UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in Informatica

Appunti di

Algoritmi e Strutture Dati

Si può fare meglio di così?

Autore

Emanuele Nardi

Revisori

Francesco Bozzo
Samuele Conti

Anno Accademico 2019/2020

A tutti coloro che lavorano per lasciare il mondo
un po' meglio di come lo hanno trovato.

Indice

I Primo semestre **7**

2 Analisi di algoritmi **1**

2.1 Introduzione . . . 1
2.2 Definizione di complessità . . . 1
2.3 Valutare la dimensione dell'input . . . 1
2.4 Misurare il tempo . . . 1
2.4.1 Calcolo della complessità . . . 2
2.5 Ordini di complessità . . . 4
2.6 Funzioni di costo, notazione asintotica . . . 5
2.7 Complessità di un algoritmo e di un problema . . . 5
2.8 Esercizi . . . 7
2.9 Complessità degli algoritmi e dei problemi a confronto . . . 10
2.9.1 Moltiplicare numeri complessi . . . 10
2.9.2 Sommare numeri binari . . . 10
2.9.3 Moltiplicare numeri binari . . . 11
2.10 Algoritmi di ordinamento . . . 14
2.10.1 Selection sort . . . 14
2.1.2 Insertion sort . . . 15
2.1.3 Merge sort . . . 16

3 Analisi delle funzioni di costo **19**

3.1 Proprietà della notazione asintotica . . . 19
3.1.1 Regola generale . . . 19
3.1.2 Funzioni di costo particolari . . . 19
3.1.3 Proprietà delle notazioni . . . 20
3.2 Analisi per livelli . . . 23
3.2.1 Esempi di analisi per livelli . . . 23
3.3 Metodo di sostituzione . . . 27
3.4 Metodo dell'esperto (o delle ricorrenze comuni) . . . 34

4 Strutture dati **39**

4.1 Strutture dati astratte . . . 39
4.2 Sequenza . . . 40
4.2.1 Implementazione delle sequenze . . . 41
4.3 Insiemi . . . 41
4.4 Dizionari . . . 42
4.5 Alberi . . . 43
4.6 Grafi . . . 43
4.7 Implementazione strutture dati elementari . . . 44
4.7.1 Lista . . . 44

4.7.2 Pila . . . 44
4.7.3 Coda . . . 48

5 Alberi **51**
5.1 Definizioni . . . 51
5.2 Terminologia . . . 51
5.3 Alberi binari . . . 52
5.3.1 Memorizzazione di un albero binario . . . 52
5.3.2 Implementazione . . . 53
5.3.3 Visite . . . 53
5.3.4 Applicazioni . . . 53
5.4 Alberi generici . . . 55
5.4.1 Visita in profondità . . . 55
5.4.2 Visita in ampiezza . . . 55
5.5 Memorizzazione . . . 56
5.5.1 Realizzazione con vettore dei figli . . . 56
5.5.2 Realizzazione basata su primo figlio, prossimo fratello . . . 57
5.5.3 Realizzazione con vettore dei padri . . . 58

6 Alberi Binari di Ricerca **59**
6.1 Introduzione . . . 59
6.2 Alberi Binari di Ricerca bilanciati . . . 66
6.2.1 Inserimento di un nodo . . . 68
6.2.2 Bilanciamento dell'albero . . . 68
6.2.3 Rimozione di un nodo . . . 75

8 Insiemi **77**
8.1 Realizzazione con vettori booleani . . . 77
8.1.1 Implementazioni nei linguaggi di programmazione . . . 78
8.2 Realizzazione con vettore non ordinato . . . 78
8.3 Realizzazione vettore ordinato . . . 79
8.4 Reality Check . . . 79

9 Grafi **81**
9.1 Introduzione . . . 81
9.1.1 Definizioni . . . 81
9.1.2 Terminologia . . . 81
9.1.3 Ragionamenti sulla complessità . . . 82
9.1.4 Casi speciali . . . 82
9.1.5 Specifica . . . 84
9.1.6 Memorizzazione . . . 84
9.2 Visite dei grafi . . . 87
9.2.1 Visita in ampiezza . . . 87
9.2.2 Cammini più brevi . . . 90
9.2.3 Complessità della visita in ampiezza . . . 93

9.2.4 Visita in profondità . . . 93
9.2.5 Componenti connesse . . . 95
9.3 Verifica ciclicità . . . 98
9.4 Ordinamento topologico . . . 104
9.5 Componenti fortemente connesse . . . 106
9.5.1 L'algoritmo di Kosaraju . . . 107
9.5.2 Dimostrazione di correttezza . . . 110

12 Divide et Impera **113**
12.1 Risoluzione di problemi . . . 113
12.1.1 Classificazione dei problemi . . . 113
12.1.2 Caratterizzazione della soluzione . . . 113
12.2 La tecnica del Dividi-et-Impera . . . 114
12.3 La torre di Hanoi . . . 115
12.4 Algoritmo di ordinamento . . . 115
12.5 Conclusioni . . . 117
12.6 Applicazione della tecnica . . . 118
12.6.1 Gap . . . 118

Elenco delle figure

2.1 Notazione asintotica Θ 5
2.6 Moltiplicazione di due numeri binari 11
2.7 Analisi per livelli 17

5.1 Realizzazione di un albero tramite vettore dei figli 56
5.2 Realizzazione di un albero tramite primo figlio, prossimo fratello 57

6.5 Esempio di rotazione a sinistra 68
6.6 Esempio di rotazione a destra 69

9.5 Cammino in un grafo diretto 83
9.9 Esempio di attraversamento di un grafo tramite la procedura di visita di un albero 88
9.10 Esempio di visita di un grafo tramite la procedura di visita in ampiezza 92
9.11 Esempio di identificazione delle componenti connesse in un grafo non orientato 97
9.13 Un grafo aciclico. 98
9.15 Esempio di grafo aciclico e ciclico 100
9.20 Esistenza di più ordinamenti topologici 104
9.21 Esempio di ordinamento topologico 104
9.22 Esempio di ordinamento topologico alternativo 105
9.23 Identificazione della componente fortemente connessa 106
9.26 Identificazione delle componenti fortemente connesse 109
9.28 Grafo delle componenti del grafo e del grafo trasposto 110

Elenco delle tabelle

2.1 Classi di complessità degli algoritmi 4

4.1 Differenza fra specifica ed implementazione 39
4.2 Implementazione delle strutture dati nei vari linguaggi 40

6.1 Possibili implementazioni della struttura dati dizionario e relative complessità 59

8.1 Implementazione `java.util.BitSet` 78
8.2 Implementazioni e relative complessità delle operazioni sugli insiemi 79

12.1 Valutazione delle prestazioni degli algoritmi scritti in python 119

Parte I

Primo semestre

Capitolo 2

Analisi di algoritmi

> *Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop.*
>
> Lewis Carroll, *Alice in Wonderland*, 1899

2.1 Introduzione

Il nostro obiettivo è stimare la complessità *in tempo* degli algoritmi. Dovremmo stimare anche quella in spazio, ma la complessità in spazio dipende da quella in tempo. Daremo delle definizioni, parleremo di modelli di calcolo, faremo qualche esempio di valutazione precisa e introdurremo una notazione. Faremo tutto questo per stimare il tempo per un dato input, per stimare il più grande input gestibile in tempi ragionevoli, per avere un metodo di misura per confrontare algoritmi diversi ed in particolare per ottimizzare le parti più importanti dell'algoritmo.

2.2 Definizione di complessità

La complessità viene definita come una funzione che, data la dimensione dell'input, restituisce il tempo, considerato come un valore intero.

Come definiamo quindi la dimensione dell'input? Come misuriamo il tempo?

2.3 Valutare la dimensione dell'input

Esistono due criteri per valutare la dimensione dell'input:

1. il criterio di **costo logaritmico**: dove la dimensione dell'input è il numero di bit necessari per rappresentarlo (un esempio è la moltiplicazione di numeri binari, lunghi n bit);
2. il criterio di **costo uniforme**: dove la dimensione dell'input è il numero di elementi di cui è costituito (un esempio è a ricerca del minimo in un vettote di n elementi).

Ad esempio consideriamo n interi rappresentati tramite 32 bit. Nel criterio di costo uniforme hanno un costo pari a n, mentre nel criterio di costo logaritmico hanno un costo pari a $32n$.

In molti casi, infatti, possiamo assumere che gli "elementi" siano rappresentati da un numero costante di bit e che le due misure coincidano a meno di una costante moltiplicativa.

Il criterio che abbiamo utilizzano fin'ora — e che useremo d'ora in poi — è il criterio del costo uniforme (in casi particolari utilizzeremo il criterio di costo logaritmico).

2.4 Misurare il tempo

Consideriamo un'istruzione come elementare se può essere eseguita in tempo "costante" dal processore. Facciamo qualche esempio:

- `a *= 2` effettua un'operazione di shift, è una singola operazione macchina;

- `Math.cos(d)` può essere considerata come un'operazione elementare;
- `min(A, n)` non può essere considerata un'operazione elementare poiché si richiede il minimo di un vettore *arbitrariamente lungo*.

Ma allora come possiamo distinguere in maniera precisa un'operazione elementare da una che non lo è? Per farlo abbiamo bisogno di un modello di calcolo, ossia una rappresentazione astratta di un calcolatore. Il quale deve:

1. permettere di nascondere i dettagli (tramite astrazione)
2. riflettere la situazione reale (realismo)
3. permettere di trarre conclusioni "formali" sul contesto.

La pagina di Wikipedia dei modelli di calcolo presenta centinaia di modelli di calcolo diversi. La macchina di turing ne è un esempio. È una macchina ideale che manipola – seconodo un insieme prefissato di regole – i dati contenuti su un nastro di lunghezza infinita. Ad ogni passo, la Macchina di Turing:

1. legge il simbolo sotto la testina;
2. modifica il proprio stato interno;
3. scrive il nuovo simbolo nella cella;
4. muove la testina a destra o a sinistra.

Nel corso di laurea magistrale è possibile approfondire questo aspetto, per i nostri scopi questa è una trattazione dell'argomento troppo a basso livello.

Noi utilizzeremo il modello di calcolo RAM, che sta per Random Access Machine, ossia una macchina che ha una quantità infinita di celle (di dimensione finita) e accesso in tempo costante indipendentemente dalla posizione (diversamente da ciò che avviene nei nastri); un singolo processore con un set di istruzioni simile a quelli reali i cui costi di esecuzione sono uniformi e ininfluenti ai fini della valutazione (faremo un esempio più avanti).

2.4.1 Calcolo della complessità

Proviamo a calcolare la complessità dell'algoritmo che ricerca il minimo.

Algoritmo 2.4.1: Calcolo della complessità della ricerca del minimo in un vettore

```
// calcola il minimo di un vettore arbitrariamente lungo
min(ITEM[] A, int n)
    ITEM min ← A[i]                          Costo   # Volte
    from i ← 2 until n do                    c2      n
        if A[i] < min then                   c3      n − 1
            min ← A[i]                       c4      n − 1
    return min                               c5      1
```

Ragionamento sul calcolo della complessità L'assegnazione del minimo viene eseguita solo una volta. Il ciclo viene eseguito n volte. Il controllo $A[i] < min$ viene eseguito $n-1$ volte in quanto dobbiamo guardare tutto il vettore. Consideriamo *il caso pessimo*, ovvero un vettore ordinato in modo decrescente. L'istruzione di ritorno viene eseguita una volta sola.

Bisogna tenere ben a mente che:

- ogni istruzione richiede un tempo costante per essere eseguita e

- viene eseguita un certo numero di volte, dipendente da n e
- la costante è potenzialmente diversa da istruzione a istruzione.

Sommando tutte le costanti il costo totale risultante è:

$$\begin{aligned} T(n) &= c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) \\ &= an + b \end{aligned}$$

raccogliamo / *semplifichiamo*

Possiamo quindi notare che le costanti vanno a semplificarsi nei parametri a e b.

Proviamo a calcolare la complessità dell'algoritmo che ricerca un numero intero all'interno di un vettore ordinato.

Algoritmo 2.4.2: Calcolo della complessità della ricerca di un numero intero in un vettore ordinato

```
// effettua una ricerca binaria su un vettore
binarySearch(ITEM[] A, ITEM v, int i, int j)
```

	Costo	# $i > j$	# $i \leqslant j$
if $i > j$ **then**	c_1	1	1
return 0	c_2	1	0
else			
int $m \leftarrow \lfloor \frac{(i+j)}{2} \rfloor$	c_3	0	1
if $A[m] == v$ **then**	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ **then**	c_6	0	1
return binarySearch$(A, v, m+1, j)$	$c_7 + T\left(\lfloor \frac{n-1}{2} \rfloor\right)$	0	0/1
else			
return binarySearch$(A, v, i, m-1)$	$c_7 + T(\lfloor n/2 \rfloor)$	0	1/0

Nota. Ci è permesso fare questo ragionamento poiché il vettore è ordinato in ordine decrescente.

Ragionamento sul calcolo della complessità Il vettore viene diviso due parti: la parte sinistra di dimensione $\lfloor \frac{n-1}{2} \rfloor$ e la parte destra di dimensione $\lfloor n/2 \rfloor$. Se n è pari allora il vettore viene diviso in due parti uguali, altrimenti il vettore "di destra" avrà un elemento in più. Si andrà cercare sulla metà sinistra o sulla metà destra a seconda che l'elemento cercato sia più grande o più piccolo rispettivamente. Anche in questo caso consideriamo il caso peggiore, ovvero il caso in cui l'elemento non sia presente. Non prendiamo in considerazione il caso fortunato in cui l'elemento che stiamo cercando sia l'elemento che guardiamo per primo. Nelle chiamate ricorsive dobbiamo considerare (nel costo complessivo) anche il costo delle sotto-chiamate ricorsive con dimensione dell'input pari alla dimesione del vettore passato.

Esercizio Cerca nel vettore ordinato l'elemento 0 tramite la procedura binarySearch, calcolando di volta in volta la dimensione del vettore n.

1	2	3	4	5	6	7	8
8	7	6	5	4	3	2	1

Calcolo del caso pessimo Assumiamo per semplicità che

1. n sia una potenza di 2 ($n = 2^k$, $8 = 2^3$);
2. l'elemento cercato non sia precisato
3. ad ogni passo scegliamo il vettore di destra (di dimensione $n/2$).

Si hanno due casi:

- il caso base $\boxed{i > j}$, dove $n = 0$ e la relazione di ricorrenza è pari a $T(n) = c_1 + c_2 = c$ dove c è una costante;

- il caso ricorsivo $\boxed{i \leqslant j}$, dove $n > 0$ e dobbiamo tener conto di tutte le costanti moltiplicative $T(n) = T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7$, raccogliendo le costanti $T(n) = T(n/2) + d$, dove d è la costante che racchiude tutti i costi.

La relazione di ricorrenza che ne segue è la sequente:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n > 0 \end{cases}$$

Nota. Per calcolare la complessità di una funzione ricorsiva abbiamo bisogno di una funzione di ricorrenza anch'essa ricorsiva.

Le equazioni di ricorrenza così fatte $T(n) = d\log(n) + e$ sono dette in "forma chiusa" e rappresentano la complessità dell'algoritmo (non necessano quindi ulteriori sviluppi).

Risolviamo quindi l'equazione di ricorrenza *tramite espansione*:

$$\begin{aligned} T(n) &= T(n/2) + d && \big) \left(T\left(\tfrac{n}{2} \cdot \tfrac{1}{2}\right) + d\right) + d \\ &= T(n/4) + 2d && \big) \left(T\left(\tfrac{n}{4} \cdot \tfrac{1}{2}\right) + 2d\right) + d \\ &= T(n/8) + 3d \\ &\;\;\vdots && \big) \; n = 2^k \Rightarrow k = \log n \\ &= T(1) + kd && \big) \; T(0) = c \\ &= T(0) + (k+1)\,d \\ &= kd + (c+d) && \big) \; k = \log n \\ &= d\log n + e. \end{aligned}$$

2.5 Ordini di complessità

Per ora, abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di complessità*:

- Ricerca: $T(n) = d\log n + e$. Chiamiamo questa funzione **logaritmica**, utilizzando la notazione $\mathcal{O}(\log n)$;
- Minimo: $T(n) = a + b$. Chiamiamo questa funzione **lineare**, utilizzando la notazione $\mathcal{O}(n)$.

Abbiamo visto anche una terza funzione che deriva dall'algoritmo banale (*naïf*) per la ricerca del minimo:

- Minimo: $T(n) = fn^2 + gn + h$. Chiamiamo questa funzione **quadratica**, utilizzando la notazione $\mathcal{O}(n^2)$.

Tabella 2.1: Classi di complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
$\sqrt{n}$	3	10	31	100	sublineare
n	10	100	1000	10 000	lineare
$n\log n$	30	664	9965	132 877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale

2.6 Funzioni di costo, notazione asintotica

Ora andremo a formalizzare le nozioni sui limiti superiori ed inferiori che abbiamo accennato in maniera informale nelle lezioni precedenti.

Definizione (Funzione di costo)**.** Utilizziamo il termine "funzione di costo" per indicare una funzione $f: \mathbb{N} \to \mathbb{R}$ (dall'insieme dei numeri naturali ai reali).

Definizione (Notazione $\mathcal{O}$)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\mathcal{O}(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cg(n), \forall n \geqslant m$$

Nota. Eventuali fattori moltiplicativi non ci interessano.

La notazione si legge $f(n)$ è "O grande" di $g(n)$ e si scrive $f(n) = \mathcal{O}(g(n))$. Questo è un abuso di notazione, dovemmo scrivere $f(n) \in \mathcal{O}(g(n))$, in quanto $\mathcal{O}$ è un insieme (più precisamente una famiglia di funzioni). Questa notazione è però diventata d'uso comune poiché ci si può fare una specie di aritmetica sopra infatti è la notazione che troverete nella letteratura e sta a significare che $g(n)$ è un limite asintotico superiore per $f(n)$, ossia che $f(n)$ cresce al più (al massimo) come $g(n)$.

Definizione (Notazione Ω)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : f(n) \geqslant cg(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Omega grande" (nella letteratura big-O) di $g(n)$, si scrive $f(n) = \Omega(g(n))$ e sta a significare che $g(n)$ è un limite asintotico inferiore per $f(n)$, ossia che $f(n)$ cresce almeno quanto (non di meno) come $g(n)$.

Definizione (Notazione Θ)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : c_1 g(n) \leqslant f(n) \leqslant c_2 g(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Theta" di $g(n)$, si scrive $f(n) = \Theta(g(n))$ e sta a significare che $f(n)$ cresce *esattamente* come $g(n)$ al di là di fattori moltiplicativi. Nota che $f(n) = \Theta(g(n))$ avviene se e solo se $f(n) = \mathcal{O}(g(n))$ e $f(n) = \Theta(g(n))$.

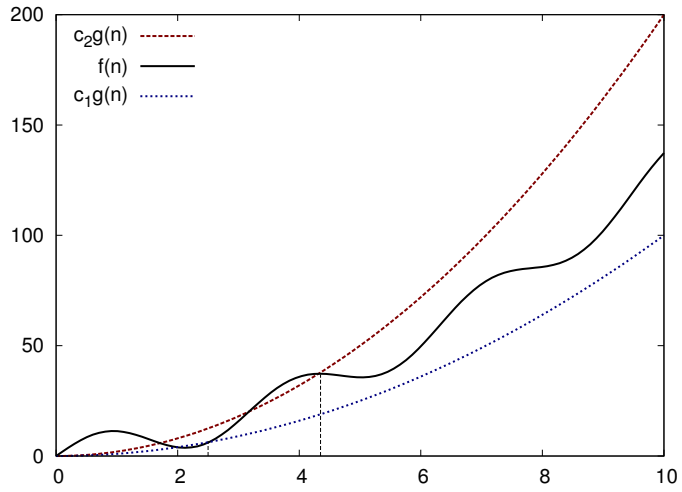


Figura 2.1: Notazione asintotica Θ

2.7 Complessità di un algoritmo e di un problema

Definizione (Complessità in tempo di un algoritmo)**.** La più grande quantità di tempo richiesta per un input di dimensione n.

- $\mathcal{O}(f(n))$: per tutti gli input, l'algoritmo costa al più $f(n)$;
- $\Omega(f(n))$: per tutti gli input, l'algoritmo costa almeno $f(n)$;
- $\Theta(f(n))$: l'algoritmo richiede $\Theta(f(n))$ per tutti gli input.

Definizione (Complessità in tempo di un problema computazionale)**.** La complessità in tempo relativa a tutte le possibili soluzioni.

- $\mathcal{O}(f(n))$: complessità del miglior algoritmo che risolve il problema;
- $\Omega(f(n))$: dimostrare che nessun algoritmo può risolvere il problema in tempo inferiore a $\Omega(f(n))$;
- $\Theta(f(n))$: abbiamo trovato l'algoritmo ottimo.

2.8 Esercizi

Iniziamo con gli esercizi banali che ci permettono di introdurre delle tecniche che utilizzeremo con le ricorrenze. In particolare ci servono a renderci conto che non stiamo dimostrando equazioni, ma disequazioni.

$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 \\
&\leqslant 10n^3 + 2n^3 + 7 && \forall n \geqslant 1 \\
&\leqslant 10n^3 + 2n^3 + 7n^3 && \textit{sommiamo i termini} \\
&= 19n^3 && \textit{esiste una certa costante } c \\
&\stackrel{?}{\leqslant} cn^3 && \textit{per la quale } f(n) \leqslant cn^3 \ ? \\
19n^3 &\leqslant cn^3 && \textit{metto a confronto} \\
19\cancel{n^3} &\leqslant c\cancel{n^3} && \textit{semplifico}
\end{aligned}$$

che è vera per ogni $c \geqslant 19$ (abbiamo così trovato la costante moltiplicativa) e per ogni $n \geqslant 1$ (introdotta nei calcoli), quindi $m = 1$.

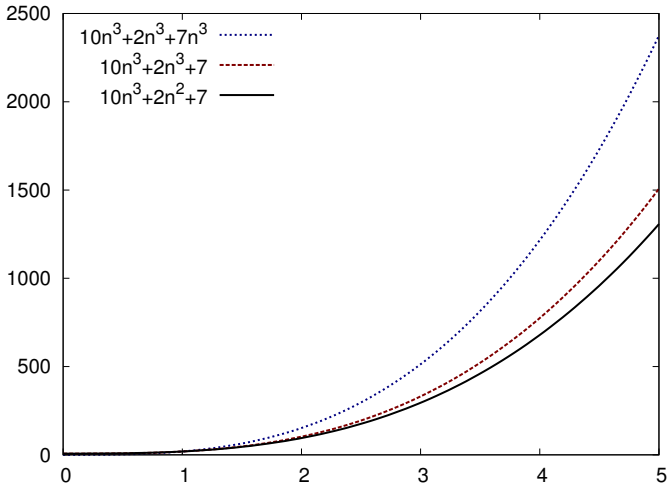


Figura 2.2: Risoluzione grafica dell'esercizio

Nota. In generale noi considereremo solo valori di n positivi, in quanto le funzioni di costo sono definite sull'insieme dei numeri naturali, non ha alcun senso definire una funzione di costo su una dimensione dell'input negativa.

Dato lo stesso esercizio posso esserci passaggi risolutivi diversi. Risolviamo quindi l'esercizio precedente diversamente.

$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$
\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 && \text{)}\ \forall n \geqslant 1 \\
&\leqslant 10n^3 + 2n^3 + 7 && \text{)}\ \forall n \geqslant \sqrt[3]{7} \\
&\leqslant 10n^3 + 2n^3 + n^3 && \text{)}\ \textit{sommiamo i termini} \\
&= 13n^3 && \text{)}\ \textit{esiste una certa costante } c \\
&\overset{?}{\leqslant} cn^3 && \text{)}\ \textit{per la quale } f(n) \leqslant cn^3\ ? \\
13n^3 &\leqslant cn^3 && \text{)}\ \textit{metto a confronto} \\
13\cancel{n^3} &\leqslant c\cancel{n^3} && \text{)}\ \textit{semplifico}
\end{aligned}
$$

che è vera per ogni $c \geqslant 13$ e per ogni $n \geqslant \sqrt[3]{7}$ (ad esempio con $n = 2$ abbiamo $n^3 = 2^3 = 8$ che soddisfa la nostra condizione), quindi usiamo $m = 2$ (abbiamo semplificato, sarebbe $m = \sqrt[3]{7}$, ma possiamo prendere un qualunque valore si trovi dopo n in modo totalmente arbitrario).

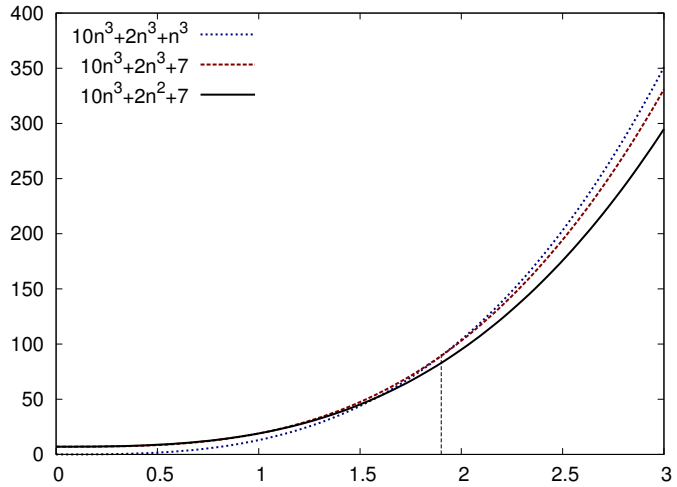


Figura 2.3: Risoluzione grafica dell'esercizio

$$\boxed{f(n) = 3n^2 + 7n \overset{?}{=} \Theta(n^2)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c_1 > 0, \exists m_1 \geqslant 0 : f(n) \geqslant c_1 n^2, \forall n \geqslant m_1$

$$
\begin{aligned}
f(n) &= 3n^2 + 7n && \text{)}\ \forall n \geqslant 0 \\
&\geqslant 3n^2 && \text{)}\ \textit{esiste una certa costante } c \\
&\overset{?}{\geqslant} c_1 n^2 && \text{)}\ \textit{per la quale } f(n) \leqslant c_1 n^2\ ? \\
3n^2 &\leqslant c_1 n^2 && \text{)}\ \textit{metto a confronto} \\
3\cancel{n^2} &\leqslant c_1\cancel{n^2} && \text{)}\ \textit{semplifico}
\end{aligned}
$$

che è vera per ogni $c_1 \leqslant 3$ e per ogni $n \geqslant 0$ (introdotta nei calcoli), quindi $m_1 = 0$.
Nota. Abbiamo dimostrato quindi che $f(n) = \Omega(n^2)$.

Limite superiore: Dobbiamo dimostrare che $\exists c_2 > 0, \exists m_2 \geqslant 0 : f(n) \leqslant c_2 n^2, \forall n \geqslant m_2$

$$
\begin{aligned}
f(n) &= 3n^2 + 2n^2 + 7n && \text{)}\ \forall n \geqslant 1 \\
&\leqslant 3n^2 + 7n^2 && \text{)}\ \textit{raccogliamo} \\
&\leqslant 10n^2 && \text{)}\ \textit{esiste una certa costante } c \\
&\overset{?}{\leqslant} c_2 n^2 && \text{)}\ \textit{per la quale } f(n) \leqslant c_2 n^2\ ? \\
10n^2 &\leqslant c_2 n^2 && \text{)}\ \textit{metto a confronto} \\
10\cancel{n^2} &\leqslant c_2\cancel{n^2} && \text{)}\ \textit{semplifico}
\end{aligned}
$$

che è vera per ogni $c_2 \geqslant 10$ e per ogni $n \geqslant 1$, quindi $m_2 = 1$.

Nota. Abbiamo dimostrato quindi che $f(n) = \mathcal{O}(n^2)$.

Notazione Θ: $\exists c_1 > 0, \exists c_2 > 0, \exists m \geqslant 0 : c_1 n^2 \leqslant f(n) \leqslant c_2 n^2, \forall n \geqslant m$.

Con questi paramentri:

- $c_1 = 3$
- $c_1 = 10$
- $m = max\{m_1, m_2\} = max\{0, 1\} = 1$, ossia un valore dopo il quale la nostra proprietà è provata

Nota. Abbiamo dimostrato quindi che $f(n) = \Theta(n^2)$

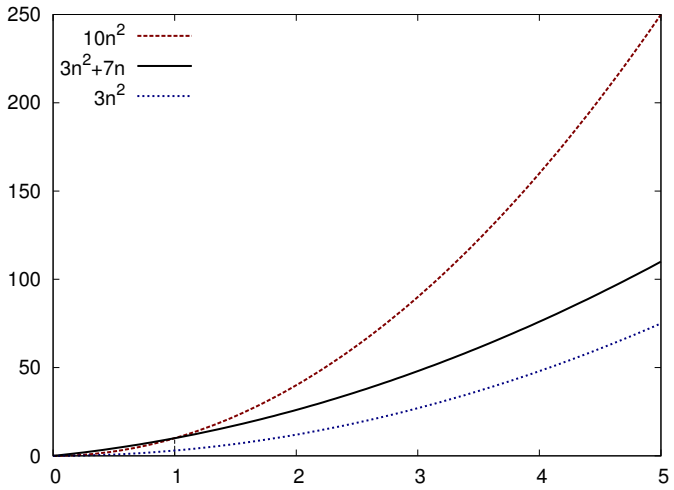


Figura 2.4: Risoluzione grafica dell'esercizio

Errori comuni durante la risoluzione degli esercizi

$$\boxed{f(n) = n^2 \stackrel{?}{=} \mathcal{O}(n)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : n^2 \geqslant cn, \forall n \geqslant m$.

Otteniamo che $n^2 \leqslant cn \Leftrightarrow c \geqslant n$, questo significa che c cresce con il crescere di n, ovvero che non possiamo scegliere una costante c.

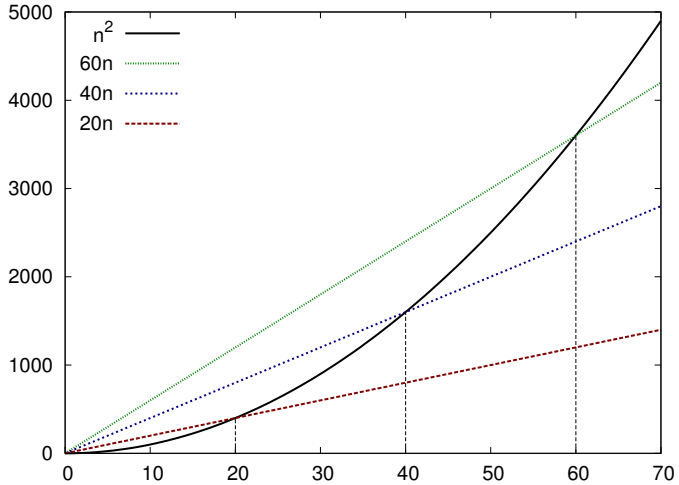


Figura 2.5: Per qualunque fattore c scelto (ossia la pendenza della retta) la curva quadratica crescerà sempre più velocemente da un certo punto in poi

$$\boxed{f(n) = n^2 \stackrel{?}{=} \Omega(n^3)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0, n^2 \geqslant cn^3, \forall n \geqslant m$.

Otteniamo che $n^2 \geqslant cn^3 \Leftrightarrow c \leqslant \frac{1}{n}$, questo significa che c diminuisce al crescere di n, ovvero che non possiamo scegliere una costante c che verifichi la proprietà.

2.9 Complessità degli algoritmi e dei problemi a confronto

In questa sezione ragioneremo su alcuni algoritmi risolutivi che ci sono stati insegnati, in alcuni casi si può migliorare la complessità, in altri è impossibile fare di meglio.

> Qual è il rapporto fra un problema computazionale e l'algoritmo?

2.9.1 Moltiplicare numeri complessi

La moltiplicazione fra numeri complessi avviene nel seguente modo: $(a+bi)(c+di) = [ac-db]+[ad+bc]i$. Abbiamo in input a, b, c, d e dobbiamo restituire in output $ac-bd$ e $ad+bc$.

Consideriamo un modello di calcolo dove la moltiplicazione costa 1 e le addizioni e sottrazioni costano 0,01.

1. Quanto costa l'algoritmo dettato dalla definizione?
2. Riesci a fare meglio di così?
3. Qual è il ruolo del modello di calcolo?

L'algoritmo banale dettato dalla definizione costa 4,02, in quanto bisogna fare 4 moltiplicazioni, 1 somma ed una sottrazione.

La seguente è la soluzione di Gauss al problema, datata 1805.

Input: a, b, c, d, Output: $A1 = ac - bd$, $A2 = ad + bc$

$$\begin{aligned} m_1 &= a \times c \\ m_2 &= b \times d \\ A_1 &= m_1 - m_2 \\ m_3 &= (a+b)\cdot(c+d) = ac + ad + bc + bd \\ A_2 &= m_3 - m_1 - m_2 = ad + bc \end{aligned}$$

calcolo i valori intermedi

evito una moltiplicazione

Il costo totale è 3,05.

> Si può fare ancora meglio di così? Oppure, è possibile dimostrare che non si può fare di meglio?

2.9.2 Sommare numeri binari

Nota. In questo caso usiamo il criterio del costo logaritmico.

L'algoritmo elementare della somma richiede di esaminare tutti gli n bit, il costo totale risulta cn, dove c è il costo per sommare due bit e generare il riporto.

> Esiste un metodo più efficiente?

È dimostrabile per assurdo che *non è possibile fare di meglio* di una soluzione lineare, poiché non è possibile sommare due numeri binari senza esaminare tutti gli n bit.

Limiti alla complessità di un problema

Definizione (Limite superiore, $\mathcal{O}(f(n))$)**.** Un problema ha complessità $\mathcal{O}(f(n))$ *se esiste almeno un algoritmo* che ha complessità $\mathcal{O}(f(n))$.

Nota. Il problema della somma dei numeri binari ha complessità $\mathcal{O}(n)$.

Definizione (Limite inferiore, $\Omega(f(n))$)**.** Un problema ha complessità $\Omega(f(n))$ *se tutti i possibili algoritmi* che lo risolvono hanno complessità $\Omega(f(n))$.

Nota. Il problema della somma dei numeri binari ha complessità $\Omega(n)$.

2.9.3 Moltiplicare numeri binari

L'algoritmo elementare del prodotto richiede di moltiplicare ogni bit con ogni altro bit, per un costo totale di cn^2.

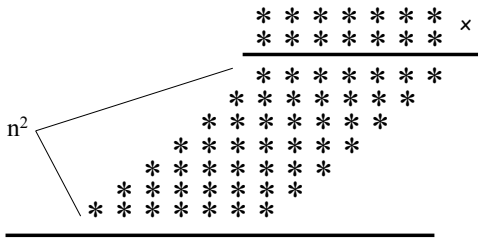


Figura 2.6: Moltiplicazione di due numeri binari

Si potrebbe concludere che il problema della moltiplicazione è molto più costoso del problema dell'addizione: ne è conferma la nostra esperienza.

Nota. Per provare che il problema del prodotto è più costoso del problema della somma, dobbiamo provare che non esiste una soluzione in tempo lineare al problema del prodotto.

Abbiamo infatti erroneamente confrontato gli algoritmi, non i problemi! Sappiamo solo che l'algoritmo della somma che ci hanno insegnato è più efficiente di quello della moltiplicazione.

Nel 1960, Kolmogorov enunciò che la moltiplicazione avesse limite inferiore pari a $\Omega(n^2)$, una settimana dopo un suo studente Karatsuba riuscì a provare il contrario. Osserviamo la sua soluzione.

Moltiplicazione di Karatsuba

Karatsuba adottò un approccio divide-et-impera.

Definizione 2.9.1 (Approccio divide-et-impera)**.** Si svolge in tre parti:

- **Divide**: dividi il problema in sottoproblemi di dimensione inferiore;
- **Impera**: risolvi i sottoproblemi in maniera ricorsiva;
- (**Combina**): unisci le soluzioni dei sottoproblemi in modo da ottenere la risposta del problema principale.

$$
\begin{aligned}
X &= a \cdot 2^{n/2} + b \\
Y &= c \cdot 2^{n/2} + d \\
XY &= ac \cdot 2^{n} + (ad + bc) \cdot 2^{n/2} + db
\end{aligned}
$$

Algoritmo 2.9.1: Algoritmo della moltiplicazione di due numeri binari

```
// moltiplica due numeri binari
bool[ ] pdi(bool[ ] X, bool[ ] Y, int n)
    // X:      numero binario
    // Y:      numero binario
    // n:      numero di bit contenuti
    if n == 1 then
        return X[1] · Y[1] // eseguo la moltiplicazione di due bit
    else
        spezza X in a;b e Y in c;d
        return pdi(a, c, n/2) · 2^n + (pdi(a, d, n/2) + pdi(b, c, n/2)) · 2^(n/2) + pdi(b, d, n/2)
```

Complessità Moltiplicare per 2^t è equivalente ad eseguire uno shift di t posizioni, in tempo lineare, quindi l'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Nota. Non sappiamo ancora trattare questo genere di problemi, quindi facciamo solo degli accenni.

Analisi della ricorsione Al primo passo la chiamata ricorsiva avviene su una dimesione n, al secondo passo vengono effettuate 4 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $4^2 = 16$ chiamate ricorsive su una dimesione $n/2^2 \ldots$ al livello i-esimo vengono effettuate 4^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $4^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $4^{\log n} = n^{\log 4} = n^2$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^2)$. È possibile ridurre ulteriormente la complessità.

$$\begin{aligned} A_1 &= a \times c \\ A_2 &= b \times d \\ m &= (a+b) \times (c+d) = ac + ad + bc + bd \\ A_3 &= m - A_1 - A_2 = ad + bc \end{aligned}$$

calcolo un valore intermedio

evito una moltiplicazione

Principio Effettuo un'unica moltiplicazione che mi permette di calcolare un valore intemedio che contiene la somma di tutte le combinazioni (m), e ricavo $ad + bc$ tramite due sottrazioni (nello stesso modo in cui lavorava Gauss), evitando così una moltiplicazione.

Algoritmo 2.9.2: Algoritmo della moltiplicazione di Karatsuba

```
bool[ ] karatsuba(bool[ ] X, bool[ ] Y, int n)
    if n == 1 then
        return X[1] · Y[1] // rimane invariato
    else
        spezza X in a;b e Y in c;d
        bool[ ] A1 = karatsuba(a, c, n/2)
        bool[ ] A3 = karatsuba(b, d, n/2)
        bool[ ] m = karatsuba(a+b, c+d, n/2) // potrebbe essere n/2 + 1
        bool[ ] A2 = m − A1 − A3 // ottengo A2 tramite sottrazione
        return A1 · 2^n + A2 · 2^(n/2) + A3 // effettuo degli shift
```

Complessità L'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Analisi della ricorsione Al primo passo la chiamata ricorsiva avviene su una dimesione n, al secondo passo vengono effettuate 3 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $3^2 = 9$ chiamate ricorsive su una dimesione $n/3^2 \ldots$ al livello i-esimo vengono effettuate 3^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $3^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $3^{\log n} = n^{\log 3} = n^{1.58\ldots}$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^{1.58\ldots})$.

Nota. L'algoritmo ingenuo (naïf) non è sempre il migliore a meno che non sia possibile dimostrare il contrario.

Negli anni sono stati proposti diversi algoritmi, che il limite inferiore al problema della moltiplicazione sia $\Omega(n \log n)$ è una congettura. Una congettura è un'affermazione o un giudizio fondato sull'intuito, ritenuto probabilmente vero, ma non ancora rigorosamente dimostrato, cioè dunque relegato solamente a rango di ipotesi.

Nella GNU Multiple Precision Arithmetic Library vengono utilizzati diversi algoritmi al crescere di n, il valore soglia per cui si predilige un algoritmo rispetto ad un altro dipende dal tipo di architettura.

2.10 Algoritmi di ordinamento

In questa lezione impareremo a capire quando è meglio utilizzare un algoritmo di ordinamento rispetto ad un altro.

In alcuni casi, gli algoritmi si comportano diversamente a seconda delle caratteristiche dell'input. Conoscere in anticipo tali caratteristiche permette di scegliere l'algoritmo migliore in quella determinata situazione.

Tipologia di analisi

Esistono tre tipi di analisi:

1. Analisi del *caso pessimo*: è la tipologia più importante, il tempo di esecuzione nel caso peggiore è il limite superiore al tempo di esecuzione per qualsiasi input. Per alcuni algoritmi il caso peggiore si verifica molto spesso (ad esempio nella ricerca di dati non presenti nel database);
2. Analisi del *caso medio*: è difficile da definire (cosa si intende per "medio"?), dobbiamo avere una conoscenza pregressa sulle distribuzioni;
3. Analisi del *caso ottimo*: può avere senso se si conoscono informazioni particolari sull'input.

Problema dell'ordinamento

Data una sequenza $A = a_1, a_2, \ldots, a_n$ di n valori in input, il problema dell'ordinamento consiste nel restituire in output una sequenza $B = b_1, b_2, \ldots, b_n$ che sia una permutazione di A, tale per cui $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$ (ovvero che ci sia un ordinamento *totale*).

Un approccio "demente" è quello di generare tutte le possibili permutazioni (complessità $n!$) fino a quando non se ne trova una ordinata.

2.10.1 Selection sort

Un approccio banale (*naïf*) è quello di cercare il minimo e metterlo nella posizione corretta, riducendo il problema agli $n-1$ valori rimanenti.

Algoritmo 2.10.1: Algoritmo selectionSort

```
// effettua l'ordinamento di un vettore
selectionSort(ITEM[ ] A, int n)
    from int i ← 1 until n − 1 do // l'ultimo elemento è ordinato
        int j ← min(A, i, n) // ricerca il nuovo minimo
        swap(A[i], A[j]) // lo metto nella posizione corretta

// cerca l'indice dell'elemento più piccolo
int min(ITEM[ ] A, int i, int j)
    int min ← i // posizione del minimo parziale
    from int j ← i + 1 until n do
        if A[j] < A[min] then // ho trovato un nuovo minimo
            min ← j // nuovo minimo parziale
    return min // restituisco l'indice dell'elemento più piccolo
```

Suggerimento. Provalo su carta! Un algoritmo dev'essere provato per essere capito davvero!

Analisi della complessità Il ciclo effettua n chiamate della funzione min (una per ciascuna iterazione). Ad ogni iterazione il vettore su cui viene calcolato il minimo risulta più piccolo di un elemento.

$$\begin{aligned}
&\sum_{i=1}^{n-1}(n-1) \\
&= \sum_{i=1}^{n-1} i && \text{(} 10+9+\cdots+1 \Leftrightarrow 1+2+\cdots+10 \text{)} \\
&= \frac{n(n-1)}{2} \\
&= n^2 - \frac{n}{2} && \text{(svolgo i calcoli)} \\
&= \Theta(n^2)
\end{aligned}$$

Posso dire che è $\Theta(n^2)$, e non solo che $\Omega(n^2)$, perché indipendentemente dall'ordine dei numeri ci metterà sempre lo stesso tempo. In altre parole, il caso migliore, peggiore e medio coincidono.

2.1.2 Insertion sort

Un algoritmo che si basa sul principio di ordinamento di una "mano" di carte da gioco è il seguente:

Algoritmo 2.1.2: Algoritmo insertionSort

```
// effettua l'ordinamento di un vettore
insertionSort(ITEM[] A, int n)
    from int i = 2 until n do // il 1° elemento verrà ordinato in seguito
        ITEM temp ← A[i] // elemento da ordinare
        int j ← i
        while j > 1 and A[j − 1] > temp do
            A[j] ← A[j − 1] // copio l'elemento
            j++ // mi sposto
        A[j] ← temp
```

Questo è un algoritmo molto efficiente per ordinare piccoli insiemi di elementi.

Analisi della complessità Il costo di esecuzione di questo algoritmo non dipende solo dalla dimensione del vettore, ma anche dalla distribuzione dei dati in ingresso. Nel caso in cui il vettore sia *già ordinato* il costo è $\mathcal{O}(n)$, in quanto non si entra mai nel secondo ciclo dato che la condizione risulta falsa. Nel caso in cui il vettore sia *ordinato in ordine inverso* è $\Omega(n^2)$. In media (informalmente) possiamo assumere che metà dei valori sia ordinata rispetto la loro disposizione finale e quindi metà di loro dovrannno fare n passi per arrivare alla destinazione per una complessità di $n \cdot n/2 = \mathcal{O}(n^2)$.

Infine quando sappiamo che i valori sono quasi ordinati o che n è molto piccolo (nell'ordine di 16 o 32 elementi) allora questo algoritmo risulta efficiente.

2.1.3 Merge sort

MergeSort è basato sulla tecnica divide-et-impera vista in precendenza. Ma come la utilizza?

Definizione 2.1.1 (Approccio divide-et-impera di MergeSort)**.** Si svolge in tre parti:

- **Divide**: spezza il vettore di n elementi in 2 sottovettori di $\frac{n}{2}$ elementi;
- **Impera**: chiama mergeSort ricorsivamente sui due sottovettori (ottenendo due metà ordinate);
- (**Combina**): unisce le due sequenze ordinate (`merge`).

L'idea alla base di questo algoritmo sfrutta il fatto che è possibile unire due sottovettori ordinati in un vettore ordinato in tempo lineare.

Algoritmo 2.1.3: merge

```
// ordina i sottovettori
mergeSort(ITEM[] A, int primo, int ultimo)
    if primo < ultimo then // devono esistere almeno due elementi
        int mezzo ← ⌊(primo+ultimo)/2⌋
        mergeSort(A, primo, mezzo)
        mergeSort(A, mezzo+1, ultimo)
        merge(A, primo, ultimo, mezzo)  // unisce le soluzioni

// effettua l'ordinamento dei sotto-vettori
merge(ITEM A, int primo, int ultimo, int mezzo)
    int i, j, k, h

    // inizializzo i puntatori
    i ← primo
    j ← mezzo
    k ← primo
    // k:  indica la prossima posizione di scrittura

    while i ⩽ mezzo and j ⩽ ultimo do
        // B è il vettore di appoggio in cui memorizzo la porzione di vettore già ordinata
        if A[i] ⩽ A[j] then
            // l'elemento è gia ordinato
            B[k] ← A[i]
            i++
        else
            B[k] ← A[j]
            j++

        // in entrambi i casi ho inserito un valore
        k++

    // se uno dei due vettori finisce ricopio la parte ordinata alla fine del vettore d'appoggio
    j ← ultimo
    from h ← mezzo until i do
        A[j] ← A[h]
        j++

    // ricopio il vettore d'appoggio del vettore originale
    from j ← primo until k − 1 do
        A[j] ← B[j]
```

Analisi della complessità Assumiamo (per semplicità) che $n = 2^k$ (ovvero che l'altezza dell'albero di suddivisioni sia esattamente $k = \log_2 n$) e che tutti i sottovettori abbiano dimensioni che sono potenze

esatte di 2. L'equazione di ricorrenza risultante è la seguente:

$$
\begin{aligned}
T &= \begin{cases} \Theta(1) & n = 1 \\ T\left(\lfloor n/2 \rfloor\right) + T\left(\lfloor n/2 \rfloor\right) + \Theta(n) & n > 1 \end{cases} \\
&= \begin{cases} c & n = 1 \\ 2T\left(n/2\right) + dn & n > 1 \end{cases}
\end{aligned}
$$

Qual è il costo computazionale di mergeSort?

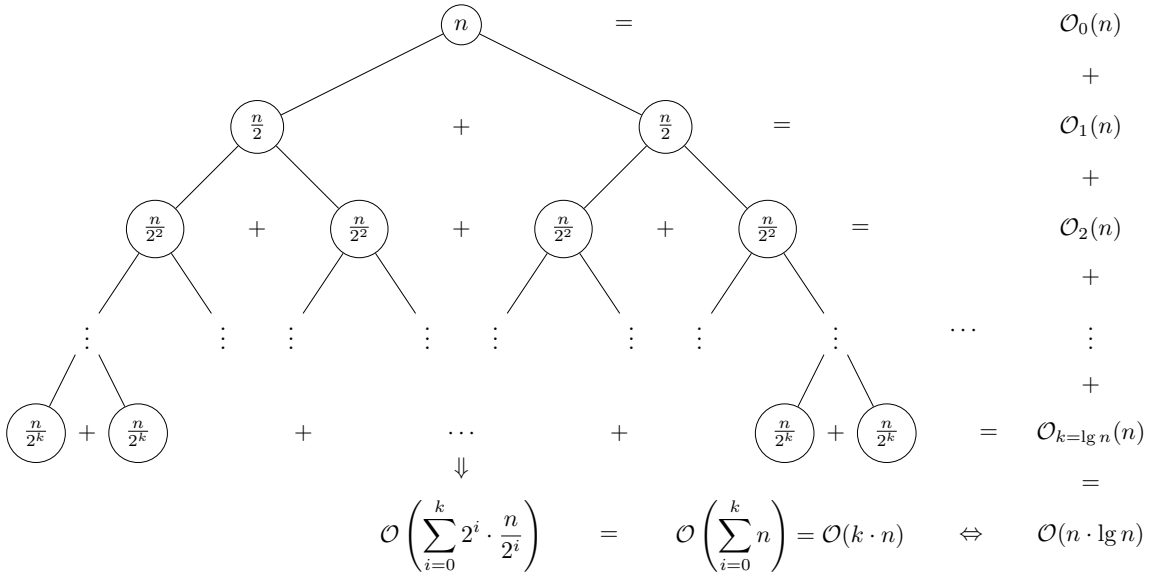


Figure 2.7: Analisi per livelli

L'analisi per livelli è la seguente:

$$
\begin{aligned}
&\mathcal{O}\left(\textstyle\sum_{i=0}^{k} 2^i \frac{n}{2^i}\right) \\
&= \mathcal{O}\left(\textstyle\sum_{i=0}^{k} n\right) && \textit{semplifico} \\
&= \mathcal{O}((k+1) \cdot n) && \textit{equivalente} \\
&= \mathcal{O}(k \cdot n) && \textit{$k+1$ elementi, semplico perché è una costante} \\
&= \mathcal{O}(n \log n) && \textit{$k = \log n$}
\end{aligned}
$$

$\mathcal{O}(n \log n)$ è asintoticamente migliore di $\mathcal{O}(n^2)$. Questo algoritmo è preferibile — per grandi dimensioni di n — al selectionSort e all'insertionSort.

Capitolo 3

Analisi delle funzioni di costo

Abbiamo concluso la prima parte delle lezioni che ci introduceva alle equazioni di ricorrenza: ora andremo a capire i fondamenti matematici che ci permetteranno di analizzarne una famiglia più grande.

3.1 Proprietà della notazione asintotica

3.1.1 Regola generale

Teorema 1 (Regola generale)**.** $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots a_1 n + a_0, a_k > 0 \Rightarrow f(n) = \Theta(n^k)$

Dimostrazione. **Limite superiore**: $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^k, \forall n \geqslant m$

$$
\begin{aligned}
f(n) &= a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \\
&\leqslant a_k n^k + |a_{k-1}| n^{k-1} + \dots + |a_1| n + |a_0| \\
&\leqslant a_k n^k + |a_{k-1}| n^k + \dots + |a_1| n^k + |a_0| n^k \\
&= (a_k + |a_{k-1}| + \dots + |a_1| + |a_0|)\, n^k \\
&\overset{?}{\leqslant} cn^k
\end{aligned}
$$

$a_k > 0$ per def., rendo gli altri positivi
$\forall n \geqslant 1$, elevo tutte le potenze a k
raccolgo n^k
esiste una costante c che rende la disequazione vera?

che è vera per $c \geqslant (|a_k| + |a_{k-1}| + \dots + |a_1| + |a_0|)$ (il coefficiente) e per $m = 1$. □

Dimostrazione. **Limite inferiore**: $\exists d > 0, \exists m \geqslant 0 : f(n) \geqslant dn^k, \forall n \geqslant m$

$$
\begin{aligned}
f(n) &= a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \\
&\geqslant a_k n^k - |a_{k-1}| n^{k-1} - \dots - |a_1| n - |a_0| \\
&\geqslant a_k n^k - |a_{k-1}| n^{k-1} - \dots - |a_1| n^{k-1} - |a_0| n^{k-1} \\
&\overset{?}{\geqslant} dn^k
\end{aligned}
$$

normalizzo aggiungendo un segno negativo
$\forall n \geqslant 1$, elevo alla $k-1$

L'ultima equazione è vera se:

$$
d \leqslant a_k - \frac{|a_{k-1}|}{n} - \frac{|a_{k-2}|}{n} - \dots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \iff n > \frac{|a_{k-1}| + \dots + |a_0|}{a_k} = m
$$

□

Abbiamo dimostrato sia il limite superiore che quello inferiore, possiamo affermare che è un Θ.

Ad esempio $17n^3 - 47n^2 + 123n + 17 = \Theta(n^3)$. Oppure $2n^3 + 7 = \Theta(n^2)$.

3.1.2 Funzioni di costo particolari

La complessità di $f(n) = 5$ è pari a $\Theta(1)$, questa classe di complessità rappresenta quegli algoritmi che sono così ben congegnati che indipendentemente dalla dimensione dell'input impiegano un tempo costante a risolvere il problema. Ad esempio richiedere il minimo in un vettore ordinato.

Dimostriamolo.

$$
\begin{aligned}
f(n) = 5 \geqslant c_1 n^0 &\Rightarrow c_1 \leqslant 5 \\
f(n) = 5 \leqslant c_2 n^0 &\Rightarrow c_2 \geqslant 5 \\
f(n) = \Theta(n^0) &= \Theta(1)
\end{aligned}
$$

La complessità di $f(n) = 5 + \sin(n)$ è pari a $\Theta(1)$, in quanto $\sin(n)$ oscilla fra 1 e -1.

3.1.3 Proprietà delle notazioni

Teorema 2 (Dualità)**.** *Se $f(n)$ è limitata superiormente da $g(n)$, allora $g(n)$ è limitata inferiormente da $f(n)$.*

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

La dimostrazione avviene tramite passaggi algebrici.

Dimostrazione.

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) &\Leftrightarrow f(n) \leqslant \boldsymbol{c}\, g(n), \forall n \geqslant m \\ &\Leftrightarrow g(n) > \tfrac{\mathbf{1}}{\boldsymbol{c}}\, f(n), \forall n \geqslant m \\ &\Leftrightarrow g(n) > \boldsymbol{c'}\, f(n), \forall n \geqslant m, c' = \tfrac{1}{c} \\ &\Leftrightarrow g(n) = \Omega(f(n)) \end{aligned}$$

ribalto la disequazione, $c > 0$

rinomino $\frac{1}{c}$ a c'

passo alla notazione dei quantificatori

□

Teorema 3 (Eliminazione delle costanti)**.**

$$f(n) = \mathcal{O}(g(n)) \Leftrightarrow af(n) = \mathcal{O}(g(n)), \forall a > 0$$
$$f(n) = \Omega(g(n)) \Leftrightarrow af(n) = \Omega(g(n)), \forall a > 0$$

La dimostrazione avviene sempre tramite semplici passaggi algebrici.

Dimostrazione.

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) &\Leftrightarrow f(n) \leqslant cg(n), \forall n \geqslant m \\ &\Leftrightarrow f(n) \leqslant acg(n), \forall n \geqslant m, \forall a \geqslant 0 \\ &\Leftrightarrow f(n) \leqslant c'g(n), \forall n \geqslant m, c' = ac > 0 \\ &\Leftrightarrow af(n) = \mathcal{O}(g(n)) \end{aligned}$$

Introduco una costante a

raccolgo ac sotto un'unica costante c'

per definizione

□

Ad esempio $2 \log n = \Theta(\log n)$, ignoriamo quindi le costanti numeriche.

Teorema 4 (Sommatoria, sequenza di algoritmi)**.**

$$\begin{cases} f_1(n) = \mathcal{O}(g_1(n)) \\ f_2(n) = \mathcal{O}(g_2(n)) \end{cases} \Rightarrow f_1(n) + f_2(n) = \mathcal{O}(\mathsf{max}(g_1(n), g_2(n)))$$
$$\begin{cases} f_1(n) = \Omega(g_1(n)) \\ f_2(n) = \Omega(g_2(n)) \end{cases} \Rightarrow f_1(n) + f_2(n) = \Omega(\mathsf{max}(g_1(n), g_2(n)))$$

Dimostrazione.

$$\begin{aligned} f_1(n) = \mathcal{O}(g_1(n)) \wedge f_2 n = \mathcal{O}(g_2(n)) &\Rightarrow \\ f_1(n) \leqslant c_1 g_1(n) \wedge f_2(n) \leqslant c_2 g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leqslant c_1 g_1(n) + c_2 g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leqslant \mathsf{max}\{c_1, c_2\}(2 \cdot \mathsf{max}(g_1(n), g_2(n)) &\Rightarrow \\ f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n)) & \end{aligned}$$

per definizione

raccolgo

per definizione

□

Ad esempio se ripeto due volte un algoritmo lineare, l'algoritmo risultante sarà comunque lineare. Mentre se ho un algoritmo lineare ed un algoritmo quadratico, la complessità risultante è una combinazione delle due, il limite superiore della complessità totale sarà quindi quadratica.

Teorema 5 (Cicli annidati)**.** *Se $f_1(n)$ viene ripetuto $f_2(n)$ volte, allora $f_1(n) \cdot f_2(n)$ è limitato superiormente dal prodotto delle complessità e limitato inferiormente dal prodotto delle complessità.*

$$\begin{cases} f_1(n) = \mathcal{O}(g_1(n)) \\ f_2(n) = \mathcal{O}(g_2(n)) \end{cases} \Rightarrow f_1(n) \cdot f_2(n) = \mathcal{O}(g_1(n) \cdot g_2(n))$$
$$\begin{cases} f_1(n) = \Omega(g_1(n)) \\ f_2(n) = \Omega(g_2(n)) \end{cases} \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

La dimostrazione è molto semplice.

Dimostrazione.

$$\begin{aligned} f_1(n) = \mathcal{O}(g_1(n)) \wedge f_2 n = \mathcal{O}(g_2(n)) &\Rightarrow \\ f_1(n) \leqslant c_1 g_1(n) \wedge f_2(n) \leqslant c_2 g_2(n) &\Rightarrow \\ f_1(n) \cdot f_2(n) \leqslant c_1 c_2 g_1(n) g_2(n) & \end{aligned} \quad \begin{aligned} &\text{per definizione} \\ &\text{raccolgo, } c_1 c_2 > 0 \end{aligned}$$

□

Ad esempio, se ripeto n volte un algoritmo di costo $\log n$, l'algoritmo complessivo avrà un costo di $n \log n$.

Teorema 6 (Simmetria)**.** *Se $f(n)$ è limitata superiormente ed inferiormente da $g(n)$, allora anche $g(n)$ è limitata inferiormente e superiormente da $f(n)$.*

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Vuol dire semplicemente che $2n^2 + 7 = \Theta(n^2) \Leftrightarrow n^2 = \Theta(2n^2 + 7)$.

La dimostrazione avviene tramite la proprietà di dualità:

Dimostrazione.

$$\begin{aligned} f(n) = \Theta(g(n)) \quad &\Rightarrow \quad f(n) = \mathcal{O}(g(n)) \quad \Rightarrow \quad g(n) = \Omega(f(n)) \\ f(n) = \Theta(g(n)) \quad &\Rightarrow \quad f(n) = \Omega(g(n)) \quad \Rightarrow \quad g(n) = \mathcal{O}(f(n)) \end{aligned}$$

□

Teorema 7 (Transitività)**.** *Se $f(n)$ è limitata superiormente da $g(n)$, e $g(n)$ è limitata superiormente da $h(n)$, allora $f(n)$ è limitata superiormente da $h(n)$.*

$$\begin{cases} f(n) = \mathcal{O}(g(n)) \\ g(n) = \mathcal{O}(h(n)) \end{cases} \Rightarrow f(n) = \mathcal{O}(h(n))$$

La dimostrazione è banale:

Dimostrazione.

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}(h(n)) &\Rightarrow \\ f(n) \leqslant c_1 g(n) \wedge g(n) \leqslant c_2 h(n) &\Rightarrow \\ f(n) \leqslant c_1 c_2 h(n) &\Rightarrow \\ f(n) = \mathcal{O}(h(n)) & \end{aligned} \quad \begin{aligned} &\text{applico la definizione} \\ &\text{sostituisco } g(n) \text{ con } c_2 h(n) \\ &c_1 c_2 > 0, \text{ elimino la costante} \end{aligned}$$

□

Altre funzioni di costo

Vogliamo provare che $\boxed{\log n = \mathcal{O}(n)}$

Dimostriamo per induzione che $\exists c > 0, \exists m \geqslant 0 : \log n \leqslant cn, \forall n \geqslant m$.

- **caso base** ($n = 1$):

$$\begin{aligned} \log n &\leqslant cn \\ \log 1 &\leqslant c \cdot 1 \\ 0 &\leqslant c \end{aligned} \quad \begin{array}{l} \\ \searrow\ n = 1 \\ \searrow\ \textit{semplifico} \end{array}$$

- **ipotesi induttiva**: $\log k \leqslant ck, \forall k \leqslant n$
- **passo induttivo** dimostriamo la proprietà per $n + 1$:

$$\begin{aligned} \log(n+1) \leqslant \log(n+n) &= \log 2n \quad \forall n \geqslant 1 && \searrow\ \log ab = \log a + \log b \\ &= \log 2 + \log n && \searrow\ \log_2 2 = 1 \\ &= 1 + \log n && \searrow\ \textit{per ipotesi induttiva} \\ &\leqslant 1 + cn && \searrow\ \textit{obiettivo} \\ &\overset{?}{\leqslant} c(n+1) && \searrow\ \textit{metto a confronto} \\ 1 + cn &\leqslant c(n+1) && \searrow\ \textit{moltiplico} \\ 1 + \cancel{cn} &\leqslant \cancel{cn} + c && \searrow\ \textit{semplifico} \\ 1 &\leqslant c \end{aligned}$$

Classificazione delle funzioni

È possibile trarre un ordinamento dalle principali espressioni, estendendo le relazioni che abbiamo dimostrato fino ad ora. Per ogni $r < s, h < k, a < b$:

$$\mathcal{O}(1) \subset \mathcal{O}(\log^r n) \subset \mathcal{O}(\log^s n) \subset \mathcal{O}(n^h) \subset \mathcal{O}(n^h \log^r n) \subset \mathcal{O}(n^h \log^s n) \subset \mathcal{O}(n^k) \subset \mathcal{O}(a^n) \subset \mathcal{O}(b^n)$$

Ricordati che $\log^r(n) = (\log n)^r$. Non è detto che gli esponenti siano interi. $\sqrt[1000]{n}$ cresce più velocemente di $\log^2 n$. n cresce meno velocemente di $n \log n$, il quale a sua volta cresce meno velocemente di $n \log^2 n$.

3.2 Analisi per livelli

Nell'analisi per livelli (o dell'albero di ricorsione) andiamo sostanzialmente a "srotolare" la ricorrenza in un albero, i cui nodi rappresentano i costi dei vari livelli della ricorsione.

3.2.1 Esempi di analisi per livelli

Analisi per livelli dell'algoritmo della ricerca binaria

Equazione di ricorrenza della ricerca binaria:

$$T(n) = \begin{cases} T(n/2) + b & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Assumiamo per semplicità che $n = 2^k$, ovvero che $k = \log n$. È possibile quindi risolvere questa ricorrenza nel seguente modo:

$$\begin{aligned} T(n) &= b + T\left(\tfrac{n}{2}\right) && \text{espando } T\left(\tfrac{n}{2}\cdot\tfrac{1}{2}\right)+b \\ &= b + b + T\left(\tfrac{n}{4}\right) && \text{espando } T\left(\tfrac{n}{4}\cdot\tfrac{1}{2}\right)+b \\ &= b + b + b + T\left(\tfrac{n}{8}\right) \\ &= \cdots && \text{svolgo } \log n \text{ operazioni} \\ &= \underbrace{b + b + \ldots + b}_{\log n} + T(1) && \text{semplifico} \\ &= b \log n + T(1) && \text{eliminazione delle costanti} \\ &= \Theta(\log n) \end{aligned}$$

Analisi per livelli del primo tentativo della moltiplicazione di Karatsuba

Equazione di ricorrenza:

$$T(n) = \begin{cases} 4T(n/2) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

È possibile risolvere questa ricorrenza nel modo seguente:

$$\begin{aligned} T(n) &= n + 4T(n/2) \\ &= n + 4\left(4T\left(\frac{n}{2}\cdot\frac{1}{2}\right) + \frac{n}{2}\right) && 4T\left(\tfrac{n}{2}\cdot\tfrac{1}{2}\right)+\tfrac{n}{2} \\ &= n + 4n/2 + 16T(n/4) && \text{semplifico} \\ &\quad n + 2n + 16\left(4T\left(\frac{n}{4}\cdot\frac{1}{2}\right) + \frac{n}{4}\right) && 4T\left(\tfrac{n}{4}\cdot\tfrac{1}{2}\right)+\tfrac{n}{4},\ 4n/2 = 2n \\ &= n + 2n + 16n/4 + 64T(n/8) && \text{semplifico} \\ &= \cdots && \text{svolgo } \log n \text{ operazioni} \\ &= \underbrace{n + 2n + 4n + 8n + \cdots + 2^{\log n - 1} n}_{sommatoria} + 4^{\log n} T(1) \\ &= n \sum_{j=0}^{\log n - 1} 2^j + 4^{\log n} && \text{raccolgo} \end{aligned}$$

Ciò che abbiamo ottenuto è una forma chiusa, non più un'equazione di ricorrenza: non è ancora nella sua forma definitiva, dobbiamo ancora trattarla.

$$
\begin{aligned}
T(n) &= n \sum_{j=0}^{\log n - 1} 2^j + 4^{\log n} && \text{Applico } \forall x \neq 1 : \textstyle\sum_{j=0}^{k} x^j = \frac{x^{k+1}-1}{x-1}, \text{ dove } k = \log n - 1 \\
&= n \cdot \frac{2^{\log n} - 1}{2 - 1} + 4^{\log n} && 2^{\log n} = n^{\log_2 2} = n^1 \\
&= n(n-1) + 4^{\log n} && \text{moltiplico: } n(n-1) = n^2 - n \\
&= n^2 - n + n^2 && \text{semplifico: } 4^{\log n} = n^{\log 4} = n^2 \\
&= 2n^2 - n && \text{raccolgo } n^2 \\
&= \Theta(n^2) && \text{regola generale } \uparrow
\end{aligned}
$$

Possiamo concludere che $T(n) = n + 4T(n/2) = \Theta(n^2)$.

Modifichiamo l'esempio precendente

Esaminiamo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 4T(n/2) + n^3 & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Per semplicità consideriamo $n = 2^\ell$.

livello	dim.	costo chiam.	no. chiamate	costo livello
0	n	n^3	1	n^3
1	$n/2$	$(n/2)^3$	4	$4(n/2)^3$
2	$n/4$	$(n/4)^3$	16	$16(n/4)^3$
3	$n/8$	$(n/8)^3$	64	$64(n/8)^3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$n/2^i$	$(n/2^i)^3$	4^i	$4^i(n/2^i)^3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\ell - 1$	$n/2^{\ell-1}$	$(n/2^{\ell-1})^3$	$4^{\ell-1}$	$4^{\ell-1}(n/2^{\ell-1})^3$
$\ell = \log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Sommando il costo di tutti i $\ell - 1$ livelli ed il livello ℓ-esimo otteniamo:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log n - 1} 4^i \cdot \frac{n^3}{2^{3i}} + 4^{\log n} && \text{semplifico: } 4^i = 2^{2i}, \text{ porto fuori: } n^3 \\
&= n^3 \sum_{i=0}^{\log n - 1} \frac{2^{2i}}{2^{3i}} + 4^{\log n} && \text{semplifico: } \tfrac{2^{2i}}{2^{3i}} = \left(\tfrac{1}{2}\right)^i \\
&= n^3 \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i + 4^{\log n} && \text{cambio di base, } 4^{\log n} = n^{\log 4} = n^2 \\
&= n^3 \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i + n^2 && \text{estensione della sommatoria ad } \infty \\
&\leqslant n^3 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + n^2 && \text{Serie geometrica infinita decrescente: } \forall x, |x| < 1 : \textstyle\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \text{ dove } x = \frac{1}{2} \\
&= n^3 + \frac{1}{1 - \frac{1}{2}} + n^2 && \text{semplifico: } \tfrac{1}{1-\frac{1}{2}} = 2 \\
&= 2n^3 + n^2
\end{aligned}$$

Abbiamo dimostrato che $T(n) \leqslant 2n^3 + n^2$, possiamo quindi affermare che $T(n) = \mathcal{O}(n^3)$, ma non possiamo affermare che $T(n) = \Theta(n^3)$ poiché abbiamo dimostrato solo un limite superiore.

Suggerimento. Tutte le volte che notiamo in un'equazione di riccorrenza una parte polinomiale, come ad esempio n^3, possiamo dire con certezza che l'equazione è $\Omega(n^3)$.

Ora possiamo affermare che $T(n) = \Theta(n^3)$.

Modifichiamo l'esempio precendente ulteriormente

Esaminiamo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 4T(n/2) + n^2 & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Cambia il costo della chiamata, che passa dall'essere n^3 all'essere n^2. Di conseguenza cambia anche il costo del livello.

Livello	Dim.	Costo chiam.	no. chiamate	Costo livello
0	n	n^2	1	n^2
1	$n/2$	$(n/2)^2$	4	$4(n/2)^2$
2	$n/4$	$(n/4)^2$	16	$16(n/4)^2$
3	$n/8$	$(n/8)^2$	64	$64(n/8)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$n/2^i$	$(n/2^i)^2$	4^i	$4^i(n/2^i)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\ell - 1$	$n/2^{\ell-1}$	$(n/2^{\ell-1})^2$	$4^{\ell-1}$	$4^{\ell-1}(n/2^{\ell-1})^2$
$\ell = \log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Sommando il costo di tutti i $\ell - 1$ livelli ed il livello ℓ-esimo otteniamo:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n - 1} \frac{n^2}{2^{2i}} \cdot 4i + 4^{\log_2 n} && \text{semplifico: } 4^i = 2^{2i},\ 4^{\log n} = n^{\log_2 4} = n^2 \\ &= n^2 \sum_{i=0}^{\log n - 1} \frac{2^{2i}}{2^{2i}} + n^2 && \text{porto fuori: } n^2 \\ &= n^2 \sum_{i=0}^{\log n - 1} 1 + n^2 && \text{semplifico: } \tfrac{2^{2i}}{2^{2i}} = 1 \\ &= n^2 \log n + n^2 && \text{svolgo la sommatoria: } \textstyle\sum_{i=0}^{\log n - 1} 1 = \log n \\ &= \Theta(n^2 log n) && \text{regola generale} \end{aligned}$$

Conclusioni

Per riassumere, se consideriamo i termini non ricorrenti, n ha prodotto n^2, n^2 ha prodotto $n^2 \log n$ ed n^3 ha prodotto n^3. Quando avremo a disposizione lo strumento "master theorem" riusciremo semplicemente guardando l'equazione di ricorrenza a capire quale complessità essa produce.

3.3 Metodo di sostituzione

Vediamo ora un ulteriore meccanismo per risolvere le equazioni di ricorrenza in quanto il metodo precendente in alcuni casi può non esserci di aiuto. Il metodo di sostituzione è un metodo in cui si cerca di "indovinare" (*guess*) una soluzione in base alla propria esperienza ed in seguito si dimostra che questa intuizione è corretta tramite dimostrazione per induzione.

Primo esercizio

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Risolvendolo tramite il metodo precendente otteniamo:

$$\begin{aligned} T(n) &= n \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i && \text{estensione della sommatoria ad } \infty \\ &\leqslant n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i && \text{Serie geometrica decrescente infinita: } \forall x, |x| < 1 : \textstyle\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \text{ dove } x = \frac{1}{2} \\ &\leqslant n \frac{1}{1-\frac{1}{2}} && \frac{1}{1-\frac{1}{2}} = 2 \\ &= 2n \end{aligned}$$

Sapendo già il risultato proviamo – per tentativi – a dimostrare che $\boxed{T(n) = \mathcal{O}(n)}$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn, \forall n \geqslant m$

- **caso base** dimostriamo $T(1)$:
$$T(1) = 1 \overset{?}{\leqslant} c \cdot 1 \iff c \geqslant 1$$
- **ipotesi induttiva** $\forall k < n : T(n) \leqslant ck$, ossia assumiamo che per tutti i valori più piccoli di n la dimostrazione sia già stata fatta.
- **ipotesi induttiva** dimostriamo la disequazione per $T(n)$:
$$\begin{aligned} T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{2} \rfloor \\ &\leqslant c\lfloor \tfrac{n}{2} \rfloor + n && \text{semplifico l'intero inferiore} \\ &\leqslant c\tfrac{n}{2} + n && \text{raccolgo } n \\ &= (\tfrac{c}{2} + 1)n && \text{obiettivo} \\ &= (\tfrac{c}{2} + 1)n \overset{?}{\leqslant} cn \\ &\Rightarrow \tfrac{c}{2} + 1 \leqslant c && \text{semplifico} \\ &\Rightarrow c \geqslant 2 \end{aligned}$$

  Abbiamo quindi provato che $T(n) \leqslant cn$, con due diversi valori della nostra costante c: nel caso base è risultata $c \geqslant 1$, mentre nel passo induttivo è risultata pari a $c \geqslant 2$. Quindi la nostra ipotesi è valida per qualsiasi valore di c t.c. $c \geqslant a$ e $c \geqslant 2$, ovvero $\forall c \geqslant 2$.

  Quanto abbiamo dimostrato vale per $n = 1$ e per tutti i valori di n successivi, di conseguenza $m = 1$.

Al solo scopo didattico (in quanto lo potremmo dedurlo dal termine non ricorsivo) proviamo passo passo anche il limite inferiore, ossia che $\boxed{T(n) = \Omega(n)}$

Limite inferiore: Dobbiamo dimostrare che $\exists d > 0, \exists m \geqslant 0 : T(n) \geqslant dn, \forall n \geqslant m$.

Nota. Usiamo la costante d al solo scopo di non confonderla la costante c usata nella dimostrazione precendente.

- **caso base** $T(1)$:

$$T(1) = 1 \overset{?}{\geqslant} d \cdot 1 \iff d \leqslant 1$$

- **ipotesi induttiva** $\forall k < n : T(k) \geqslant ck$, ossia assumiamo che per tutti i valori più piccoli di n la mia dimostrazione sia già stata fatta.

- **ipotesi induttiva** dimostriamo la disequazione per $T(n)$

$$\begin{aligned}
T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + n && \text{sost. ip. ind. con } d = \tfrac{n}{2} \\
&\geqslant d\lfloor \tfrac{n}{2} \rfloor + n && \text{semplifico l'intero inferiore} \\
&\geqslant d\tfrac{n}{2} - 1 + n && \text{raccolgo } n \\
&= \left(\tfrac{d}{2} - \tfrac{1}{n} + 1\right) n && \text{obiettivo} \\
&= \left(\tfrac{d}{2} - \tfrac{1}{n} + 1\right) n \overset{?}{\geqslant} dn && \\
&\Rightarrow \tfrac{d}{2} - \tfrac{1}{n} + 1 \geqslant n && \text{semplifico} \\
&\Rightarrow d \leqslant 2 - \tfrac{2}{n}
\end{aligned}$$

  Abbiamo quindi provato che $T(n) \geqslant dn$, con due diversi valori della nostra costante d: nel caso base è risultata $d \leqslant 1$, mentre nel passo induttivo è risultata pari a $d \leqslant 2 - \frac{2}{n}$. $d = 1$ è valore che soddisfa entrambre le disequazioni, dimostra quindi la nostra tesi.

  Quanto abbiamo dimostrato vale per $n = 1$ e per tutti i valori di n successivi, di conseguenza $m = 1$.

  Per concludere abbiamo provato che $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n$ è limitata sia superiormente $T(n) = \mathcal{O}(n)$, sia inferiormente $T(n) = \Omega(n)$, possiamo affermare quindi con assoluta certezza che $T(n) = \Theta(n)$.

Nota che avremmo potuto evitare la dimostrazione del limite inferiore osservando che

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + n \geqslant n \overset{?}{\geqslant} dn$$

l'ultima disequazione risulta vera per $d \leqslant 1$, la quale è una condizione identica a quella del caso base. Nota che non abbiamo fatto nemmeno ricordo all'ipotesi induttiva.

Cosa succede se si sbaglia l'intuizione

$$T(n) = \begin{cases} T(n-1) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Nota. L'equazione di riccorrenza rappresenta il caso in cui selection sort sia espresso in forma ricorsiva.

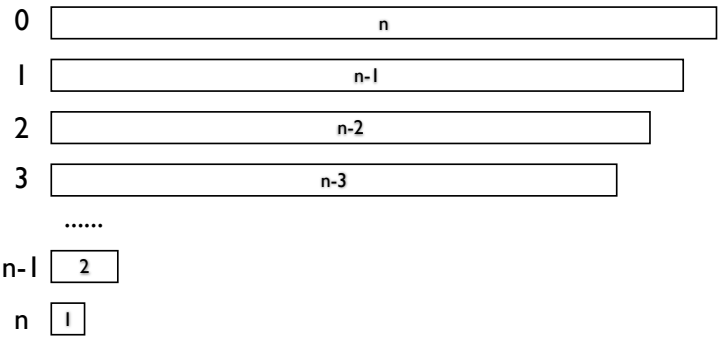


Figure 3.1: Rappresentazione della ricorrenza lineare.

Possiamo rappresentare la funzione nel seguente modo:

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

Effettuiamo un tentativo e proviamo a dimostrare che $\boxed{T(n) = \mathcal{O}(n)}$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn, \forall n \geqslant m$.

- **caso base** lo saltiamo perché vedremmo subito che è sbagliato.
- **ipotesi induttiva** $\forall k < n : T(k) \leqslant ck$.
- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$
\begin{aligned}
T(n) &= T(n-1) + n && \text{sost. ip. ind. con } k = n-1 \\
&= c(n-1) + n && \text{moltiplico} \\
&= cn - c + n && \text{raccolgo } n \\
&= (c+1)n - c && \text{rimuovo l'elemento negativo} \\
&\leqslant (c+1)n && \text{obiettivo} \\
&= (c+1)n \overset{?}{\leqslant} cn && \text{semplifico} \\
&\Rightarrow c + 1 \leqslant c
\end{aligned}
$$

Possiamo notare che l'ultima disequazione risulta impossibile. Dunque quando proviamo a dimostrare qualcosa di sbagliato non riusciremo a dimostrarlo.

Difficoltà matematica

$$
T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1 & n > 1 \\ 1 & n \leqslant 1 \end{cases}
$$

Nota. È possibile ottenere questa equazione di riccorrenza dall'algoritmo che calcola il minimo di un vettore non ordinato in maniera ricorsiva.

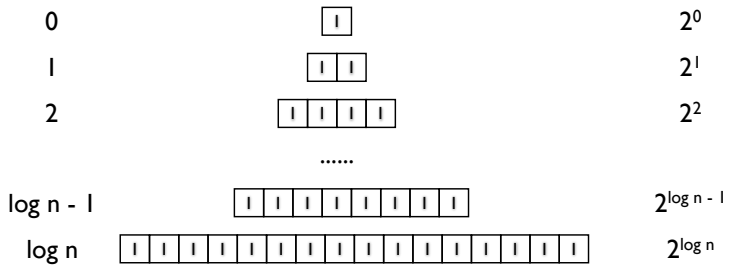


Figure 3.2: Rappresentazione della ricorsione.

$$
T(n) = \sum_{i=0}^{\log n} 2^i = n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = \mathcal{O}(n)
$$

Effettuiamo un tentativo per $\boxed{T(n) = \mathcal{O}(n)}$

- **ipotesi induttiva**: $\forall k < n : T(k) \leqslant ck$.
- **passo induttivo**: dimostriamo la disequazione per $T(n)$:

$$
\begin{aligned}
T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + 1 && \text{sost. ip. ind.}\\
&= c(\lfloor \tfrac{n}{2} \rfloor) + c(\lceil \tfrac{n}{2} \rceil) + 1 && \text{semplifichiamo, } \lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil = n\\
&= cn + 1 && \text{obiettivo}\\
&= cn + 1 \overset{?}{\leqslant} cn && \text{semplifico}\\
&\Rightarrow 1 \leqslant 0
\end{aligned}
$$

Anche in questo caso notiamo che l'ultima disequazione risulta impossibile, ma – a differenza del caso precendente – non riusciamo a dimostrare il passo induttivo per un termine di ordine inferiore. Il tentativo risulta quindi errato.

Proviamo quindi ad utilizzare un'ipotesi induttiva *più stretta*.

- **ipotesi induttiva più stretta**: $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn - b, \forall n \geqslant m,\ b > 0$.

  Abbiamo introdotto una costante $b > 0$ nella nostra tesi, questa modifica ci permetterà di dimostrare correttamente il passo induttivo.

- **ipotesi induttiva** $\exists b > 0, \forall k < n : T(k) \leqslant ck - b$.

- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$
\begin{aligned}
T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + 1 && \text{sost. ip. ind.}\\
&= c(\lfloor \tfrac{n}{2} \rfloor) - b + c(\lceil \tfrac{n}{2} \rceil) - b + 1 && \text{semplifichiamo}\\
&= cn - 2b + 1 && \text{obiettivo}\\
&= cn - 2b + 1 \overset{?}{\leqslant} cn - b && \text{semplifico}\\
&\Leftrightarrow -2b + 1 \leqslant -b\\
&\Leftrightarrow b \geqslant 1
\end{aligned}
$$

- **caso base**

$$T(1) = 1 \overset{?}{\leqslant} c \cdot 1 - b \Leftrightarrow c \geqslant b + 1$$

  Per concludere abbiamo provato che $T(n) \leqslant cn - b \leqslant cn$ con diversi valori delle costanti c e b, nel passo induttivo $\forall b \geqslant 1, \forall c$, nel caso base $\forall c \geqslant b + 1$. Una coppia di valori di b e c che rispettano queste disequazioni sono $b = 1$, $c = 2$.

  Questo vale per $n = 1$, e per tutti i valori di n successivi, quindi per $m = 1$.

  Abbiamo quindi provato che $T(n) = \mathcal{O}(n)$.

Dimostriamo il limite inferiore facendo un tentativo per $\boxed{T(n) = \Omega(n)}$

Dobbiamo dimostrare che $\exists d > 0, \exists m \geqslant 0 : T(n) \geqslant dn, \forall n \geqslant m$.

- **passo induttivo**

$$
\begin{aligned}
T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + 1 && \text{sost. ip. ind.}\\
&\geqslant d(\lfloor \tfrac{n}{2} \rfloor) + d(\lceil \tfrac{n}{2} \rceil) + 1 && \text{semplifichiamo}\\
&= dn + 1 && \text{obiettivo}\\
&= dn + 1 \overset{?}{\geqslant} dn
\end{aligned}
$$

  L'ultima disequazione risulta vera $\forall d$.

- **caso base**

$$T(n) = 1 \geqslant d \cdot 1 \iff d \leqslant 1$$

  Abbiamo quindi provato che $T(n) = \Omega(n)$

Problemi con i casi base

$$T(n) = \begin{cases} 2T(\lfloor \frac{n}{2} \rfloor) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Proviamo a visualizzarlo graficamente così:

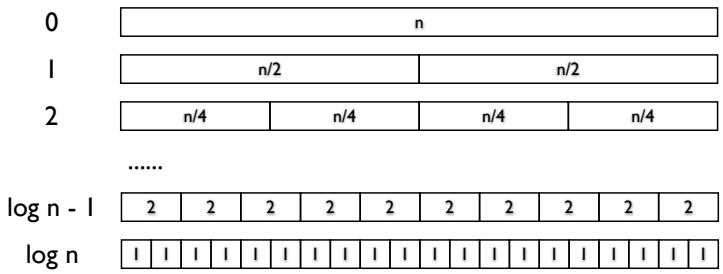


Figure 3.3: Rappresentazione della ricorsione.

Nota. È molto simile all'equazione dell'algoritmo mergeSort che sappiamo avere una complessità di $\mathcal{O}(n \log n)$.

Effettuiamo un tentativo per $\boxed{T(n) = \mathcal{O}(n \log n)}$

Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn \log n, \forall n \geqslant m$.

- **Ipotesi induttiva**: $\exists c > 0, \forall k < n : T(k) \leqslant ck \log k$
- **Passo di induzione**: Dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= 2T(\lfloor \tfrac{n}{2} \rfloor) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{2} \rfloor \\ &\leqslant 2c\lfloor \tfrac{n}{2} \rfloor \log \lfloor \tfrac{n}{2} \rfloor + n && \text{rimuovo l'intero inferiore} \\ &\leqslant 2c\tfrac{n}{2} \log \tfrac{n}{2} + n && \text{semplifico} \\ &= cn \log \tfrac{n}{2} + n && \log \tfrac{n}{2} = \log n - \log_2 2 = \log n - 1 \\ &= cn(\log n - 1) + n && \text{moltiplico} \\ &= cn \log n - cn + n && \text{obiettivo} \\ &\leqslant cn \log n - cn + n \overset{?}{\leqslant} cn \log n && \text{semplifico} \\ &\Leftrightarrow -cn + n \leqslant 0 \\ &\Leftrightarrow c \geqslant 1 \end{aligned}$$

- **caso base**: dimostriamo la disequazione per $T(1)$

$$T(1) = 1 \overset{?}{\leqslant} 1 \cdot c \log 1 = 0 \Rightarrow 1 \nleq 0$$

  È falso, ma non è un problema, non a caso si chiama notazione asintotica: il valore di m lo possiamo scegliere noi.

- **caso base**: dimostriamo la disequazione per $T(2)$, $T(3)$

$$\begin{aligned} T(2) &= 2T(\lfloor \frac{2}{2} \rfloor) + 2 = 4 \leqslant 1 \cdot c \cdot 2 \log 2 \Leftrightarrow c \geqslant 2 \\ T(3) &= 2T(\lfloor \frac{3}{2} \rfloor) + 3 = 5 \leqslant 1 \cdot c \cdot 3 \log 3 \Leftrightarrow c \geqslant \frac{5}{3 \log 3} \\ T(4) &= 2T(\lfloor \frac{4}{2} \rfloor) + 4 = 2T(\lfloor 2 \rfloor) + 4 \end{aligned}$$

  Non è necessario provare la terza disequazione, in quanto viene espressa in base ai casi base diversi da $T(1)$ che sono già stati dimostrati e quindi possono costituire la base per la nostra induzione.

Riassumendo:

Abbiamo provato che $T(n) \leqslant cn \log n$

- nel passo induttivo: $\forall c \geqslant 1$
- nel caso base: $\forall c \geqslant 2, c \geqslant \frac{5}{3 \log 3}$

Visto che sono tutte disequazioni con il segno $\geqslant$, è sufficiente utilizzare $c \geqslant \max\left\{1, 2, \frac{5}{3 \log 3}\right\}$

Questo vale per $n = 2$, $n = 3$, e per tutti i valori di n successivi, quindi $m = 2$.

Ultimo esercizio

$$T(n) = \begin{cases} 9T(\lfloor n/3 \rfloor) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Effettuiamo un tentativo $\boxed{T(n) = \mathcal{O}(n^2)}$

Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn^2, \forall n \geqslant m$

- **ipotesi induttiva**: $\exists c > 0 : T(k) \leqslant ck^2, \forall k < n$
- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= 9T\left(\lfloor \tfrac{n}{3} \rfloor\right) + n \\ &\leqslant 9c\left(\lfloor \tfrac{n}{3} \rfloor\right)^2 + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{3} \rfloor \\ &\leqslant 9c\left(\tfrac{n^2}{9}\right) + n && \text{rimuovo l'intero inferiore} \\ &= cn^2 + n && \text{semplifico il 9} \\ &= cn^2 + n \leqslant cn^2 && \text{obiettivo} \end{aligned}$$

L'ultima disequazione risulta falsa per un termine di ordine inferiore: proviamo quindi a modificare l'ipotesi induttiva e a ripetere il passo induttivo.

- **ipotesi induttiva più stretta** $\exists c > 0 : T(k) \leqslant c(k^2 - k), \forall k < n$
- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= 9T\left(\lfloor \tfrac{n}{3} \rfloor\right) + n \\ &\leqslant 9c\left(\lfloor \tfrac{n}{3} \rfloor^2 - \lfloor \tfrac{n}{3} \rfloor\right) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{3} \rfloor \\ &\leqslant 9c\left(\left(\tfrac{n}{3}\right)^2 - \tfrac{n}{3}\right) + n && \text{rimuovo l'intero inferiore} \\ &\leqslant 9c\left(\tfrac{n^2}{9} - \tfrac{n}{3}\right) + n && \text{svolgo la potenza} \\ &\leqslant cn^2 - 3cn + n && \text{moltiplico} \\ &\leqslant cn^2 - 3cn + n \overset{?}{\leqslant} cn^2 - cn && \text{obiettivo} \\ &\Leftrightarrow 2cn \geqslant cn \\ &\Leftrightarrow c \geqslant \tfrac{1}{2} && \text{semplifico} \end{aligned}$$

- **caso base**

$$\begin{aligned} &T(1) = 1 \leqslant c(1^2 - 1) = 0, falso \\ &T(2) = 9T(0) + 2 = 11 \leqslant c(2^2 - 2) \Leftrightarrow c \geqslant {}^{11}/_{2} \\ &T(3) = 9T(1) + 3 = 12 \leqslant c(3^2 - 3) \Leftrightarrow c \geqslant {}^{12}/_{6} \\ &T(4) = 9T(1) + 4 = 13 \leqslant c(4^2 - 4) \Leftrightarrow c \geqslant {}^{13}/_{12} \\ &T(5) = 9T(1) + 5 = 14 \leqslant c(5^2 - 5) \Leftrightarrow c \geqslant {}^{14}/_{20} \\ &T(6) = 9T(2) + 6 \end{aligned}$$

Non è necessario andare oltre poiché T(6) dipende da T(2) che è già stato dimostrato.

Riassumendo i parametri scelti sono:

- $c \geqslant \max\left\{\frac{1}{2}, \frac{11}{2}, \frac{12}{6}, \frac{13}{12}, \frac{14}{20}\right\}$
- $m = 1$

Nota che l'esempio combina le due difficoltà insieme, ma è stato creato artificiosamente: infatti se avessimo scelto come ipotesi più stretta $T(n) \leqslant cn^2 - bn$, il problema sui casi base non si sarebbe posto.

Abbiamo quindi dimostrato che $T(n) \leqslant c(n^2 - n) \leqslant cn^2, \forall n \geqslant 1, \forall c \geqslant \frac{14}{20}$, ossia che $T(n) = \mathcal{O}(n)$

Riassumendo

Il metodo di sostituzione è composto da tre parti:

1. si *indovina* una possibile soluzione e si formula un'ipotesi induttiva;
2. si *sostituisce* nella ricorrenza le espressioni $T(\cdot)$, utilizzando l'ipotesi induttiva;
3. si *dimostra* che la soluzione è valida anche per il caso base.

Bisogna fare attenzione:

- ad ipotizzare soluzioni troppo "strette"
- ad alcuni casi particolari che richiedono astuzie matematiche
- ai casi base in cui compare il logaritmo in quanto potrebbe complicare le cose

3.4 Metodo dell'esperto (o delle ricorrenze comuni)

Esiste un'ampia classe di ricorrenze che possono essere risolte facilmente facendo ricorso ad alcuni teoremi, ognuno dei quali si occupa di una classe particolare di equazioni di ricorrenza.

Teorema 8 (Ricorrenze lineari con partizione bilanciata)**.** *Siano a e b costanti intere tale che $a \geqslant 1$ e $b \geqslant 2$, e c, β costanti reali tali che $c > 0$ e $\beta \geqslant 0$. Sia $T(n)$ data dalla relazione di ricorrenza:*

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + cn^{\beta} & n > 1 \\ d & n \leqslant 1 \end{cases}$$

Posto $\alpha = \frac{\log a}{\log b} = \log_b a$, allora:

$$T(n) = \begin{cases} \Theta(n^{\alpha}) & \alpha > \beta \\ \Theta(n^{\alpha} \log n) & \alpha = \beta \\ \Theta(n^{\beta}) & \alpha < \beta \end{cases}$$

Commento Affrontiamo le equazioni di ricorrenza in cui la dimensione viene divisa in b parti, dove b dev'essere almeno pari a 2; l'algoritmo ricorsivo dev'essere richiamato a volte, dove a è almeno 1. Nella versione estesa, che vedremo fra poco, vedremo che i parametri a e b verranno "rilassati".

Prendiamo ad esempio il primo esercizio $T(n) = 4T(\frac{n}{2})$ e vediamo che si può risolvere semplicemente calcolando $\alpha = \log_b a = \log_2 4 = 2 > \beta = 1$, possiamo quindi concludere che $T(n) = \Theta(n^{\alpha}) = \Theta(n^2)$.

Dimostrazione del teorema delle ricorrenze lineari con partizione bilanciata. Assumiamo che n sia una potenza intera di b, ossia che $n = b^k$, $k = \log_b n$ poiché ci permetterà di semplificare i calcoli successivi ed è ininfluente sul risultato. Ad esempio supponiamo che l'input abbia dimensione $b^k + 1$ ($2^8 + 1 = 257$ bit), se estendiamo l'input fino ad una dimensione b^{k+1} ($2^8 + 1 = 512$ bit, facendo del *padding*, l'input sarebbe stato esteso al massimo di un fattore costante b (2 nel nostro caso), il che è ininfluente al fine della complessità computazionale.

Calcoliamo l'albero delle ricorrenze per la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + cn^{\beta} & n > 1 \\ d & n \leqslant 1 \end{cases}$$

livello	dim.	costo chiam.	no. chiamate	costo livello
0	b^k	$c\,b^{k\beta}$	1	$cb^{k\beta}$
1	b^{k-1}	$c\,b^{(k-1)\beta}$	a	$a \cdot c\,b^{(k-1)\beta}$
2	b^{k-2}	$c\,b^{(k-2)\beta}$	a^2	$a^2 \cdot c\,b^{(k-2)\beta}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	b^{k-i}	$c\,b^{(k-i)\beta}$	a^i	$a^i \cdot c\,b^{(k-i)\beta}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	b	$cb^{(k-(k-1))\beta} = c\,b^{\beta}$	a^{k-1}	$a^{k-1} \cdot c\,b^{\beta}$
k	1	d	a^k	$a^k \cdot d$

Sommando i costi totali del k-esimo livello e dei livelli fino al $k-1$, si ottiene:

$$
\begin{aligned}
T(n) &= da^k + \sum_{i=0}^{k-1} a^i \cdot cb^{(k-i)\beta} && \text{moltiplico} \\
&= da^k + \sum_{i=0}^{k-1} a^i \cdot cb^{k\beta} \cdot b^{-i\beta} && \text{porto fuori i termini non dipendenti da } i \\
&= da^k + cb^{k\beta} \sum_{i=0}^{k-1} \frac{a^i}{b^{i\beta}} && \text{raccolgo } i \\
&= da^k + cb^{k\beta} \sum_{i=0}^{k-1} \left(\frac{a}{b^{\beta}}\right)^i
\end{aligned}
$$

A questo punto ho una formula chiusa ma non ancora nella sua forma definitiva.

Facciamo alcune osservazioni.

- $a^k = a^{\log_b n} = a^{\frac{\log n}{\log b}} = 2^{\log_2 a \frac{\log n}{\log b}} = 2^{\log_2 n \frac{\log a}{\log b}} = n^{\frac{\log a}{\log b}} = n^{\alpha}$
- $\alpha = \frac{\log a}{\log b} \Leftrightarrow \alpha \log b = \log a \Leftrightarrow \log b^{\alpha} = \log a \Leftrightarrow a = b^{\alpha}$
- poniamo $q = \frac{a}{b^{\beta}} = \frac{b^{\alpha}}{b^{\beta}} = b^{\alpha-\beta}$

Grazie alle osservazioni appena fatto possiamo sostituire i parametri nell'equazione finale:

$$
\begin{aligned}
T(n) &= da^k + cb^{k\beta} \sum_{i=0}^{k-1} \left(\frac{a}{b^{\beta}}\right)^i && \text{sostituisco: } \alpha^k \to n^{\alpha} \\
&= dn^{\alpha} + cb^{k\beta} \sum_{i=0}^{k-1} q^i && \text{sostituisco: } a \to b^{\alpha},\ q = \tfrac{b^{\alpha}}{b^{\beta}}
\end{aligned}
$$

Da qui si aprono tre possibilità, ossia che 1. $\alpha > \beta$ 2. $\alpha = \beta$ 3. $\alpha < \beta$ Studiamole una per una.

1. Caso $\boxed{\alpha > \beta}$, ne segue che $q = b^{\alpha-\beta} > 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i && \text{serie geometrica finita}\\
&= n^\alpha d + cb^{k\beta}\left[\frac{q^k-1}{q-1}\right] && \text{introduco la disequazione}\\
&\leqslant n^\alpha d + cb^{k\beta}\frac{q^k}{q-1} && \text{sostituisco: } q = \tfrac{a}{b^\beta} \Rightarrow q^k = \tfrac{a^k}{b^{k\beta}}\\
&= n^\alpha d + \frac{cb^{k\beta}a^k}{b^{k\beta}}\frac{1}{q-1} && \text{semplifico: } b^{k\beta}\\
&= n^\alpha d + \frac{ca^k}{q-1} && \text{sostituisco: } a^k = n^\alpha\\
&= n^\alpha d + \frac{cn^\alpha}{q-1} && \text{raccolgo per } n^\alpha\\
&= n^\alpha\left[d + \frac{c}{q-1}\right]
\end{aligned}
$$

   Visto che d, c e q sono tutti termini positivi e costanti possiamo concludere che n^α limita superiormente l'espressione e che quindi $T(n) = \mathcal{O}(n^\alpha)$. Infine per via della componente non ricorsiva dn^α, $T(n)$ è anche $\Omega(n^\alpha)$, possiamo concludere che $T(n) = \Theta(n^\alpha)$.

2. Caso $\boxed{\alpha = \beta}$, ne segue che $q = b^{\alpha-\beta} = 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i && q^i = 1^i = 1\\
&= n^\alpha d + cn^\beta k && 1 + 2 + \cdots + k - 1 = k\\
&= n^\alpha d + cn^\alpha k && \text{sostituisco: } \beta = \alpha\\
&= n^\alpha(d + ck) && \text{raccolgo per } n^\alpha\\
&= n^\alpha(d + c\tfrac{\log n}{\log b}) && \text{sostituisco: } k = \log_b n
\end{aligned}
$$

   Visto che d, c e $\log b$ sono tutti termini positivi e costanti e che non abbiamo introdotto disequazioni, possiamo affermare che $T(n) = \Theta(n^\alpha \log n)$.

3. Caso $\boxed{\alpha < \beta}$, ne segue che $q = b^{\alpha-\beta} < 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^{k}\sum_{i=0}^{k-1} q^i && \text{serie geometrica finita}\\
&= dn^\alpha + cb^{k}\left[\frac{q^k-1}{q-1}\right] && \text{inversione, } 1-q>0\\
&= dn^\alpha + cb^{k}\left[\frac{1-q^k}{1-q}\right] && \text{introduco la disequazione}\\
&\leqslant dn^\alpha + cb^{k}\left[\frac{1}{1-q}\right] && \text{sostituisco: } b^k = n\\
&= n^\alpha d + \frac{cn^\beta}{1-q}
\end{aligned}
$$

   $n^\alpha < n^\beta$ quindi considero il polinomio di grado maggiore, di conseguenza $T(n)$ è $\mathcal{O}(n^\beta)$. Poiché $T(n) = \Omega(n^\beta)$ per via del termine non ricorsivo, possiamo affermare che $T(n) = \Theta(n^\beta)$.

Fine dimostrazione. □

Teorema 9 (Ricorrenze lineari con partizione bilanciata estesa)**.** *Sia $a \geqslant 1$, $b > 1$, $f(n)$ asintoticamente positiva, e sia*

$$
T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ d & n \leqslant 1 \end{cases}
$$

Sono dati tre casi:

1. $\exists \varepsilon > 0 : f(n) = \mathcal{O}(n^{\alpha-\varepsilon})$ *allora* $T(n) = \Theta(n^{\alpha})$;
2. $f(n) = \Theta(n^{\alpha})$ *allora* $T(n) = \Theta(f(n) \log n)$;
3. $\exists \varepsilon > 0 : f(n) = \mathcal{O}(n^{\alpha+\varepsilon}) \wedge$
   $\exists c : 0 < c < 1, \exists m > 0 :$
   $af(\frac{n}{b}) \leqslant cf(n), \forall n \geqslant m$ *allora* $T(n) = \Theta(f(n))$.

Non vedremo la dimostrazione poiché è troppo complessa.

Commento Il teorema precendente funzionava con n^{β}, aveva una serie di condizioni semplificative, ora non ci sono più. Nel secondo caso se $f(n) = n^{\beta}$ ritorniamo esattamente al secondo caso del teorema precendente, ma qui possiamo prendere in considerazione funzioni più complesse.

Esercizi

Le soluzioni sono indicate fra parentesi.

- $T(n) = 9T(\frac{n}{3}) + n$ [$\mathcal{O}(n^{2-\varepsilon})$, con $\varepsilon < 1$]
- $T(n) = T(\frac{2}{3}n) + 1$ [$\Theta(n^0)$]
- $T(n) = 3T(\frac{n}{4}) + n \log n$ [$c = 3/4$, $m = 1$]
- $T(n) = 2T(\frac{n}{2}) + n \log n$ [nessun caso applicabile]

Teorema (Ricorrenze lineari di ordine costante)**.** *Siano* $\{a_1, a_2, \ldots, a_n\}$ *costanti intere non negative, con* h *costante positiva,* c *e* β *costanti reali tali che* $c > 0$ *e* $\beta \geqslant 0$*, e sia* $T(n)$ *definita dalla relazione di ricorrenza:*

$$T(n) = \begin{cases} \sum\limits_{1 \leqslant i \leqslant h} a_i T(n-1) + cn^{\beta} & n > m \\ \Theta(1) & n \leqslant m \leqslant h \end{cases}$$

Posto $a = \sum_{1 \leqslant i \leqslant h} a_i$*, allora:*

1. $T(n)$ *è* $\Theta(n^{\beta+1})$*, se* $a = 1$;
2. $T(n)$ *è* $\Theta(a^n n^{\beta})$*, se* $a \geqslant 2$.

Commento Questo teorema tratta ricorrenze lineari *di ordine costante* perché tutte le volte rimuoviamo dalla dimensione di input n una quantità costante.

Esercizi

Le soluzioni sono indicate fra parentesi.

- $T(n) = T(n-10) + n^2$ [costo polinomiale]
- $T(n) = T(n-2) - T(n-1) + 1$ [costo esponenziale]

Capitolo 4

Strutture dati

> *Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.*
>
> Steven S. Skiena, *The Algorithm Design Manual*

4.1 Strutture dati astratte

Alcune definizioni

Definizione 4.1.1 (Tipo di dato)**.** In un linguaggio di programmazione, un dato è un valore che una variabile può assumere.

Definizione 4.1.2 (Tipo di dato astratto)**.** Un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori.

Definizione 4.1.3 (Tipi di dato primitivi)**.** Sono dei tipi di dati che vengono forniti direttamente dal linguaggio. Come ad esempio: int (`+,-,*,/, %`), boolean (`!, &&, ||`).

Ogni tipo di dato deve distinguere *specifica* ed *implementazione* di un tipo di dato astratto. La *specifica* è astratta, il "manuale d'uso" che nasconde i dettagli implementativi all'utente, mentre l'*implementazione* è la realizzazione vera e propria del tipo di dato.

Tabella 4.1: Differenza fra specifica ed implementazione

Specifica	Implementazione
Numeri reali	IEEE-754
Pile	Pile basate su vettori Pile basate su puntatori
Code	Code basate su vettori circolari Code basate su puntatori

Definizione 4.1.4 (Strutture di dati)**.** Le strutture di dati sono collezioni di dati, caratterizzate più dall'organizzazione della collezione piuttosto che dal tipo dei dati in esse contenute.

Le strutture dati sono un modo sistematico per organizzare i dati e su di esse sono definite un insieme di operatori che permettono di manipolare la struttura stessa. Le strutture dati possono essere caratterizzate in vari modi:

- *lineari*/*non lineari*: presentano (o meno) una sequenza al loro interno;
- *statiche*/*dinamiche*: possono variare (o meno) di dimensione o di contenuto;
- *omogenee*/*disomogenee*: si riferisce ai dati contenuti al loro interno.

Tabella 4.2: Implementazione delle strutture dati nei vari linguaggi.
Nota che Java distingue chiaramente la specifica dall'implementazione

Tipo	Java	C++	Python
Sequenze	`List`, `Queue`, `Deque`, `LinkedList`, `ArrayList`, `Stack`, `ArrayDeque`	`list`, `forward_list`, `vector`, `stack`, `queue`, `dequeue`	`list`, `tuple`
Insiemi	`Set`, `TreeSet`, `HashSet`, `LinkedHashSet`	`set`, `unordered_set`	`set`, `fronzenset`
Dizionari	`Map`, `HashTree`, `HashMap`, `LinkedHashMap`	`map`, `unordered_map`	`dict`
Alberi	-	-	-
Grafi	-	-	-

4.2 Sequenza

Una sequenza è una struttura dati *dinamica*, *lineare* che rappresenta una sequenza *ordinata* di valori, dove un valore può comparire più di una volta. L'ordine all'interno della sequenza è importante.

Le operazioni ammesse su una sequenza sono:

- L'aggiunta e la rimozione elementi, specificando la posizione (tipicamente un intero), l'elemento s_1 si trova in posizione pos_i ed esistono posizioni fittizie pos_0 e pos_{n+1};
- Accesso diretto alla testa e coda;
- Accesso sequenziale a tutti gli altri elementi.

Specifica SEQUENCE

Una struttura dati *dinamica*, *lineare* che rappresenta una sequenza *ordinata* di valori, dove lo stesso valore può comparire più volte.

Sequence

```
// INTERPRETARE
bool isEmpty          // true se la sequenza è vuota
bool finished         // true se p è uguale a pos_0 o a
 pos_{n+1}

// LEGGERE
Pos head              // posizione del primo elemento
Pos tail              // posizione dell'ultimo elemento

// ITERARE
Pos next      // posizione dell'elem. che segue p
Pos prev      // posizione dell'elem. che precede p

// MODIFICA

// inserisce l'elemento di tipo Item nella
   posizione p,
// ritorna la nuova posizione,
// che diviene il predecessore di p
Pos insert(Pos p, Item v)
// rimuove l'elemento contenuto nella pos. p,
// ritorna il successore di p
Pos remove(Pos p)

// legge l'elemento di tipo Item
// contenuto nella posizione p
read(Pos p)

// scrive l'elemento v di tipo Item
// nella posizione p
write(Pos p, Item v)
```

4.2.1 Implementazione delle sequenze

Di seguito vengono presentati alcuni esempi d'utilizzo dell'implementazione delle sequenze nei diversi linguaggi di programmazione utilizzati oggigiorno.

Codice 4.1: Implementazione delle liste in Java

```
List<String> lista = new LinkedList<String>();
lista.add("two");
lista.addFirst("one");
lista.addLast("three");

Result: [ "one", "two", "three" ]
```

Codice 4.2: Implementazione delle liste in C++

```
std::list<int> lista;
lista.push_front(2);
lista.push_front(1);
lista.push_back(3);

Result: [1,2,3]
```

Codice 4.3: Implementazione delle liste in Python

```
lista = ["one", "three"]
lista.insert(1, "two")

Result: [ 'one', 'two', 'three' ]
```

4.3 Insiemi

Un insieme è una struttura dati *dinamica*, *non lineare* che memorizza una *collezione non ordinata di elementi* senza valori ripetuti. L'ordinamento fra elementi è dato dall'eventuale relazione d'ordine definita sul tipo degli elementi stessi.

Le operazioni ammesse su un'insieme sono:

- operazioni di base: come inserimento, cancellazione e verifica di contenimento;
- operazione di ordinamento: massimo, minimo;
- operazioni insiemistiche: unione, intersezione, differenza;
- iteratori: effettuare operazione per ogni elemento contenuto nell'insieme.

Struttura dati SET

Una struttura dati *dinamica*, *non lineare* che memorizza una *collezione non ordinata di elementi* senza valori ripetuti.

Set

```
// INTERPRETARE
int size                  // cardinalità dell'insieme
bool contains               // true se x è contenuto
```

```
// OPERAZIONI DI BASE
// inserisce x nell'insieme, se assente
insert(ITEM k)
// rimuove x nell'insieme, se presente
remove(ITEM k)

// OPERAZIONI INSIEMISTICHE
static SET union(SET A, SET B)
static SET intersection(SET A, SET B)
static SET difference(SET A, SET B)
```

Codice 4.4: Implementazione degli insiemi in Java

```
List<String> lista = new LinkedList<String>();
Set<String> docenti = new TreeSet<>();
docenti.add("Alberto");
docenti.add("Cristian");
docenti.add("Alessio");

Result: { "Alberto", "Alessio", "Cristian" }
```

Codice 4.5: Implementazione degli insiemi in C++

```
std::set<std::string> frutta;
frutta.insert("mele");
frutta.insert("pere");
frutta.insert("banane");
frutta.insert("mele");
frutta.remove("mele")

Result: { "banane", "pere" }
```

Codice 4.6: Implementazione degli insiemi in Python

```
items = { "rock", "paper", "scissors", "rock" }
print(items)
print("Spock" in items)
print("lizard" not in items)

Result: { "rock", "paper", "scissors" }
False
True
```

4.4 Dizionari

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* $R : D \to C$, o associazione chiave-valore, dove:

- l'insieme D è il dominio (gli elementi sono detti *chiavi*);
- l'insieme C è il codominio (gli elementi sono detti *valori*).

Le operazioni ammesse sui dizionari sono:

- ottenere il valore associato ad una particolare chiave (se presente) o **nil** se assente;
- inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti per la stessa chiave;
- rimuovere un'associazione chiave-valore esistente.

Specifica dizionario

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* o associazione chiave-valore.

DICTIONARY

ITEM lookup(ITEM k)	// restituisce il valore associato alla chiave k, **nil** altrimenti
ITEM insert(KEY k, ITEM v)	// associa il valore v alla chiave k
remove(KEY k)	// rimuove l'associazione della chiave k

Codice 4.7: Implementazione dei dizionari in Java

```
Map<String, String> capoluoghi = new HashMap<>();
capoluoghi.put("Toscana", "Firenze");
capoluoghi.put("Lombardia", "Milano");
capoluoghi.put("Sardegna", "Cagliari");
```

Codice 4.8: Implementazione dei dizionari in C++

```
std::map<std::string, int> wordcounts;
std::string s;

while (std::cin >> s && s != "end")
  ++wordcounts[s];
```

Codice 4.9: Implementazione dei dizionari in Python

```
v = {}
v[10] = 5
v["alberto"] = 42
v[10]+v["alberto"]

Result: 47
```

4.5 Alberi

Un albero ordinato è dato da un insieme finito di elementi detti nodi. Uno di questi nodi è designato come radice. I rimanenti nodi, se esistono sono partizionati in insiemi *ordinati* e *disgiunti*, anch'essi alberi ordinati.

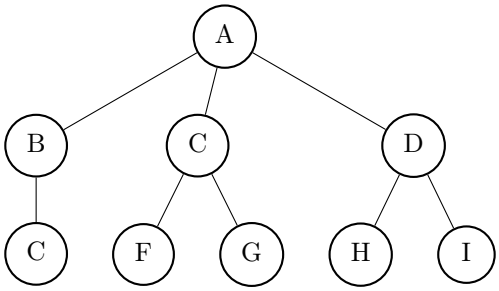


Figura 4.1: Un albero

Non vedremo implementazioni nei vari linguaggi in quanto non esiste una struttura dati definita riconosciuta universalmente.

4.6 Grafi

La struttura dati grafo è composta da:

- un insieme di elementi detti nodi o vertici;
- un insieme di coppie (ordinate oppure no) di nodi detti archi.

Tutte le operazioni su alberi e grafi ruotano attorno alla possibilità di effettuare visite su di essi, vedremo la specifica completa più avanti.

Nota. La scelta della struttura dati si riflette sull'efficienza e sulle operazioni ammesse.

4.7 Implementazione strutture dati elementari

4.7.1 Lista

Una lista è una sequenza di nodi, contenenti dati arbitrari e 1-2 puntatori all'elemento successivo e/o precedente.

La contiguità nella lista non implica che ci sia continuità nella memoria. Tutte le operazioni effettuate sulla lista hanno complessità $\mathcal{O}(1)$, ma per fare una ricerca dobbiamo spendere $\mathcal{O}(n)$.

Esistono diverse implmentazioni della lista, le quali possono essere:

- bidirezionale o monodirezionale;
- con sentinella o senza;
- circolare o non circolare.

Struttura dati lista bidirezionale con sentinella in pseudocodice

```
LIST                                  // bidirezionale con sentinella
LIST pred                             // predecessore
LIST succ                             // successore
LIST value                            // elemento

LIST List
    // la sentinella fa riferimento a sé stessa
    t.pred = t
    t.succ = t
    return t

POS head
    return succ

POS tail
    return pred

POS next
    return p.succ

POS prev
    return p.pred

bool finished(POS p)
    return p = this

ITEM read(POS p)
    return p.value

write(POS p)
    return p.value

// posso fare queste operazioni essendo sicuro
// di avere sempre un predecessore
POS insert(POS p, ITEM v)
    LIST t = List t.value = v
    t.pred = p.pred
    p.pred.succ = t
    t.succ = p
    p.pred = t
    return p

POS remove(POS p)
    p.pred.succ = p.succ
    p.succ.pred = p.pred
    LIST t = p.succ
    delete p
    return t
```

Il costo delle operazioni di lettura, scrittura, inserimento e rimozione per questa struttura è $\mathcal{O}(1)$.

4.7.2 Pila

La pila è una struttura dati *dinamica*, *lineare* in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, ed è quello che "è rimasto per meno tempo nell'insieme" (con strategia LIFO, *Last-In-First-Out*).

Specifica STACK

```
bool isEmpty                          // restituisce vero se la pila è vuota
push(ITEM v)                          // inserisce v in cima alla pila
ITEM pop                              // estrae l'elemento in cima alla pila e lo restituisce al chiamante
ITEM top                              // legge l'elemento in cima alla pila
```

Codice 4.10: Lista bidirezionale *senza* sentinella in Java

```
class Pos {
    Pos succ;    /** Prossimo elemento della lista */
    Pos pred;    /** Precedente elemento della lista */
    Object v;    /** Valore */

    Pos(Object v) {
        succ = pred = null;
        this.v = v;
    }
}

public class List {
    private Pos head;        /** Primo elemento della lista */
    private Pos tail;        /** Ultimo elemento della lista */

    public List() {
        head = tail = null;
    }

    public Pos head()                  { return head; }
    public Pos tail()                  { return tail; }
    public boolean finished(Pos pos)   { return pos == null; }
    public boolean isEmpty()           { return head == null; }
    public Object read(Pos p)          { return p.v; }
    public void write(Pos p, Object v) { p.v = v; }

    public Pos next(Pos pos) {
        return (pos != null ? pos.succ : null);
    }

    public Pos prev(Pos pos) {
        return (pos != null ? pos.pred : null);
    }

    public void remove(Pos pos) {
        if (pos.pred == null) // sto inserendo in testa
            head = pos.succ;
        else
            pos.pred.succ = pos.succ;

        if (pos.succ == null) // sto inserendo in coda
            tail = pos.pred;
        else
            pos.succ.pred = pos.pred;
    }

    public Pos insert(Pos pos, Object v) {
        Pos t = new Pos(v);

        if (head == null) {
            head = tail = t; // Inserisci in una lista vuota
        } else if (pos == null) {
            t.pred = tail; // Inserisci alla fine
            tail.succ = t;
            tail = t;
        } else {
            t.pred = pos.pred; // Iserimento davanti ad una posizione esistente
            if (t.pred != null)
                t.pred.succ = t;
            else
                head = t;

            t.succ = pos;
            pos.pred = t;
    }

        return t;
    }
}
```

Figura 4.2: `xkcd no. 379`

Ogni volta che viene effettuata una chiamata a funzione si usa implicitamente una pila, che memorizza tutti i record di attivazione delle chiamate effettuate. Sfrutteremo questo meccanismo implicito per visitare gli alberi, attraverso una visita in profondità.

Le pile possono essere implementate come:

- liste bidirezionali, dove il puntatore punta all'elemento top (non utilizzate);
- tramite vettore, dove la dimesione è limitata quindi si crea un *overhead* più basso.

Struttura dati pila basata su vettore in pseudocodice

```
ITEM[] A                          // elementi
int n                             // cursore
int m                             // dimesione massima

// crea una pila vuota
STACK Stack(int dim)
    STACK t = new STACK
    t.A = new int[0...dim-1]
    t.m = dim
    t.n = 0
    return t

// leggi l'elemento in cima alla pila
ITEM top
    precondition: n > 0
    return A[n]

// restituisce true se la pila è vuota
bool isEmpty
    return n == 0

// estrae l'elemento in cima alla pila e lo
   restituisce al chiamante
ITEM pop
    precondition: n > 0
    ITEM t = A[n]
    n++
    return t

// inserisce v in cima alla pila
push(ITEM v)
    precondition: n < m
    n++
    A[n] = v
```

Codice 4.11: Pila basata su vettore circolare in Java

```
public class VectorStack implements Stack {

    /** Vector containing the elements */
    private Object[] A;

    /** Number of elements in the stack */
    private int n;

    public VectorStack(int dim) {
        n = 0;
        A = new Object[dim];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public Object top() {
        if (n == 0)
            throw new IllegalStateException("Stack is empty");

        return A[n-1];
    }

    public Object pop() {
        if (n == 0)
            throw new IllegalStateException("Stack is empty");

        return A[--n];
    }

    public void push(Object o) {
        if (n == A.length)
            throw new IllegalStateException("Stack is full");

        A[n++] = o;
    }

}
```

4.7.3 Coda

La coda è una struttura dati *dinamica lineare* in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, ed è quello che "è rimasto per più tempo nell'insieme" (con strategia, FIFO, *First-In-First-Out*).

Specifica QUEUE

```
bool isEmpty                                   // restituisce vero se la coda è vuota
ITEM enqueue(ITEM v)                           // inserisce v in fondo alla coda
ITEM dequeue   // estrae l'elemento in cima alla coda e lo restituisce al chiamante
ITEM top                                       // legge l'elemento in testa alla coda
```

Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda. La politica FIFO è onesta (*fair*) rispetto l'ordine in cui i processi sono stati inseriti.

Le code possono essere implementate come:

- liste monodirezionali, dove sono presenti due puntatori: uno alla testa (*head*) per l'estrazione, ed uno alla coda per l'inserimento;
- vettori circolari, il quale ha una dimensione limitata e crea un *overhead* più basso.

Struttura dati coda basata su vettore circolare in pseudocodice

```
ITEM[] A                          // elementi
int n                             // dimensione attuale
int testa                         // testa
int m                             // dimesione massima

// crea una cosa vuota
QUEUE Queue(int dim)
    QUEUE t = new QUEUE
    t.A = new int[0...dim - 1]
    t.m = dim
    t.testa = 0
    t.n = 0
    return t

// legge l'elemento in testa alla coda
ITEM top
    precondition: n > 0
    return A[testa]

// restituisce true de la coda è vuota
ITEM isEmpty
    return n == 0

// estrae l'elemento in testa alla coda e lo
   restituisce al chiamante
ITEM dequeue
    precondition: n > 0
    ITEM t = A[testa]
    testa = (testa + 1) mod m
    n++
    return t

// inserisce v in fondo alla coda
ITEM enqueue
    precondition: n < m
    A[(testa + n) mod m] = v
    n++
```

Codice 4.12: Coda basata su vettore in Java

```
public class VectorQueue implements Queue {

    /** Element vector */
    private Object[] A;

    /** Current number of elements in the queue */
    private int n;

    /** Top element of the queue */
    private int head;

    public VectorQueue(int dim) {
        n = 0;
        head = 0;
        A = new Object[dim];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public Object top() {
        if (n == 0)
            throw new IllegalStateException("Queue is empty");

        return A[head];
    }

    public Object dequeue() {
        if (n == 0)
            throw new IllegalStateException("Queue is empty");

        Object t = A[head];
        head = (head+1) % A.length;
        n = n-1;
        return t;
    }

    public void enqueue(Object v) {
        if (n == A.length)
            throw new IllegalStateException("Queue is full");

        A[(head+n) % A.length] = v;
        n = n+1;
    }
}
```

Capitolo 5

Alberi

5.1 Definizioni

Definizione 5.1.1 (Albero radicato, *rooted tree*)**.** Un albero consiste di un insieme di nodi e un insieme di archi orientati che connettono coppie di nodi, con le seguenti proprietà:

- un nodo dell'albero è designato come nodo radice;
- ogni nodo n, a parte la radice, ha esattamente un arco entrante;
- esiste un cammino unico dalla radice ad ogni nodo;
- l'albero è connesso.

Definizione 5.1.2 (Albero radicato, definizione ricorsiva)**.** Un albero è dato da:

- un insieme vuoto, oppure
- una radice e zero o più sottoalberi, ognuno dei quali è albero; la radice è connessa alla radice di ogni sottoalbero con un arco orientato.

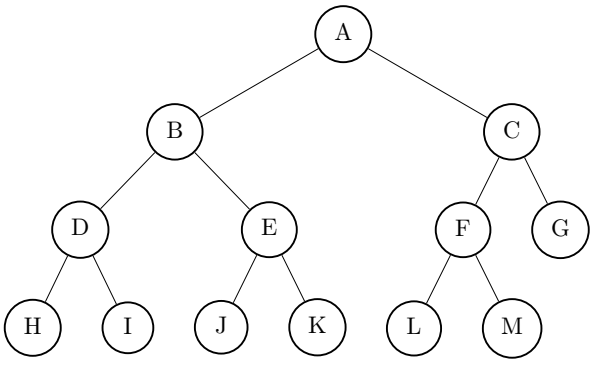
Definizione 5.1.3 (Profondità – *depth*)**.** La lunghezza del cammino semplice dalla radice al nodo (misurato in archi).

Definizione 5.1.4 (Livello – *level*)**.** L'insieme dei nodi alla stessa profondità.

Definizione 5.1.5 (Altezza dell'albero – *height*)**.** La profondità massima delle sue foglie.

5.2 Terminologia

- A è la radice (*root*);
- B, C sono radici dei sottoalberi (*roots of their subtrees*);
- D, E sono fratelli (*siblings*);
- D, E sono figli (*children*) di B;
- B è il padre (*parent*) di D, E;
- H, I, J, K, L, M, G sono foglie (*leafs*);
- gli altri nodi sono nodi interni (*internal nodes*);
- E è lo zio (il fratello del padre) di I;
- B è il nonno di I, I è il nipote di B.



5.3 Alberi binari

Definizione 5.3.1 (Albero binario)**.** Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, che vengono identificati come figlio sinistro e figlio destro.

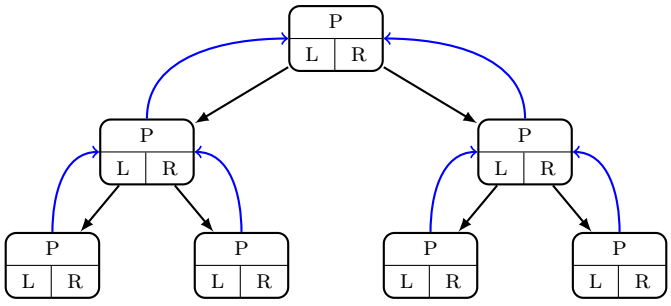
Nota. Due alberi T e U che hanno gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, sono distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T come figlio destro del medesimo nodo in U. In altre parole, anche se due alberi hanno lo stesso numero di nodi ed ognuno di questi nodi ha lo stesso numero di figli non è che detto che l'albero risultante sia identico.

Specifica albero binario

```
// GESTIONE ALBERO
Tree(ITEM v)  // costruisce un nuovo nodo, contenente v, senza figli o genitori
ITEM read  // legge il valore memorizzato nel nodo
write(ITEM v)  // modifica il valore memorizzato nel nodo
TREE parent  // restituisce il padre, oppure nil se questo nodo è radice

// GESTIONE STRUTTURA

// restituiscono il figlio sinistro (destro) di questo nodo,
// restituisce nil se assente
TREE left
TREE right

// inserisce il sottoalbero radicato in t
// come figlio sinistro (destro) di questo nodo
insertLeft(TREE t)
insertRight(TREE t)

// distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo
deleteLeft
deleteRight
```

Nota. Le funzioni *senza parametri* sono indicate con un carattere senza grazie e privi di parentesi tonde vuote al fine di alleggerire la lettura del codice.

5.3.1 Memorizzazione di un albero binario



Vengono memorizzati i seguenti campi:

- *parent*: riferimento al nodo padre;
- *left*: riferimento al figlio sinistro;
- *right*: riferimento al figlio destro.

Uno qualunque di questi oggetti potrebbe essere pari a **nil**, stando ad indicare che non esiste nessun sottoalbero.

5.3.2 Implementazione

Algoritmo 5.3.1: Implementazione albero binario in pseudocodice

```
// crea un nuovo albero
// restituisce la radice dell'albero creato
TREE Tree(ITEM v)
    TREE t = new TREE
    t.parent ← nil
    t.left ← t.right ← nil
    t.value ← v
    return t
insertLeft(TREE t)
    if left ≠ nil then
        t.parent ← this
        left ← t
insertRight(TREE t)
    if right ≠ nil then
        t.parent ← this
        right ← t
```

```
// elimina ricorsivamente il sottoalbero sinistro
deleteLeft
    if left ≠ nil then
        left.deleteLeft
        left.deleteRight
        left ← nil
// elimina ricorsivamente il sottoalbero destro
deleteRight
    if right ≠ nil then
        right.deleteLeft
        right.deleteRight
        right ← nil
```

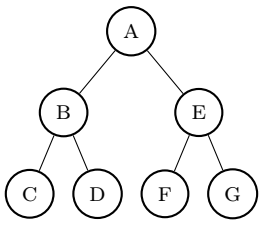
5.3.3 Visite

La visita di un albero (o la ricerca) è una strategia per passare attraverso (visitare) tutti i nodi di un albero. Si possono distinguere due tipi di visite:

1. visita in profondità: chiamata anche *Depth-First Search* (DFS), per visitare un albero visita ricorsivamente ognuno dei suoi sottoalberi; esistono tre varianti in base a quando il nodo viene visitato (pre, in o post-ordine); questa particolare visita sfrutta implicitamente il meccanismo di una pila (*stack*) tramite le chiamate ricorsive effettuate;
2. visita in ampiezza: chiamata anche *Breadth First Search* (BFS), per visitare un albero visita ogni livello, uno dopo l'altro partendo dalla radice; richiede esplicitamente l'utilizzo di una coda (*queue*).

Algoritmo 5.3.2: Schema per visita in profondità

```
dfs-schema(TREE t)
    if t ≠ nil then
        // pre-order visit
        stampa t
        dfs(t.left)
        // in-order visit
        stampa t
        dfs(t.right)
        // post-order visit
        stampa t
```



pre-visita	A B C D E F G
in-visita	C B D A F E G
post-visita	C D B F G E A

A seconda di dove scrivo il codice in questo schema ottengo una visita diversa.

5.3.4 Applicazioni

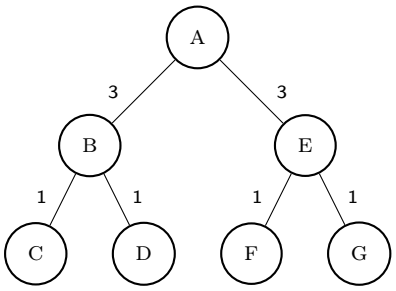
In genere post-visita e in-visita sono quelle più applicate, la pre-visita meno.

Visita in post-ordine

Una possibile applicazione della visita post-ordine è quella di effettuare un conteggio dei nodi presenti nell'albero.

Algoritmo 5.3.3: Conteggio dei nodi in un albero

```
count(TREE t)
    if t == nil then
        // è un albero vuoto
        return 0
    else
        // conto ricorsivamente i nodi
        C_ℓ = count(t.left)
        C_r = count(t.right)
        return C_ℓ + C_r + 1
```

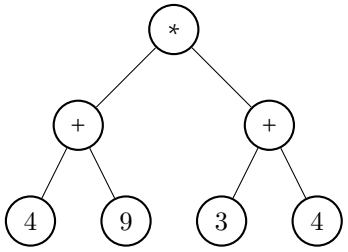


Visita in ordine (in-visita)

Una possibile applicazione della visita post-ordine è quella di stampare espressioni con operatori binari.

Algoritmo 5.3.4: Stampa espressioni con operatori binari

```
int stampaEspressioni(TREE t)
    if t.left == nil and t.right == nil then
        // siamo in una foglia
        stampa t.read
    else
        // sono su un nodo interno
        stampa "("
        stampaEspressioni(t.left)
        stampa t.read
        stampaEspressioni(t.right)
        stampa ")"
```



Stampa: `( 4 + 9 ) * ( 3 + 4 )`

Complessità di una visita

Il costo di una visita di un albero contenente n nodi è $\Theta(n)$, in quanto ogni nodo viene visitato al massimo una volta.

5.4 Alberi generici

Algoritmo 5.4.1: Specifica albero generico

```
// GESTIONE ALBERO
Tree(ITEM v)  // costruisce un nuovo nodo, contenente v, senza figli o genitori
ITEM read  // legge il valore memorizzato nel nodo
write(ITEM v)  // modifica il valore memorizzato nel nodo
TREE parent  // restituisce il padre, oppure nil se questo nodo è radice

// GESTIONE STRUTTURA
// restituiscono il primo figlio,
// oppure nil se questo nodo è una foglia
TREE leftmostChild

// restituisce il prossimo fratello,
// oppure nil se assente
TREE rightSibling

// inserisce il sottoalbero t
// come primo figlio di questo nodo
insertChild(TREE t)

// inserisce il sottoalbero t
// come prossimo fratello di questo nodo
insertSibling(TREE t)

// distuggi l'albero radicato
// identificato dal primo fratello
deleteChild

// distuggi l'albero radicato
// identificato dal primo figlio
deleteSibling
```

5.4.1 Visita in profondità

Un albero binario è anche un albero generale e lo visitiamo esattamente come lo visitavamo prima.

Algoritmo 5.4.2: Schema per visita in profondità

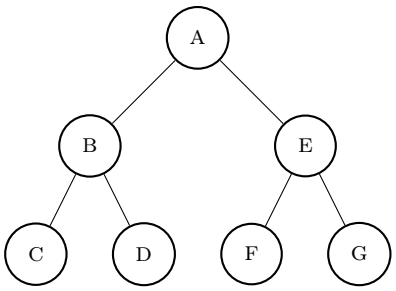
```
dfs(TREE t)
    if t ≠ nil then
        // pre-order visit
        stampa t
        dfs(t.left())
        // effettuo visita
        TREE u ← t.leftmostChild
        while u ≠ nil do
            dfs(u)
            u.rightSibling
        // post-order visit
        stampa t
```

5.4.2 Visita in ampiezza

Mentre nella visita in profondità il meccanismo della pila (*stack*) era implicito nelle chiamate ricorsive, in questo caso è necessario utilizzare *esplicitamente* una coda (*queue*). Un'altra differenza fra i due algoritmi è che quello in profondità è un algoritmo ricorsivo, l'altro è iterativo. Quando tutti i nodi di un livello vengono estratti dalla coda, la coda contiene solo ed unicamente i nodi del livello successivo.

Algoritmo 5.4.3: Schema per visita in ampiezza

```
bfs(TREE t)
    QUEUE Q ← Queue
    Q.enqueue(t)  // inserisci la radice
    while not Q.isEmpty do
        // fintanto che la coda non è vuota
        // estraggo un nodo dalla coda
        TREE u ← Q.dequeue
        // visita per livelli del nodo u
        stampa u
        // fintanto che ho almeno un figlio
        u ← u.leftmostChild
        while u ≠ nil do
            // metto in coda il figlio
            Q.enqueue(u)
            // passo al figlio destro
            u ← u.rightSibling
```



Sequenza: A B E C D F G

Commento Mettiamo in coda tutti i nodi che vogliamo visitare passo passo. Qui la stampa è in pre-visita ma qui – a differenza dei grafi – non ha molta importanza se la visita la facciamo prima o dopo. Visito tutti i figli prima di passare al livello successivo.

5.5 Memorizzazione

Esistono diversi modi per memorizzare un albero, più o meno indicati a seconda del numero massimo e medio di figli presenti. Le realizzazioni possibili sono:

1. con vettore dei figli;
2. primo figlio, prossimo fratello;
3. con vettore dei padri

5.5.1 Realizzazione con vettore dei figli

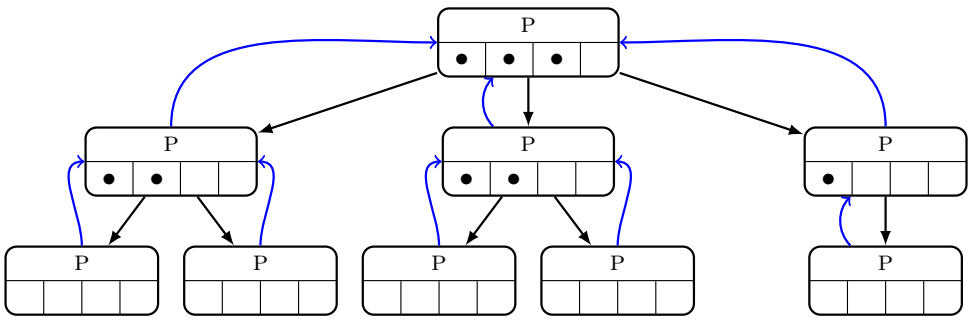


Figure 5.1: Realizzazione con vettore dei figli

Vengono memorizzati i seguenti campi:

- *parent* che è il riferimento al nodo padre;
- vettore dei figli il quale a seconda del numero dei figli può comportare una discreta quantità di spazio sprecato.

5.5.2 Realizzazione basata su primo figlio, prossimo fratello

Viene implementato come una lista di fratelli.

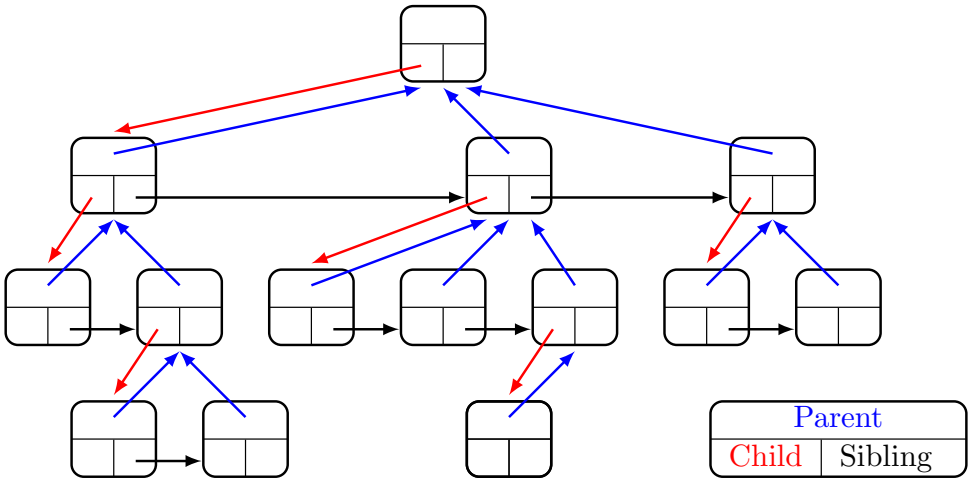


Figure 5.2: Realizzazione basata su primo figlio, prossimo fratello

La memorizzazione che viene utilizzata nel *file system* è esattamente questa.

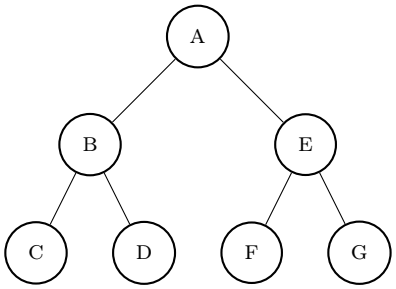
Algoritmo 5.5.1: Implementazione albero "primo figlio, prossimo fratello" in pseudocodice

```
TREE parent                                   // Riferimento al padre
TREE child                                    // Riferimento al primo figlio
TREE sibling                                  // Riferimento al prossimo fratello
ITEM value                                    // Valore memorizzato nel nodo

TREE Tree(ITEM v)
    TREE t = new TREE
    t.value ← v
    t.parent ← t.child ← t.sibling ← nil
    return t

insertChild(TREE t)
    t.parent ← self
    // inserisci t prima dell'attuale primo figlio
    t.sibling ← child
    child ← t

insertSibling(TREE t)
    t.parent ← parent
    // inserisci t prima dell'attuale prossimo
       fratello
    t.sibling ← sibling
    sibling ← t

deleteChild
    TREE newChild ← child.rightSibling
    delete(child)
    child ← newChild

deleteSibling
    TREE newBrother ← sibling.rightSibling
    delete(sibling)
    sibling ← newBrother

// metodo ausiliare
delete(TREE t)
    TREE u ← t.leftmostChild
    while u ≠ nil do
        TREE next ← u.rightSibling
        delete(u)
        u ← next
```

5.5.3 Realizzazione con vettore dei padri

Nella realizzazione con vettore dei padri, l'albero è rappresentato da un vettore i cui elementi contengono il valore associato al nodo e l'indice della posizione del padre del vettore.

1	A	0
2	B	1
3	E	1
4	C	2
5	D	2
6	F	3
7	G	3



Questa realizzazione può sembrare particolarmente assurda poiché dato un nodo non permette di stabilire direttamente quali sono i suoi figli, ma ci sono molti algoritmi che sono interessati solo ai padri. Questa è la rappresentazione più compatta che possiamo creare, vedremo la sua utilità quando andremo a studiare le visite sui grafi.

Capitolo 6

Alberi Binari di Ricerca

6.1 Introduzione

Facciamo un breve ripasso della struttura dati dizionario. La struttura dati dizionario è un insieme dinamico che implementa le seguenti funzionalità:

- ITEM lookup(ITEM v) permette di cercare per una certa chiave;
- insert(ITEM k, ITEM v) permette di associare una chiave ad un valore;
- remove(ITEM k) permette di rimuovere una certa associazione chiave-valore.

Tabella 6.1: Possibili implementazioni della struttura dati dizionario e relative complessità

Struttura dati	lookup	insert	remove
Vettore ordinato	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Vettore non ordinato	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$
Lista non ordinata	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$

* assumendo che l'elemento sia già stato trovato, $\mathcal{O}(n)$ altrimenti.

Ora vedremo la struttura dati dizionario implementata come un albero binario di ricerca.

L'idea che ha portato allo sviluppo degli alberi binari di ricerca è stata quella di portare la ricerca binaria (o dicotomica) negli alberi, avendo quindi un meccanismo dinamico per la memorizzazione delle informazioni ma basandosi sul meccanismo della ricerca binaria per recuperarle.

Le associazioni chiave-valore vengono memorizzate in un albero binario. Ogni nodo contiene una coppia ($u.key$, $u.value$). Le chiavi devono appartenere ad un insieme *totalmente ordinato*, ossia dev'essere possibile stabilire, date due chiavi, una relazione di precendenza fra di loro.

Proprietà. *Le seguenti proprietà permettono di realizzare un algoritmo di ricerca dicotomica:*

1. *Le chiavi contenute nei nodi del sottoalbero sinistro di u sono minori di $u.key$;*
2. *Le chiavi contenute nei nodi del sottoalbero destro di u sono maggiori di $u.key$.*

Nota. Queste proprietà valgono per ogni nodo e riguardano l'intero sottoalbero.

Vedremo un algoritmo per verificare se un albero binario è un albero binario di ricerca più avanti (verifyABR), il quale controllerà se queste proprietà sono soddisfatte.

Specifica Alberi Binari di Ricerca (ABR)

```
// CONTENUTO DI UN NODO
TREE parent
TREE left
TREE right
ITEM key
ITEM value

// GETTERS
TREE parent
TREE left
TREE right
ITEM key
ITEM value

// ORDINAMENTO
TREE successorNode(TREE t)
TREE predecessorNode(TREE t)
TREE min
TREE max

// FUNZIONI DIZIONARIO
ITEM lookup(ITEM k)
insert(ITEM k, ITEM v)
remove(ITEM k)

// FUNZIONI INTERNE
ITEM lookupNode
insertNode(TREE T, ITEM k, ITEM v)
removeNode(TREE T, ITEM k)
```

Ricerca di un nodo

Algoritmo 6.1.1: Ricerca di un nodo in un dizionario realizzato tramite ABR

```
int lookup(ITEM k)
    TREE t ← lookupNode(tree, k)
    if t ≠ nil then
        return t.value
    else
        return nil

// RICERCA DI UN NODO, iterativa
TREE lookupNode(TREE T, ITEM k)
    TREE u ← T  // parto dalla radice
    while u ≠ nil and u.key ≠ k do
        u ← iif(k < u.key, u.left, u.right)

// RICERCA DI UN NODO, ricorsiva
TREE lookupNode(TREE T, ITEM k)
    if T == nil or T.key == k then
        return T
    else
        return lookupNode(iif(k < u.key, u.left, u.right), k)
```

Ricerca del minimo e del massimo

Algoritmo 6.1.2: Ricerca del minimo e del massimo in un dizionario realizzato tramite ABR

```
// RICERCA DEL MINIMO
TREE min(TREE T)
    TREE u = T  // parto dalla radice
    while u.left ≠ nil do
        u ← u.left
    return u
```

```
// RICERCA DEL MASSIMO
TREE max(TREE T)
    TREE u = T  // parto dalla radice
    while u.right ≠ nil do
        u ← u.right
    return u
```

Ricerca del predecessore, successore

Algoritmo 6.1.3: Ricerca del predecessore e del successore di un nodo in un dizionario realizzato tramite ABR

```
// RICERCA DEL PREDECESSORE
TREE predecessorNode(TREE t)
    if t == nil then
        return t
    if t.left ≠ nil then
(1)     return max(t.left)
    else
(2)     TREE p ← t.parent
        while p ≠ nil and t == p.left do
            t ← p  // padre
            p ← p.parent  // nonno
    return p
```

```
// RICERCA DEL SUCCESSORE
TREE successorNode(TREE t)
    if t == nil then
        return t
    if t.right ≠ nil then
(3)     return min(t.right)
    else
(4)     TREE p ← t.parent
        while p ≠ nil and t == p.right do
            t ← p
            p ← p.parent
    return p
```

(1) u ha figlio sinistro: il predecessore è il massimo del sottoalbero sinistro di u;

(2) u non ha figlio sinistro: risalendo attraverso i padri, il predecessore è il primo avo v tale per cui u sta nel sottoalbero destro di v;

(3) u ha figlio destro: il successore è il minimo del sottoalbero destro di u;

(4) u non ha figlio destro: risalendo attraverso i padri, il sucessore è il primo avo v tale per cui u sta nel sottoalbero sinistro di v.

Nota. Posso trovare **nil** se passo alla funzione successorNode il nodo massimo o alla funzione predecessorNode il minimo (usciranno dal ciclo restituendo p che sarà pari a **nil**).

Inserimento di un nodo

La funzione insertNode inserisce un'associazione chiave-valore (k, v) nell'albero T. Se la chiave è già presente, sostituisce il valore associato; altrimenti, viene inserita una nuova associazione. Se l'albero è vuoto ($T == \textbf{nil}$) restituisce il primo nodo dell'albero, altrimenti restituisce la radice di T inalterata.

La funzione ausiliaria link si occupa di inserire il nodo collegandolo al corretto genitore.

Algoritmo 6.1.4: Inserimento di un nodo in un DICTIONARY realizzato tramite ABR

```
// IMPLEMENTAZIONE DIZIONARIO
insert(ITEM k, ITEM v)
    tree ← insertNode(tree, k, v)

// INSERIMENTO DI UN NODO
TREE insertNode(TREE T, ITEM k, ITEM v)
    TREE p ← nil  // padre
    TREE u ← T  // parto dalla radice

    // cerco posizione inserimento
    while u ≠ nil and u.key ≠ k do
        p ← u
        u ← iif(k < u.key, u.left, u.right)

    if u ≠ nil and u.key == k then
        // la chiave è già presente, aggiorno il valore
        u.value ← v
    else
        // la chiave non è presente
        // creo un nodo coppia chiave-valore
        TREE new ← Tree(k, v)

        // collego il nodo creato
        link(p, new, k)

        if p == nil then
            T ← new  // primo nodo ad essere inserito

    // restituisco l'albero non modificato o il nuovo nodo
    return T

// collega un nodo padre p ad un nodo figlio u
link(TREE p, TREE u, ITEM x)
    if u ≠ nil then
        // il nodo è stato cancellato
        u.parent ← p  // registro il padre

    if p ≠ nil then
        // collego il nodo sul figlio corretto
        u ← iif(x < p.key, p.left, p.right)
```

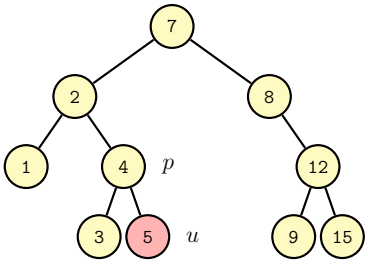
Rimozione di un nodo

Rimuove il nodo contenente la chiave k dall'albero T, restituisce la radice dell'albero (potenzialmente cambiata).

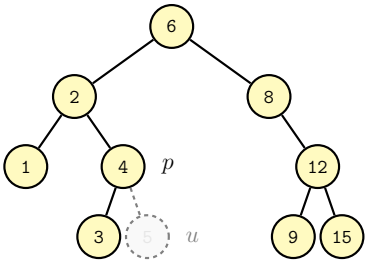
Algoritmo 6.1.5: Rimozione di un nodo in un DICTIONARY realizzato tramite ABR

```
// IMPLEMENTAZIONE DIZIONARIO
remove(ITEM k)
    TREE tree ← removeNode(tree, k)

// RIMOZIONE DI UN NODO
TREE removeNode(TREE T, ITEM k)
    // individuo il nodo da rimuovere
    TREE u ← lookupNode(T, k)

    // se il nodo da rimuovere è presente nell'albero...
    if u ≠ nil then
(1)     // ...e non ha figli
        if u.left == nil and u.right == nil then
            if u.parent ≠ nil then // se esiste il padre
                link(u.parent, nil, k) // rimuovo il puntatore al figlio

            // rimuovo direttamente il nodo
            delete u
(3)     // ...ed ha due figli
        if u.left ≠ nil and u.right ≠ nil then
            TREE s ← successorNode // individuo il successore

            link(s.parent, s.right, s.key) // collego il sottoalbero destro

            // copio il successore
            // nella posizione del nodo rimosso
            u.key ← s.key
            u.value ← s.value

            // rimuovo il successore
            delete s
(2)     // ...ed ha un solo figlio (sinistro)
        if u.left ≠ nil and u.right == nil then
            link(u.parent, u.left, k) // collega il figlio al padre

            if u.parent == nil then // se il padre non esiste
                T == u.right // il figlio diventa la radice
        // ...ed ha un solo figlio (sinistro)
        else
            link(u.parent, u.right, k) // collega il figlio al padre

            if u.parent == nil then // se il padre non esiste
                T == u.right // il figlio diventa la radice

    // restituisco la radice
    return T
```

(1) se il nodo da eliminare u non ha figli: lo si elimina semplicemente, in quanto togliere una foglia non altera le proprietà di ordinamento dell'albero;

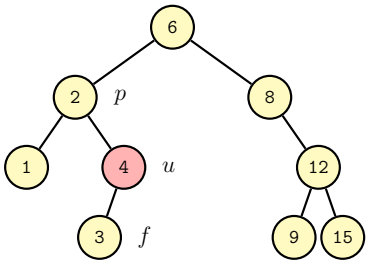


(a) Individuazione nodo foglia

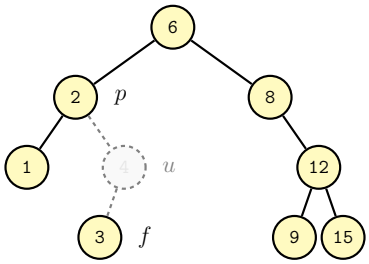


(b) Rimozione del nodo foglia

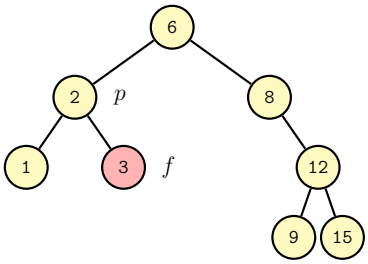
(2) se il nodo da eliminare ha un solo figlio f (destro o sinistro): si elimina u e si collega f all'ex-padre p di u in sostituzione di u (tramite la funzione link); le proprietà di ordinamento non vengono alterate in quanto tutti i nodi del sottoalbero destro di p sono maggiori di p stesso;



(a) Individuazione nodo u da eliminare

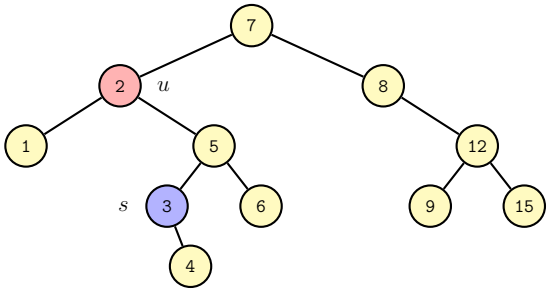


(b) Rimozione nodo u

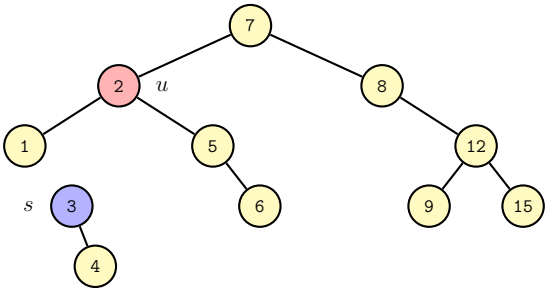


(c) Collegamento del sottoalbero f di u al padre p di u

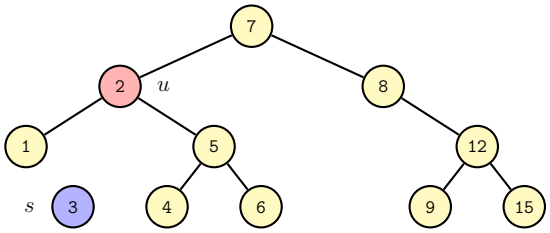
(3) se il nodo da eliminare u ha due figli: cerchiamo di ricadere nel caso (2); (a) individuiamo il successore (predecessore) s di u, il quale è il più piccolo valore maggiore di u (il più grande valore minore di u) e di conseguenza non ha figli sinistri (non ha figli destri); (b) si "stacca" il successore s; (c) si collega l'eventuale figlio destro di s al padre (tramite la funzione link) in quanto trovandosi nel sottoalbero sinistro del nonno vuol dire che sicuramente il suo valore non è maggiore del padre di s; (d) si copia s su u, si rimuove il nodo s, così facendo rispetto comunque l'ordine parziale.



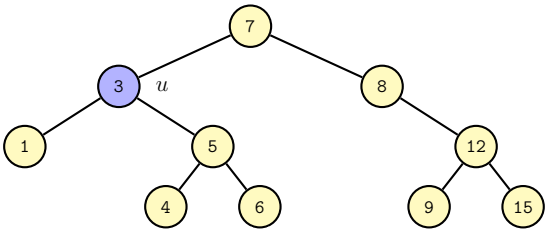
(a) Individuiamo il successore s di u



(b) Si "stacca" il successore s



(c) Si collega l'eventuale figlio destro di s al padre



(d) Si copia s su u, si rimuove il nodo s

Costo computazionale delle operazioni

Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia. Quindi se l'altezza dell'albero è definita come h, il tempo di ricerca ha complessità $\mathcal{O}(h)$.

Il caso pessimo è rappresentanto da un albero sbilanciato completamente a destra o completamente a sinistra. Questo caso può accadere quando si inseriscono ordinatamente i dati nell'albero. Questo caso, dove l'altezza $h = n$ porta ad una complessità $\mathcal{O}(n)$.

Mentre il caso ottimo è rappresentanto da un albero perfettamente bilanciato. Nell'esempio è mostrato un albero perfetto con $2^h - 1$ nodi, dove h è l'altezza. In questo caso la complessità è pari a $\mathcal{O}(\log n)$, ad esempio con $h = 2^3 - 1$ la complessità è $\mathcal{O}(\log h) = \mathcal{O}(\log 7) < 3$.

Ci domandiamo quindi quale sia l'altezza media di un albero binario di ricerca. Il caso "semplice" è quello di considerare che gli inserimenti avvengano in maniera statisticamente uniforme, è possibile dimostrare che l'altezza media è $\mathcal{O}(\log n)$, mentre il caso generale, ossia quello in cui avvengono sia inserimenti che cancellazioni è di difficile trattazione. Per evitare questa casistica si utilizzano varie tecniche per mantenere l'albero bilanciato. Per capire queste tecniche abbiamo prima bisogno di fissare un concetto.

Definizione 6.1.1 (Fattore di bilanciamento)**.** Il fattore di bilanciamento $\beta(v)$ di un nodo v è la massima differenza di altezza fra i sottoalberi di v.

Negli anni sono state usate diverse tecniche, ora in disuso:

- Alberi AVL (1962): $\beta(v) \leqslant 1$ per ogni nodo v, il bilanciamento dell'albero avveniva tramite rotazioni;
- B-Alberi (1972): $\beta(v) = 0$ per ogni nodo v, sono specializzati per strutture in memoria secondaria;
- Alberi 2-3 (1983): $\beta(v) = 0$ per ogni nodo v, in cui ogni nodo può avere o 2 o 3 figli, se ad un nodo viene aggiunto un ulteriore figlio, il ramo viene spezzato in due rami con 2 figli ciascuno, mentre se ad un ramo con 2 figli ne viene tolto uno allora l'unico figlio rimanente viene collegato al padre, questo potrebbe riportare il problema al primo caso; il bilanciamento viene ottenuto quindi tramite merge/split, il grado è variabile.

Nota (Meccanismo di rotazione)*.* Il meccanismo di rotazione ci permette di abbassare il fattore di sbilanciamento rispettando le proprietà di ordinamento parziale.

6.2 Alberi Binari di Ricerca bilanciati

Definizione 6.2.1 (Albero Red-Black)**.** Un albero red-black è un albero binario di ricerca in cui:

- ogni nodo è colorato di rosso o di nero;
- le chiavi vengono mantenute solo nei nodi interni dell'albero;
- le foglie sono costituite solo da nodi speciali **Nil**.

I nodi speciali **Nil** sono dei nodi sentinella il cui unico scopo è quello di evitare di trattare diversamente i puntatori ai nodi, dai puntatori **nil**; infatti al posto di un puntatore **nil** si usa un puntatore ad un nodo **Nil**; in memoria ne esiste solo uno per motivi di economia. I nodi con figli **Nil** sono le foglie nell'albero binario di ricerca corrispondente.

Un albero red-black deve rispettare i seguenti vincoli:

1. la radice è nera;
2. tutte le foglie sono nere;
3. entrambi i figli di un nodo rosso sono neri;
4. ogni cammino semplice da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u ha lo stesso numero di nodi neri.

Algoritmo 6.2.1: Specifica Red-Black Tree

```
// CONTENUTO DI UN NODO
Tree parent
Tree left
Tree right
int color  // red o black
Item key
Item value
// GETTERS
Tree parent
Tree left
Tree right
int color
Item key
Item value
```

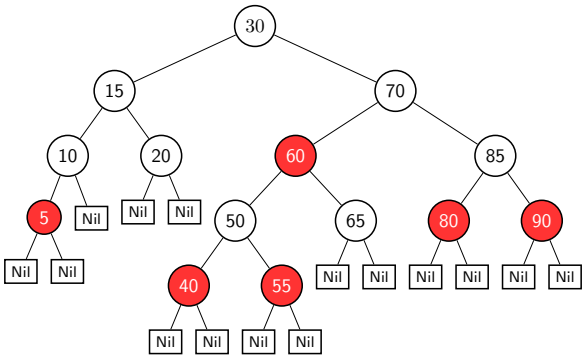
Proprietà (Altezza nera di un nodo v)**.** *L'altezza nera $b(v)$ di un nodo v è il numero di nodi neri lungo ogni percorso da v (escluso) ad ogni foglia (inclusa) del suo sottoalbero.*

Proprietà (Altezza nera di un albero Red-Black)**.** *L'altezza nera di un albero Red-Black è pari all'altezza nera della sua radice.*

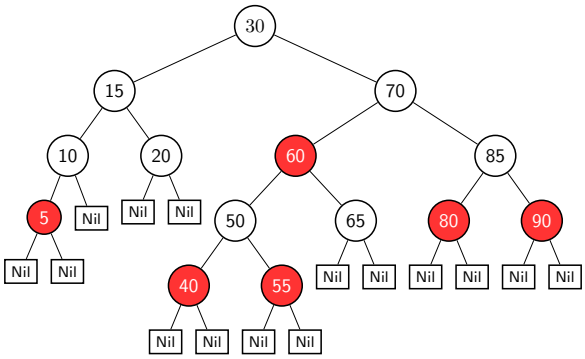
Entrambe le proprietà sono ben definite perché tutti i percorsi hanno lo stesso numero di nodi neri (per via della regola no. 4).

Esempi

Nelle suguenti figure i nodi neri sono segnati in bianco, mentre quelli rossi sono segnati.

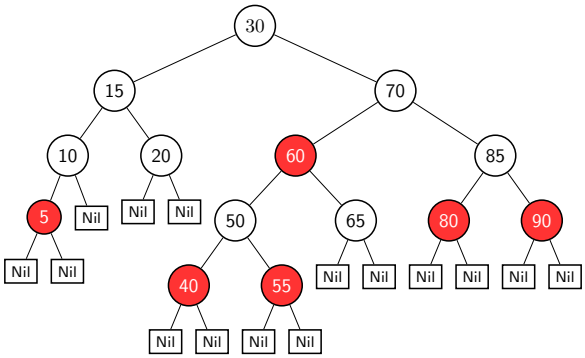


Es. 1 Entrambi i figli di un nodo rosso sono neri (3), ma un nodo nero può avere figli neri.

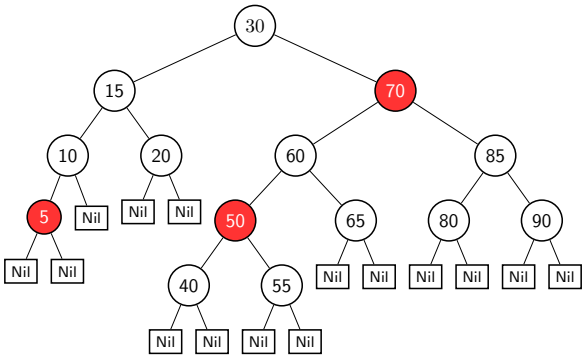


Es. 2 Ogni percorso da un nodo interno ad un nodo **Nil** ha lo stesso numero di nodi neri (4). L'altezza nera di quest'albero è 3

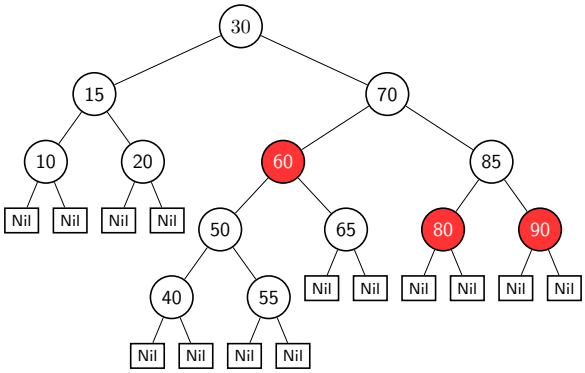
Lemma 10. *L'altezza totale di un albero è al più il doppio della sua altezza nera.*



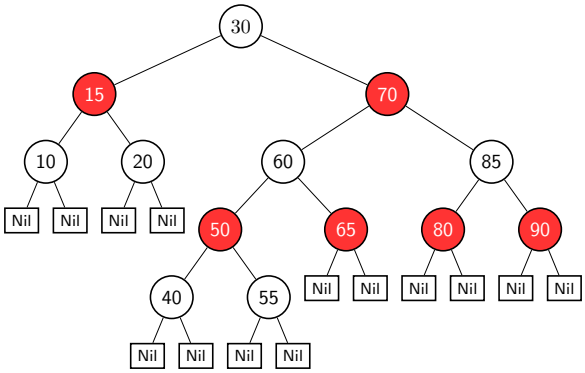
Es. 3 Più colorazioni sono possibili (versione 1). L'altezza di questo albero è 3



Es. 4 Più colorazioni sono possibili (versione 2). L'altezza di questo albero è 3



Es. 5 Cambiare colorazione può cambiare l'altezza nera. L'altezza di questo albero è 3



Es. 6 Cambiare colorazione può cambiare l'altezza nera. Stesso albero, l'altezza nera di questo albero è 2

6.2.1 Inserimento di un nodo

Durante la modifica di un albero Red-Black è possibile che le condizioni di bilanciamento risultino violate. Quando i vincoli Red-Black vengono violati si può agire in due modi:

- modificando i colori nella zona della violazione;
- operando dei bilanciamenti dell'albero tramite rotazioni (a destra o a sinistra)

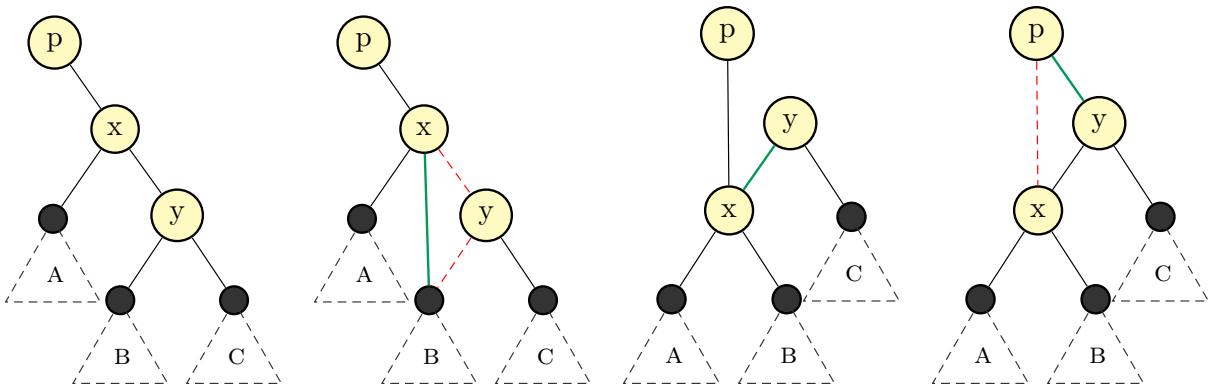
6.2.2 Bilanciamento dell'albero

Rotazione a sinistra

Algoritmo 6.2.2: Bilanciamento dell'albero tramite rotazione a sinistra

```
// effettua una rotazione verso sinistra
TREE rotateLeft(TREE x)
(1)  TREE y ← x.right
     TREE p ← x.parent
(2)  x.right ← y.left  // il sottoalbero B diventa figlio destro di x
     if y.left ≠ nil then
         y.left.parent ← x
(3)  y.left ← x  // x diventa figlio sinistro di y
     x.parent ← y
(4)  y.parent ← p  // y diventa figlio di p
     if p ≠ nil then
         if p.left == x then
             p.left ← y
         else
             p.right ← y
     return y
```

Nota. Il disegno differisce minimamente da quello che si trova sulle slide per motivi di comodità nel disegnarli con LaTeX



(1) Stato iniziale

(2) Il sottoalbero B diventa figlio dx di x

(3) x diventa figlio sx di y

(4) y diventa figlio di p, il vecchio padre di x

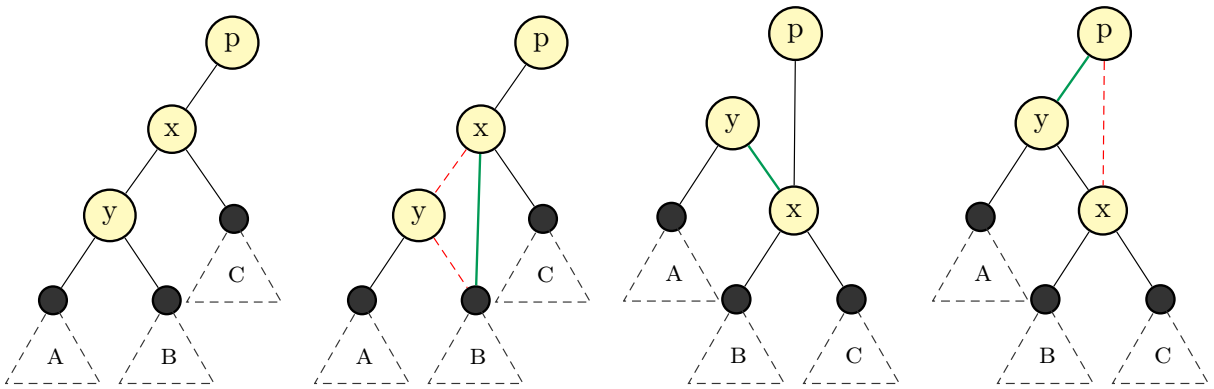
Figura 6.5: Esempio di rotazione a sinistra

Rotazione a destra

La rotazione a destra è simmetrica e viene spiegata ed illustrata per completezza.

Algoritmo 6.2.3: Bilanciamento dell'albero tramite rotazione a destra

```
// effettua una rotazione verso destra
TREE rotateRight(TREE x)
    // entrambi potrebbero essere nil
(1) TREE y ← x.left
    TREE p ← x.parent
(2) x.left ← y.right  // il sottoalbero B diventa figlio sinistro di x
    if y.right ≠ nil then
        y.right.parent ← x
(3) y.right ← x  // x diventa figlio destro di y
    x.parent ← y
(4) y.parent ← p  // y diventa figlio di p
    if p ≠ nil then
        if p.right == x then
            p.right ← y
        p.left ← y
    return y
```



(1) Stato iniziale

(2) Il sottoalbero B diventa figlio sx di x

(3) x diventa figlio dx di y

(4) y diventa figlio di p, il vecchio padre di x

Figura 6.6: Esempio di rotazione a destra

Inserimento di un nodo in un albero binario bilanciato

Per inserire un nodo in un albero Red-Black si usa la stessa procedura usata per gli alberi binari di ricerca e si colora il nuovo nodo di RED. Il vincolo che potremmo violare è il terzo, quello che prevede che entrambi i figli di un nodo rosso siano neri.

Algoritmo 6.2.4: Inserimento di un nodo in un RED-BLACK TREE

```
// Inserimento di un nodo in un albero Red-Black
TREE insertNode(TREE T, TREE k, ITEM x)
    TREE p ← nil  // riferimento al padre
    TREE u ← T  // riferimento alla radice

    // cerco posizione inserimento
    while u ≠ nil and u.key ≠ k do
        p ← u
        u ← iif(k < u.key, u.left, u.right)

    if u ≠ nil and u.key == k then
        // la chiave è già presente, aggiorno il valore
        u.value ← v
    else
        // la chiave non è presente
        // creo un nodo coppia chiave-valore
        TREE new ← Tree(k, v)

        // collego il nodo creato
        link(p, new, k)
        balanceInsert(new)

        if p == nil then
            T ← new  // primo nodo ad essere inserito

    // restituisco l'albero non modificato o il nuovo nodo
    return T
```

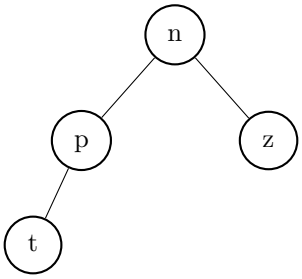
Nel caso in cui l'inserimento vìoli il terzo vincolo ci sposteremo verso l'alto lungo il percorso di inserimento; cercheremo di ripristinare il terzo vincolo; sposteremo le violazioni verso l'alto rispettando il quarto vincolo (mantenendo l'altezza nera dell'albero); al termine, coloreremo la radice di nero (onorando il primo vincolo).

Nota. Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi, altrimenti non sono necessarie.

Algoritmo 6.2.5: Bilanciamento dell'albero in seguito all'inserimento di un nodo rosso

```
balanceInsert(TREE t)
    t.color ← RED  // coloro il nodo da inserire di rosso
    // t == nil è la condizione di fino ciclo
    while t ≠ nil do
(0)     TREE p ← t.parent                                                    // riferimento al padre
        TREE n ← iif(p ≠ nil, p.parent, nil)                                 // riferimento al nonno
        TREE z ← iif(n == nil, nil, iif(n.left == p, n.right, n.left))       // riferimento allo zio
(1)     if p == nil then
            t.color ← BLACK
            t ← nil  // fine
(2)     else if p.color == BLACK then
            t ← nil  // fine
(3)     else if z.color == RED then
            p.color ← z.color ← BLACK
            n.color ← RED
            t ← n  // passo il problema al nonno
        else
(4a)        if (t == p.right) and (p == n.left) then
                rotateLeft(p)
                t ← p  // passo il problema al padre
(4b)        if (t == p.left) and (p == n.right) then
                rotateRight(p)
                t ← p  // passo il problema al padre
            else
(5a)            if (t == p.left) and (p == n.left) then
                    rotateRight(n)
(5b)            else if (t == p.right) and (p == n.right) then
                    rotateLeft(n)

                p.color ← BLACK
                n.color ← RED
                t ← nil  // fine
```

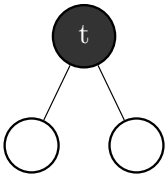
(0) dichiaro dei riferimento al padre, al nonno e allo zio



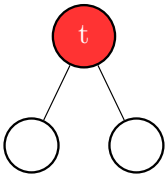
(1) il nuovo nodo t non ha padre; Questo può accadere in due casi:

- è il primo nodo ad essere inserito, oppure
- quando abbiamo spostato la violazione verso l'alto fino a raggiungere la radice

Ricoloriamo t di BLACK in quanto trovandosi sulla radice dell'albero non viola nessun vincolo.

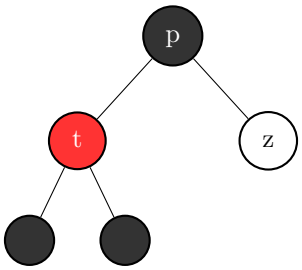


(a) Possibile violazione del primo vincolo

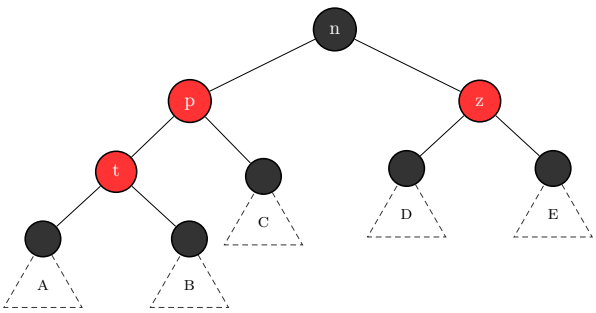


(b) Ricolorazione del nodo t

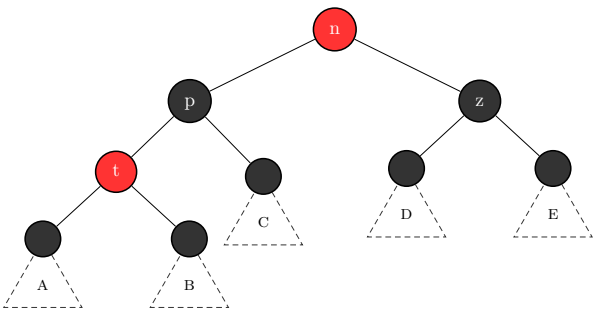
(2) il padre p di t è nero; anche in questo caso non abbiamo violato nessun vincolo perché avendo inserito un nodo rosso la lunghezza dei cammini neri non cambia e avendo inserito un nodo rosso figlio di un nodo nero non violiamo il terzo vincolo;



(3) il padre p e lo zio z sono rossi; Se z è rosso è possibile ricolorare di nero p e z, e di rosso n; poiché tutti i cammini che passano per z e p passano anche per n, l'altezza nera non è cambiata (non abbiamo violato il quarto vincolo); Abbiamo spostato così il problema verso l'alto, più precisamente sul nonno che potrebbe aver violato il primo o il terzo vincolo, ovvero che n può essere una radice rossa o che abbia un padre rosso. Per risolvere il problema poniamo $t \leftarrow n$ e continuiamo il ciclo.



(a) Possibile violazione del primo e del terzo vincolo

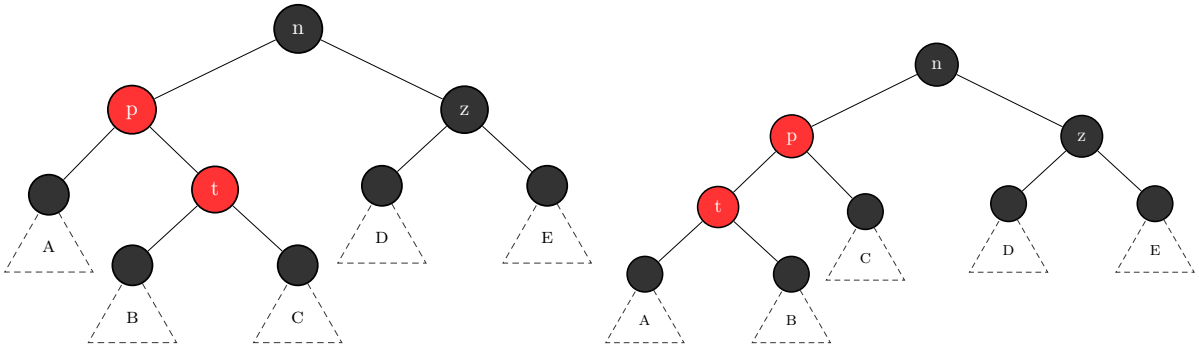


(b) Ricolorazione del padre e dello zio di nero

(4a) il padre p è rosso e lo zio z è nero; si assuma che t sia figlio *destro* di p e che p sia figlio *sinistro* di n; effettuando una rotazione a sinistra a partire dal nodo p scambiamo i ruoli di t e d p ottenendo il caso (5a), dove i nodi in conflitto sul terzo vincolo sono entrambi figli *sinistri* dei loro padri; i nodi

coinvolti nel cambiamento sono p e t, entrambi rossi, quindi l'altezza nera non cambia; abbiamo spostato il problema al padre, quindi poniamo $t \leftarrow p$ e continuiamo il ciclo.

`(4b)` speculare al caso `(4a)` (ossia che che t sia figlio *sinistro* di p e che p sia figlio *destro* di n)



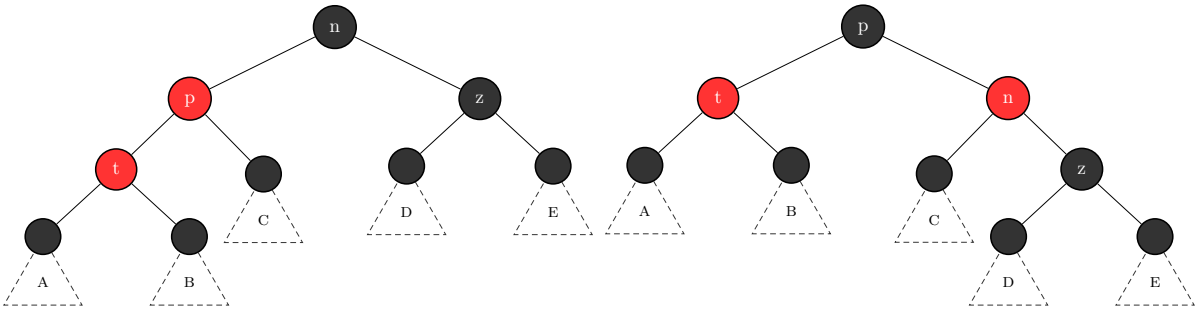
(a) Possibile violazione del primo e del terzo vincolo

(b) Ricolorazione del padre e dello zio di nero

Figura 6.9: Rappresentazione grafica del caso `(4a)`

`(5a)` anche in questo caso il padre p è rosso e lo zio z è nero; ma si assuma che t sia figlio *sinistro* di p e che p sia figlio *sinistro* di n; effettuando una rotazione a destra a partire dal nodo n ci porta ad una situazione in cui t e n sono figli di p; colorando p di nero ed n di rosso ci troviamo in una situazione in cui tutti i vincoli vengono rispettati (in particolare, l'altezza nera che passano per la radice è uguale a quella iniziale).

`(5b)` speculare al caso `(5a)` (ossia che che che t sia figlio *destro* di p e che p sia figlio *destro* di n)



(a) Possibile violazione del primo e del terzo vincolo

(b) Ricolorazione del padre e dello zio di nero

Figura 6.10: Rappresentazione grafica del caso `(5a)`

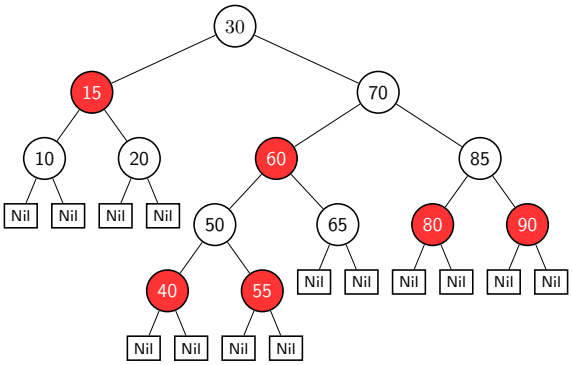
Complessità Ognuna di queste operazione avviene in tempo costante. Ogni volta il problema può salire di uno o due livelli, in quanto l'altezza dell'albero è limitata da $\log n$, il nostro algoritmo è limitato superiormente da $\log n$, ossia $\mathcal{O}(\log n)$.

Più precisamente:

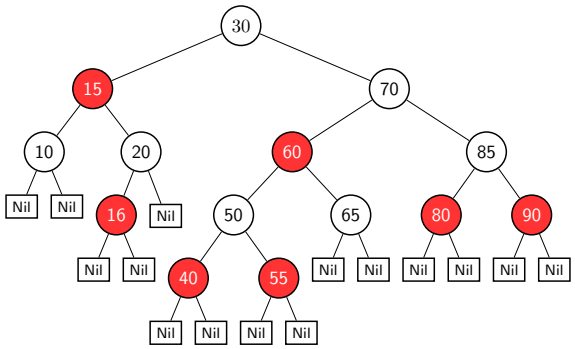
- $\mathcal{O}(\log n)$ per scendere fino al punto di inserimento;
- $\mathcal{O}(1)$ per effettuare l'inserimento;
- $\mathcal{O}(\log n)$ per risalire ed "aggiustare" (caso `3`)

Esempi di inserimento

Proviamo ad inserire il nodo 16 nell'albero red-black sottostante.

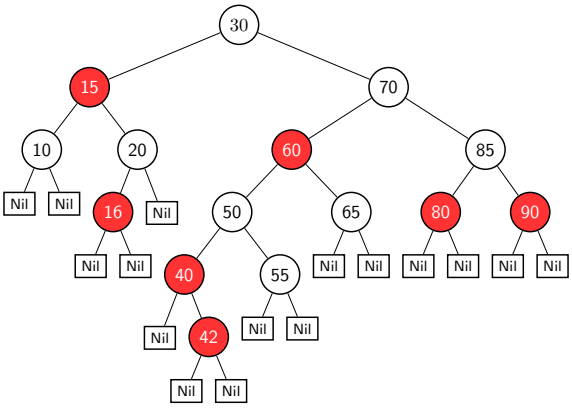


(a) Stato attuale

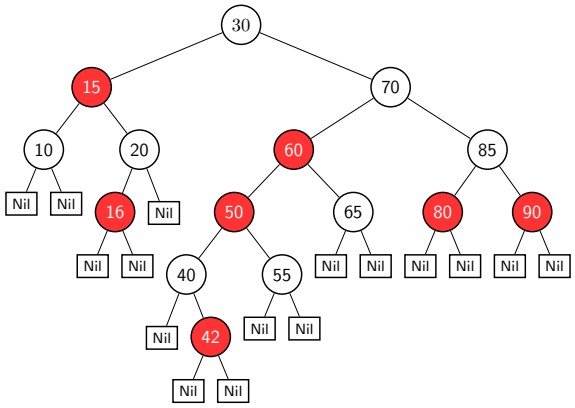


(b) Inserimento del nodo 16 andato a buon fine

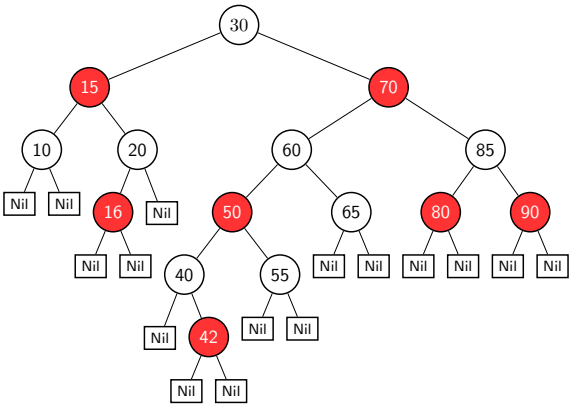
Non violiamo alcun vincolo in quanto il padre di 16 è nero e non abbiamo modificato l'altezza nera. Questo caso rappresenta il caso `2`, l'inserimento del nodo 16 è quindi andato a buon fine. Alternativamente proviamo ad inserire il nodo 42 sempre nello stesso albero.



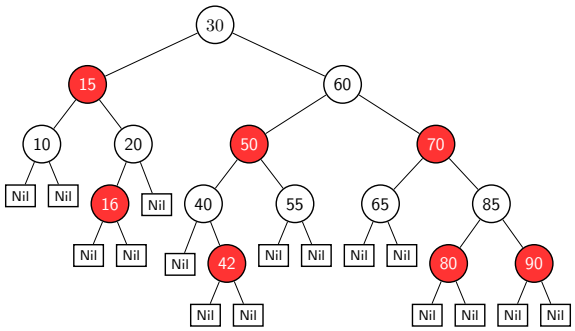
(a) Inserimento del nodo 42, violiamo il secondo vincolo, ci troviamo nel caso `3` (z rosso), quindi coloriamo di nero p e z e di rosso n, il problema si sposta ad n



(b) Violiamo il terzo vincolo, entrambi i nodi rossi sono figli *sinistri* quindi ci troviamo nel caso `5a`, quindi coloriamo di nero p, di rosso n ...



(c) ... ed effettuiamo una rotazione a destra con perno n



(d) Abbiamo ripristinato il terzo vincolo, gli altri vincoli non sono mai stati violati, quindi abbiamo finito

Questo esempio ci mostra chiaramente come gli alberi Red-Black, attraverso il rispetto dei vincoli, tendano a mantenere bilanciato l'albero. Esiste una versione *top-down* dell'algoritmo di inserimento che scende fino interessato "aggiustando" l'albero man mano.

6.2.3 Rimozione di un nodo

Se il nodo rimosso è rosso l'altezza nera rimane invariata, non sono stati creati nodi rossi consecutivi e la radice resta nera. Il problema sorge quando si rimuovono nodi neri in quanto è possibile che siano stati violati il primo ed il terzo vincolo e sicuramente è stato violato il quarto vincolo in quanto cambia l'altezza nera. L'algoritmo balanceDelete(T, t) ripristina la proprietà Red-Black con rotazioni e cambiamenti di colore. Ci sono quattro casi possibili (e 4 simmetrici).

Algoritmo 6.2.6: Rimozione di un nodo in un RED-BLACK TREE

```
balanceDelete(TREE T, TREE t)
    t.color ← RED  // coloro il nodo da inserire di rosso
    while (t ≠ T) and (t.color == BLACK) do
        TREE p ← t.parent                                   // riferimento al padre
        if t == p.left then
            TREE f ← p.right                                // riferimento al fratello
            TREE ns ← f.left                                // riferimento al nipote sinistro
            TREE nd ← f.right                               // riferimento al nipote destro
(1)         if f.color == RED then
                p.color ← RED
                f.color ← BLACK
                rotateLeft(p)
                // t viene lasciato inalterato, quindi si ricade nei casi 2, 3, 4
            else
(2)             if ns.color == nd.color == BLACK then
                    f.color == RED
                    t ← p  // passo il problema al padre
(3)             else if (ns.color == RED) and (nd.color == BLACK) then
                    ns.color ← BLACK
                    f.color ← RED
                    rotateRight(f)
                    // t viene lasciato inalterato, quindi si ricade nel caso 4
(4)             else if nd.color == RED then
                    f.color == p.color
                    p.color ← BLACK
                    nd.color ← BLACK
                    rotateLeft(p)
                    t ← T
        else
            // casi speculari
```

La cancellazione è concettualmente complicata, ma è efficiente.

(1) si passa ad uno dei casi 2, 3, 4;

(2) si torna ad uno degli altri casi, ma risalendo di un livello l'albero;

(3) si passa al caso 4;

(4) si termina.

È possibile visitare al massimo un numero $\mathcal{O}(\log n)$ di casi, ognuno dei quali è gestito in $\mathcal{O}(1)$.

Capitolo 8

Insiemi

Possono essere implementati con molte delle strutture dati viste fin'ora. Ognuna delle quali rappresenta valntaggi e svantaggi.

8.1 Realizzazione con vettori booleani

L'insieme viene rappresentato attraverso un vettore booleano di m elementi. Il quale è notevolmente semplice da implementare ed estremamente efficiente verificare se un elemento appartiene all'insieme. Sfortunatamente la memoria occupata è $\mathcal{O}(m)$, indipendentemente dalle dimensioni effettive, inoltre alcune operazioni dipendono dalla memoria utilizzata per memorizzare questi oggetti, piuttosto che dal numero di oggetti effettivamente memorizzati, il che porta ad una complessità di queste operazione di $\mathcal{O}(m)$.

Struttura dati SET implementata come Vettore Booleano

```
bool[ ] V
int size
int dim

SET Set(int m)
    SET t ← new SET
    t.size ← 0
    t.dim ← m
    t.V ← [false] * m
    return t

SET contains(int x)
    if 1 ⩽ x ⩽ dim then
        return V[x]
    else
        return false

int size
    return size

insert(int x)
    if 1 ⩽ x ⩽ dim then
        if not V[x] then
            size++
            V[x] ← true

remove(int x)
    if 1 ⩽ x ⩽ dim then
        if V[x] then
            size++
            V[x] ← false
```

```
SET union(SET A, SET B)
    // crea un insieme della capacità max
    SET C ← Set(max(A.dim, A.dim))

    // inserisci gli elementi di A
    from i ← 1 until A.dim do
        if A.contains(i) then
            C.insert(i)

    // inserisci gli elementi di B
    from i ← 1 until B.dim do
        if A.contains(i) then
            C.insert(i)

SET intersection(SET A, SET B)
    // crea un insieme della capacità min
    SET C ← Set(min(A.dim, A.dim))

    from i ← 1 until min(A.dim, A.dim) do
        // se è contenuto in entambi
        if A.contains(i) and B.contains(i) then
            C.insert(i)  // aggiungilo

SET difference(SET A, SET B)
    SET C ← Set(A.dim)

    from i ← 1 until A.dim do
        // se è contenuto A e non in B
        if A.contains(i) and not B.contains(i)
         then
            C.insert(i)  // aggiungilo
```

8.1.1 Implementazioni nei linguaggi di programmazione

Esistono alcune implementazioni nei linguaggi attualmente utilizzati. In Java esiste la struttura dati `BitSet` i cui meotodi sono illustrati nella tabella 8.1. Mentre in `C++ STL` esistono due implementazioni `std::bitset` e `vector<bool>`:

- `bitset` è una struttura dati con dimensione fissata nel template al momento della compilazione;
- `vector<bool>` è una specializzazione di `vector` per ottimizzare la memorizzazione, ha dimensione dinamica.

Tabella 8.1: Implementazione `java.util.BitSet`

Operazione	Metodo
contains	`boolean get(int i)`
size	`int cardinality()`
insert	`void set(int i)`
remove	`void clear(int i)`
union	`void and(BitSet set)`
intersection	`void or(BitSet set)`

8.2 Realizzazione con vettore non ordinato

Struttura dati SET implementata come vettore non ordinato

```
SET difference(SET A, SET B)
    SET C ← Set  // non ha bisogno della dimensione
    from s ∈ A do
        // se non è contenuto in B
        if not B.contains(s) then
            C.insert(s)  // aggiungilo
```

Costo delle operazioni Le operazioni di ricerca, inserimento e cancellazione costano $\mathcal{O}(n)$, le operazioni di inserimento (assumendo che non esista l'elemento) costano $\mathcal{O}(1)$, le operazioni di unione, intersezione e differenza $\mathcal{O}(nm)$.

8.3 Realizzazione vettore ordinato

Struttura dati SET implementata come vettore ordinato

```
SET intersection(LIST A, LIST B)
    LIST C ← Set  // non ha bisogno della dimensione
    // creo puntatori alle liste
    POS p ← A.head
    POS q ← B.head
    while not A.finished(p) and B.finished(q) do
        if A.read(p) == B.read(q) then // se gli elementi coincidono
            C.insert(C.tail, A.read(p))  // inseriscilo nell'intersezione
            // scorri i puntatori
            p ← A.next(p)
            q ← B.next(q)
        else if A.read(p) < B.read(q) then
            p ← A.next(p)  // scorro puntatore di A
        else
            q ← B.next(q)  // scorro puntatore di B
```

Costo delle operazioni Le operazioni di ricerca costano $\mathcal{O}(n)$ con le liste e $\mathcal{O}(\log n)$ con i vettori, le operazioni di inserimento e cancellazione costano $\mathcal{O}(n)$, le operazioni di unione, intersezione e differenza $\mathcal{O}(n)$.

8.4 Reality Check

In realtà si utilizzano strutture dati complesse che permettono di ottimizzare le performance. Se abbiamo bisogno dell'ordinamento si utilizzano alberi bilanciati, mentre se abbiamo bisogno di sapere smeplicemente se un elemento è contenuto o meno si utilizzano le tabelle hash.

Per le operazioni di ricerca, inserimento e cancellazione negli alberi bilanciati hanno una complessità di $\mathcal{O}(\log n)$, mentre nelle tabelle hash di $\mathcal{O}(1)$.

Le implementazioni più diffuse degli alberi bilanciati sono `TreeSet` in Java, `OrderedSet` in Python e `set` in `C++`, mentre le implementazioni più diffuse delle tabelle hash sono `HashSet` in Java, `set` in Python e `unordered_set` in `C++`.

Tabella 8.2: Implementazioni e relative complessità delle operazioni
$m \equiv$ dimensione del vettore o della tabella hash

	contains ricerca	insert	remove	min max	foreach (memoria)	Ordine
Vettore booleano	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$	Sì
Lista non ordinata	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	No
Lista ordinata	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Sì
Vettore ordinato	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Sì
Alberi bilanciati	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Sì
Hash (mem. interna)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(n)$	No
Hash (mem. esterna)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$	No

Capitolo 9

Grafi

9.1 Introduzione

Un grafo non è altro che un insieme di entità collegate da un insieme di relazioni che possono essere interpretate in vari modi.

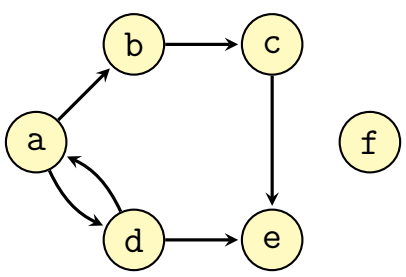
Il questa parte della lezione affronteremo i problemi che possono essere risolti tramite grafi non pesati, ossia:

- ricerca del cammino più breve, misurato in numero di archi (che differisce dal problema del cammino minimo definito su grafi pesati che cerca di stabilire quale sia il cammino meno costoso);
- componenti (fortemente) connesse;
- verifica ciclicità;
- ordinamento topologico.

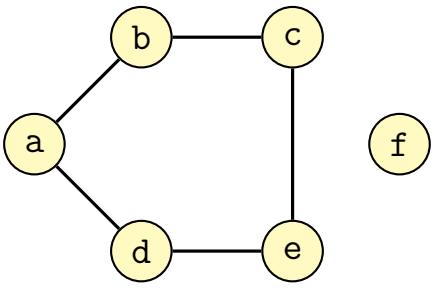
9.1.1 Definizioni

Definizione 9.1.1 (Grafo orientato, *directed*)**.** Un grafo orientato è una coppia $G = (V, E)$ dove V è un insieme di nodi (node) o vertici (vertex), mentre E è un insieme di coppie ordinate (u, v) di nodi detti archi (edges).

Definizione 9.1.2 (Grafo non orientato, *undirected*)**.** Un grafo non orientato è una coppia $G = (V, E)$ dove è un insieme di nodi (node) o vertici (vertex), mentre E è un insieme di coppie non ordinate (u, v) dette archi (edges).



(a) Grafo diretto

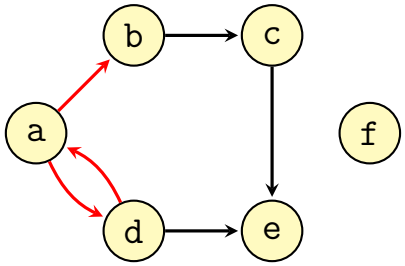


(b) Grafo indiretto

9.1.2 Terminologia

Proprietà (Adiacenza)**.** *Un vertice v è detto adiacente a u se esiste un arco (u, v).*

Proprietà (Incidenza)**.** *Un arco (u, v) è detto incidente da u a v.*



- (a, b) è incidente da a a b
- (a, d) è incidente da a a d
- (d, a) è incidente da d a a
- b è adiacente ad a
- d è adiacente ad a
- a è adiacente a d

Nota. In un grafo non orientato, la relazione di adiacenza è simmetrica.

9.1.3 Ragionamenti sulla complessità

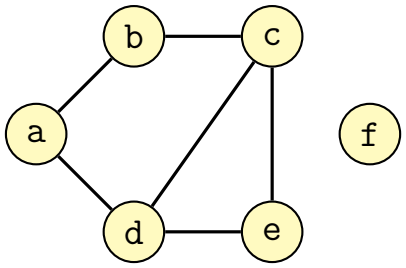
Definiamo il numero di nodi con $n = |V|$, ed il numero di archi con $m = |E|$. C'è una relazione precisa fra n ed m. In un grafo non orientato $m \leqslant \frac{n(n-1)}{2} = \mathcal{O}(n^2)$, mentre in un grafo orientato $m \leqslant n^2 - n = \mathcal{O}(n^2)$. Questi ordini di grandezza ci serviranno a valutare quale algoritmo utilizzare in base al numero di possibili archi. La complessità viene quindi espressa in termini sia di n che di m, ad esempio $\mathcal{O}(n + m)$.

9.1.4 Casi speciali

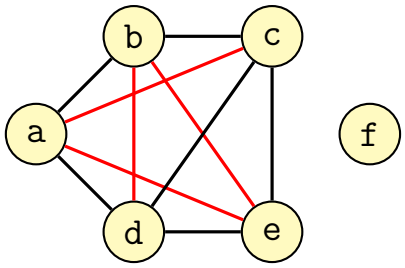
Completezza di un grafo

Un grafo con un arco fra tutte le coppie di nodi è detto *completo*. Informalmente (non c'è accordo sulla definizione) parleremo di:

- grafo *sparso* se ha "pochi archi"; ad esempio grafi con m pari a $\mathcal{O}(n)$ o $\mathcal{O}(n \log n)$ sono considerati tali;
- grafo *denso* se ha "tanti archi"; ad esempio grafi con m pari a $\Omega(n^2)$.



(a) Grafo sparso



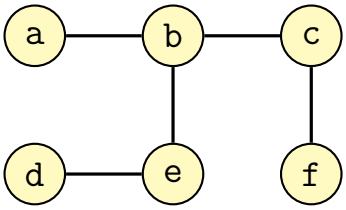
(b) Grafo denso

Alberi radicati

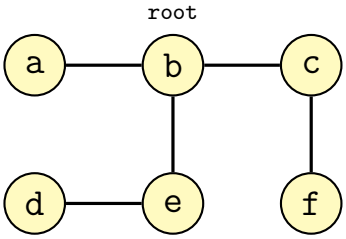
Definizione 9.1.3 (Albero non radicato, libero)**.** Un albero libero (free tree) è un grafo connesso con $m = n - 1$, dove non viene identificata una radice.

Definizione 9.1.4 (Albero radicato)**.** Un albero radicato (rooted tree) è un grafo connesso con $m = n - 1$ nel quale uno dei nodi è designato come radice.

Definizione 9.1.5 (Foresta)**.** Un insieme di alberi è un grafo detto foresta.



(a) Albero libero



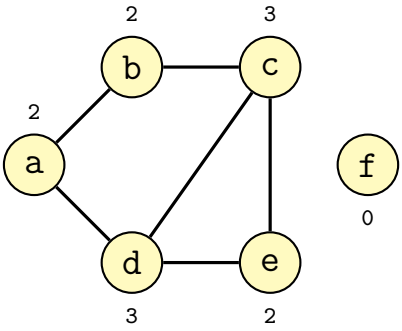
(b) Albero radicato

Definizioni

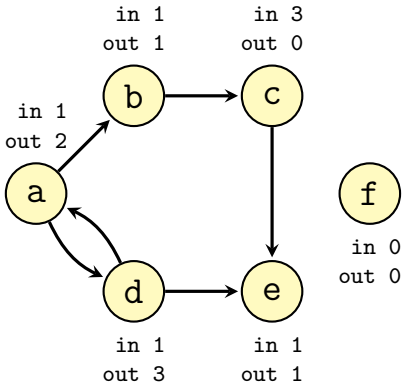
Definizione 9.1.6 (Grado, *degree*)**.** In un grafo non orientato il grado di un nodo è il numero di archi incidenti su di esso.

Definizione 9.1.7 (Grado entrante, *in-degree*)**.** In un grafo orientato il grado entrante di un nodo è il numero di archi entranti su di esso.

Definizione 9.1.8 (Grado uscente, *out-degree*)**.** In un grafo orientato il grado uscente di un nodo è il numero di archi uscenti da esso.



(a) Grado di un grafo indiretto



(b) Grado di un grafo diretto

Definizione 9.1.9 (Cammino, *path*)**.** In un grafo $G = (V, E)$, un cammino C di lunghezza k è una sequenza di nodi $u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leqslant i \leqslant k-1$.

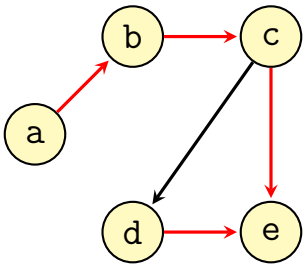


Figura 9.5: Cammino in un grafo diretto, a, b, c, e, d è un cammino di lunghezza 4

Nota. Un cammino è detto semplice se tutti i suoi nodi sono distinti.

9.1.5 Specifica

Nella versione più generale, il grafo è una struttura di dati dinamica che permette di aggiungere e rimuovere nodi e archi. La specifica che utilizzeremo non prevede la rimozione dei nodi dal grafo.

Algoritmo 9.1.1: Specifica della struttura dati GRAPH

Graph	// crea un grafo vuoto
SET V	// restituisce l'insieme di tutti i nodi
int size	// restituisce il numero di nodi
SET adj(NODE u)	// restituisce l'insieme di nodi adiacenti a u
insertNode(NODE u)	// aggiunge il nodo u al grafo
deleteNode(NODE u)	// rimuove il nodo u dal grafo
insertEdge(NODE u, NODE v)	// aggiunge l'arco (u, v) al grafo
deleteEdge(NODE u, NODE v)	// rimuove l'arco (u, v) dal grafo

9.1.6 Memorizzazione

Esistono due diversi modi per memorizzare un grafo:

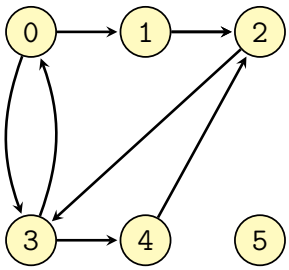
1. **matrici di adiacenza**: utilizza una matrice contenente un bit per indicare la presenza di ciascun arco: questo permette di controllare in tempo costante se un determinato arco è presente. La matrice di adiacenza viene ottenuta nel seguente modo:

$$m_{uv} = \begin{cases} 1 & (u, v) \in E \\ 0 & (u, v) \notin E \end{cases}$$

2. **lista di adiacenza**: una lista delle adiacenze presenti fra i nodi.

Grafo orientato

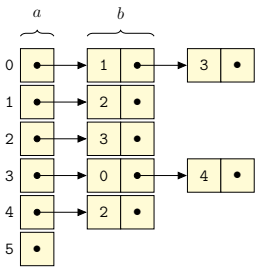
Memorizzare un grafo orientato attraverso matrice di adiacenza occupa uno spazio pari a n^2 bit, mentre se si utilizza una lista di adiacenza vengono occupati $an + bm$ bit.



(a) Grafo orientato

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	1	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

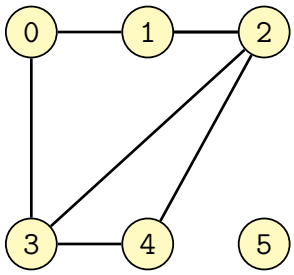
(b) Matrice di adiacenza



(c) Lista di adiacenza

Grafo non orientato

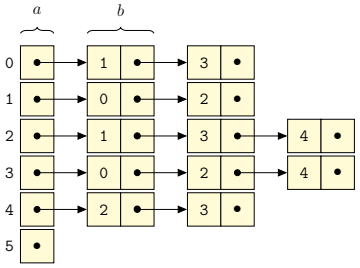
Se il grafo non è orientato ed utilizziamo una matrice di adiacenza per memorizzarlo, è sufficiente memorizzare solo la metà superiore, occupando uno spazio pari a $n(n-1)/2$ bit, mentre con lista di adiacenza dobbiamo raddoppiare i puntatori occupando $an + 2 \cdot bm$ bit.



(a) Grafo non orientato

	0	**1**	**2**	**3**	**4**	**5**
0		1	0	1	0	0
1			1	0	0	0
2				1	1	0
3					1	0
4						0
5						

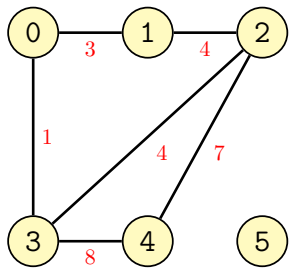
(b) Matrice di adiacenza



(c) Lista di adiacenza

Grafo pesato

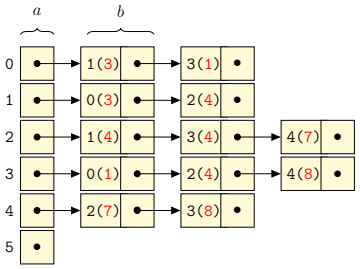
Infine se il grafo è pesato ed usiamo le matrici per rappresentarlo possiamo memorizzare il peso al posto del valore booleano, l'assenza dell'arco è quindi segnalata da un valore particolare (come -1 o ∞ in base alla procedura che vogliamo risolvere). Mentre se utilizziamo una lista di adiacenza memorizzeremo semplicemente il peso.



(a) Grafo pesato

	0	**1**	**2**	**3**	**4**	**5**
0		3	0	1	0	0
1			4	0	0	0
2				4	7	0
3					8	0
4						0
5						

(b) Matrice di adiacenza pesata



(c) Lista di adiacenza pesata

Dettagli sull'implementazione

Nel seguito, se non diversamente specificato, assumeremo che:

- l'implementazione sia basata su vettori di adiacenza (statici o dinamici);
- l'accesso alle informazioni abbia costo costante (ossia che la classe NODE sia equivalente a **int**);
- le operazioni per aggiungere nodi e archi abbiano costo ammortizzato $\mathcal{O}(1)$;
- dopo l'inizializzazione, il grafo sia statico.

Iterazione su nodi e archi

Negli algoritmi che seguiranno utilizzeremo questi schemi di codice per iterare su nodi ed archi.

Algoritmo 9.1.2: Schemi di iterazione per nodi ed archi

```
// iterare sui nodi
foreach u ∈ G.V do
    { Esegui operazioni sul nodo u }

// iterare sugli archi
foreach u ∈ G.V do
    { Esegui operazioni sul nodo u }
    foreach v ∈ G.adj(u) do
        { Esegui operazioni sull'arco u, v }
```

Complessità dell'iterazione Quest'operazione costa $\mathcal{O}(m+n)$ con le liste di adiacenza, mentre $\mathcal{O}(n^2)$ con le matrici di adiacenza.

Riassumendo

Le matrici sono ideali per grafi *densi*, occupano uno spazio $\mathcal{O}(n^2)$ e iterare su tutti gli archi costa $\mathcal{O}(n^2)$, verificare se u è adiacente a v richiede tempo costante $\mathcal{O}(1)$.

Le liste di adiacenza sono ideali per grafi *sparsi*, occupano uno spazio $\mathcal{O}(n+m)$ e iterare su tutti gli archi costa $\mathcal{O}(n+m)$, verificare se u è adiacente a v richiede tempo lineare $\mathcal{O}(n)$.

9.2 Visite dei grafi

Dato un grafo $G = (V, E)$ e un vertice $r \in V$ di partenza (che prende il nome di *radice* o di *sorgente*), si vuole visitare una e una volta sola tutti i nodi del grafo che possono essere raggiunti da r.

In ampiezza La visita in ampiezza effettua una visita dei nodi per livelli: prima visita la radice, poi i nodi a distanza uno dalla radice, poi i nodi a distanza due e così via... Una possibile applicazione di questa visita è quella di calcolare i cammini più brevi da una singola sorgente.

In profondità La visita in profondità effettua una visita ricorsiva: per ogni nodo adiacente, si visita il nodo e tutti i suoi nodi adiacenti ricorsivamente. Delle possibili applicazioni sono l'ordinamento topologico, la verifica della ciclicità e le componenti connesse e fortemente connesse.

9.2.1 Visita in ampiezza

Un approccio ingenuo alla visita di un grafo potrebbe essere il seguente:

Algoritmo 9.2.1: Primo tentativo di visita di un grafo

```
visita(GRAPH G)
    foreach u ∈ G.V do
        { visita nodo u }
        foreach v ∈ G.adj(u) do
            { visita arco (u, v) }
```

Ma la struttura del grafo non viene presa in considerazione, poiché si itera su tutti i nodi e gli archi senza alcun criterio. Un possibile approcio potrebbe essere quello di sfruttare l'algoritmo delle visite sugli alberi

Algoritmo 9.2.2: Algoritmo adatto all'attraversamento degli alberi

```
BFSTraversal(GRAPH G, int r)
    QUEUE Q ← Queue
    Q.enqueue(r)
    while not Q.isEmpty do
        NODE u ← Q.dequeue
        { visita il nodo u }
        foreach u ∈ G.adj(u) do
            Q.enqueue(v)
```

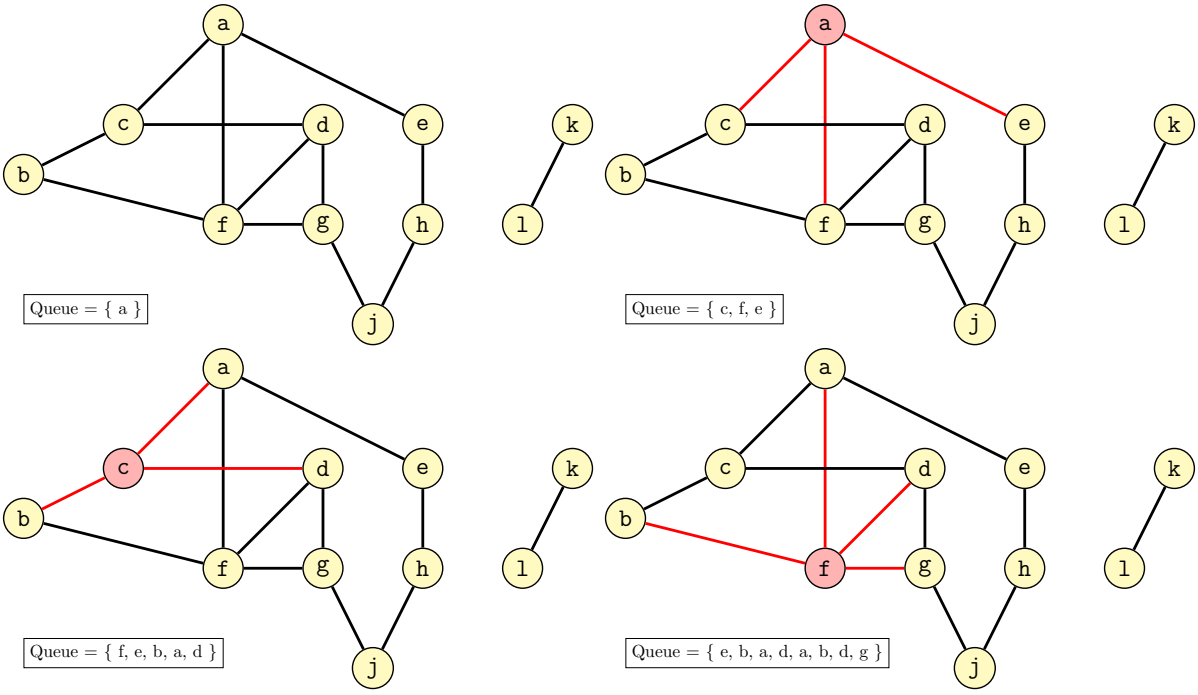


Figura 9.9: Esempio di attraversamento di un grafo tramite la procedura di visita di un albero

Dall'esempio possiamo notare come i nodi vengano reinseriti all'infinito all'interno della coda e questo non permette all'algoritmo di terminare. Negli alberi, data la loro struttura, abbiamo la sicurezza che non visiteremo mai lo stesso nodo più di una volta. Abbiamo bisogno di un meccanismo che ci permetta di evitare che questo avvenga. Possiamo farlo memorizzando i nodi che abbiamo già visitato.

Algoritmo 9.2.3: Algoritmo adatto all'attraversamento dei grafi

```
visita(GRAPH G, NODE r)
    SET S ← Set  // insieme generico, da specificare (STACK, QUEUE)
    S.insert(r)  // inserisco il nodo, da specificare

    // ho visitato il nodo
    { marca il nodo r come "scoperto" }

    // fintanto che l'insieme non è vuoto
    while S.size > 0 do
        // la politica di rimozione dipende dal problema da risolvere
        NODE u ← S.remove
        { esamina il nodo u }
        foreach v ∈ G.adj(u) do
            { esamina l'arco (u, v) }
            if v non è già stato scoperto then
                // serve a non inserire il nodo più di una volta
                { marca il nodo v come "scoperto" }
                S.insert(v)  // inserisce il nodo nell'insieme, da specificare
```

Gli obiettivi della visita in ampiezza sono:

- visitare i nodi a distanze crescenti dalla sorgente: visitare quindi i nodi a distanza k prima di quelli a distanza $k+1$;
- calcolare il cammino più breve da r a tutti gli altri nodi, dove il cammino più breve è il percorso con il minor numero di archi;
- generare un albero in ampiezza (*breadth-first*): l'albero in ampiezza è un albero contenente tutti i nodi raggiungibili da r, tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u nel grafo.

Algoritmo 9.2.4: Procedura specializzata per la visita in ampiezza di un grafo

```
// visitare tutti i nodi a distanza k prima di visitare i nodi a distanza k+1
bfs(GRAPH G, NODE r)
    QUEUE S ← Queue // creo una pila
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    bool[ ] visitato ← bool[1...G.n] // della dimensione del no. di nodi
    foreach u ∈ G.V − {r} do visitato[u] ← false // devo ancora visitarli
    visitato[r] ← true // radice visitata

    // visita del grafo
    while not S.isEmpty do
        NODE u ← S.dequeue // rimuovo un nodo
        { esamina il nodo u }
        foreach v ∈ G.adj(u) do // per ciascun nodo adiacente "v"
            { esamina l'arco (u, v) }
            if not visitato[v] then // se non ho ancora visitato "v"
                visitato[v] ← true // marcalo come visitato
                S.enqueue // inseriscilo nella coda
```

9.2.2 Cammini più brevi

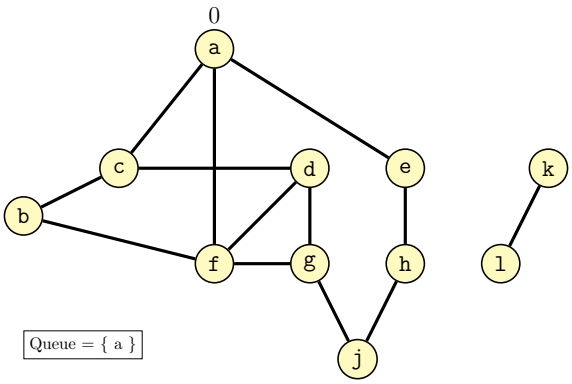
Vediamo ora un'applicazione della visita in ampiezza: la ricerca dei cammini più brevi e lo facciamo tramite un esempio particolare: calcolare il numero di erdös.

Paulo Erdös è stato uno dei matematici più prolifici al mondo (1500+ articoli e 500+ co-autori). Definiamo il numero di erdos nel seguente modo:

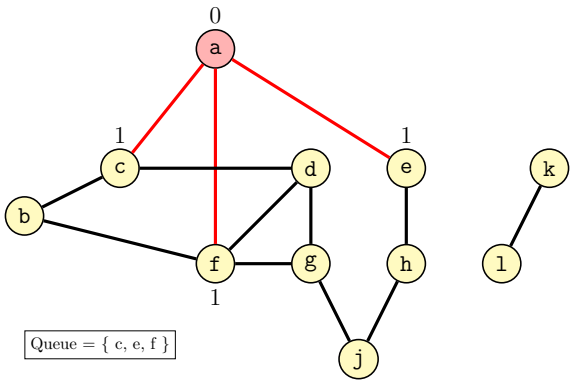
- Erdös ha valore $erdos = 0$;
- i co-autori di Erdös hanno $erdos = 1$;
- se X è co-autore di qualcuno con $erdos = k$ e non è co-autore con qualcuno con $erdos < k$, allora X ha $erdos = k + 1$;
- le persone non raggiunte da questa definizione hanno $erdos = +\infty$.

Algoritmo 9.2.5: Ricerca dei cammini minimi più brevi dalla radice

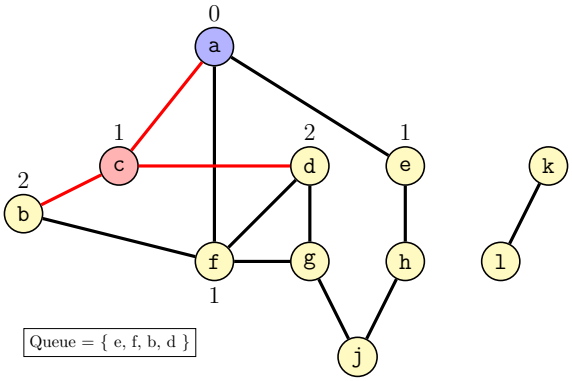
```
// il camminimo più breve fra due vertici viene memorizzato tramite il vettore dei padri p
erdos(GRAPH G, NODE r, int[] erdos, NODE parent)
    // struttura di supporto
    QUEUE S ← Queue // creo una pila
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    foreach u ∈ G.V − {r} do erdos[u] ← ∞ // nodi non ancora raggiunti
    erdos[r] ← true // erdõs ha distanza 0 da se stesso
    parent[r] ← nil // per la stampa del cammino

    // visita del grafo
    while not S.isEmpty do
        NODE u ← S.dequeue
        foreach u ∈ G.adj(u) do
            { esamina l'arco (u, v) }
            if erdos[v] == ∞ then
                // il nodo non è stato ancora stato scoperto
                erdos[v] ← erdos[u] + 1 // gli assegno un livello di erdõs+1
                parent[v] ← u // memorizzo il padre del nodo attuale nel v. dei padri
                S.enqueue(v) // è la prima volta che lo raggiungo quindi lo metto in coda
```



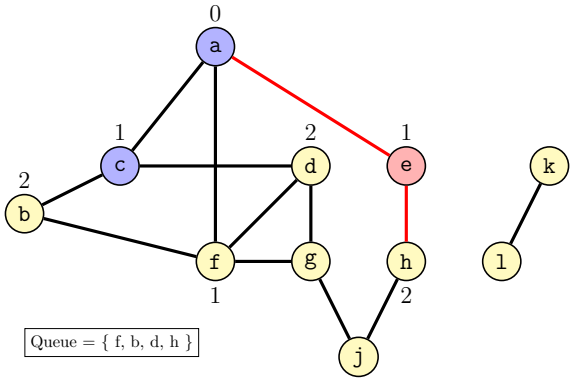
(a) Assegno al nodo a distanza 0, lo aggiungo alla coda



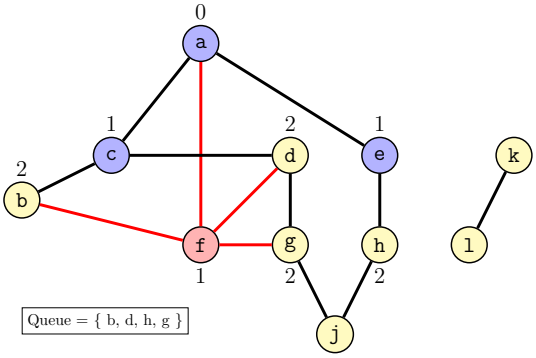
(b) Estraggo il nodo a, assegno ai suoi nodi adiacenti non ancora visitati (c, f, e) distanza $a + 1 = 1$ e li aggiungo alla coda



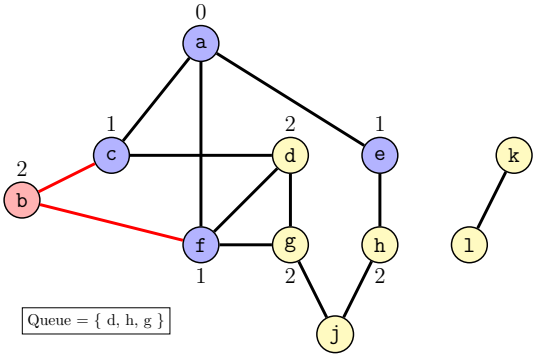
(c) Estraggo il nodo c, assegno ai suoi nodi adiacenti non ancora visitati (b, d) distanza $c + 1 = 2$ e li aggiungo alla coda



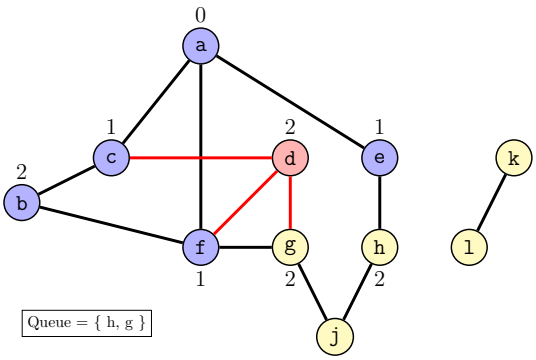
(d) Estraggo il nodo e, assegno ai suoi nodi adiacenti non ancora visitati (h) distanza $e + 1 = 2$ e lo aggiungo alla coda



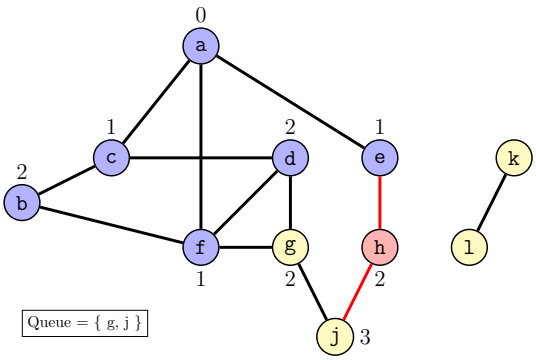
(e) Estraggo il nodo f, assegno ai suoi nodi adiacenti non ancora visitati (b, g) distanza $f + 1 = 2$ e li aggiungo alla coda



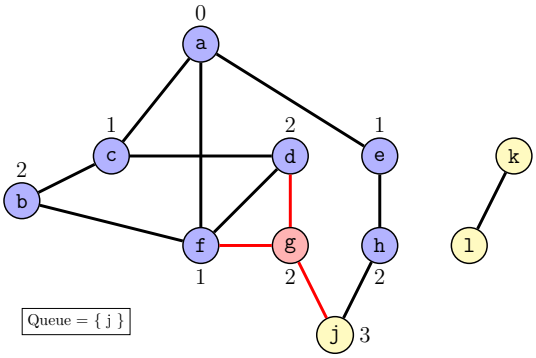
(f) Estraggo il nodo b, il quale non ha nodi adiacenti non ancora visitati



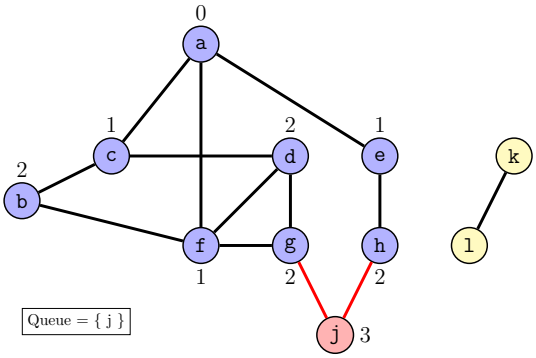
(g) Estraggo il nodo d, il quale non ha nodi adiacenti non ancora visitati



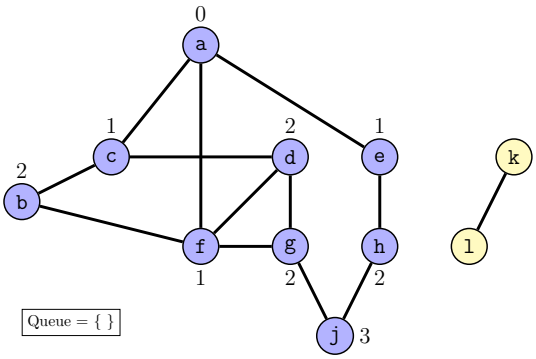
(h) Estraggo il nodo h, assegno al suo nodo adiacente non ancora visitato (j) distanza $h + 1 = 3$ e lo aggiungo alla coda



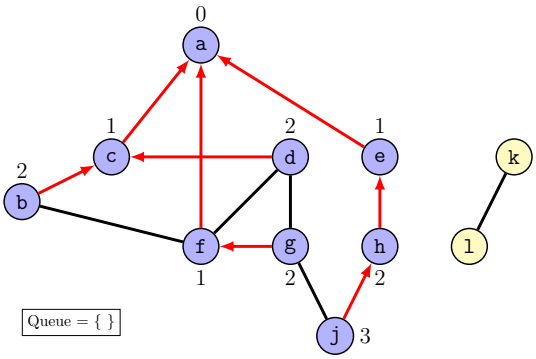
(i) Estraggo il nodo g, il quale non ha nodi adiacenti non ancora visitati



(j) Estraggo il nodo j, il quale non ha nodi adiacenti non ancora visitati



(k) Questo è l'albero risultante



(l) Rappresentazione dell'albero di copertura bfs, attraverso il vettore dei padri

Figura 9.10: Esempio di visita di un grafo tramite la procedura di visita in ampiezza

L'albero "di copertura" con radice r viene memorizzato tramite vettore dei padri (che nell'algoritmo di erdos abbiamo chiamato *parent*).

Algoritmo 9.2.6: Stampa del cammino

```
// stampa il cammino da r a s nell'ordine corretto
stampaCammino(NODE r, NODE s, NODE[] parent)
    // parent: vettore dei padri
    if r == s then // se è la radice
        stampa s // la stampo
    else if parent[s] == nil then
        stampa "nessun cammino da r a s"
    else
        // la chiamata ricorsiva prima della stampa per stampare in ordine
        stampaCammino(r, parent[s], p)
        stampa s
```

9.2.3 Complessità della visita in ampiezza

Ognuno degli n nodi viene inserito nella coda al massimo una volta, ed ogni volta che un nodo viene estratto tutti i suoi archi vengono analizzati una volta sola, per una complessità totale di $\mathcal{O}(m+n)$. Il numero di archi analizzati è quindi

$$m = \sum_{u \in V} out_d(u)$$

dove $out_d(u)$ è il grado uscente (*out-degree*) del nodo u.

9.2.4 Visita in profondità

Molto spesso la visita in profondità è solo una parte di una soluzione ad un altro problema, viene utilizzata per esplorare un intero grafo, e non solo i nodi raggiungibili da una singola sorgente.

L'output dell'algoritmo non è più un singolo albero, ma una foresta *depth-first* (ossia un insieme di alberi *depth-first*).

La struttura dati utilizzata non è più una coda, bensì uno *stack* implicito (attraverso la ricorsione) o esplicito (vedremo un algoritmo nel seguito).

Algoritmo 9.2.7: Visita in profondità, ricorsiva con stack implicito

```
// genera un albero depth-first
dfs(GRAPH G, NODE u, bool[] visitato)
    visitato[u] = true // ho visitato il nodo
(5) { esamina il nodo u (caso pre-visita) }
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u, v) }
        if not visitato[v] then // se non l'ho ancora visitato
            // chiamata ricorsiva
            dfs(G, v, visitato) // lo visito ricorsivamente
(6) { esamina il nodo u (caso post-visita) }
```

Analisi della complessità Questo algoritmo ha la stessa complessità della visita in ampiezza: $\mathcal{O}(n+m)$ con il grafo implementato tramite liste di adiacenza, e $\mathcal{O}(n^2)$ con matrice di adiacenza.

Si presenta però un problema, mentre con gli alberi possiamo assumere che la loro profondità sia limitata, con i grafi non possiamo fare lo stesso discorso. Eseguire una visita in profondità basata su chiamate ricorsive può essere rischioso (errore di stack overflow) in grafi molto grandi e connessi (in particolare se non sono orientati, in quanto la visita analizza tutti i nodi). È possibile che la profondità raggiunta sia troppo grande per la dimensione dello stack del linguaggio. In tali casi, si preferisce utilizzare una visita in ampiezza, oppure in profondità ma basata su stack esplicito.

Algoritmo 9.2.8: Visita in profondità, iterativa con stack esplicito, visita in *pre*-ordine

```
// effettua una visita in profondità iterativa
dfs(GRAPH G, NODE r)
    STACK S ← Stack
    S.push(r) // inserisco la radice nella pila
    bool[ ] visitato ← new bool[1 ... G.size]
    foreach u ∈ G.V − {r} do visitato[u] ← false

    visitato[r] ← true // marco la radice come visitata
    while not S.isEmpty do
        NODE u ← S.pop // estraggo un nodo
        if not visitato[v] then // se non l'ho ancora visitato
            { esamina il nodo u in pre-ordine }
            visitato[v] ← true // lo segno come visitato
            foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
                { esamina l'arco (u, v) }
                S.push(v) // lo inserisco nella pila
```

La procedura si ottiene semplicemente sostituendo una pila alla coda utilizzata nella visita in ampiezza.

Un nodo può essere inserito nella pila più volte (tante volte quanti sono gli archi entranti in quel nodo, il numero di archi entranti in tutti i nodi è limitato superiormente dal numero di archi m, quindi $\mathcal{O}(m)$, in quanto il controllo se un nodo è già stato inserito viene fatto all'estrazione, non all'inserimento come avveniva in precedenza.

Complessità della visita in ampiezza con stack esplicito Inserimenti ed estrazioni sono pari al numero degli archi, quindi $\mathcal{O}(m)$, visitare gli archi costa $\mathcal{O}(m)$ e le visite dei nodi costano $\mathcal{O}(n)$, per una complessità risultante di $\mathcal{O}(m+n)$, invariata rispetto agli algoritmi precedenti.

Visita in post-ordine È possibile effettuare una visita in profondità con una procedura con stack esplicito e visita in *post*-ordine ma abbiamo bisogno di aggiungere due "flag": discovery e finish.

Quando un nodo viene scoperto viene inserito nello stack con il tag discovery; quando un nodo viene estratto dalla coda con tag discovery: viene re-inserito con il tag finish e tutti i suoi vicini vengono inseriti; Quando un nodo viene estratto dalla coda con tag finish, viene effettuata la post-visita. Non vedremo il codice poiché è complicato e i dettagli non sono interessanti.

9.2.5 Componenti connesse

Vogliamo identificare le componenti connesse di un grafo. Prima di tutto vogliamo capire se è connesso, dopodiché vogliamo sapere quante sono le sue componenti connesse. Il motivo per cui vogliamo farlo è che molti algoritmi che operano sui grafi iniziano decomponendo il grafo nelle sue componenti connesse per poi ri-comporre i risultati assieme. Sono definite due tipologie di problemi:

- componenti connesse (*Connected Components, CC*);
- componenti fortemente connesse (*Strongly Connected Components, SCC*)

Entrambi i problemi sono definiti su grafi non orientati.

Per capire meglio il problema abbiamo bisogno di qualche definizione:

Definizione 9.2.1 (Raggiungibilità di un nodo)**.** Un nodo v è raggiungibile da un nodo u se esiste almeno un cammino da u a v.

Definizione 9.2.2 (Grafo non orientato connesso)**.** Un grafo non orientato $G = (V, E)$ è connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.

Definizione 9.2.3 (Componente connessa)**.** Un grafo $G' = (V', E')$ è una componente connessa di G se e solo se G' è un sottografo connesso e massimale di G.

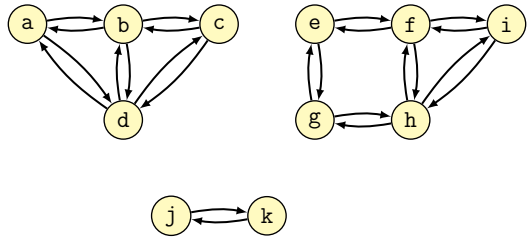
Definizione 9.2.4 (Sottografo)**.** G' è un sottografo di G $(G' \subseteq G) \Leftrightarrow V' \subseteq V$ e $E' \subseteq E$

Definizione 9.2.5 (Grafo massimale)**.** G' è massimale se e solo se non esiste nessun altro sottografo G'' di G tale che G'' è connesso e più grande di G' (ad esempio $G' \subseteq G'' \subseteq G$)

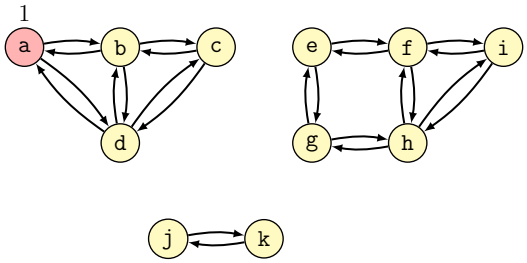
Vogliamo quindi verificare se un grafo è connesso oppure no, ed identificare le sue componenti connesse. Per farlo utilizzeremo di un vettore (che chiameremo id), il quale conterrà l'indentificatore alla quale la componente appartiene ($id[u]$ è l'indentificatore della c.c. a cui appartiene u). Un grafo risulta connesso se, al termine della visita in profondità, tutti i suoi nodi risultano marcati. Altrimenti la visita deve ricominciare da capo da un nodo non marcato, identificando una nuova componente.

Algoritmo 9.2.9: Identifica le componenti connesse di un grafo non orientato

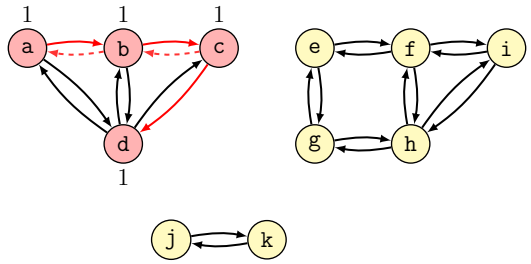
```
// parte iterativa
int[ ] cc(Graph G, Stack S)
    // creo un vettore della dimensione dei nodi del grafo
    int[ ] id ← new int[1 ... G.size]
    // inizializzo il vettore
    foreach u ∈ G.V do id[u] ← 0
    int counter = 0  // contatore delle componenti connesse
    foreach u ∈ G.V do // per ogni nodo del grafo
        if id[u] == 0 then // ho trovato una nuova componente connessa
            counter++  // aggiorno il contatore
            // effettuo una chiamata ricorsiva sul nodo scoperto
            ccdfs(G, counter, u, id)
    // restituisco l'identificativo della componente connessa
    return id
// visita ricorsiva di ciascuna componente
ccdfs(Graph G, int counter, Node u, int[ ] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u:       il nodo che sto visitando
    // id:      l'identificativo della componente
    // memorizzo l'identificativo della cc
    id[v] ← counter
    foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
        if id[v] == 0 then // non è ancora stato visitato
            // v:  il nodo su cui vado ad operare
            ccdfs(G, counter, v, id)  // lo visito ricorsivamente
```



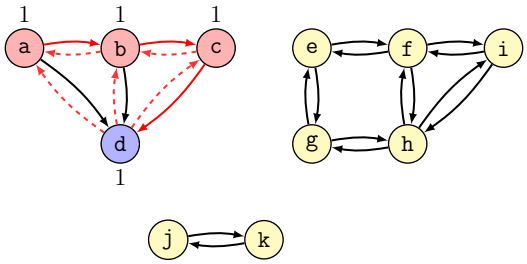
(a) Stato iniziale



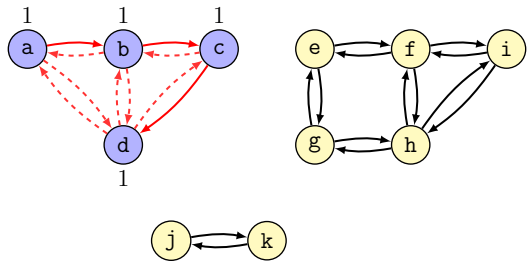
(b) Partiamo dal nodo a per comodità, gli assegniamo il valore 1



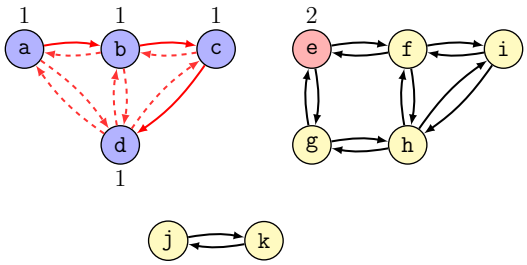
(c) Visito tutti i nodi adiacenti al nodo a



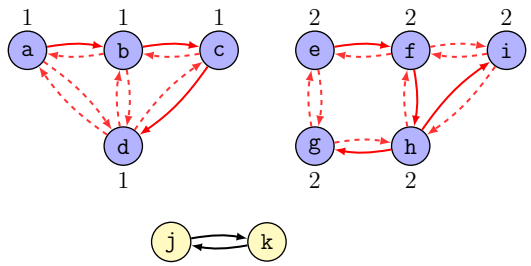
(d) Una volta arrivati al nodo d abbiamo già visitato tutti i suoi possibili vicini



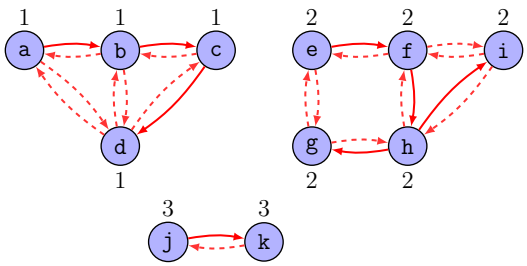
(e) Abbiamo completato la visita della prima componente connessa



(f) Abbiamo ricontrollato se potevamo ripartire da uno qualsiasi dei nodi della prima componente ma aveva già un *id* assegnato, partiamo da e per comodità



(g) Completo così anche la seconda componente...



(h) ...e la terza

Figure 9.11: Esempio di identificazione delle componenti connesse in un grafo non orientato

9.3 Verifica ciclicità

Grafi aciclici non orientati

Definizione 9.3.1 (ciclo, *cycle*)**.** In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1, \dots u_k$ tale che $(u_i, u_{i+1} \in E)$ (un cammino) per $0 \leqslant i \leqslant k-1$ e $u_0 = u_k$ (il primo e l'ultimo nodo coincidono, ossia è chiuso).

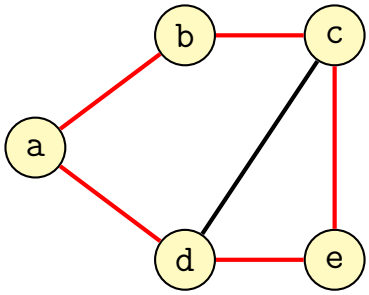


Figure 9.12: $k > 2$ esclude cicli banali composti da coppie di archi, i quali sono onnipresenti nei grafi non orientati. Questo grafo contine 3 cicli, uno di lunghezza 3, uno di lunghezza 4 ed uno di lunghezza 5.

Definizione 9.3.2 (Grafo aciclico)**.** Un grafo non orientato che non contiene cicli è detto aciclico.

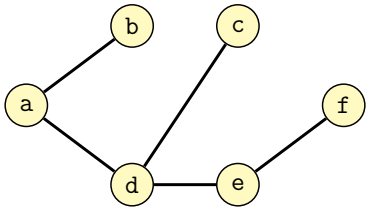


Figure 9.13: Un grafo aciclico.

Un grafo che contiene cicli è detto *ciclico*, altrimenti viene detto *aciclico*. Dato un grafo non orientato, vogliamo poter essere in grado di determinare se contiene cicli o meno. Dobbiamo scrivere quindi un algoritmo che restituisca **true** se il grafo contiene un ciclo, **false** altrimenti.

Nota. Se è un grafo connesso ed è aciclico, allora è un albero.

L'idea è che se visito un nodo che ho già visitato, allora ho identificato un ciclo.

Algoritmo 9.3.1: Ricerca di un ciclo in un grafo non orientato

```
// restituisce true se trova un ciclo
bool hasCycleRec(GRAPH G, NODE u, NODE p, bool[ ] visited)
    // G:       grafo esplorato
    // u:       nodo da esaminare
    // p:       nodo da cui provengo (padre)
    // visited: vettore dei nodi visitati
    visited[u] ← true  // lo visito per la prima volta
    foreach v ∈ G.adj(u) − {p} do // visito tutti i suoi vicini
        // G.adj(u) − {p}:  non considero il nodo da cui provengo (è un grafo orientato)
        if visited[v] then // ho già visitato il nodo
            return true  // ho trovato un ciclo
            // altrimenti effettuo una visita ricorsiva sul nodo vicino v
        else if hasCycleRec(G, v, u, visited) then
            // se una qualsiasi delle sottochiamate ritorna vero, allora ho trovato un ciclo
            return true
    // non ho trovato alcun ciclo
    return false

// ricerca di un ciclo per grafi disconnessi
bool hasCycle(GRAPH G)
    bool[ ] visited ← new bool[ ]1G.size  // creo il vettore
    foreach u ∈ G.V do // lo inizializzo
        visited[u] ← false
    foreach u ∈ G.V do // per ciascun nodo appartenente al grafo
        if not visited[u] then // il primo nodo non sarà stato visitato
            if hasCycleRec(G, v, null, visited) then
                // effettuo una visita ricorsiva sul nodo vicino v
                return true
    return false
```

La parte non ricorsiva dell'algoritmo ci permettere di cercare cicli in grafi non connessi, in quanto stiamo osservando grafi e non alberi (i quali sono connessi).

Grafi aciclici orientati

Cosa succede se iniziamo a considerare i grafi orientati?

Definizione 9.3.3 (ciclo, *cycle*)**.** In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k \geqslant 2$ è una sequenza di nodi $u_0, u_1, \ldots u_k$ tale che $(u_i, u_{i+1} \in E)$ (un cammino) per $0 \leqslant i \leqslant k-1$ e $u_0 = u_k$ (il primo e l'ultimo nodo coincidono, ossia è chiuso).

La definizione è identica a parte che ora consideriamo come cicli anche i cammini composti da due soli nodi ($k \geqslant 2$)

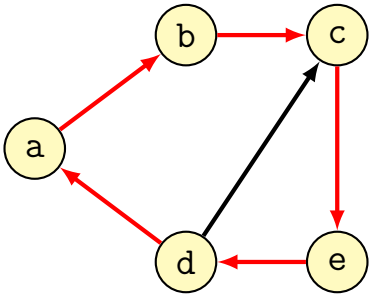
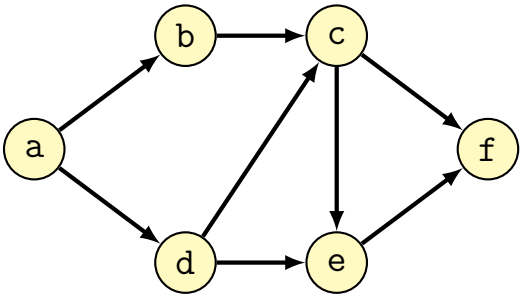


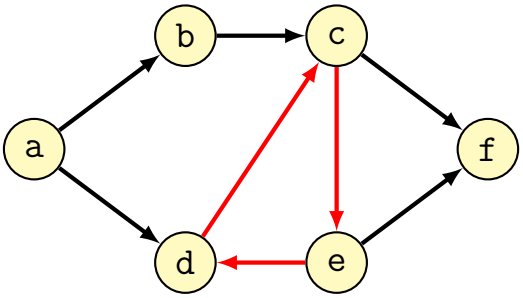
Figure 9.14: a, b, c, e, d, a è un cammino di lunghezza 5

Nota. Un ciclo è detto semplice se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo)

Definizione 9.3.4 (Grafo diretto aciclico, *Directed Acyclic Graph*)**.** Un grafo orientato che non contiene cicli è detto DAG (Directed Acyclic Graph)



(a) Un grafo aciclico.



(b) Un grafo ciclico. Il ciclo è dato dai nodi c, e, f.

Figure 9.15: Esempio di grafo aciclico e ciclico

I *Directed Acyclic Graph*, DAG d'ora in poi, rappresentano una classe di problemi ben precisi e vedremo degli algoritmi che trattano nello specifico questi grafi. Il problema è il medesimo: dato un grafo non orientato, vogliamo poter essere in grado di determinare se contiene cicli o meno. Dobbiamo scrivere quindi un algoritmo che restituisca **true** se il grafo contiene un ciclo, **false** altrimenti.

L'algoritmo precedente riesce a risolvere anche queto problema? Riesci a pensare ad un grafo orientato per cui l'algoritmo appena visto non si comporta correttamente?

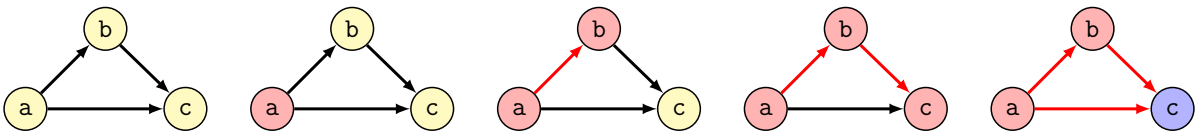
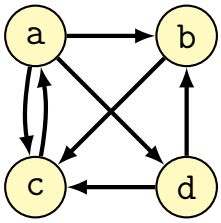


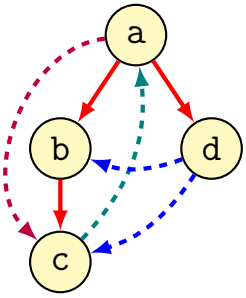
Figure 9.16: Se in un grafo esistono due cammini che portano allo stesso nodo l'algoritmo precedente dirà erroneamente che esiste un ciclo.

Classificazione degli archi

Ogni volta che si esamina un arco da un nodo marcato (visitato) ad un nodo non marcato (non visitato), tale arco viene detto arco dell'albero. Gli archi non inclusi nell'albero possono essere divisi in tre categorie:

- se u è un antenato di v in T, (u, v) è detto arco in avanti;
- se u è un discendente di v in T, (u, v) è detto arco all'indietro;
- altrimenti viene detto arco di attraversamento.





Algoritmo 9.3.2: Schema per visita dell'albero in profondità

```
// classifica i lati di un grafo
dfs-schema(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
    // time:    contatore
    // dt:      tempo di scoperta
    // ft:      tempo di fine
    esamina il nodo u (caso pre-visita)
    time++ // incremento il contatore
    dt[u] ← time // lo memorizzo nel vettore di scoperta
    // effettuo una visita in profondità
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u, v) (qualsiasi) } // qui si sviluppa la logica dell'algoritmo
        if dt[v] == 0 then // non ho ancora esaminato il nodo
            { esamina l'arco (u, v) (albero) }
            dfs-schema(G, v, time, dt, ft) // effettuo la chiamata ricorsiva
        else if dt[u] > dt[v] and ft[v] == 0 then
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita, allora ho
               trovato un arco all'indietro
            { esamina l'arco (u, v) (indietro) }
        else if dt[u] < dt[v] and ft[v] ≠ 0 then
            // se raggiungo un mio discendente e ho terminato la mia visita, allora ho trovato un arco
               in avanti
            { esamina l'arco (u, v) (avanti) }
        else
            // l'ultimo caso rimanente
            { esamina l'arco (u, v) (attraversamento) }
    { visita il nodo u (post-visita) }
    time++ // aggiorno il contatore
    ft[u] ← time // lo memorizzo nel vettore di fine
```

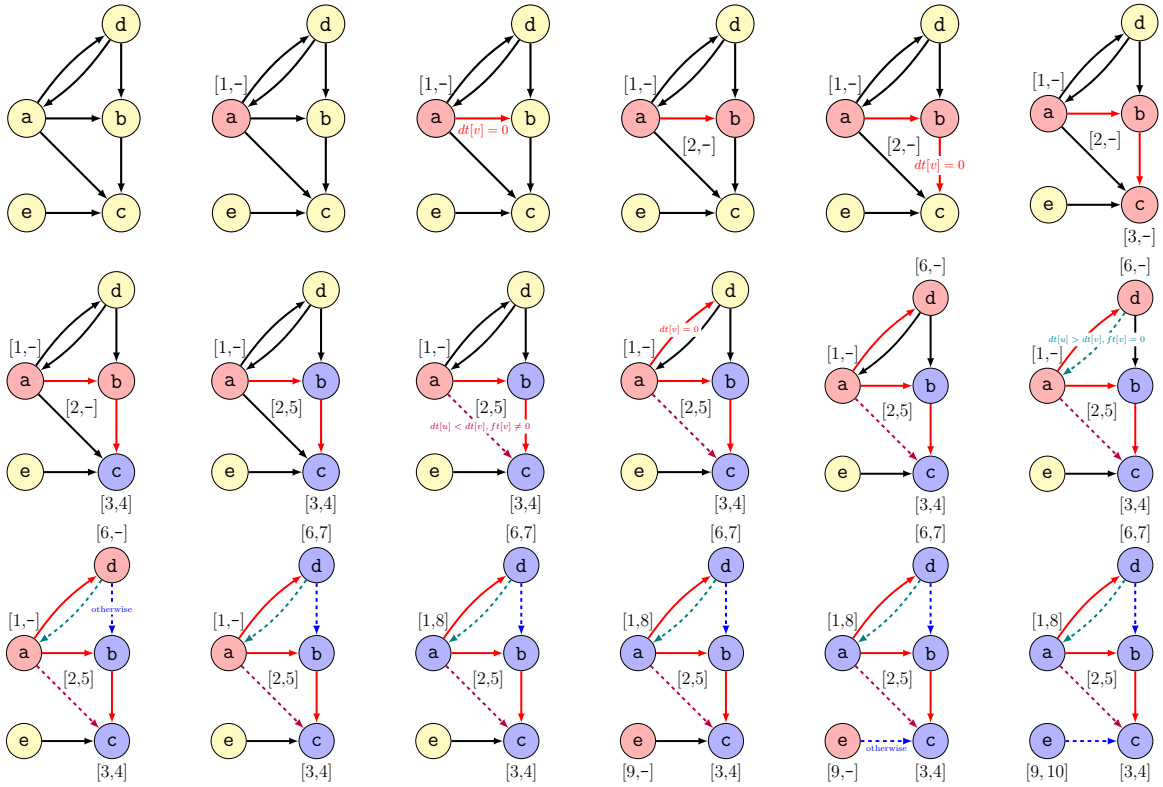


Figure 9.18: Visitati in ordine alfabetico per comodità

Osserviamo i numeri assegnati ai nodi; se li consideriamo come intervalli allora $a \subset b$ perché $[1,8] \subset [2,5]$. Osservando gli intervalli possiamo quindi dedurre le relazioni di discendenza fra i nodi.

Ma perché li classifichiamo? Perché possiamo dimostrare delle proprietà sul tipo di archi e usarle per costruire algoritmi migliori.

Teorema. *Data una visita in profondità di un grafo $G = (V, E)$, per ogni coppia di nodi $(u, v) \in V$, solo una delle condizioni seguenti è vera:*

- *Gli intervalli $[dt[u], ft[u]$ e $[dt[v], ft[v]]$ sono non-sovrapposti; u, v non sono discendenti l'uno dell'altro nella foresta* depth-first*;*
- *L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$; u è un discendente di v in un albero* depth-first*;*
- *L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$; v è un discendente di u in un albero* depth-first*.*

Teorema. *Un grafo orientato è aciclico se e solo se non esistono archi all'indietro nel grafo*

Dimostrazione. Abbiamo due casi:

1. se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e sia (u, v) un arco del ciclo. Il cammino che connette u a v verrà prima o poi visitato, e da v verrà scoperto l'arco all'indietro (v, u) (se esiste un ciclo prima o poi nella visita lo vado a toccare);
2. se esiste un arco all'indietro (u, v) dove v è un antenato di u, allora esiste un cammino da v a u e un arco da u a v, ovvero un ciclo.

□

Sfruttando questa dimostrazione possiamo quindi semplificare l'algoritmo precedente per la ricerca di un ciclo in un grafo aciclico diretto.

Algoritmo 9.3.3: Ricerca di un ciclo in un grafo aciclico diretto

```
// applicabile solo ai DAG, in quanto non hanno archi all'indietro
bool hasCycle(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
    // u:  il primo nodo che viene visitato

    time++ // aumento il contatore
    dt[u] ← time // memorizzo il tempo di scoperta

    foreach v ∈ G.adj(u) do
        if dt[v] == 0 then // non ho ancora scoperto questo nodo
            // effettuo una visita ricorsiva
            if hasCycle(G, v, time, dt, ft) then
                return true

        // logica dell'algoritmo
        else if dt[u] > dt[v] and ft[v] == 0 then
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita, allora ho
               trovato un arco all'indietro (un ciclo)
            return true

    time++ // aumento il contatore
    ft[u] ← time // memorizzo il tempo di fine

    // non ho trovato un ciclo
    return false
```

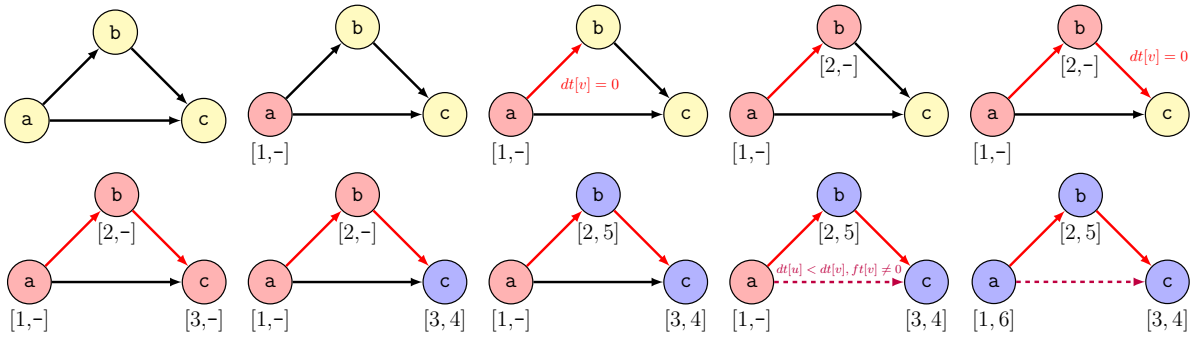


Figure 9.19: Questo è il particolare esempio sulla quale l'algoritmo precedente falliva

Nota. Se nella schema degli intervalli ci sono due intervalli sovrapposti, allora esiste un ciclo.

9.4 Ordinamento topologico

L'obiettivo è quello di scrivere un algoritmo che prende in input un DAG e che ne restituisca un possibile ordinamento topologico.

Definizione 9.4.1 (ordinamento topologico)**.** Dato un grafo diretto e aciclico (DAG) G, un ordinamento topologico di G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.

Nota. Se il grafo contiene un ciclo, non esiste un ordinamento topologico.

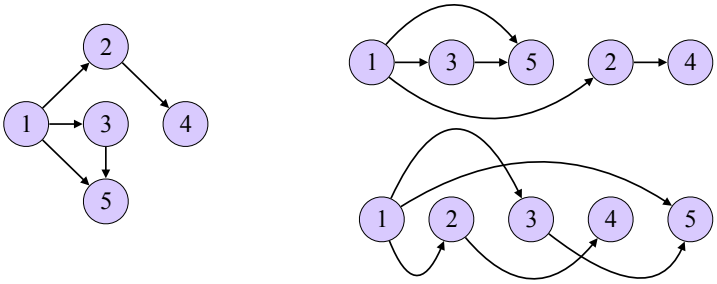


Figure 9.20: Possono esistere più ordinamenti topologici.

Un approccio banale potrebbe essere il seguente:

- trovo un nodo senza archi entranti;
- aggiungo questo nodo nell'ordinamento e lo rimuovo dal grafo insieme a tutti i suoi archi;
- ripeto questa procedura fino a quando tutti i nodi sono stati rimossi.

Si può fare meglio di così. Eseguiamo una visita in profondità nel quale l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista in *post*-ordine. E restituiamo la lista così ottenuta. Restituiamo in output la sequenza dei nodi ordinati per tempo decrescente di fine.

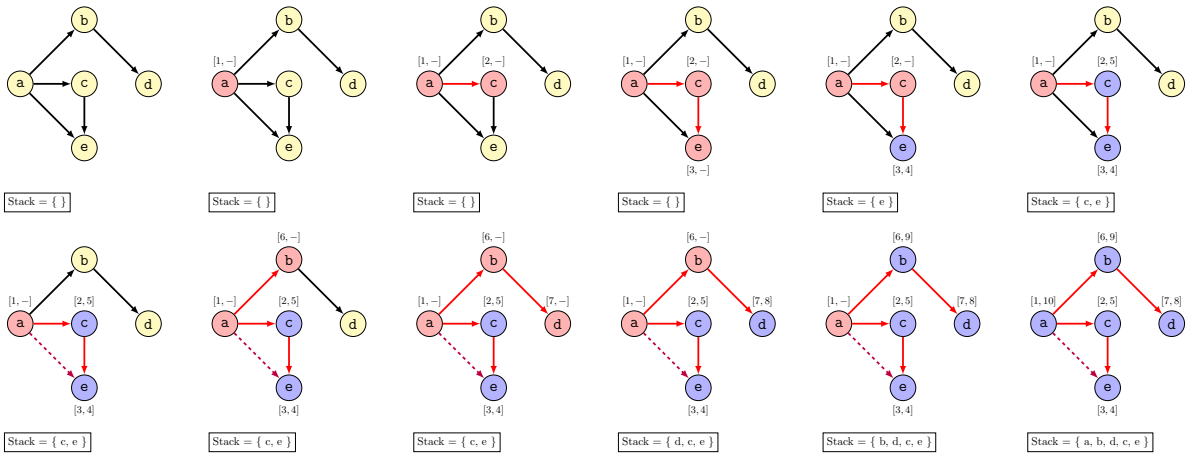


Figure 9.21: Esempio di ordinamento topologico

Algoritmo 9.4.1: Ordinamento topologico di un grafo orientato aciclico

```
// ritorna una pila in cui il primo elemento è il primo elemento dell'ordinamento
STACK topSort(GRAPH G)
    STACK S ← Stack
    bool[ ] visited ← new bool[1 ... G.size]
    foreach u ∈ G.V do
        visited[u] ← false
    foreach u ∈ G.V do // per ogni nodo del grafo
        if not visitato[u] then // se non l'ho visitato
            // effettua una chiamata ricorsiva
            ts-dfs(G, u, visitato, S)
    return S

// restituisce l'ordinamento topologico dei nodi di un DAG
int ts-dfs(GRAPH G, NODE u, bool[ ] visitato, STACK S)
    visitato[u] ← true // imposta il nodo come visitato
    foreach v ∈ G.adj(u) do
        // è una grafo diretto aciclico quindi non ho bisogno di ricordarmi da dove sono venuto
        if not visitato[v] then
            // effettua una visita in profondità
            i ← ts-dfs(G, u, visitato, S)
    S.push(u) // aggiungi il nodo in testa alla pila
```

Quando termino tutte le chiamate ricorsive l'algoritmo restituisce un ordinamento topologico dei nodi del grafo dato in input; quando un nodo è "finito" tutti i suoi discendenti sono stati scoperti e aggiunti alla lista. Aggiungendolo in testa alla lista, il nodo si trova prima dei nodi a cui i suoi archi puntano, ossia i suoi discendenti.

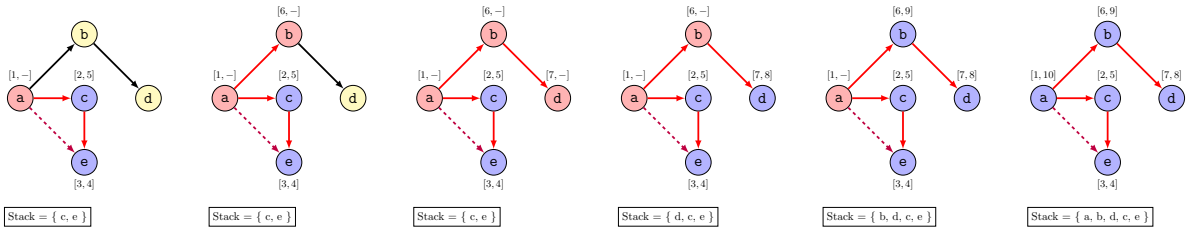


Figure 9.22: Esempio di ordinamento topologico alternativo al precendente, il quale dimostra informalmente che l'algoritmo funziona partendo da qualsiasi nodo

9.5 Componenti fortemente connesse

Definizione 9.5.1 (Grafo fortemente connesso)**.** Un grafo orientato $G = (V, E)$ è fortemente connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.

Definizione 9.5.2 (Componente fortemente connessa)**.** Un grafo $G' = (V', E')$ è una componente fortemente connessa di G se e solo se G' è un sottografo connesso e massimale di G.

Le definizioni di fortemente connessa è identica alla definizione di componente connessa, ma si opera su grafi orientati, mentre prima operavamo su grafi non orientati.

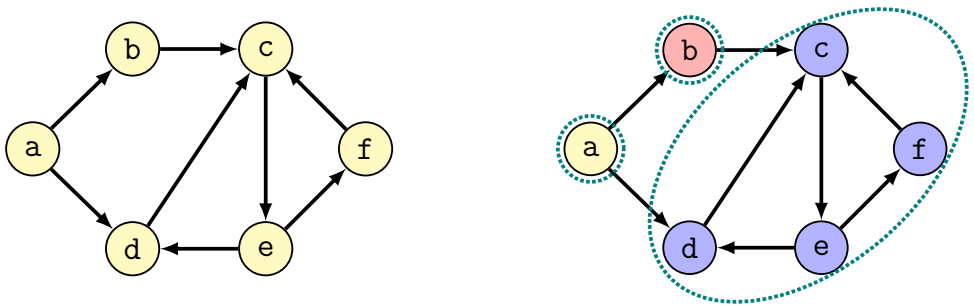


Figure 9.23: (a) e (b) sono componenti connesse massimali, (c, d, e, f) è una componente fortemente connessa.

Per definire le componenti fortemente connesse potremmo applicare l'algoritmo cc; purtroppo il risultato dipende dal nodo di partenza.

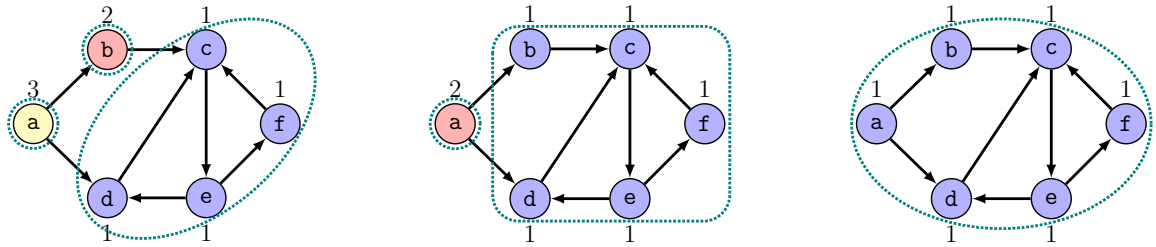


Figure 9.24: Il risultato dipende dal nodo di partenza.

9.5.1 L'algoritmo di Kosaraju

L'algoritmo di Kosaraju:

1. effettua una visita in profondità del grafo G
2. ne calcola il grafo trasposto G^T (ossia il grafo con la direzione degli archi invertiti)
3. esegue una visita in profondità sul grafo trasposto G^T utilizzando l'algoritmo cc, esaminando i nodi nell'ordine inverso di tempo di fine della prima visita;

Le componenti connesse (e i relativi alberi *depth-first*) rappresentano le componenti fortemente connesse di G.

Algoritmo 9.5.1: Algoritmo di Kosaraju

```
// indentifica le componenti fortemente connesse
int[ ] scc(GRAPH G)
    STACK S ← topSort(G)  // prima visita
    G^T ← transpose(G)  // trasposizione del grafo
    return cc(G^T, S)  // seconda visita
```

Restituisce un vettore di interi che associa ad ogni nodo l'id della sua componente fortemente connessa. Applicando l'algoritmo di ordinamento topologico *su un grafo generale*, siamo sicuri che:

- se un arco (u, v) non appartiene ad un ciclo, allora u viene lista prima di v nella sequenza ordinata;
- gli archi di un ciclo vengono listati in qualche ordine (che è ininfluente).

Utilizziamo quindi la procedura topSort per ottenere i nodi in ordine decrescente di tempo di fine.

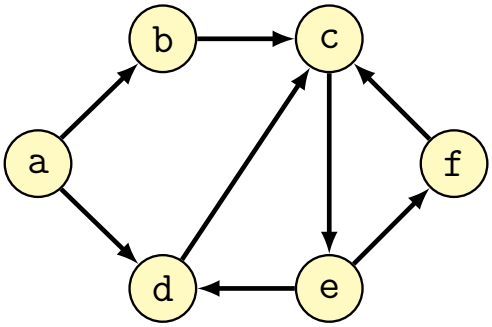
Nella topSort non calcoliamo nemmeno i tempi di fine, li utilizziamo semplicemente per dimostrare che i nodi vengono ordinati in ordine inverso di tempo di fine.

Definizione 9.5.3 (Grafo trasposto)**.** Dato un grafo orientato $G = (V, E)$, il grafo trasposto $G_t = (V, E_T)$ ha gli stessi nodi e gli archi orientati in senso opposto: $E_T = \{(u, v) \mid (v, u) \in E\}$

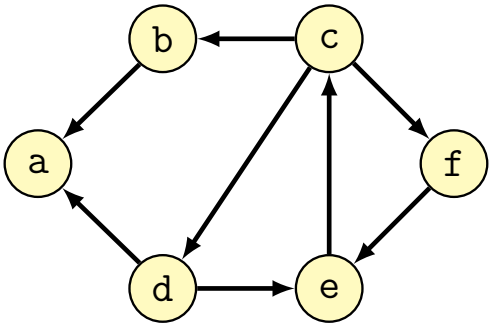
Algoritmo 9.5.2: Calcolo del grafo trasposto

```
// restituisce il grafo transposto
int[ ] transpose(GRAPH G)
    GRAPH G^T ← Graph  // creo il grafo
    foreach u ∈ G.V do
        G^T.insertNode(u)  // aggiungo gli stessi nodi
    foreach u ∈ G.V do
        foreach v ∈ G.adj(u) do
            G^T.insertEdge(v, u)  // li aggiungo in ordine inverso
    // restituisco il grafo trasposto
    return G^T
```

Complessità Il costo computazionale totale ammonta a $\mathcal{O}(m+n)$, in quanto aggiungere i nodi costa $\mathcal{O}(n)$, gli archi $\mathcal{O}(m)$ ed ogni operazione costa $\mathcal{O}(1)$.



(a) Grafo originale



(b) Grafo trasposto

Algoritmo 9.5.3: Identificazione delle componenti connesse alternativa

```
// parte iterativa
int[ ] cc(GRAPH G, STACK S)
    // creo un vettore della dimensione dei nodi del grafo
    int[ ] id ← new int[1 ... G.size]
    // inizializzo il vettore
    foreach u ∈ G.V do
        id[u] ← 0
    // contatore delle componenti connesse
    int counter ← 0
    while not S.isEmpty do // fintanto che la pila non è vuota
        u ← S.pop
        if id[u] == 0 then // ho trovato una nuova componente connessa
            counter++ // aggiorno il contatore
            // effettuo una chiamata ricorsiva sul nodo scoperto
            ccdfs(G, counter, u, id)
    // restituisco l'identificativo della componente connessa
    return id
// visita ricorsiva di ciascuna componente
ccdfs(GRAPH G, int counter, NODE u, int[ ] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u:       il nodo che sto visitando
    // id:      l'identificativo della componente
    // memorizzo l'identificativo della cc
    id[v] ← counter
    foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
        if id[v] == 0 then // non è ancora stato visitato
            // v:  il nodo su cui vado ad operare
            ccdfs(G, counter, v, id) // lo visito ricorsivamente
```

Invece di esaminare i nodi in ordine arbitrario, questa versione di cc li esamina nell'ordine LIFO memorizzato nello stack.

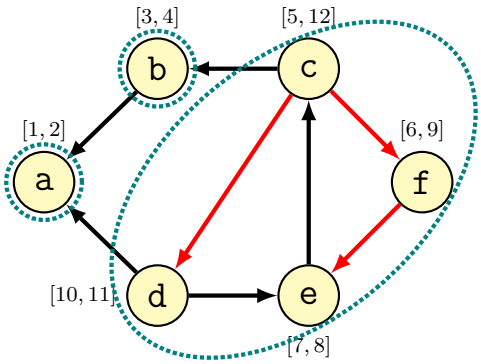


Figure 9.26: Identificazione delle componenti fortemente connesse;
l'ordine in cui li visito è quello della pila `{ a, b, c, e, d, f }`

Algoritmo 9.5.4: Identificazione delle componenti fortemente connesse

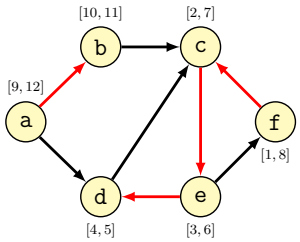
---

```
// indentifica le componenti fortemente connesse
int[ ] scc(GRAPH G)
    STACK S ← topSort(G)  // prima visita
    G^T ← transpose(G)  // trasposizione del grafo
    return cc(G^T, S)  // seconda visita
```

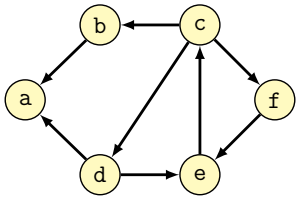
Complessità Ognuna delle fasi che compongono l'algoritmo:

1. visita in profondità della topSort;
2. la trasposizione del grafo di transpose;
3. la visita delle componenti connesse.

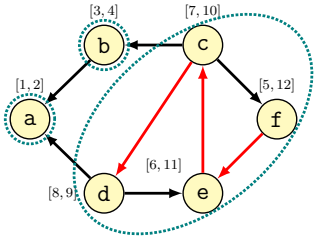
richiede un costo di $\mathcal{O}(m+n)$. Quindi la complessità è lineare nel numero di nodi e nel numero di archi, ossia $\mathcal{O}(m+n)$.



(a) La prima visita parte da f, la seconda dal nodo a



(b) Calcolo il grafo trasposto G^T



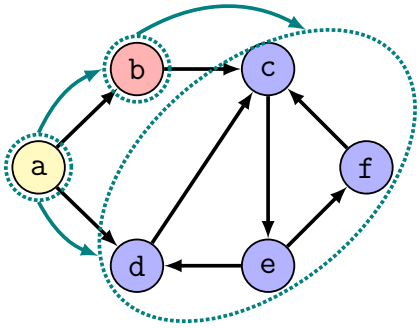
(c) Identificazione delle componenti fortemente connesse

Figure 9.27: Una seconda esecuzione dell'ordinamento topologico

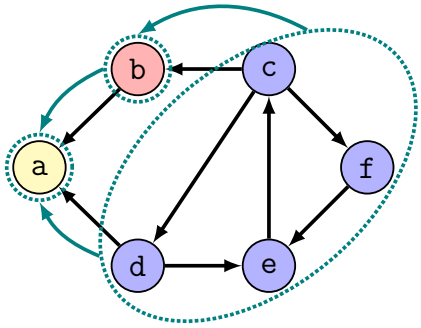
9.5.2 Dimostrazione di correttezza

Definizione 9.5.4 (Grafo delle componenti)**.** Il grafo delle componenti si definisce come il grafo $C(G) = (V_c, E_c)$, dove:

- $V_c = \{C_1, C_2, \ldots, C_k\}$, dove C_i è la i-esima componente fortemente connessa del grafo G;
- $E_c = \{(C_i, C_j) \mid \exists (u_i, v_i) \in E : u_i \in C_i \wedge u_j \in C_j\}$



(a) Grafo delle componenti del grafo



(b) Grafo delle componenti del grafo trasposto

Figure 9.28

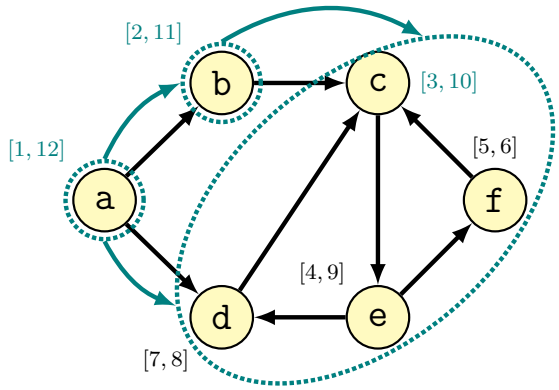
Quando si traspone un grafo fortemente connesso, l'insieme di nodi che compongono il grafo delle componenti connesse rimane lo stesso, mentre gli archi sono in direzione inversa.

Nota. Il grafo delle componenti è aciclico poiché se contenesse un ciclo, il ciclo stesso sarebbe una più grande componente fortemente connessa, il che è assurdo.

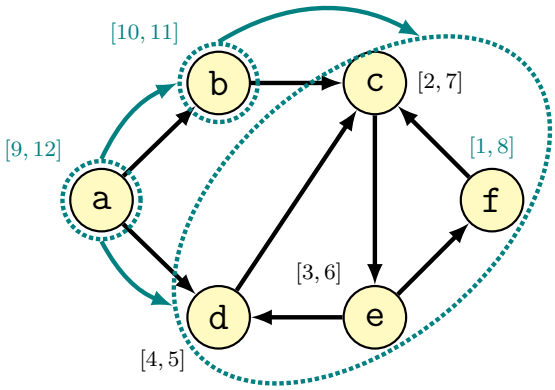
Se il grafo delle componenti è aciclico, allora posso farne un'ordinamento topologico. Possiamo definire quindi $dt(C) = \min\{dt(u) \mid u \in C\}$ e $ft(C) = \max\{ft(u) \mid u \in C\}$, i quali corrisponderanno ai tempo di inizio e di fine del primo nodo visitato in C.

Teorema. *Siano C e C' due distinte componenti fortemente connesse nel grafo orientato $G = (V, E)$. Se c'è un arco $(C, C') \in E_c$, allora $ft(C) > ft(C')$.*

La componente che finisce la visita per ultima è la componente dalla quale si possono raggiungere le altre componenti.



(a) Grafo delle componenti del grafo



(b) Grafo delle componenti del grafo trasposto

Corollario. Siano C_u e C_v due componenti fortemente connesse distinte del grafo orientato $G = (V, E)$. Se c'è un arco $(u, v) \in E_t$ tale che $u \in C_u$ e $v \in C_v$, allora $ft(C_u) < ft(C_v)$.

In generale:

$$(u, v) \in E_t \Rightarrow (v, u) \in E \Rightarrow (C_v, C_u) \in E_c \Rightarrow ft(C_v) > ft(C_u) \Rightarrow ft(C_u) < ft(C_v)$$

Nel nostro caso:

$$(b,a) \in E_t \Rightarrow (a,b) \in E \Rightarrow (C_a, C_b) \in E_c \Rightarrow \underset{12}{ft(C_a)} > \underset{11}{ft(C_b)} \Rightarrow \underset{11}{ft(C_b)} < \underset{12}{ft(C_a)}$$

Se la componente C_u e la componente C_v sono connesse da un arco $(u,v) \in E_t$, allora possiamo dedurne che $ft(C_u) < ft(C_v)$ (dal corollario) e che la visita di C_v inizierà prima della visita di C_u (dal teorema). Non esistendo cammini fra C_v e C_u in G_t (altrimenti il grafo sarebbe ciclico) la visita di C_v non raggiungerà C_u. In altre parole, l'algoritmo cc assegnerà correttamente gli indentificatori delle componenti ai nodi.

Algoritmo di Tarjan

L'algoritmo di Tarjan è preferito a quello di Kosaraju il quale, avendo comunque la medesima complessità computazionale ($\mathcal{O}(m+n)$), è preferito a quest'ultimo in quanto necessita di una sola visita e non richiede la trasposizione del grafo (al posto di una doppia visita e di memoria aggiuntiva).

Applicazioni

Gli algoritmi sulle componenti fortemente connesse possono essere utilizzati per risolvere il problema "2-satisfiability" (2-SAT), un problema di soddisfacibilità booleana con clausole composte da coppie di letterali.

Capitolo 12

Divide et Impera

12.1 Risoluzione di problemi

Dato un problema non esistono "ricette originali" per risolverlo in modo efficiente; tuttavia è possibile evidenziare quattro fasi:

1. **classificazione del problema**: è il primo passo verso la risoluzione;
2. **caratterizzazione della soluzione**: bisogna caratterizzare matematicamente la soluzione, evitando di escludere soluzioni banali;
3. **tecnica di progetto**: quando è possibile dividere il problema in più sottoproblemi di complessità minore allora la tecnica "divide et impera" potrebbe essere quella più appropriata (più avanti vedremo delle tecniche più interessanti quali: programmazione dinamica (Capitolo 13), algoritmi ingordi (Capitolo 14) e backtrack (Capitolo 16));
4. **utilizzo di strutture dati**: bisogna scegliere la struttura dati più adatta alla risoluzione del nostro particolare problema (spesso sarà una tabella hash o un albero binario di ricerca, più avanti vedremo delle strutture dati specializzate per risolvere problemi specifici, a differenza di quelle che abbiamo visto fin'ora che sono generiche).

Queste fasi non sono necessariamente sequenziali, l'ordine dipende da come stiamo affrontando il problema.

12.1.1 Classificazione dei problemi

Ma come possiamo classificare un problema? Le classi di problemi che affronteremo possono essere raggruppate in quattro macro-categorie:

- **problemi decisionali**: consistono nel determinare se il dato in ingresso soddisfa o meno una certa proprietà ed hanno una risposta binaria (si/no, true/false); come ad esempio stabilire se un grafo risulta connesso o meno. Su questo genere di problemi spesso non esistono delle tecniche standard e bisogna creare algoritmi ad-hoc;
- **problemi di ricerca**: consistono nel trovare nello spazio di soluzioni possibili una soluzione ammissibile che rispetti certi vincoli, come ad esempio la ricerca della posizione di una sottostringa in una stringa. In questi problemi la tecnica "divide et impera" può rincorrere in nostro aiuto;
- **problemi di ottimizzazione**: ad ogni soluzione è associata una funzione di costo e vogliamo trovare quella di costo minimo, come ad esempio il cammino (pesato) più breve fra due nodi. Questa classe di problemi può essere risolta tramite la programmazione dinamica o algoritmi ingordi;
- **problemi di approssimazione**: a volte, trovare la soluzione ottima è computazionalmente impossibile e ci si accontenta di una soluzione approssimata, in questo caso il costo rimane basso ma non sappiamo se è ottimale; un esempio di questo genere di problemi è quello del commesso viaggiatore.

12.1.2 Caratterizzazione della soluzione

È fondamentale definire bene il problema dal punto di vista matematico. La formulazione del problema può suggerire una prima idea, seppur banale, alla risoluzione dello stesso. Lo si può osservare nella formulazione del seguente problema: data una sequenza di n elementi, una permutazione ordinata è data dal minimo seguito da una permutazione ordinata dei restanti $n-1$ elementi. Questa formulazione produce l'algoritmo selectionSort. La definizione matematica può suggerire una possibile tecnica, ad esempio:

- se troveremo una *sottostruttura ottima* allora potremmo applicare la programmazione dinamica (Capitolo 13);
- se troveremo la *proprietà greedy* allora potremmo applicare un algoritmo ingordo (Capitolo 14).

Tecniche di soluzione dei problemi

Come vengono affrontati i problemi dalle varie tecniche?

- nella tecnica divide-et-impera un problema viene suddiviso in sotto-problemi indipendenti, i quali vengono risolti ricorsivamente (avendo quindi un approccio dall'alto verso il basso, detto *top-down*); Abbiamo già visto diversi esempi dell'applicazione di questa tecnica, provate a pensare all'algoritmo mergeSort: ordinare due sottovettori sono due problemi indipendenti (ordinare il sottovettore di sinistra non richiede conoscere il contenuto del vettore di destra e viceversa);
- nella programmazione dinamica la soluzione viene costruita (dal basso verso l'altro, *bottom-up*) a partire da un insieme di sotto-problemi potenzialmente ripetuti.
- la tecnica della *memoization* (annotazione) è la versione *top-down* della programmazione dinamica.
- la tecnica *greedy* effettua sempre la scelta localmente ottima (necessita di una dimostrazione).
- il backtrack procede per "tentativi", tornando ogni tanto sui suoi passi;
- nella ricerca locale la soluzione ottima viene trovata "migliorando" via via soluzioni esistenti; Negli algoritmi probabilistici si dimostra che talvolta è meglio scegliere casualmente, ma in modo "gratuito", che con giudizio, ma in maniera costosa.

12.2 La tecnica del Dividi-et-Impera

La tecnica del Divide-et-Impera si suddivide in tre fasi principali:

- **Divide**: divide il problema in sotto-problemi più piccoli è indipendenti;
- **Impera**: risove i sottoproblemi ricorsivamente;
- (**Combina**): "unisce" le soluzioni dei sottoproblemi.

Sfortunatamente non esiste una ricetta unica per applicare questa tecnica: ad esempio l'algoritmo mergeSort ha una fase "divide" banale (basta calcolare il valore mediano) ma, allo stesso tempo una fase di unione delle soluzioni complessa, diversamente nel quickSort la fase "divide" è complessa ma non esiste una fase "combina". È quindi necessario fare uno sforzo creativo, in quanto la tecnica ci dà una modalità con cui ad arrivare alla soluzione, ma bisogna applicarla caso per caso.

Minimo divide-et-impera

La tecnica "divide-et-impera" non è un proiettile d'argento, a volte utilizzarla crea più danni di quanti ne risolva. Osserva questo esempio nella quale è presentato un algoritmo di ricerca del minimo con questa tecnica.

Algoritmo 12.2.1: Algoritmo di ricerca del minimo con tecnica divide-et-impera

```
minrec(int[ ] A, int i, int j)
    if i == j then
        return A[i]
    else
        m ← ⌊(i+j)/2⌋
        return min(minrec(A, i, m), minrec(A, m + 1, j))
```

Complessità L'algoritmo divide il vettore a metà, cerca il minimo nella metà di sinistra e nella metà di destra, il risultato è il minimo dei due minimi.

$$T = \begin{cases} 2T(n/2) + 1 & n > 1 \\ 1 & n = 1 \end{cases}$$

La complessità ammonta a $\alpha = 1, \beta = 0, T(n) = \Theta(n)$, non ne vale la pena, tanto vale fare la ricerca del minimo come abbiamo spiegato a lezione.

12.3 La torre di Hanoi

La torre di Hanoi è un gioco matematico che prevede tre pioli e n dischi di dimensioni diverse. Inizialmente i dischi sono impilati in ordine decrescente nel piolo di sinistra. Lo scopo del gioco è quello di impilare i dischi sul piolo di destra, senza mai impilare un disco più grande su uno più piccolo, muovendo al massimo un disco alla volta ed utilizzando il piolo centrale come appoggio. Questo problema può essere risolto tramite la tecnica "divide-et-impera".

Algoritmo 12.3.1: Versione ricorsiva della soluzione al problema della torre di Hanoi

```
hanoi(int n, int src, int dest, int middle)
    if n = 1 then
        stampa src → dest
    else
        hanoi(n − 1, src, middle, dest)  // sposta n − 1 dischi da src a middle
        stampa src → dest  // sposta 1 disco da src a dest
        hanoi(n − 1, middle, dest, src)  // sposta n − 1 dischi da middle a dest
```

Nella prima parte l'algoritmo sposta $n-1$ dischi da *src* a *middle* utilizzando *dest* come punto d'appoggio. Dopodiché sposta l'ultimo disco rimanente dalla *src* alla *dest*. Infine sposta $n-1$ dischi da *src* a *dest* utilizzando *src* come punto d'appoggio.

Complessità L'equazione di ricorrenza prodotta da questo algoritmo è $T = 2T(n-1) + 1 = \Theta(2^n)$. Si può dimostrare che questa soluzione è ottima (non si può fare meglio di così).

12.4 Algoritmo di ordinamento

L'algoritmo di ordinamento quickSort è basato sulla tecnica "divide et impera", nel caso medio ha una complessità di $\mathcal{O}(n \log n)$, mentre nel caso pessimo è di $\mathcal{O}(n^2)$. Fino a qualche anno fa era l'algoritmo di eccellenza per l'ordinamento. Infatti presenta molti aspetti a suo favore:

- il fattore costante del quickSort è migliore di quello del mergeSort;
- non utilizza memoria addizionale in quanto svolge i calcoli "in-memory" (a differenza di mergeSort che ha bisogno di un vettore di appoggio);
- esistono delle tecniche "euristiche" per evitare il caso pessimo.

Quindi spesso è preferito ad altri algoritmi. All'interno dell'ultimo capitolo riassumeremo tutti gli algoritmi di ordinamento visti fin'ora e ne vedremo di nuovi, tra questi anche gli algoritmi attualmente utilizzati negli attuali linguaggi di programmazione (c, java, python).

Spiegazione Sono dati in input un vettore $A[1 \dots n]$, gli indici *start*, *end* tali che $1 \leqslant start \leqslant end \leqslant n$, tali indici indicano quale parte del vettore stiamo ordinando, come avviene in mergeSort.

1. la parte del "divide" avviene nel seguente modo:

- scegliamo un valore $p \in A[start \dots end]$ detto perno (*pivot*);
- spostiamo gli elementi del vettore $A[start \dots end]$ in modo tale che:
  - $\forall i \in [start \dots j-1] : A[i] \leqslant p$;
  - $\forall i \in [j+1 \dots end] : A[i] \geqslant p$

  l'indice j viene calcolato per rispettare tale condizione;
- il perno viene messo in posizione $A[j]$.

2. la parte "impera" ordina i due sottovettori $A[start \dots j-1]$ e $A[j+1 \dots end]$ richiamando ricorsivamente quickSort;
3. la parte "combina" non fa nulla.

Algoritmo 12.4.1: quickSort

```
quickSort(ITEM[] A, int primo, int ultimo)
    // su almeno due elementi
    if primo < ultimo then
        int j ← perno(A, primo, ultimo)  // logica dell'algoritmo
        // richiamo l'algoritmo su entrambi i sottovettori
        quickSort(A, primo, j − 1)
        quickSort(A, j + 1, ultimo)
```

Algoritmo 12.4.2: perno

```
// sposta gli elementi più piccoli a sinistra del perno, i più grandi a destra
int perno(ITEM[] A, int primo, int ultimo)
    ITEM x ← A[primo] // il perno è il primo elemento
    int j ← primo // il cursore parte dal primo elemento

    // spostamenti "in-place"
    from i ← primo until ultimo do
        if A[i] < x then // l'elemento è più piccolo del perno
            j++ // sposta il cursore j
            swap(A[i], A[j]) // scambia gli elementi: i ↔ j

    /* a questo punto tutti gli elementi posizionati prima della posizione j sono più piccoli del perno,
       rimane solo da riposizionare il perno nella sua posizione finale (è ordinato)                  */

    // riposiziono il perno
    A[primo] ← A[j]
    A[j] ← x

    // restituisco la posizione del perno
    return j
```

Complessità computazionale Il costo della funzione perno è $\Theta(n)$ (deve guardare $n-1$ valori ed effettuare i confronti). Il costo di quickSort dipende dal partizionamento:

- il partizionamento *peggiore* si verifica quando il perno è l'elemento minimo (o massimo), questo particolare caso accade quando il vettore è ordinato in ordine crescente (decrescente). La complessità risultante è $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.
- il partizionamento *migliore* avviene quando il vettore di dimensione n viene diviso in due sottoproblemi di dimensione $n/2$. La complessità risultante è $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$;
- il partizionamento nel *caso medio* è molto più vicino al caso ottimo che al caso peggiore, prendiamo ad esempio il partizionamento 9-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn = \Theta(n \log n)$$

  Prendiamo un altro esempio, il partizionamento 99-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{99n}{10}\right) + cn = \Theta(n \log n)$$

  Nota. In questi esempi, il partizionamento ha proporzionalità limitata e i fattori moltiplicativi possono essere importanti.

  Il costo computazionale dipende dall'ordine degli elementi e non dai loro valori. Dobbiamo quindi considerare tutte le possibili permutazioni, il che è difficile dal punto di vista analitico. Alcuni partizionamenti saranno parzialmente bilanciati, altri pessimi; in media questi si alterneranno nella sequenza di partizionamenti, ma quelli parzialmente bilanciati "dominano" quelli pessimi.

Moltiplicazione di catena di matrici

Viene fatto un accenno ad un argomento che verrà affrontato in modo più approfondito nel capitolo successivo.

12.5 Conclusioni

La tecnica divide-et-impera viene applicata quando i passi "divide" e "combina" sono semplici e i costi risultano migliori del corrispondente algoritmo iterativo (quindi, ad esempio, va bene per effettuare

l'ordinamento, ma non per effettuare la ricerca del minimo). Ulteriori vantaggi dell'applicazione di questa tecnica sono:

- la facile parallelizzazione: la possibilità di dividere il problema in più sottoproblemi porta ad una naturale divisione dei compiti fra più processori;
- l'utilizzo ottimale della memoria *cache* (*cache oblivious*): tutti i dati con la quale stiamo lavorando sono colocalizzati nella memoria principale.

12.6 Applicazione della tecnica

Infine vediamo una prima applicazione della tecnica e ne valutiamo le prestazioni.

12.6.1 Gap

In un vettore V contenente $n \geqslant 2$ interi, un gap è un indice i, $1 < i \leqslant n$, tale che $V[i-1] < V[i]$.

- Dimostrare che se $n \geqslant 2$ e $V[1] < V[n]$, allora V contiene almeno un gap;
- Progetta un algoritmo che, dato un vettore V contentente $n \geqslant 2$ interi e tale che $V[1] < V[n]$ (la condizione sopra), restituisca la posizione di un gap nel vettore (questo algoritmo assume che il gap esista).

Dimostrazione per assurdo. Suppponiamo che non ci sia un gap nel vettore. Allora $V[1] \geqslant V[2] \geqslant V[3] \geqslant \cdots \geqslant V[n]$, che contraddice il fatto che $V[1] < V[n]$. □

Proviamo a riformulare la proprietà tenendo conto di due indici:

- sia V un vettore di dimensione n;
- siano i, j due indici tali che $1 \leqslant i < j \leqslant n$ e $V[i] < V[j]$.

In altre parole, ci sono più di due elementi nel sottovettore $V[i \dots j]$ e il primo elemento $V[i]$ è più piccolo dell'ultimo elemento $V[j]$.

Dimostrazione per induzione. Voglia provare per induzione sulla dimensione n del sottovettore che il sottovettore contiene un gap.

- **caso base**: $n = j - i + 1 = 2$, ad esempio $j = i + 1$: $V[i] < V[j]$ implica che $V[i] < V[j]$ implica che $V[i] < V[i+1]$, che è un gap;
- **ipotesi induttiva**: dato un qualunque (sotto)vettore $V[h \dots k]$ di dimensione $n' < n$, tale che $V[h] < V[k]$, allora $V[h \dots k]$ contiene un gap;
- **passo induttivo**: consideriamo un qualunque elemento m tale che $i < m < j$. Almeno uno dei due casi seguenti è vero:
  - se $V[m] < V[j]$, allora esiste un gap in $V[m \dots j]$, per ipotesi induttiva;
  - se $V[i] < V[m]$, allora esiste un gap in $V[i \dots m]$, per ipotesi induttiva.

□

Algoritmo 12.6.1: Algoritmo che ricerca un intervallo all'interno del vettore

```
// funzione wrapper
gap(int[ ] V, int n)
    // n:       dimensione del vettore
    return gapRec(V, 1, n)

gapRec(int[ ] V, int i, int j)
    if j == i + 1 then // ho due elementi
        return j  // ritorno il secondo elemento
    m = ⌊(i+j)/2⌋  // calcolo il mediano
    if V[m] < V[j] then
        return gapRec(V, m, j)  // a destra
    else
        return gapRec(V, i, m)  // a sinistra
```

Tabella 12.1: Valutazione delle prestazioni degli algoritmi scritti in python

n	Iterativa (ms)	Ricorsiva (μs)
10^3	0,06	2,05
10^4	0,61	2,78
10^5	6,11	3,36
10^6	62,44	4,01
10^7	621,69	4,87
10^8	6205,72	5,47