi il documento non è stato revisionato. Formicola si esonera da eventuali errori.

Esercizi assegnati dal tutor

- 1) Dato un ABR T , due chiavi $k_1 \leq k_2$ ed un intero c , rimuovere da T tutti i nodi con chiave k compresa tra k_1 e k_2 ($k_1 \leq k \leq k_2$) e nei cui sottoalberi ci siano al più c nodi da rimuovere.

```
1  /*restituisce il numero di nodi che è possibile rimuovere dall'albero T*/
2  Algo(T, k1, k2, c, P)
3      rmv = 0
4      IF T != NIL THEN
5          IF T->key > k2 THEN
6              rmv = Algo(T->sx, k1, k2, c, T)
7          ELSE IF T->key < k1 THEN
8              rmv = Algo(T->dx, k1, k2, c, T)
9          ELSE
10             rmv = Algo(T->sx, k1, k2, c, T)
11             rmv = rmv + Algo(T->dx, k1, k2, c, T)
12             IF rmv <= c THEN
13                 IF T = P->sx THEN
14                     P->sx = RimuoviNodo(T)
15                 ELSE
16                     P->dx = RimuoviNodo(T)
17                 /*questo nodo è rimovibile*/
18                 rmv = rmv + 1
19             RETURN rmv
20
21 AlgoRoot(T, k1, k2, c)
22     IF T != NIL THEN
23         rmv = Algo(T->sx, k1, k2, c, T)
24         rmv = rmv + Algo(T->dx, k1, k2, c, T)
25         IF rmv <= c AND T->key >= k1 AND T->key <= k2 THEN
26             T = RimuoviNodo(T)
27     RETURN T
```

- 2) Dato un ABR T e due interi $h_1 \leq h_2$, rimuovere dall'albero tutte le chiavi pari che si trovino a distanza compresa tra h_1 ed h_2 dalla radice.

```
1  Algo(T, h1, h2, P)
2      IF T != NIL THEN
3          T->sx = Algo(T->sx, h1 - 1, h2 - 1, T)
4          T->dx = Algo(T->dx, h1 - 1, h2 - 1, T)
5          IF h1 <= 0 AND h2 >= 0 THEN
6              IF (T->key) % 2 = 0 THEN
7                  T = RimuoviNodo(T)
8          RETURN T
9
10 AlgoRoot(T, h1, h2)
11     IF T != NIL AND h1 <= h2 THEN
12         T->sx = Algo(T->sx, h1 - 1, h2 - 1, T)
13         T->dx = Algo(T->dx, h1 - 1, h2 - 1, T)
14         IF T->key % 2 = 0 AND h1 = 0 THEN
15             T = RimuoviNodo(T)
16     RETURN T
```


- 3) Dato un ABR T ed un intero d , rimuovere dall'albero tutti i nodi con chiave dispari e che si trovino a distanza da una foglia maggiore di d . **Hint:** la distanza di un nodo da una foglia è la lunghezza del percorso più breve da quel nodo ad una qualunque foglia dell'albero.

```

1  /*restituisce la distanza di T da una foglia*/
2  Algo(T, d, P)
3      dist = -1
4      IF T != NIL THEN
5          dist_sx = Algo(T->sx, d, T)
6          dist_dx = Algo(T->dx, d, T)
7          dist = 1 + min(dist_sx, dist_dx)
8          IF dist > d AND (T->key)%2 != 0 THEN
9              IF T != P->sx THEN
10                 P->dx = RimuoviNodo(T)
11             ELSE
12                 P->sx = RimuoviNodo(T)
13          Return dist
14
15  AlgoRoot(T, d)
16      IF T != NIL THEN
17          dist_sx = Algo(T->sx, d, T)
18          dist_dx = Algo(T->dx, d, T)
19          dist = 1 + min(dist_sx, dist_dx)
20          IF dist > d AND (T->key)%2 != 0 THEN
21              T = RimuoviNodo(T)
22      RETURN T

```

- 4) Si risolva la seguente equazione di ricorrenza, calcolandone l'andamento asintotico:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ T(\sqrt[3]{n}) + T(\sqrt[3]{n^2}) + \log n & \text{altrimenti} \end{cases}$$

| Livello | | Contributi singoli | Contributo totale |
|---------|--|--|--|
| 0 | $T(n) = T\left(n^{\frac{1}{3}}\right) + T\left(n^{\frac{2}{3}}\right) + \log n$ | $\log n$ | $\log n$ |
| 1 | $T\left(n^{\frac{1}{3}}\right) = T\left(n^{\frac{1}{9}}\right) + T\left(n^{\frac{2}{9}}\right) + \log n^{\frac{1}{3}}$ $T\left(n^{\frac{2}{3}}\right) = T\left(n^{\frac{2}{9}}\right) + T\left(n^{\frac{4}{9}}\right) + \log n^{\frac{2}{3}}$ | $\log n^{\frac{1}{3}}, \log n^{\frac{2}{3}}$ | $\frac{1}{3}\log n + \frac{2}{3}\log n$ $= \log n$ |
| 2 | per $n^{\frac{1}{3}}$ $\begin{cases} T\left(n^{\frac{1}{9}}\right) = T\left(n^{\frac{1}{27}}\right) + T\left(n^{\frac{2}{27}}\right) + \log n^{\frac{1}{9}} \\ T\left(n^{\frac{2}{9}}\right) = T\left(n^{\frac{2}{27}}\right) + T\left(n^{\frac{4}{27}}\right) + \log n^{\frac{2}{9}} \end{cases}$ per $n^{\frac{2}{3}}$ $\begin{cases} T\left(n^{\frac{2}{9}}\right) = T\left(n^{\frac{2}{27}}\right) + T\left(n^{\frac{4}{27}}\right) + \log n^{\frac{2}{9}} \\ T\left(n^{\frac{4}{9}}\right) = T\left(n^{\frac{4}{27}}\right) + T\left(n^{\frac{8}{27}}\right) + \log n^{\frac{4}{9}} \end{cases}$ | $\log n^{\frac{1}{9}}, \log n^{\frac{2}{9}}, \log n^{\frac{2}{9}}, \log n^{\frac{4}{9}}$ | $\frac{1}{9}\log n + \frac{2}{9}\log n$ $+ \frac{2}{9}\log n$ $+ \frac{4}{9}\log n = \log n$ |

Il ramo estremo sinistro è quello con altezza minima, dove per un generico livello i avremo un input di $n^{\frac{1}{3^i}}$.

Il ramo più a destra, invece, è quello con altezza massima, dove al livello i avremo un input di $n^{\frac{2^i}{3^i}}$.

$$n^{\frac{1}{3^{h_{\min}}}} = 2 \Rightarrow \frac{1}{3^{h_{\min}}} \log_2 n = 1 \Rightarrow h_{\min} = \log_3 \log_2 n$$

$$\begin{aligned} \frac{2^{h_{\max}}}{n^{3^{h_{\max}}}} = 2 &\Rightarrow \frac{2^{h_{\max}}}{3^{h_{\max}}} \log_2 n = 1 \Rightarrow \log_2 n = \frac{3^{h_{\max}}}{2^{h_{\max}}} \Rightarrow h_{\max} = \log_3 \log_2 n = \frac{\log_3 \log_2 n}{\log_3 \frac{3}{2}} \Rightarrow \\ &\Rightarrow h_{\max} = \frac{1}{\underbrace{1 - \log_3 2}_{>0}} \log_3 \log_2 n \end{aligned}$$

$$T_{\min}(n) \leq T(n) \leq T_{\max}(n)$$

$$\sum_{i=0}^{h_{\min}} \log n = \log n (h_{\min} + 1) = \log n \log_3 \log_2 n + \log n \leq T(n) \Rightarrow T(n) = \Omega(\log n \cdot \log_3 \log_2 n)$$

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{h_{\max}} \log n = \log n (h_{\max} + 1) = \frac{1}{1 - \log_3 2} \log n \log_3 \log_2 n + \log n \Rightarrow T(n) \\ &= O(\log n \cdot \log_3 \log_2 n) \end{aligned}$$

Dunque, possiamo concludere che $T(n) = \Theta(\log n \cdot \log_3 \log_2 n)$

Esercizio 1 del 21/07/2022

Si individuino, nel caso esistano, le costanti moltiplicative atte a mostrare la seguente relazione asintotica: $\log_2(n^{2n}) + n - \log_2(n) = \Theta(\log_2(n^n))$. In caso contrario, mostrare la falsità della relazione.

Soluzione:

$$\lim_{n \rightarrow \infty} \frac{\log n^{2n} + n - \log n}{\log n^n} = \lim_{n \rightarrow \infty} \frac{2n \log n}{n \log n} + \lim_{n \rightarrow \infty} \frac{n}{n \log n} - \lim_{n \rightarrow \infty} \frac{\log n}{n \log n} = 2 + 0 - 0 = 2$$

Poiché il limite del rapporto tende ad una costante > 0 la relazione asintotica è verificata. Inoltre, essendo 2 un asintoto orizzontale per il precedente rapporto, tale valore sarà anche una nostra costante.

Studiamo la positività della derivata prima di $\frac{\log n^{2n} + n - \log n}{\log n^n}$:

$$\frac{d}{dn} \left(\frac{\log n^{2n} + n - \log n}{\log n^n} \right) = \frac{d}{dn} (2) + \frac{d}{dn} \left(\frac{1}{\log n} \right) - \frac{d}{dn} \left(\frac{1}{n} \right) = \frac{d}{dn} (\log^{-1} n) - \frac{d}{dn} (n^{-1})$$

$$\frac{d}{dn} ((\log n)^{-1}) = -\frac{1}{(\log n)^2} \cdot \frac{d}{dn} (\log n) = -\frac{1}{(\log n)^2} \left(\frac{1}{n} \log e \right) = -\frac{\log e}{n(\log n)^2}$$

$$\frac{d}{dn} \left(\frac{\log n^{2n} + n - \log n}{\log n^n} \right) = -\frac{\log e}{n(\log n)^2} - \left(-\frac{1}{n^2} \right) = \frac{1}{n^2} - \frac{\log e}{n(\log n)^2}$$

Nota: $\log e \cong 1,4$

$$\frac{1}{n^2} - \frac{\log e}{n(\log n)^2} > 0 \Rightarrow \frac{1}{n^2} > \frac{\log e}{n(\log n)^2} \Rightarrow \frac{(\log n)^2}{n} > \log e \Rightarrow \left(\frac{1}{\sqrt{n}} \log n \right)^2 > \log e \Rightarrow \frac{\log n}{\sqrt{n}} > \sqrt{\log e}$$

La precedente relazione non è mai verificata poiché supponendo $n = 2^{2x}$ avremo $\frac{2x}{2^x}$ che è sempre un valore minore o uguale ad 1 per ogni $n \geq 1$ ma $\sqrt{\log e} \cong 1,2$; dunque la funzione sarà decrescente nell'intervallo $[1, +\infty)$.

A questo punto abbiamo tutte le informazioni:

- $\forall n \geq 1$ la funzione è decrescente e quindi incontrerà l'asintoto 2 dall'alto ($c_1 = 2$)
- L'altra costante possiamo ricavarla scegliendo $n_1 = 2$ allora $h(2) = \frac{4+2-1}{2} = 2,5$ ($c_2 = 2,5$)

Ma allora $2 \log_2(n^n) \leq \log_2(n^{2n}) + n - \log_2(n) \leq 2,5 \log_2(n^n) \quad \forall n \geq 2$ come volevasi dimostrare.

Esercizio 1 del 13/01/2021

Si verifichi, calcolando le costanti necessarie se esistono, la seguente relazione asintotica:

$$\log_2\left(\frac{4^n}{n^2}\right) = \Theta(\log_3(2^{2n}))$$

Esplicitare per esteso il procedimento di soluzione seguito.

Soluzione:

$$\log_3(2^{2n}) = \frac{\log_2(2^{2n})}{\log_2 3} = \frac{2}{\log_2 3} n = \Theta(n)$$

Infatti, $\frac{2}{\log_2 3} \cong 1,6$ e banalmente $\forall n \geq 1, n \leq \frac{2}{\log_2 3} n \leq 2n$. Dunque, la soluzione si riduce a calcolare delle costanti per cui vale $\log_2\left(\frac{4^n}{n^2}\right) = \Theta(n)$ così da avere per transitività le costanti richieste, essendo:

$$\forall n \geq 1, \quad n \leq \frac{2}{\log_2 3} n \leq 2n \Rightarrow \left(\frac{1}{2}\right) \frac{2}{\log_2 3} n \leq n \leq \frac{2}{\log_2 3} n$$

Allora se $\log_2\left(\frac{4^n}{n^2}\right) = \Theta(n)$ significa che

$$\exists n'_0 \geq 0, \exists c'_1, c'_2 > 0: \forall n \geq n_0 \quad c'_1 n \leq \log_2\left(\frac{4^n}{n^2}\right) \leq c'_2 n$$

E quindi risulterà per transitività che

$$\exists n_0 = \max\{1, n'_0\}, \exists c_1 = \frac{c'_1}{2}, c_2 = c'_2: \quad \forall n \geq n_0 \quad c_1 \frac{2}{\log_2 3} n \leq \log_2\left(\frac{4^n}{n^2}\right) \leq c_2 \frac{2}{\log_2 3} n$$

Ovvero $\log_2\left(\frac{4^n}{n^2}\right) = \Theta(\log_3(2^{2n}))$.

Risulta $\log_2\left(\frac{4^n}{n^2}\right) = \Theta(n)$ poiché

$$\lim_{n \rightarrow \infty} \frac{\log \frac{4^n}{n^2}}{n} = \lim_{n \rightarrow \infty} \frac{n \log 2^2 - 2 \log n}{n} = 2 - 2 \underbrace{\lim_{n \rightarrow \infty} \frac{\log n}{n}}_0 = 2$$

$\log n$ cresce meno velocemente di n

Dunque, 2 sarà asintoto orizzontale per la funzione $h(n) = \frac{\log \frac{4^n}{n^2}}{n}$ nonché una delle nostre costanti.

Studiamo la derivata prima di $h(n)$ così da ricavare il resto delle informazioni:

$$\frac{d}{dn} \left(2 - \frac{\log n}{n} \right) = - \frac{d}{dn} \left(\frac{\log n}{n} \right) = - \frac{1}{n^2} \left(\log e \cdot n - \log n \right) = \frac{\log n - \log e}{n^2}$$

Dunque, $h'(n) > 0$ per $\log n \geq \log e \Rightarrow n \geq e (\cong 2,72)$. Ma allora la funzione è crescente nell'intervallo $[3, +\infty)$ e quindi $n'_0 = 3, c'_1 = h(3) \cong 1,4, c'_2 = 2$ (poiché la funzione tenderà all'asintoto dal basso).

Ma allora $\exists n_0 = 3, \exists c_1 = 0,85, c_2 = 2: \quad \forall n \geq 3 \quad 0,85 \frac{2}{\log_2 3} n \leq \log_2\left(\frac{4^n}{n^2}\right) \leq 2 \frac{2}{\log_2 3} n$.

Esercizio 1 del 04/03/2020

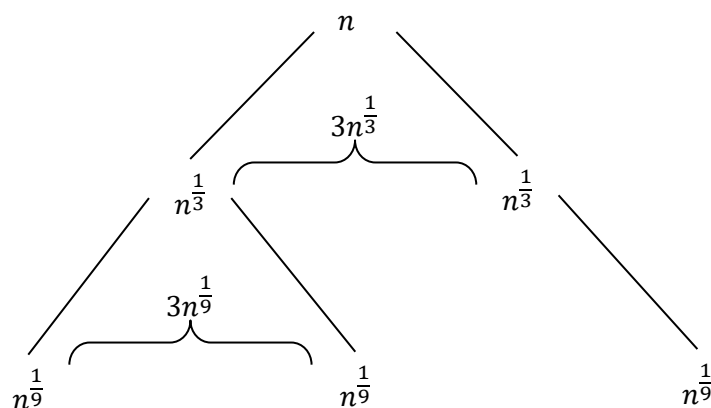
Si risolva la seguente equazione di ricorrenza, utilizzando il metodo degli alberi di ricorrenza:

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

Soluzione:

Dal caso ricorsivo $T(n) = 3n^{\frac{1}{3}} \cdot T\left(n^{\frac{1}{3}}\right) + n^{\frac{1}{2}}$ ricaviamo:

| Livello | Taglia dell'input | Contributo singolo nodo | Numero di rami | Contributo Totale |
|---------|--|---|---|--|
| 0 | n | $n^{\frac{1}{2}}$ | 1 | $n^{\frac{1}{2}}$ |
| 1 | $n^{\frac{1}{3}}$ | $\left(n^{\frac{1}{3}}\right)^{\frac{1}{2}} = n^{\frac{1}{6}}$ | $3n^{\frac{1}{3}}$ | $3n^{\frac{1}{3}} \cdot n^{\frac{1}{6}} = 3n^{\frac{3}{6}} = 3n^{\frac{1}{2}}$ |
| 2 | $\left(n^{\frac{1}{3}}\right)^{\frac{1}{3}} = n^{\frac{1}{9}}$ | $\left(n^{\frac{1}{9}}\right)^{\frac{1}{2}} = n^{\frac{1}{18}}$ | $3n^{\frac{1}{3}} \cdot \left(3n^{\frac{1}{3}}\right)^{\frac{1}{3}} = 9n^{\frac{4}{9}}$ | $9n^{\frac{4}{9}} \cdot n^{\frac{1}{18}} = 9n^{\frac{9}{18}} = 9n^{\frac{1}{2}}$ |
| ... | | | | |
| i | $n^{\left(\frac{1}{3}\right)^i}$ | | | $3^i n^{\frac{1}{2}}$ |
| ... | | | | |



L'altezza dell'albero sarà la soluzione della seguente equazione:

$$n^{\left(\frac{1}{3}\right)^h} = 3 \Rightarrow \log_3 n^{\frac{1}{3^h}} = \log_3 3 \Rightarrow \frac{1}{3^h} \log_3 n = 1 \Rightarrow 3^h = \log_3 n \Rightarrow h = \log_3(\log_3 n)$$

Dunque:

$$T(n) = \sum_{i=0}^h 3^i n^{\frac{1}{2}} = n^{\frac{1}{2}} \sum_{i=0}^h 3^i = n^{\frac{1}{2}} \cdot \frac{3^{h+1} - 1}{2} = \frac{n^{\frac{1}{2}}}{2} (3^{\log_3(\log_3 n)} \cdot 3) - \frac{n^{\frac{1}{2}}}{2} = \frac{3}{2} n^{\frac{1}{2}} \log n - \frac{n^{\frac{1}{2}}}{2} = \Theta(\sqrt{n} \log n)$$

Esercizio 1 del 21/06/2022

Si individuino, nel caso esistano, le **costanti moltiplicative** atte a mostrare la seguente relazione asintotica:

$$\ln\left(\frac{n}{e}\right) = \Theta(\ln(n^e))$$

(si ricorda che con 'ln' si indica il logaritmo naturale e con 'e' la costante di Nepero). In caso contrario, mostrare la falsità della relazione.

Soluzione:

La relazione precedente è vera, essendo:

$$\lim_{n \rightarrow \infty} \frac{\ln \frac{n}{e}}{\ln n^e} = \lim_{n \rightarrow \infty} \frac{\ln n - \ln e}{e \ln n} = \frac{1}{e} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{\ln n}\right) = \frac{1}{e} > 0$$

Allora $\exists c_1, c_2, n_0 > 0 : \forall n > n_0, c_1 \ln(n^e) \leq \ln\left(\frac{n}{e}\right) \leq c_2 \ln(n^e)$ che è equivalente alla seguente relazione:

$$c_1 e \ln n \leq \ln n - 1 \leq c_2 e \ln n$$

Cominciamo con c_1 :

$$c_1 e \ln n \leq \ln n - 1 \xrightarrow{\frac{1}{2} \ln n \geq 1 \Rightarrow \ln n \geq 2 \Rightarrow n \geq e^2} c_1 e \ln n \leq \ln n - \frac{1}{2} \ln n \leq \ln n - 1 \Rightarrow c_1 \leq \frac{1}{2e} \quad \forall n \geq e^2$$

Mentre per c_2 avremo:

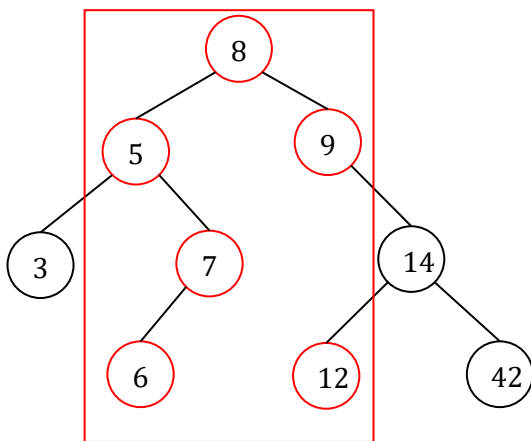
$$\ln n - 1 \leq c_2 e \ln n \xrightarrow{\ln n - 1 \leq \ln n \quad \forall n > 0} \ln n - 1 \leq \ln n \leq c_2 e \ln n \Rightarrow c_2 \geq \frac{1}{e}$$

Dunque, per $c_1 = \frac{1}{2e}, c_2 = \frac{1}{e}$ e $n_0 = e^2$ risulta vera la relazione $\ln\left(\frac{n}{e}\right) = \Theta(\ln(n^e))$.

Esercizio 2 del 21/06/2022

Si scriva un **algoritmo ricorsivo** che, dati in ingresso un albero binario di ricerca su interi T e due valori $k_1, k_2 \in \mathbb{N}$, inserisca in una lista L le chiavi k contenute in T comprese tra k_1 e k_2 ($k_1 \leq k \leq k_2$), in modo che al termine L contenga valori ordinati in modo decrescente. Tale algoritmo dovrà avere **complessità lineare** nella dimensione dell'albero.

Soluzione:



Se $k_1 = 5$ e $k_2 = 13$ tolto il caso base dove $T = \text{NIL}$ e quindi non ho nulla da fare posso imbattermi nei seguenti casi:

- Il nodo ha chiave $k < k_1$: vado solo a destra (succede al nodo 3)
- Il nodo ha chiave $k > k_2$ vado solo a sinistra (vedi nodo 14)
- Se il nodo ha chiave $k_1 \leq k \leq k_2$:
 - Vado a destra
 - Inserisco il nodo
 - Vado a sinistra

Applichiamo praticamente una visita in order "inversa", infatti visitando prima il sottoalbero destro (nodi tutti \geq della radice) poi la radice stessa e infine il sottoalbero sinistro (nodi \leq della radice) avrò una visita in ordine decrescente. Nell'esempio precedente avremo il seguente ordinamento: (12,9,8,7,6,5)

```

1  CreaLista(T, k1, k2)
2      L = NIL
3      IF T != NIL THEN
4          IF T->key < k1 THEN
5              L = CreaLista(T->dx, k1, k2)
6          ELSE IF T->key > k2 THEN
7              L = CreaLista(T->sx, k1, k2)
8          ELSE
9              L = CreaLista(T->dx, k1, k2)
10             L = insert(L, T->key)
11             L = CreaLista(T->sx, k1, k2)
12     RETURN L

```

```

insert(L, k)
x = allocaNodoLista()
x->key = k
x->next = L
RETURN x

```

Esercizio 1 del 29/03/2019

Si dimostri per esteso la verità o la falsità della seguente affermazione:

$$\sum_{i=1}^n \log_2 i = \Theta(n \log_2 n)$$

Soluzione: Poniamo $f(n) = \sum_{i=1}^n \log_2 i$

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{f(n)}{n \log n} &= \lim_{n \rightarrow \infty} \left(\frac{1}{n \log n} \sum_{i=1}^n \log i \right) \xrightarrow{\log a + \log b = \log(a \cdot b)} \lim_{n \rightarrow \infty} \left(\frac{1}{n \log n} \log \left(\prod_{i=1}^n i \right) \right) = \\ &= \lim_{n \rightarrow \infty} \left(\frac{1}{n \log n} \log(n!) \right) \xrightarrow{\log(n!) = \Theta(n \log n)} \lim_{n \rightarrow \infty} \left(\frac{n \log n}{n \log n} \right) = 1 \end{aligned}$$

Per maggiori informazioni sulla risoluzione del fattoriale si veda [l'approssimazione di Stirling](#).

Poiché la relazione è verificata, mostriamo che esistono le costanti per cui:

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

Cominciamo con la costante c_2 , $\forall n \geq 1$ risulta:

$$\sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \log n \Rightarrow f(n) \leq n \log n \leq c_2 n \log n \Rightarrow c_2 \geq 1$$

Per la costante c_1 procediamo nel seguente modo:

$$\sum_{i=1}^n \log i = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \log i + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^n \log i \geq \sum_{i=\lfloor \frac{n}{2} \rfloor}^n \log i \geq \sum_{i=\lfloor \frac{n}{2} \rfloor}^n \log \frac{n}{2} = \log \frac{n}{2} \sum_{i=\lfloor \frac{n}{2} \rfloor}^n 1 = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} \log n - \frac{n}{2}$$

Inoltre, essendo $\frac{n}{2} \leq \frac{n}{4} \log n \quad \forall n \geq 4$ si ha $\frac{n}{2} \log n - \frac{n}{2} \geq \frac{n}{2} \log n - \frac{n}{4} \log n \Rightarrow f(n) \geq \frac{n}{4} \log n$; dunque:

$$c_1 n \log n \leq \frac{n}{4} \log n \leq f(n) \Rightarrow c_1 \leq \frac{1}{4} \quad \forall n \geq 4$$

Ma allora per $n_0 = 4, c_1 = \frac{1}{4}, c_2 = 1$ vale la relazione $f(n) = \Theta(n \log n)$

Traccia B del 16/06/2021

Scrivere un algoritmo iterativo che simuli precisamente il comportamento ricorsivo dell'algoritmo sotto riportato, dove i parametri i e j sono valori interi, A un array di interi e x un valore Booleano.

```
1  Algoritmo(A, i, j, x)
2      IF j - i >= 1 THEN
3          y = Rand() % 2
4          IF x = 1 THEN
5              ret = Algoritmo(A, (i + j)/2 + 1, j, y)
6              IF ret % 2 = 0 THEN
7                  ret = Algoritmo(A, i, (i + j)/2, 1 - y)
8          ELSE
9              ret = Algoritmo(A, i, (i + j)/2, y)
10             IF ret % 2 = 1 THEN
11                 ret = ret + Algoritmo(A, (i + j)/2 + 1, j, 1 - y)
12     ELSE
13         ret = A[i]
```

Soluzione:

```
1  AlgoritmoIter(A, i, j, x)
2      ci = i
3      cj = j
4      cx = x
5      st_i = st_j = NIL
6      st_x = NIL
7      st_ret = NIL
8      last = NIL
9      start = true
10     WHILE start || st_x != NIL DO
11         IF start THEN
12             IF cj - ci >= 1 THEN
13                 y = Rand()%2
14                 st_x = push(st_x, cx)
15                 st_i = push(st_i, ci)
16                 st_j = push(st_j, cj)
17                 IF cx = 1 THEN
18                     ci = (ci + cj)/2 + 1
19                 ELSE
20                     cj = (ci + cj)/2
21                     cx = y
22             ELSE
23                 ret = A[ci]
24                 lret = ret
25                 last = cj
26                 start = false
27         ELSE
28             cx = top(st_x)
29             cj = top(st_j)
30             ci = top(st_i)
31             IF cx = 1 && last = cj THEN
32                 y = cx
33                 ret = lret
34             IF ret%2 = 0 THEN
35                 cj = (ci + cj)/2
36                 cx = 1 - y
37                 start = true
38             ELSE
39                 ELSE IF cx = 1 THEN
40                     ret = lret
41                     last = cj
42                     st_i = pop(st_i)
43                     st_j = pop(st_j)
44                     st_x = pop(st_x)
45                 ELSE IF last != cj THEN
46                     ret = lret
47                     IF ret%2 = 1 THEN
48                         y = cx
49                         st_ret = push(st_ret, ret)
```

Nella linea 39 andrebbe questo blocco per essere precisi. Ma è superfluo poiché le seguenti variabili hanno già i valori assegnati

```
ELSE
    lret = ret
    last = cj
    start = false
```

```

50         ci = (ci + cj)/2 + 1
51         cx = 1 - y
52         start = true
53     ELSE
54         last = cj
55 ELSE
56     ret = top(st_ret)
57     ret = ret + lret
58     lret = ret
59     st_i = pop(st_i)
60     st_ret = pop(st_ret)
61     st_j = pop(st_j)
62     st_x = pop(st_x)
63 RETURN ret

```

Esercizio 3 del 29/03/2019

Sia dato un grafo orientato $G = (V, E)$, rappresentato con liste di adiacenza, e un array A contenente un sottoinsieme dei vertici di G (quindi, $A \subseteq V$) e due vertici $u, v \in V$. Si scriva un algoritmo che, dati in ingresso unicamente G, A, u e v , calcoli, in **tempo lineare sulla dimensione di G** , l'insieme di vertici $A' \subseteq A$ contenente **tutti e soli** i vertici di A che soddisfano la seguente condizione:

- $a \in A'$ se e solo se esiste un percorso π di G che parte da u , arriva a v e passa per a , cioè π è della forma $u \rightsquigarrow a \rightsquigarrow v$.

Soluzione: faremo una visita DFS su G che parte da u . Dopo questa DFS seguirà un'altra DFS sul grafo trasposto precedentemente costruito partendo stavolta da v . Una volta terminate le due DFS i vertici che rispetteranno la condizione saranno quelli colorati di nero in entrambi gli array di colori. Infatti, un vertice sarà doppio nero se e solo se è raggiunto da u e raggiunge v .

```

1  Algo(G, A, u, v)
2      Init(G)
3      DFS_Visit(G, u)
4      A' = NIL
5      IF Color[v] = 'n' THEN
6          Gt = GrafoTrasposto(G)
7          Init(Gt)
8          DFS_Visit(Gt, v)
9          FOR EACH a IN A DO
10             IF Color[a] = 'n' && Color_t[a] = 'n' THEN
11                 A' = add(A', a)
12      RETURN A'

1  GrafoTrasposto(G)
2      Vt = NIL
3      n = 0
4      FOR EACH v IN V DO
5          Vt = add(Vt, v)
6          n = n + 1
7      Adj_t = allocaArray(n)
8      FOR EACH v IN V DO
9          FOR EACH u IN Adj[v] DO
10             Adj_t[u] = Insert(Adj_t[u], v)
11      Gt = costruisciGrafo(Vt, Adj_t)
12      RETURN Gt

1  DFS_Visit(G, s)
2      Color[s] = 'g'
3      FOR EACH v IN Adj[s] DO
4          IF Color[v] = 'b' THEN
5              DFS_Visit(G, v)
6      Color[v] = 'n'

```


Esercizio 1 del 25/03/2022

Si risolva la seguente equazione di ricorrenza, utilizzando il metodo degli alberi di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ 3 \cdot T(\sqrt{n}) + 2 \cdot T(\sqrt[4]{n}) + \log n & \text{se } n > 3 \end{cases}$$

Soluzione:

| Livello | | Numero di rami | Contributo Totale |
|---------|---|----------------------------|---|
| 0 | $T(n) = 3T\left(n^{\frac{1}{2}}\right) + 2T\left(n^{\frac{1}{4}}\right) + \log n$ | 1 | $\log n = 2^0 \log n$ |
| 1 | $T\left(n^{\frac{1}{2}}\right) = 3T\left(n^{\frac{1}{4}}\right) + 2T\left(n^{\frac{1}{8}}\right) + \frac{1}{2} \log n$ $T\left(n^{\frac{1}{4}}\right) = 3T\left(n^{\frac{1}{8}}\right) + 2T\left(n^{\frac{1}{16}}\right) + \frac{1}{4} \log n$ | 3 + 2 | $3 \cdot \frac{1}{2} \log n + 2 \cdot \frac{1}{4} \log n$ $= 2 \log n$ |
| 2 | per $n^{\frac{1}{2}}$ $\begin{cases} T\left(n^{\frac{1}{4}}\right) = 3T\left(n^{\frac{1}{8}}\right) + 2T\left(n^{\frac{1}{16}}\right) + \frac{1}{4} \log n \\ T\left(n^{\frac{1}{8}}\right) = 3T\left(n^{\frac{1}{16}}\right) + 2T\left(n^{\frac{1}{32}}\right) + \frac{1}{8} \log n \end{cases}$ per $n^{\frac{1}{4}}$ $\begin{cases} T\left(n^{\frac{1}{8}}\right) = 3T\left(n^{\frac{1}{16}}\right) + 2T\left(n^{\frac{1}{32}}\right) + \frac{1}{8} \log n \\ T\left(n^{\frac{1}{16}}\right) = 3T\left(n^{\frac{1}{32}}\right) + 2T\left(n^{\frac{1}{64}}\right) + \frac{1}{16} \log n \end{cases}$ | $3(3 + 2)$ $+ 2(3 + 2)$ | $3\left(\frac{3}{4} \log n + \frac{1}{4} \log n\right)$ $+ 2\left(\frac{3}{8} \log n + \frac{1}{8} \log n\right)$ $= 3 \log n + \log n$ $= 2^2 \log n$ |

Per livello i -esimo abbiamo un contributo di $2^i \log n$, mentre l'input è $\begin{cases} n^{\frac{1}{2^i}} & \text{per percorso estremo sinistro} \\ n^{\frac{1}{4^i}} & \text{per percorso estremo destro} \end{cases}$

Altezza dell'albero minimo:

$$n^{\frac{1}{4^{h_{\min}}}} = 2 \Rightarrow \log_2 n = 4^{h_{\min}} \Rightarrow \log_2 \log_2 n = 2h_{\min} \Rightarrow h_{\min} = \frac{1}{2} \log_2 \log_2 n$$

Altezza dell'albero massimo:

$$n^{\frac{1}{2^{h_{\max}}}} = 2 \Rightarrow \log_2 n = 2^{h_{\max}} \Rightarrow h_{\max} = \log_2 \log_2 n$$

$$\begin{aligned}
 T_{\min}(n) &= \sum_{i=0}^{h_{\min}} 2^i \log n = \log n \sum_{i=0}^{h_{\min}} 2^i = \log n \left(2^{\frac{1}{2} \log_2 \log_2 n + 1} - 1 \right) = \log n \left(2 \cdot 2^{\log_2 (\log_2 n)^{\frac{1}{2}}} - 1 \right) \\
 &= 2 \log n \cdot \sqrt{\log_2 n} - \log n = 2 \log n \cdot \sqrt{\frac{\log n}{\log 2}} - \log n = \frac{2}{\sqrt{\log 2}} (\log n)^{\frac{3}{2}} - \log n \\
 &= \Theta\left(\log^{\frac{3}{2}} n\right)
 \end{aligned}$$

$$\begin{aligned}
 T_{\max}(n) &= \sum_{i=0}^{h_{\max}} 2^i \log n = \log n \sum_{i=0}^{h_{\max}} 2^i = \log n \left(2^{\log_2 \log_2 n + 1} - 1 \right) = \log n \left(2 \cdot 2^{\log_2 \log_2 n} - 1 \right) \\
 &= 2 \log n \cdot \log_2 n - \log n = \Theta(\log^2 n)
 \end{aligned}$$

Poiché $T_{\min}(n)$ non cresce allo stesso modo di $T_{\max}(n)$ possiamo solo concludere che

$$T(n) = \Omega\left(\log^{\frac{3}{2}} n\right) \wedge T(n) = O(\log^2 n)$$