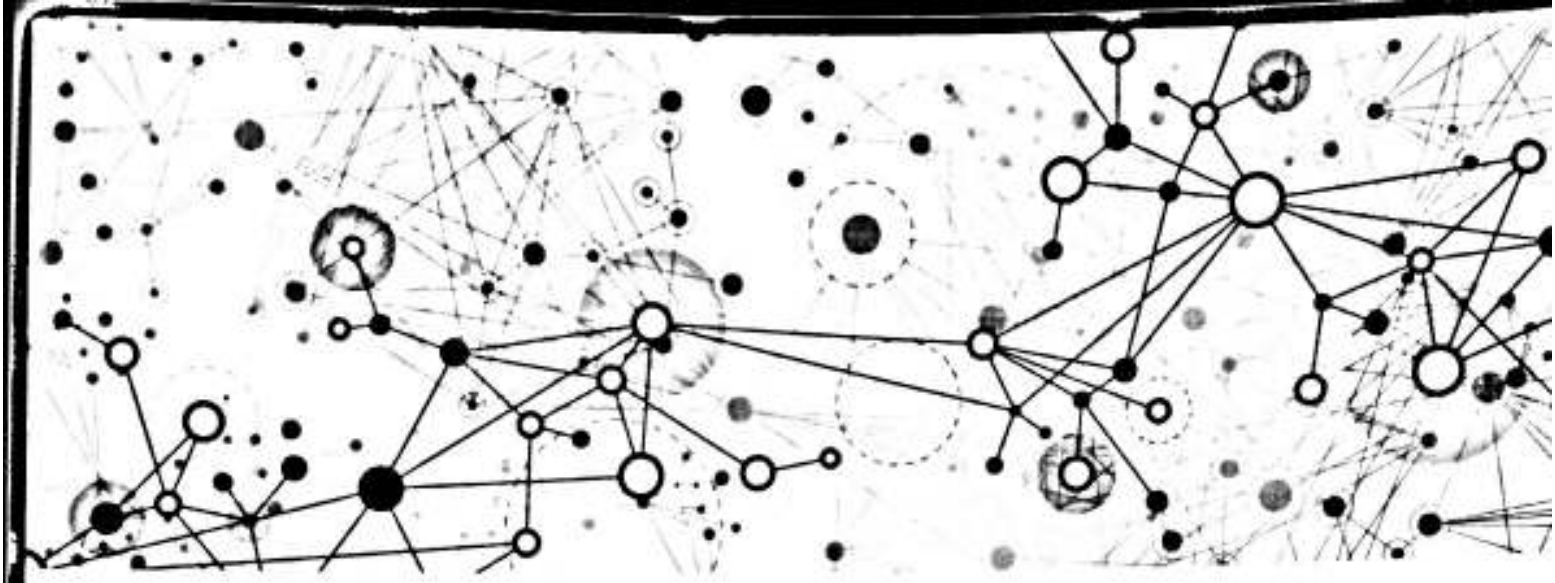


ALAN BERTOSSI ALBERTO MONTRESOR

ALGORITMI E STRUTTURE DI DATI

Terza edizione

CittàStudi
EDIZIONI

Alan Bertossi, Alberto Montresor

Algoritmi e strutture di dati

Terza edizione

CittàStudi
EDIZIONI

CittaStudi
EDIZIONI

www.cittastudi.it

Proprietà letteraria riservata
© 2014 De Agostini Scuola SpA – Novara
1ª edizione: febbraio 2014
Printed in Italy

Tutti i diritti riservati. Nessuna parte del materiale protetto da questo copyright potrà essere riprodotta in alcuna forma senza l'autorizzazione scritta dell'Editore.

Fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, comma 4, della legge 22 aprile 1941 n.633.
Le riproduzioni ad uso differente da quello personale potranno avvenire, per un numero di pagine non superiore al 15% del presente volume/fascicolo, solo a seguito di specifica autorizzazione rilasciata da AIDRO – Corso di Porta Romana 108 – 20122 Milano – e-mail: segreteria@aidro.org; www.aidro.org.

Stampa: Tipografia Gravinese - Torino

Ristampe:	0 1	2 3	4 5	6 7	8 9	10	11
Anno:	2014	2015	2016	2017	2018	2019	2020

INDICE

XI *Prefazione*

3 PARTE PRIMA – Concetto di algoritmo

5 Capitolo 1 – Introduzione

6 1.1 Algoritmi e problemi
7 1.2 Descrizione di algoritmi
1.2.1 Sintassi di base, p. 10 - 1.2.2 Tipi di dato primitivi, p. 11 - 1.2.3 Tipi di dato composti, p. 12 - 1.2.4 Procedure e funzioni, p. 14 - 1.2.5 Ricorsione, p. 15
16 1.3 Valutazione di algoritmi
18 1.4 Progettazione di algoritmi
19 1.5 Reality check

20 Capitolo 2 – Analisi di algoritmi

20 2.1 Tempo di calcolo
24 2.2 Ordine di grandezza e complessità
27 2.3 Algoritmi efficienti e inefficienti
33 2.4 Complessità di problemi e algoritmi ottimi
36 2.5 Tecniche di analisi
36 2.6 Analisi per livelli
37 2.7 Risoluzione di relazioni di ricorrenza "comuni"
41 2.8 Analisi per tentativi
42 2.9 Analisi ammortizzata
45 2.10 Reality check
46 Esercizi
49 Soluzioni

53 PARTE SECONDA – Strutture di dati

55 Capitolo 3 – Tipi di dato e strutture di dati

56 3.1 Strutture di dati: definizione
57 3.2 Strutture di dati: specifica

3.2.1 Sequenze, p. 57 - 3.2.2 Insiemi, p. 60 - 3.2.3 Dizionari, p. 61 - 3.2.4 Alberi e grafi, p. 62 - 3.2.5 Discussione, p. 63

64 3.3 Strutture di dati: realizzazione
65 3.4 Reality check
67 Esercizi
68 Soluzioni

69 Capitolo 4 – Strutture di dati elementari

70 4.1 Liste con puntatori
72 4.2 Realizzazione con vettori
74 4.3 Pile e code

4.3.1 Specifica, p. 75 - 4.3.2 Realizzazione di una pila con vettore, p. 76 - 4.3.3 Realizzazione di una coda con vettore circolare, p. 77

78 4.4 Pile e procedure ricorsive
82 4.5 Reality Check
84 Esercizi
86 Soluzioni

88 Capitolo 5 – Alberi

90 5.1 Specifica
92 5.2 Visite
94 5.3 Realizzazione con puntatori padre/primo-figlio/fratello
95 5.4 Realizzazione con vettore dei padri
97 5.5 Alberi binari
100 5.6 Altre realizzazioni
101 5.7 Reality check
102 Esercizi
104 Soluzioni

107 Capitolo 6 – Alberi bilanciati di ricerca

107 6.1 Alberi binari di ricerca
112 6.2 Alberi Red-Black
123 6.3 Reality check
128 Esercizi
129 Soluzioni

131 Capitolo 7 – Tabelle hash

134 7.1 Funzioni hash
137 7.2 Metodi di scansione
141 7.3 Complessità media
144 7.4 Ristrutturazione
145 7.5 Reality check
146 Esercizi
148 Soluzioni

149 Capitolo 8 – Insiemi e dizionari

149 8.1 Insiemi realizzati con vettori booleani
151 8.2 Insiemi realizzati con liste non ordinate
152 8.3 Insiemi realizzati con liste ordinate
153 8.4 Realizzazione con tabelle hash
154 8.5 Realizzazione con alberi bilanciati
154 8.6 Reality check
155 Esercizi
156 Soluzioni

159 Capitolo 9 – Grafi

159 9.1 Definizioni

9.1.1 Grafi orientati, p. 159 - 9.1.2 Grafi non orientati, p. 160 - 9.1.3 Connessione e connessione forte, p. 161 - 9.1.4 Relazioni fra alberi e grafi, p. 162

164 9.2 Specifica
165 9.3 Realizzazioni con matrici
167 9.4 Realizzazioni con liste di adiacenza
168 9.5 Esplorazione di un grafo

9.5.1 Visita BFS, p. 169 - 9.5.2 Applicazione BFS: il numero di Erdős, p. 170 - 9.5.3 Alberi di copertura BFS, p. 171 - 9.5.4 Visita DFS, p. 172 - 9.5.5 Applicazioni DFS: componenti connesse, p. 172 - 9.5.6 Alberi di copertura DFS, p. 173 - 9.5.7 Applicazione schema DFS: grafo aciclico, p. 175 - 9.5.8 Applicazione schema DFS: componenti fortemente connesse, p. 176 - 9.5.9 Applicazione schema DFS: componenti fortemente connesse, p. 177

179 9.6 Reality check
180 Esercizi
182 Soluzioni

185 Capitolo 10 – Code con priorità e insiemi disgiunti

185 10.1 Code con priorità

10.1.1 Realizzazione con liste, p. 186 - 10.1.2 Realizzazione con alberi bilanciati, p. 187 - 10.1.3 Realizzazione con heap, p. 187 - 10.1.4 Heapsort, p. 192

195 10.2 Unione di insiemi disgiunti (Merge-Find)
10.2.1 Realizzazione basata su liste, p. 196 - 10.2.2 Realizzazione basata su foresta, p. 196 - 10.2.3 Euristica del rango, p. 198 - 10.2.4 Compressione dei percorsi, p. 199 - 10.2.5 Componenti connesse incrementali, p. 200
201 10.3 Reality check
203 Esercizi
205 Soluzioni

207 PARTE TERZA – Tecniche di progettazione

209 Capitolo 11 – Strutture di dati e progettazione di algoritmi
210 11.1 Cammini minimi
212 11.2 Algoritmo prototipo
214 11.3 Algoritmo di Dijkstra
215 11.4 Algoritmo di Johnson
216 11.5 Algoritmo di Fredman-Tarjan
216 11.6 Algoritmo di Bellman-Ford-Moore
218 11.7 Utilizzo di una pila
219 11.8 Algoritmo di Pape-D'Esopo
220 11.9 Reality check
221 Esercizi
223 Soluzioni

225 Capitolo 12 – Divide et impera
226 12.1 Ricerca binaria e ricerca per interpolazione
227 12.2 Quicksort
232 12.3 Moltiplicazione di polinomi
234 12.4 Moltiplicazione di matrici
236 12.5 Reality check
238 Esercizi
240 Soluzioni

247 Capitolo 13 – Programmazione dinamica
249 13.1 String matching approssimato
252 13.2 Insieme indipendente di intervalli pesati
253 13.3 Moltiplicazione catena di matrici
255 13.4 Occupazione di memoria
257 13.5 Cammini minimi tra tutte le coppie
260 13.6 Memoization
262 13.7 Reality check
265 Esercizi
268 Soluzioni

270 Capitolo 14 – Greedy

271 14.1 Zaino reale
273 14.2 Insieme indipendente di intervalli
274 14.3 Minimo albero di copertura
14.3.1 Algoritmo di Kruskal, p. 274 - 14.3.2 Algoritmo di Prim, p. 276 - 14.3.3 Dimostrazione di correttezza, p. 278
278 14.6 Reality check
281 Esercizi
283 Soluzioni

285 Capitolo 15 – Ricerca locale

288 15.1 Shell Sort
291 15.2 Flusso massimo
15.2.1 Algoritmo di Ford-Fulkerson, p. 294 - 15.2.2 Algoritmo di Edmonds-Karp, p. 294 - 15.2.3 Algoritmo dei tre indiani, p. 294
299 15.3 Reality check
300 Esercizi
302 Soluzioni

305 Capitolo 16 – Backtrack

305 16.1 Inviluppo convesso
16.1.1 Algoritmo di Graham, p. 308
311 16.2 Enumerazione tramite backtrack
314 16.3 Reality check
316 Esercizi
317 Soluzioni

319 Capitolo 17 – Algoritmi probabilistici

320 17.1 Selezione
322 17.2 Primalità
325 17.3 Reality check
327 Esercizi
328 Soluzioni

329 PARTE QUARTA – Problemi intrattabili

331 Capitolo 18 – Teoria dell'NP-completezza

334 18.1 Certificati polinomiali
335 18.2 Non determinismo
338 18.3 Enumerazione
339 18.4 Le classi $\mathbb{P}$ ed $\mathbb{NP}$

341 18.5 Riducibilità polinomiale
343 18.6 Teorema di Cook-Levin
344 18.7 Prove di NP-completezza
351 18.8 Reality check
353 Esercizi
355 Soluzioni

357 Capitolo 19 – Tecniche risolutive per problemi intrattabili

357 19.1 Algoritmi pseudo-polinomiali
19.1.1 NP-completezza "forte", p. 359

360 19.2 Algoritmi di approssimazione
19.2.1 Approssimazione assoluta, p. 361 - 19.2.2 Non approssimabilità, p. 363
19.2.3 ε-approssimazione, p. 366

368 19.3 Algoritmi branch-&-bound
19.3.1 Commesso viaggiatore, p. 370

373 19.4 Algoritmi euristici
19.4.1 Greedy, p. 373 - 19.4.2 Ricerca locale, p. 375

377 19.5 Reality Check
379 Esercizi
382 Soluzioni

385 *Conclusione*
389 *Bibliografia*
397 *Indice analitico*

PREFAZIONE

L'*algoritmica* è quella branca dell'informatica che riguarda la definizione e la progettazione degli algoritmi, l'analisi della loro correttezza ed efficienza, la dimostrazione delle loro inerenti limitazioni e complessità, lo studio delle strutture di dati da essi elaborati e la definizione di nuovi modelli di calcolo.

Il termine *algoritmo* indica la descrizione precisa delle azioni che un esecutore deve compiere per giungere alla soluzione di un problema computazionale di qualsiasi genere. Gli algoritmi vengono comunemente descritti tramite programmi, che si avvalgono di costrutti e istruzioni tipici dei linguaggi di programmazione e che devono essere eseguiti da macchine (o calcolatori elettronici). Le proprietà degli algoritmi, però, sono talmente "fondamentali", "generali" e "robuste" da essere virtualmente indipendenti dalle caratteristiche di specifici linguaggi di programmazione o di particolari calcolatori elettronici.

Ovunque si impieghi un calcolatore elettronico occorrono algoritmi corretti ed efficienti per utilizzarlo al massimo delle sue potenzialità. Per esempio, algoritmi sofisticati e veloci sono usati per controllare il volo di aerei, regolare reazioni nucleari, reperire informazioni in archivi di dati, risolvere problemi algebrici e matematici, pagare stipendi a impiegati di aziende, giocare con i videogiochi, smistare comunicazioni telefoniche, controllare la produzione di catene di montaggio robotizzate, progettare nuovi calcolatori elettronici, riconoscere immagini di oggetti tridimensionali, elaborare testi letterari, comporre musica.

L'algoritmica è molto vasta e riguarda essenzialmente le seguenti aree:

- studio delle *caratteristiche dei dati* elaborati dagli algoritmi per risolvere certe classi di problemi e del modo più opportuno di organizzare e strutturare tali dati al fine di agevolarne l'uso da parte degli algoritmi che li elaborano;
- studio di *metodologie per la progettazione* di algoritmi corretti (cioè che risolvono sempre e solo certe classi di problemi cui si è interessati) ed efficienti (cioè che li risolvono il più velocemente possibile);

– studio delle *inerenti limitazioni* dei problemi da risolvere. Molti problemi ben definiti e di indubbia utilità pratica non potranno mai essere risolti da alcun algoritmo in un lasso di tempo "ragionevole" (p.e. la vita di un essere umano), mentre altri problemi non potranno mai essere risolti neanche in un lasso di tempo "irragionevole". Occorre allora individuare metodologie per affrontare tali problemi inerentemente intrattabili.

Il presente testo è suddiviso in quattro parti principali: dopo una prima introduzione al concetto di algoritmo, si tratterà di strutture di dati, progettazione di algoritmi e problemi intrattabili, che corrispondono alle suddette aree dell'algoritmica.

Nella prima parte verranno introdotti i concetti di problema computazionale e di algoritmo che lo risolve; si presenterà lo pseudocodice utilizzato per descrivere gli algoritmi e si discuteranno le principali tecniche utilizzate per valutare l'efficienza degli algoritmi proposti. Nella seconda parte saranno trattati metodi di specifica e realizzazione di strutture di dati, cioè particolari tipi di dato formati da aggregati di oggetti elementari (e non) che hanno rilevante interesse in informatica (ad esempio, sono strutture di dati i vettori, le liste e gli insiemi). Si vedrà come la stessa struttura di dati possa essere rappresentata in vari modi che influenzano direttamente l'efficienza e la complessità delle operazioni ammesse sui componenti della struttura. Sarà anche illustrato come l'esecuzione di procedure ricorsive possa essere gestita agevolmente utilizzando una particolare struttura, nota col nome di pila.

Nella parte seguente sarà dapprima data una classificazione dei problemi computazionali più comuni nei quali ci si imbatte nella pratica. Saranno successivamente trattate tecniche per la progettazione di algoritmi efficienti per risolvere problemi appartenenti alle suddette classi, quali il *divide-et-impera*, il *backtrack*, la programmazione dinamica, il *greedy* e la ricerca locale. Sarà anche mostrato come la scelta di una struttura di dati, rispetto ad un'altra, influenzi direttamente l'efficienza di un algoritmo che la usi.

Nella quarta e ultima parte si introdurranno classi di problemi "intrattabili" di rilevante interesse pratico. Tali problemi, detti NP-completi, sono tra loro computazionalmente correlati: per essi sono conosciuti soltanto algoritmi del tipo "prova tutte le possibilità" e si dubita fortemente che anche uno solo di essi sia risolubile con un algoritmo polinomiale! Si vedranno tecniche per trattare problemi di questa classe, quali algoritmi pseudopolinomiali, l'approssimazione, il *branch-&-bound* e le tecniche euristiche. Si accennerà infine anche all'esistenza di classi di problemi ancora più difficili, per i quali è possibile dimostrare limitazioni inferiori del tempo di calcolo che sono esponenziali.

Particolare enfasi è posta sui due momenti successivi di astrazione e realizzazione di strutture di dati e sulla trattazione delle metodologie di progettazione di algoritmi. Da un lato, si è seguito l'approccio classico che prevede l'enunciazione del problema computazionale, la formalizzazione della sua soluzione, l'individuazione delle operazioni richieste per trovare tale soluzione, l'applicazione di opportune tecniche di progettazione di algoritmi risolutivi, e l'analisi di complessità degli algoritmi proposti. Dall'altro lato, si è cercato di integrare i problemi astratti e gli algoritmi teorici con applicazioni concrete tratte dal mondo reale informatico: ciò avviene in appositi paragrafi denominati *Reality check* che trattano svariati argomenti, come Java Collections, file system, simulazione ad

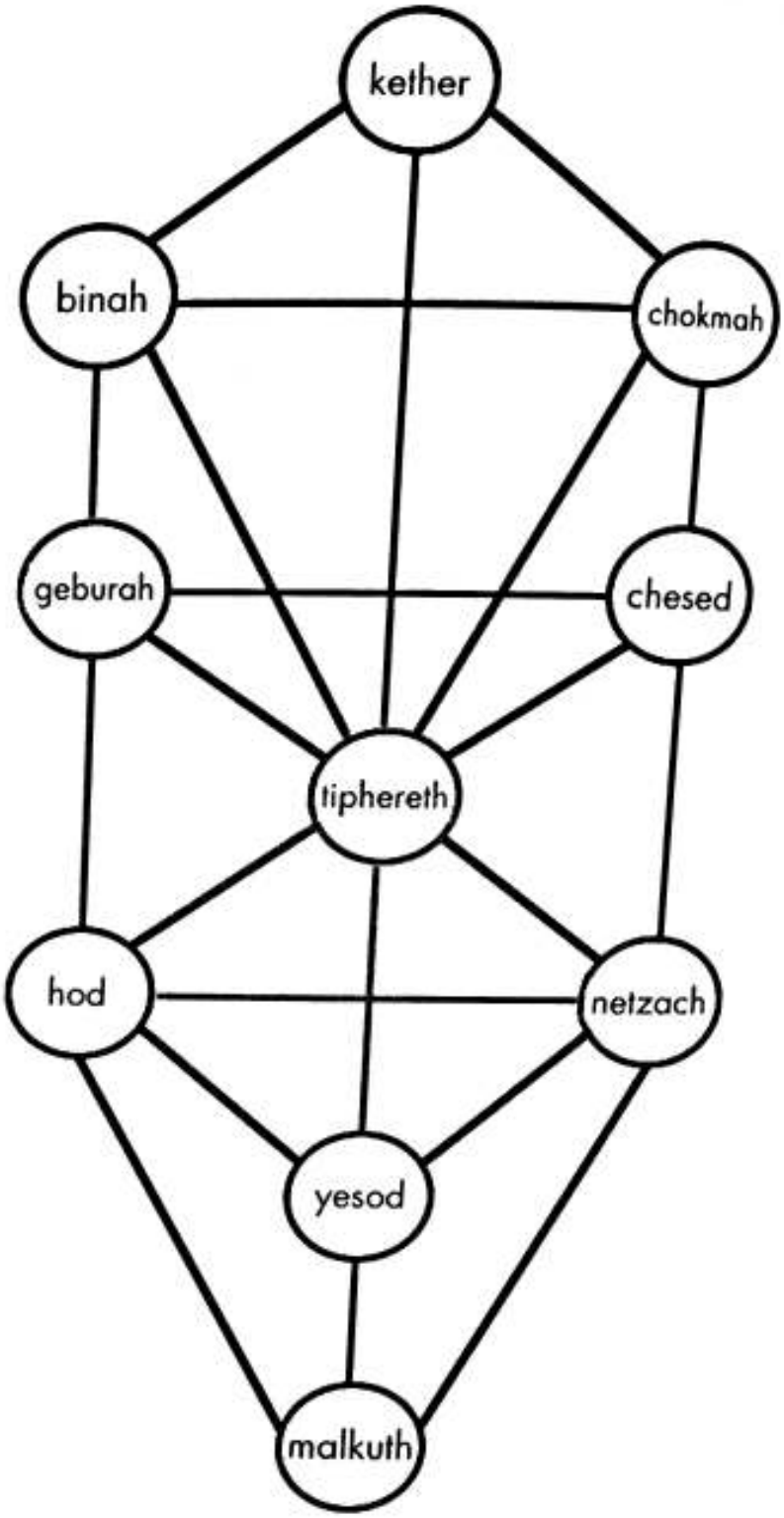
eventi, protocolli di *routing* per Internet, comandi del sistema operativo Unix, codifiche compresse di dati, sistemi crittografici a chiave pubblica, e giochi come il Sudoku e il quindici.

Gli algoritmi presentati sono descritti in uno pseudocodice orientato ad oggetti, ma indipendente da un particolare linguaggio di programmazione. I costrutti dello pseudocodice sono comunque facilmente traducibili in linee di codice di un particolare linguaggio come il C++ o il Java. Per ben capire lo pseudocodice ed eventualmente tradurlo in codice eseguibile è sufficiente che il lettore abbia un po' di dimestichezza con i linguaggi di programmazione (acquisita in un corso come "Programmazione" o in un altro ad esso equivalente) ed in particolare con i puntatori e le strutture di dati elementari come i vettori. Invece, per la comprensione delle dimostrazioni di correttezza e di complessità, è opportuno che il lettore abbia familiarità con dimostrazioni per assurdo e per induzione e con le basi del calcolo combinatorio (acquisite in corsi matematici come "Analisi matematica" e "Matematica discreta", o altri ad essi equivalenti).

Il corso, eventualmente sfrondato di alcune parti più specialistiche (come gli alberi rosso-neri, i cammini minimi, la moltiplicazione di catene di matrici, il flusso massimo e il *branch-&-bound*), può essere agevolmente insegnato in un corso di "Algoritmi e strutture di dati" di 12 crediti formativi (circa 60 ore di lezione) al primo o al secondo anno della laurea triennale in Informatica, eventualmente diviso in due moduli di 6 crediti ciascuno, comprendenti rispettivamente le strutture di dati (Parti I e II) e le tecniche algoritmiche (Parti III e IV). Sempre con le stesse sfrondature, le Parti I-III possono essere insegnate in un modulo di 9 crediti, lasciando gli eventuali argomenti tagliati in precedenza e la Parte IV ad un modulo di 6 crediti da insegnare al terzo anno della laurea triennale oppure al primo anno della successiva laurea magistrale.

Nel testo sono proposti più di 200 esercizi (per la maggior parte tema d'esame), dei quali oltre 95 sono completamente svolti, mentre per molti altri sono forniti opportuni suggerimenti. Prima di seguire i suggerimenti o di guardare gli svolgimenti, però, è bene che il lettore provi a risolvere gli esercizi per conto proprio.

Gli autori desiderano ringraziare quanti hanno letto e commentato porzioni preliminari del presente testo, suggerendo modifiche e miglioramenti, proponendo nuovi esercizi ed indicando riferimenti bibliografici aggiornati. In particolare, gli autori sono riconoscenti a Roberto Battiti, Maurizio Bonuccelli, Ferruccio Damiani, Lorenzo Donatiello, Floriana Esposito, Moreno Falaschi, Paolo Ferragina, Elio Giovannetti, Roberto Grossi, Donato Malerba, Luigi Mancini, Andrea Masini, Alessandro Mei, Carlo Montangero, Linda Pagli, Stefano Pallottino, Cristina Pinotti, Geppino Pucci, Romeo Rizzi, Davide Rossi e Maddalena Zacchi. Gli autori desiderano altresì ricordare i numerosi studenti delle Università di Pisa, Bologna e Trento che hanno fornito molte soluzioni originali agli esercizi proposti, alcune delle quali sono riportate nel testo. In ogni caso, la responsabilità di eventuali errori resta degli autori, che saranno riconoscenti a chiunque glieli segnali.



Scacciò l'uomo e pose ad oriente del giardino di Eden
i cherubini e la fiamma della spada sfolgorante,
per custodire la via all'albero della vita.
GENESI 3,24

CONCETTO DI ALGORITMO

Parte prima

INTRODUZIONE

Capitolo 1

Il termine *algoritmo*, che significa "procedimento di calcolo", è una versione moderna del termine latino medievale *algorismus*, che deriva dal nome del matematico usbeco Mohammed ibn-Musa al-Khwārizmī, vissuto nel IX secolo d.C. e dalle cui opere è nata, fra l'altro, la moderna algebra. Tale termine indica la descrizione precisa di una sequenza di azioni che un esecutore deve compiere per giungere alla soluzione di un qualsiasi problema computazionale. Per poter risolvere il problema, l'algoritmo ha bisogno di rappresentare, organizzare ed elaborare l'ingresso, l'uscita ed altre informazioni intermedie utili al calcolo. Questo è possibile manipolando oggetti elementari, detti *dati*. In pratica, algoritmi e dati sono inscindibilmente legati. L'algoritmo può essere considerato come un elaboratore di dati che, a fronte di certi dati d'ingresso consistenti con la natura del problema da risolvere (*dati di input*), li rielabora fino a produrre altri dati come risultato del problema (*dati di output*).

ESEMPIO – [Ricetta gastronomica]

Una cucina di un ristorante è provvista di ingredienti, pentole, fornelli e altri utensili vari. Per cucinare un buon minestrone alla toscana, il cuoco esegue delle azioni specificate da una ricetta. Eccone una per un numero arbitrario p di persone:

Ingredienti: $75p$ grammi di fagioli, $50p$ grammi di pasta, p patate, $50p$ grammi di cavolo, $50p$ grammi di cardi, $40p$ grammi di pomodori maturi, $p/4$ mazzetti di basilico, $p/4$ cipolle, p pizzichi di sale e pepe, p cucchiai d'olio e p cucchiai di formaggio grana grattugiato.

Mettere in una pentola l'olio, le cipolle tritate e il basilico, lasciare dorare e unire tutte le verdure tagliate a dadini, mescolare, unire p cucchiai d'acqua dei fagioli, precedentemente cotti e passati al setaccio. Aggiungere la purea di fagioli, sale, pepe e continuare la cottura. Dopo un'oretta aggiungere la pasta e lasciarla cuocere per circa sei minuti. Servire con il formaggio grattugiato e, a piacere, con un cucchiaio di olio di oliva di frantoio.

Cucinare il minestrone è un procedimento che viene eseguito dal cuoco, a partire dagli ingredienti, con l'aiuto di pentole, fornelli e altri utensili, in accordo alla ricetta. Gli ingredienti sono i dati di input, il minestrone è l'output, la ricetta è l'algoritmo e il cuoco è l'esecutore (o elaboratore, o calcolatore). Volendo spingere oltre l'analogia tra la cucina di un ristorante e un moderno sistema di calcolo, si può dire che le ricette rappresentano il software, mentre il cuoco, i fornelli, le pentole e gli altri utensili rappresentano l'hardware.

1.1 ALGORITMI E PROBLEMI

Esiste una distinzione precisa fra il *problema computazionale* che si vuole risolvere e gli algoritmi che lo risolvono. Mentre l'enunciato del problema specifica quale relazione si desideri tra l'ingresso e l'uscita (cioè il risultato), l'algoritmo descrive la sequenza di azioni da eseguire per ottenere l'uscita desiderata a partire dall'ingresso. Per meglio comprendere la differenza, consideriamo due problemi di esempio:

MINIMO Il minimo di un insieme A è l'elemento di A che è minore o uguale ad ogni elemento di A: $min(A) = a \Leftrightarrow \forall b \in A: a \leq b$

RICERCA Sia $A = a_1,\ldots,a_n$ una sequenza di dati ordinati e distinti, $a_1 < a_2 < \ldots < a_n$. Eseguire una ricerca della posizione di un dato v nella sequenza A consiste nel restituire l'indice corrispondente, se v è presente, oppure 0, se v non è presente.

$$ricerca(A,v) = \begin{cases} i & \exists i \in \{1,\ldots,n\}: a_i = v \\ 0 & \text{altrimenti} \end{cases}$$

Non abbiamo specificato che tipo di valori siano inclusi nell'insieme; potrebbero essere interi o reali, ad esempio. Per i nostri scopi, è sufficiente sapere che esiste una relazione di ordinamento totale "$\leq$" su questi valori.

Così come sono, queste definizioni matematiche non rappresentano certo un insieme di istruzioni che possano essere eseguite (da un calcolatore o da un essere umano) per risolvere il problema. Facciamo quindi un primo tentativo, cercando di risolvere i due problemi proposti. Con un po' di fantasia possiamo trarre ispirazione dalle definizioni stesse, inventando i seguenti "algoritmi":

- per trovare il minimo di un insieme, confronta ogni elemento con tutti gli altri: l'elemento che è minore di tutti è il minimo;
- per trovare un valore v nella sequenza A, confronta v con tutti gli elementi di A, in ordine, e restituisci la posizione corrispondente; restituisci 0 se nessuno degli elementi corrisponde.

A prima vista queste soluzioni potrebbero sembrare soddisfacenti; in realtà vi sono diversi problemi.

DESCRIZIONE DI ALGORITMI

Innanzitutto, le soluzioni proposte sono descritte in linguaggio naturale e fanno uso di una terminologia che andrebbe definita più precisamente (cosa significano esattamente "con tutti gli altri" e "corrispondente"?). Risolvere un problema richiede di saper descrivere il procedimento risolutivo nel modo più formale possibile.

VALUTAZIONE DI ALGORITMI

Una volta descritto un algoritmo in qualche linguaggio formale, possiamo ritenerci soddisfatti? Oppure esistono degli algoritmi "migliori" del nostro? Ovviamente dovremmo definire esattamente il significato di "migliore"; per il momento, è possibile limitarsi alla seguente osservazione: questi non sembrano algoritmi particolarmente "furbi". Ad esempio, la nostra proposta per la ricerca del minimo richiede che ogni valore sia confrontato con tutti gli altri, per un totale di $n(n-1)$ confronti, dove n è la dimensione dell'insieme. Tuttavia, molti di questi confronti sono inutili: se $a < b$ e $a < c$, è inutile confrontare b e c; nessuno dei due potrà essere il minimo. Allo stesso modo, nel problema RICERCA non abbiamo sfruttato l'ipotesi che i numeri siano ordinati; tutti i confronti effettuati dopo aver incontrato il primo elemento maggiore di v sono inutili, poiché tutti i valori successivi sono sicuramente maggiori di v (e quindi diversi).

PROGETTAZIONE DI ALGORITMI

I due problemi appena proposti sono estremamente semplici; non c'è bisogno di un esperto programmatore per suggerire delle soluzioni (tanto più se sono così inefficienti!). Ma non sempre è così; molti problemi non hanno soluzioni "ispirate dalla definizione", e richiedono un notevole sforzo di progettazione. Fortunatamente, esistono tecniche di progettazione generali che possono venire in aiuto. È fondamentale padroneggiare queste tecniche per essere in grado di risolvere al meglio il maggior numero di problemi.

1.2 DESCRIZIONE DI ALGORITMI

Nel descrivere un algoritmo, si utilizzano *azioni elementari* che debbono dare in tempo limitato un risultato certo, unico e ripetibile. In altri termini, l'effetto di un'azione elementare su un dato è determinato univocamente dall'azione e dal dato, cioè non dipende da variabili casuali. Se viene ripetuta la stessa azione sullo stesso dato si ottiene sempre lo stesso risultato. Il ritardo per ottenere il risultato è finito e noto a priori. Inoltre, le azioni debbono essere comprensibili ed eseguibili senza possibilità di ambiguità alcuna da parte dell'esecutore.

ESEMPIO – [Mescolare]
Si consideri l'istruzione "mescolare". Perché invece non è scritto: "Impugnare un mestolo, immergerlo nella pentola fino a toccare il fondo, con un angolo di 25° rispetto alla perpendicolare e muovere il braccio in modo rotatorio ad una velocità di circa mezzo metro al secondo"?

La risposta è ovvia: perché il cuoco sa benissimo come si fa a mescolare! Oppure perché non è scritto più semplicemente: "Prepara un minestrone alla toscana"? La risposta è altrettanto ovvia: perché questa azione sarebbe eseguibile da chi conoscesse già la ricetta ma ben difficilmente da una persona che non sapesse come si prepara un minestrone!

Il *livello di dettaglio* con cui si specificano le azioni elementari è molto importante. I diversi livelli di dettaglio dipendono dalle capacità dell'esecutore; nel caso dei calcolatori, sono necessari moltissimi dettagli. Infatti, anche i più moderni ed evoluti calcolatori altro non sono che esecutori di algoritmi espressi da lunghe sequenze di bit. Un *bit* è un oggetto che può assumere uno e un solo valore tra due possibili, indicati di solito con i valori booleani 0 e 1 (falso e vero, **false** e **true**). Può sembrare incredibile, eppure le uniche azioni che la macchina può effettuare sono semplicemente verificare il valore di un singolo bit e modificarne il valore! Fortunatamente, per descrivere algoritmi eseguibili da macchine elettroniche non è necessario usare tutti i dettagli ed i tecnicismi richiesti dalla manipolazione di sequenze di bit, ma si possono utilizzare costrutti tipici di linguaggi di programmazione ad alto livello. Tali costrutti sono espressi in una forma intermedia tra il linguaggio discorsivo tipico degli esseri umani, che troppo spesso ha un significato ambiguo, ed i comandi specifici di una macchina che "lavora" sui bit, che non sono ambigui ma troppo "tecnici" e dettagliati e quindi di difficile interpretazione da parte di un essere umano. I costrutti dei linguaggi di programmazione ad alto livello permettono concisione e chiarezza e hanno carattere di universalità (non dipendono dalla struttura circuitale di alcuna specifica macchina elettronica), ma nel contempo possono essere tradotti automaticamente (cioè tramite l'esecuzione di altri algoritmi, detti *compilatori*) in sequenze di bit eseguibili direttamente da circuiti di un vero calcolatore.

In questo testo, gli algoritmi saranno descritti in *pseudocodice*, un linguaggio di programmazione fittizio, non associato ad alcun compilatore o interprete realmente esistente, il cui scopo è quello di rappresentare, per quanto possibile, l'essenza degli algoritmi evitando di perdersi nei dettagli di un particolare linguaggio di programmazione. I linguaggi vanno e vengono; qualche anno fa era "di moda" il Pascal, oggi è il turno di Java, C, C++, C# (e qualunque altra variante di C vi venga in mente), domani chissà. Proprio mentre stiamo scrivendo Google ha annunciato il proprio linguaggio di programmazione, il Go: sta per nascere una nuova moda? I concetti alla base degli algoritmi discussi in questo libro, invece, sono immutabili: lo scopo dello pseudocodice è quello di comunicare, in maniera compatta, questi concetti.

Non esiste uno pseudocodice standard; ogni testo di programmazione o di algoritmi introduce il proprio e noi non saremo da meno. Accenniamo qui i principali costrutti, sottolineando le similitudini con alcuni dei linguaggi più diffusi.

Utilizziamo come esempio due algoritmi, scritti in pseudocodice, che risolvono i due problemi computazionali descritti in precedenza, così da introdurre i principali costrutti che impiegheremo. Ulteriori esempi verrano proposti per illustrare costrutti più avanzati

MINIMO

Supponiamo che un insieme di n valori di un generico tipo ITEM siano rappresentati da un vettore $A[1...n]$, di tipo ITEM[ ]. Un possibile algoritmo per identificare il minimo è il seguente: si sceglie il primo elemento di A come minimo parziale, copiandolo in una variabile *min*. Si confronta *min* con gli elementi compresi fra le posizioni 2 e n; ogni volta che si incontra un elemento minore, si aggiorna il minimo parziale. Al termine, *min* contiene il minimo dell'insieme, che viene restituito al chiamante.

```
ITEM min(ITEM[] A, integer n)
ITEM min ← A[1]                          % Minimo parziale
for i ← 2 to n do
    if A[i] < min then
        min ← A[i]                       % Nuovo minimo parziale
return min
```

RICERCA DELLA POSIZIONE

Come sopra, supponiamo che la sequenza sia memorizzata in un vettore. L'algoritmo proposto per risolvere questo problema prende il nome di *ricerca binaria* (o *ricerca dicotomica*) e parte dalla seguente osservazione: se si confronta v con un qualsiasi elemento $A[m]$ del vettore, possono darsi tre casi:

- $A[m] = v$: v è l'm-esimo elemento della sequenza;
- $A[m] < v$: gli elementi che precedono $A[m]$ nel vettore non possono contenere v, in quanto minori di $A[m]$; la ricerca deve continuare nei soli elementi che seguono $A[m]$;
- $A[m] > v$: simmetrico al caso precedente, la ricerca deve continuare nei soli elementi che precedono $A[m]$.

Con questa idea in mente, come scegliere l'indice m? In assenza di ulteriori informazioni sulla distribuzione dei numeri nella sequenza, una scelta conveniente potrebbe essere quella di prendere il mediano fra l'indice iniziale e l'indice finale del sottovettore preso in considerazione. In questo modo, siamo sicuri di scartare ad ogni passo metà degli elementi rimasti, arrivando in pochi passi alla soluzione.

L'algoritmo binarySearch() realizza queste idee. Si noti che l'algoritmo è ricorsivo; l'algoritmo termina quando l'elemento viene trovato, nel qual caso restituisce l'indice corrispondente, oppure quando i "supera" j, (ovvero quando l'insieme di elementi da considerare diventa nullo), nel qual caso restituisce 0.[1]

[1] L'algoritmo di ricerca binaria è realizzato dalla funzione `bsearch` della libreria standard del C e nella classe `java.util.Arrays` della libreria Java.

```
integer binarysearch(ITEM[ ] A, ITEM v, integer i, integer j)
  if i > j then
  |   return 0
  else
  |   integer m ← ⌊(i + j)/2⌋
  |   if A[m] = v then
  |   |   return m
  |   else if A[m] < v then
  |   |   return binarysearch(A, v, m + 1, j)
  |   else
  |   |   return binarysearch(A, v, i, m − 1)
```

Per cercare v in un vettore $A[1 \ldots n]$ si chiama binarySearch(A, v, 1, n).

ESEMPIO – Ricerca binaria

Cercando $v = 13$ nel vettore di 12 valori $A = 1, 5, 7, 8, 13, 17, 21, 27, 33, 37, 40, 42$, si ottiene:

CHIAMATA	i	j	m	$A[m]$
1	1	12	6	17
2	1	5	3	7
3	4	5	4	8
4	5	5	5	13

1.2.1 SINTASSI DI BASE

Lo pseudocodice prende ispirazione dai principali linguaggi di programmazione, quali Pascal, C e Java, con alcune importanti differenze per evitare possibili incomprensioni. Ad esempio, l'assegnamento è denotato con "←" mentre il confronto di uguaglianza è denotato da "="; vogliamo così evitare la confusione che deriva dall'utilizzare "=" e "==" nel linguaggio C, come in questo caso:

```
if (a=b) { printf("%d" , a); }
```

Questa è un'istruzione valida in C, ma non è un semplice confronto fra a e b: la variabile a prende il valore di b, e la condizione è vera se b è diverso da 0.

Per semplificare molti dei nostri algoritmi, utilizziamo il costrutto $a \leftrightarrow b$ per indicare che il contenuto della variabile a è scambiato con il contenuto della variabile b; corrisponde alla sequenza di istruzioni

$$temp \leftarrow a; \quad a \leftarrow b; \quad b \leftarrow temp$$

dove *temp* è una variabile temporanea opportunamente dichiarata.

Utilizzeremo i più comuni costrutti condizionali e iterativi, le cui parole chiave saranno rappresentate in grassetto:

- il costrutto condizionale "**if** *condizione* **then** *istruzione*" comanda l'esecuzione dell'istruzione purché la condizione sia vera; la sua forma più generale "**if** *condizione* **then** *istruzione1* **else** *istruzione2*" comprende anche un'istruzione alternativa, da eseguire qualora la condizione sia falsa;
- il costrutto iterativo illimitato "**while** *condizione* **do** istruzione" permette di ripetere iterativamente l'istruzione; la ripetizione prosegue se la condizione è verificata e si arresta non appena la condizione è falsa;
- i costrutti iterativi limitati:
  - "**for** *indice* ← *estremoInferiore* **to** *estremoSuperiore* **do** *istruzione*"
  - "**for** *indice* ← *estremoSuperiore* **downto** *estremoInferiore* **do** *istruzione*"

  che sono un modo compatto di scrivere:

```
indice ← estremoInferiore
while indice ≤ estremoSuperiore do
    istruzione
    indice ← indice + 1
```

```
indice ← estremoSuperiore
while indice ≥ estremoInferiore do
    istruzione
    indice ← indice − 1
```

La funzione iif(*condizione*, v_1, v_2) sarà utilizzata per scrivere un codice più compatto; restituisce v_1 se la condizione è vera, altrimenti restituisce v_2. Corrisponde all'operatore ternario di C/C++/Java: *condizione* ? v_1 : v_2.

Utilizzeremo anche il costrutto iterativo limitato "**foreach** *elemento* ∈ *insieme* **do** *istruzione*", che itera l'istruzione assegnando ad *elemento* tutti i valori contenuti in *insieme*, senza un particolare ordine. In un linguaggio reale tale costrutto può essere tradotto in vari modi, a seconda di come l'insieme viene rappresentato; se l'insieme è memorizzato in un vettore, viene sostituito da un ciclo **for**; se l'insieme è una delle `Collections` di Java, l'insieme può essere scandito con un iteratore (si veda il capitolo 3.4) o con il costrutto "for-each" introdotto in Java 1.5. Il costrutto **foreach** permette di scrivere algoritmi più compatti e più semplici da comprendere.

Un *blocco* è una sequenza di istruzioni che condividono la stessa indentazione (il rientro a destra delle righe), da considerarsi come una sola istruzione composita. Le linee verticali evidenziano in modo grafico dove inizia e finisce un blocco; non sono presenti marcatori espliciti come **begin/end** del Pascal o le parentesi graffe di C/C++/Java.

Infine, i commenti vengono indicati con il simbolo "%".

1.2.2 TIPI DI DATO PRIMITIVI

Ogni linguaggio di programmazione fornisce un certo numero di tipi di dato *primitivi*. Nel nostro pseudocodice, utilizzeremo interi, reali e booleani, identificati dalle parole

chiave **integer**, **real** e **boolean**. Le variabili saranno dichiarate al primo utilizzo, antecedendo il tipo al nome, come nel seguente esempio:

integer *min* ← *A*[1]

Le variabili utilizzate come contatori nei cicli **for** saranno di tipo intero, e per semplicità non verranno dichiarate; possiamo assumere che il tipo venga inferito dal contesto e dai valori di inizializzazione, come avviene per esempio in linguaggi quali il Go. Analogo discorso vale per il costrutto **foreach**, dove il tipo viene rilevato a partire dall'insieme a cui si applica il costrutto stesso.

Se non diversamente specificato, le variabili dichiarate all'interno di una procedura hanno visibilità locale alla procedura stessa.

Utilizzeremo la notazione matematica standard per scrivere espressioni su questi tipi di dato; oltre ai classici +, –, ·, /, **and**, **or**, **not**, anche le operazioni log per il logaritmo, mod per il modulo, $\lfloor x \rfloor$ e $\lceil x \rceil$ per la parte intera inferiore e superiore, e così via.

Gli operatori booleani **and** e **or** sono cortocircuitati: ovvero, nell'espressione *x* **and** *y*, viene prima valutato *x*; se questo è falso, *y* non viene valutato e l'espressione è falsa. Nell'espressione *x* **or** *y*, viene prima valutato *x*; se questo è vero, *y* non viene valutato e l'espressione è vera.

1.2.3
TIPI DI DATO COMPOSTI

Nello scrivere una procedura in un linguaggio di programmazione può presentarsi la necessità di voler definire ed utilizzare, oltre ai tipi di dato primitivi, nuovi tipi di dato, che siano in qualche modo più vicini alla natura del problema che si vuole risolvere. Ogni linguaggio mette a disposizione costrutti per comporre tipi di dato esistenti in tipi di dato più complessi.

I *vettori* (o *array*, in inglese) permettono di riunire sotto un unico nome un numero prefissato di elementi dello stesso tipo e di individuare i singoli elementi tramite *indici*. Un *record* (Pascal), *struttura* (C), *classe* (Java) è un agglomerato di elementi di tipo diverso. Ogni elemento (*campo*) è individuato con un nome di riferimento, utilizzato per accedere all'elemento in operazioni di lettura o scrittura.

I vettori possono essere unidimensionali o multidimensionali (*matrici*).

Il tipo di un vettore unidimensionale contenente elementi di tipo *T* è denotato con *T*[ ], mentre il tipo di un vettore multidimensionale (o matrice) viene dichiarato con *T*[ ]...[ ], con tante coppie di parentesi quante sono le dimensioni. Per esempio, il tipo record RETTANGOLO contenente due campi è dichiarato nel modo seguente:

RETTANGOLO

integer *lunghezza*
integer *altezza*

È possibile quindi dichiarare variabili di tipo vettore e record in maniera molto semplice:

$$\begin{array}{rcl} T[]\,A & \leftarrow & \textbf{new}\ T[1\ldots n] \\ T[][]\,B & \leftarrow & \textbf{new}\ T[1\ldots n][1\ldots m] \\ \text{RETTANGOLO}\ r & \leftarrow & \textbf{new}\ \text{RETTANGOLO} \end{array}$$

Queste tre righe dichiarano una variabile A di tipo $T[\,]$ inizializzata come vettore di n elementi, una variabile B di tipo $T[\,][\,]$ come matrice di $n \times m$ elementi e una variabile r di tipo RETTANGOLO.

Dal punto di vista di un'introduzione agli algoritmi, non interessa come la memoria venga gestita per ospitare queste strutture di tipo composto. Ogni linguaggio utilizza approcci diversi; assumeremo che tutte le variabili di tipo composto siano in realtà *puntatori* (o *reference*) alle strutture stesse, la cui memoria viene allocata all'atto della creazione tramite il comando **new**, mentre il comando **delete** serve a liberare la memoria occupata da tali oggetti. Un puntatore "vuoto" viene indicato da **nil**.

I puntatori permettono un accesso indiretto alla struttura di dati, mediante l'indirizzo della cella di memoria a partire dalla quale la struttura stessa è memorizzata. Per esempio, una variabile di tipo puntatore ad un rettangolo ha per valore l'indirizzo della cella di memoria che contiene il rettangolo.

I campi di una variabile r di tipo RETTANGOLO vengono acceduti tramite la notazione a punto: *r.lunghezza*. Per specificare un elemento di un vettore o di una matrice, scriveremo il nome del vettore seguito dagli indici dell'elemento fra parentesi quadre: per esempio, $A[i]$ e $B[i,j]$. Sebbene alcuni linguaggi (ad es., Java) permettano di ottenere la lunghezza di un vettore tramite costrutti del tipo `A.length`, espliciteremo sempre questa lunghezza fra i parametri di ingresso delle procedure.

La scelta dell'indice di partenza (1 come in Pascal, invece che 0 come nei linguaggi derivati dal C) ancora una volta è dettata da semplicità e chiarezza. In realtà, molti algoritmi e strutture di dati lavorano in modo indipendente dagli indici, con alcune notabili eccezioni (code circolari, code con priorità e tabelle hash). In alcuni casi potremo quindi dichiarare esplicitamente vettori di n elementi con $A[0\ldots n-1]$.

Infine, concludiamo con alcune osservazioni.

- Nella descrizione in linguaggio naturale degli algoritmi proposti, faremo riferimento a porzioni di vettori utilizzando la notazione $A[i\ldots j]$, ovvero l'insieme di elementi compresi fra l'indice i e l'indice j, estremi inclusi.
- Così come è definito, un record non è altro che un contenitore; se viene associato ad un insieme di procedure che fungono da operatori su di esso, diventa un *tipo di dato astratto*; ulteriori dettagli verranno forniti nel capitolo 3.
- Nei nostri esempi, abbiamo utilizzato il tipo di dato astratto ITEM per indicare un tipo generico, di cui non interessa conoscere né il contenuto né l'insieme di operazioni permesse, se non che su di esso è applicabile una relazione $\leq$ di ordinamento totale.
- Come forse il lettore attento ha già notato, per facilitare la lettura utilizziamo fonti tipografiche diverse per i diversi elementi sintattici del nostro pseudocodice. Funzioni e procedure saranno indicate dal font helvetica; le variabili dal font

times corsivo; i tipi di dato dal MAIUSCOLETTO; le parole chiave del linguaggio in **times bold**.

1.2.4 PROCEDURE E FUNZIONI

Un algoritmo descritto per mezzo di costrutti tipici di un linguaggio di programmazione è comunemente detto *procedura* o *funzione*.

Ad esempio, la funzione min() indica un algoritmo che può essere utilizzato con due dati di ingresso arbitrari (i *parametri formali*), chiamati all'interno della procedura col nome *A* (di tipo ITEM[ ]) e *n* (di tipo **integer**). La funzione può essere richiamata con *parametri attuali* del medesimo tipo:

$$\min(V, 1024)$$
$$\min(E, h)$$

Il testo della procedura che descrive un algoritmo è strutturato come una sequenza di azioni (o passi) elementari tra loro distinti ed ha lunghezza finita e costante per tutti i possibili dati di ingresso. D'altro canto, l'algoritmo è generale ed applicabile ad una infinità di dati d'ingresso, di grandezza arbitraria, ed il tempo necessario per eseguire la procedura può variare a seconda della grandezza dei dati di ingresso.

Specificare una procedura permette di definire un algoritmo una volta per tutte, di poterlo individuare con un nome e di poterlo utilizzare più volte, con valori distinti e arbitrari, purché consistenti, dei dati d'ingresso.

Ciò ha un duplice vantaggio:

- si evita di riscrivere più volte un algoritmo per effettuare la stessa computazione con valori diversi dei dati;
- è possibile sviluppare algoritmi complessi gradualmente, scrivendo dapprima procedure che, utilizzando costrutti elementari del linguaggio, risolvono sottoproblemi più semplici e successivamente procedure più complesse che utilizzano le procedure scritte precedentemente come se fossero nuove istruzioni elementari.

In questo modo, una procedura *A* per la risoluzione di un certo problema *P* può richiamare al suo interno una procedura *B* che risolve un sottoproblema *Q*, che a sua volta può richiamare *C* per risolvere *R*, e così via. Al momento della chiamata, l'esecuzione di *A* viene temporaneamente sospesa per permettere l'esecuzione di *B*, sui dati fornitigli da *A*. L'esecuzione di *A* viene poi ripresa, al termine dell'esecuzione di *B*, a partire dall'istruzione seguente alla chiamata.

I parametri attuali vengono passati per *valore*: la procedura chiamata riceve una copia dei parametri attuali, ed eventuali modifiche a tali parametri non vengono viste dalla procedura chiamante. Nell'esempio precedente, il valore della variabile *h* viene copiato nel parametro formale *n*, ed eventuali modifiche ad *n* nel corpo di min() non modificano il valore di *h*.

Quando viene passato un oggetto di tipo composto, viene passata una copia del puntatore all'oggetto; questo significa che la procedura chiamata potrà modificare gli ele-

menti presenti nell'oggetto, ma non scambiarlo con un altro (ad esempio sostituendolo con un vettore più grande).

Quando una procedura restituisce un valore al chiamante è detta *funzione*. Nella dichiarazione, il nome della funzione è preceduto dal tipo di dato che viene restituito; nel codice, la restituzione avviene tramite l'istruzione **return**. Non usiamo parole chiave per identificare una procedura o una funzione, preferendo evidenziare graficamente il nome fra due linee orizzontali.

Infine, un'osservazione sui dati di input alle procedure. In alcuni casi, può essere necessario specificare esplicitamente quali dati siano considerati accettabili e quali no. La condizione di accettabilità è detta *precondizione* ed è introdotta all'inizio delle procedure con la parola chiave **precondition**. Una realizzazione realistica dei nostri algoritmi dovrebbe verificare la precondizione, se presente; nel caso non sia verificata dovrebbe restituire un codice di errore (o un'eccezione, in Java).

1.2.5 RICORSIONE

Il meccanismo di chiamate tra procedure può essere anche ricorsivo, nel senso che una procedura può richiamare se stessa al suo interno! Ne abbiamo visto un esempio con la procedura binarySearch(). L'apparente circolarità (per eseguire binarySearch() si richiama ancora binarySearch()) è spezzata da una condizione di chiusura, che interrompe l'esecuzione della procedura quando i dati raggiungono opportuni valori (nell'esempio, quando $i > j$). Altrimenti, la procedura richiama se stessa, sospendendo temporaneamente la vecchia computazione e iniziando una nuova computazione su dati che hanno sì lo stesso nome (nell'esempio: i e j), ma valori diversi.

La procedura binarySearch() è un esempio di *ricorsione di coda* (*tail recursion*), ovvero la chiamata ricorsiva è l'ultima istruzione eseguita. In questo caso, è semplice scrivere iterativeBinarySearch(), una versione iterativa della stessa funzione, in quanto non occorre mantenere in memoria i dati della vecchia computazione quando ne viene invocata una nuova. Le versioni iterative sono di solito più efficienti, sebbene non abbiano la medesima chiarezza espositiva delle versioni ricorsive.

```
integer iterativebinarysearch(ITEM[] A, integer n, ITEM v)
  integer i ← 1
  integer j ← n
  integer m ← ⌊(i + j)/2⌋
  while i < j and A[m] ≠ v do
    if A[m] < v then
      i ← m + 1
    else
      j ← m − 1
    m ← ⌊(i + j)/2⌋
  if i > j or A[m] ≠ v then return 0 else return m
```

1.3 VALUTAZIONE DI ALGORITMI

Una volta stabilito un linguaggio per comunicare gli algoritmi, dobbiamo dotarci di strumenti per valutarli. Due sono i principi cardine che utilizzeremo.

CORRETTEZZA

L'algoritmo che abbiamo proposto è corretto? Risolve correttamente un particolare problema computazionale?

Provare la correttezza di un algoritmo richiede una dimostrazione matematica. Si considerino, ad esempio, gli algoritmi min() e binarySearch(). Nel caso di min(), una proprietà matematica che può essere dimostrata è la seguente: all'inizio di ogni iterazione del ciclo **for**, la variabile *min* contiene il minimo parziale degli elementi $A[1 \ldots i-1]$. Un'affermazione di questo tipo prende il nome di *invariante di ciclo* o, più semplicemente, *invariante*. Con il termine "all'inizio" intendiamo nel momento in cui la variabile i assume il valore che avrà durante l'iterazione, prima di eseguire il codice dell'istruzione associata al ciclo **for**.

Dobbiamo dimostrare che l'invariante è vera in due casi specifici:

(1) **passo base**: l'affermazione è vera all'inizio della prima iterazione del ciclo;

(2) **passo induttivo**: se è vera all'inizio di un'iterazione, deve essere vera al termine dell'iterazione stessa, e quindi all'inizio dell'iterazione successiva.

Nel caso di *min*, è facile dimostrare il passo base, in quanto all'inizio del primo ciclo $i = 2$ e $A[1 \ldots 1]$ contiene il solo elemento $A[1]$, che è stato precedentemente copiato in *min*. Per quanto riguarda il passo induttivo, se *min* contiene il minimo di $A[1 \ldots i-1]$ all'inizio di un'iterazione, possono darsi due casi: se $A[i]$ è minore di *min*, esso è il minimo di $A[1 \ldots i]$, e viene correttamente copiato in *min*; altrimenti *min* rimane invariato. In entrambi i casi, il passo induttivo è dimostrato.

Dimostrare un'invariante di questo tipo ha senso se fornisce informazioni sulla correttezza dell'algoritmo in generale; e, infatti, se si considera l'invariante al termine del ciclo, quando $i = n + 1$, vediamo che *min* contiene il minimo di $A[1 \ldots n]$, e quindi può essere correttamente restituita come valore finale. In questo caso, e in quelli che seguono, assumiamo tacitamente che la variabile i "esista" al termine del ciclo e contenga il valore successivo all'estremo superiore, che ha causato la terminazione del ciclo (come nei linguaggi derivati dal C).

Nel caso di binarySearch(), non utilizziamo un'invariante, ma adottiamo comunque un approccio induttivo per dimostrare la correttezza dell'algoritmo, qualunque sia la dimensione $n = j - i + 1$ della porzione di vettore in cui cercare. Se $n = 0$, il vettore è vuoto e $i = j + 1$; la condizione del primo **if** è vera, quindi binarySearch() restituisce correttamente il valore 0, ad indicare che v non è presente nella porzione in cui cercare. Questo dimostra il caso base. Supponendo che binarySearch() termini correttamente per tutte le dimensioni fino ad $n - 1$ (ipotesi induttiva), vogliamo dimostrare che termina correttamente anche per n. La dimostrazione segue banalmente dai tre casi dell'algoritmo: se si trova

il valore nell'indice mediano, l'algoritmo termina restituendo questo indice; altrimenti il valore (se presente) si trova sicuramente prima o dopo l'indice mediano, a causa dell'ordinamento. Il vettore in cui cercare avrà quindi dimensione massima $\lfloor n/2 \rfloor$, e la chiamata ricorsiva verrà eseguita correttamente per l'ipotesi induttiva.

Entrambe le dimostrazioni sono basate sull'induzione matematica. Si è detto che gli informatici sono matematici che sanno solo produrre dimostrazioni basate sull'induzione; questo non è totalmente vero (anche le dimostrazioni per assurdo e le riduzioni hanno il loro ruolo, come vedremo) ma è certamente vero che molti degli algoritmi che proporremo qui sono o ricorsivi o incrementali, e si prestano bene a dimostrazioni di questo tipo.

Nel seguito, dimostreremo la correttezza dei nostri algoritmi ogni volta che non sia immediatamente chiara dalla definizione dell'algoritmo stesso.

EFFICIENZA

Gli algoritmi che abbiamo proposto sono "veloci" o "efficienti"? Utilizzano al meglio le risorse del calcolatore? Abbiamo già discusso alcune semplici considerazioni sull'efficienza introducendo i nostri problemi computazionali; cerchiamo ora di formalizzare meglio le nostre idee, ma la domanda è molto complessa, ed una risposta approfondita è rimandata al capitolo 2.

Per quanto riguarda MINIMO, abbiamo abbozzato un algoritmo in cui ogni valore veniva confrontato con tutti gli altri, richiedendo un totale di $n(n-1)/2$ confronti. Ovviamente, l'algoritmo proposto è preferibile, in quanto richiede un numero molto inferiore di confronti, ovvero $n - 1$.

Per quanto riguarda RICERCA, abbiamo abbozzato un algoritmo che richiede n confronti, ma la ricerca binaria è più efficiente. Ogni chiamata richiede infatti 2 confronti fra elementi del vettore (nel costrutto **if-else if**). Quindi, la domanda diventa: quante volte la funzione richiamerà se stessa? Ad ogni chiamata, il vettore da chiamare si dimezza. Quindi il numero di chiamate richieste nel caso il valore cercato non sia presente è $\lfloor \log n \rfloor$. Se invece è presente, potrebbe essere incontrato prima. Quindi il numero totale di confronti sarà al massimo $\lfloor 2 \log n \rfloor$.

Si potrebbe domandare perché discutiamo di confronti e non più genericamente di tempo di calcolo (in secondi, minuti ecc.?). I motivi sono due: innanzitutto, le operazioni di confronto sono "dominanti" in questo algoritmo; se il tipo generico ITEM è sostituito con **integer**, il confronto è un'operazione primitiva che richiede ormai meno di un miliardesimo di secondo per essere eseguita; ma se ITEM è un tipo composto, l'operazione di confronto potrebbe essere molto pesante (si pensi al costo della valutazione di quale fra due stringhe molte lunghe venga prima in ordine alfabetico).

In secondo luogo, il tempo di calcolo effettivo dipende fortemente dalla piattaforma utilizzata per la misura; la frequenza del processore, la memoria cache di primo, secondo e terzo livello, la velocità del bus e così via. Ognuno di questi parametri può influenzare il tempo di calcolo, in maniera sottile.

Concludiamo con una domanda, a cui daremo risposta nel prossimo capitolo: possiamo fare meglio di così? Esistono degli algoritmi "migliori" (più efficienti) per i problemi proposti?

ULTERIORI PROPRIETÀ

Correttezza ed efficienza esauriscono le proprietà "desiderabili" degli algoritmi? Da un punto di vista più ampio, proprio dell'ingegneria del software, potremmo aggiungere proprietà sicuramente importanti quali semplicità, modularità, manutenibilità, espandibilità, ma anche aspetti particolari della correttezza quali sicurezza e robustezza (capacità di gestire dati in ingresso non previsti). Cercheremo di realizzare algoritmi che soddisfino queste proprietà, ma non ne discuteremo direttamente, ritenendo che un corso di ingegneria del software sia la sede più adatta.

Una considerazione è tuttavia importante. Spesso, molte di queste proprietà hanno un costo, in termini di prestazioni. Scrivere un codice modulare può richiedere un costo aggiuntivo per la gestione delle chiamate di procedura; utilizzare un linguaggio robusto ma interpretato come Java può rallentare l'esecuzione del codice; e la lista di esempi potrebbe continuare. Progettare algoritmi efficienti è un prerequisito fondamentale per poter "pagare" questi costi. Se il vostro algoritmo utilizza un numero log n di passi invece che n, il fatto che l'esecuzione sia rallentata di un piccolo fattore costante perde completamente importanza.

1.4 PROGETTAZIONE DI ALGORITMI

Per progettare algoritmi corretti ed efficienti, un buon progettista di algoritmi deve saper padroneggiare due distinti insiemi di conoscenze:

- deve essere in grado di identificare il problema da risolvere fra una classe più ampia di problemi, e utilizzare l'enorme catalogo di soluzioni sviluppate negli ultimi cinquant'anni per evitare di partire ogni volta da zero; come si dice, deve "salire sulle spalle dei giganti". Non conoscere la ricerca binaria, ad esempio, è un grave peccato per chiunque voglia scrivere algoritmi efficienti;
- deve saper utilizzare le principali tecniche algoritmiche, quali *divide-et-impera*, programmazione dinamica, tecnica *greedy*, *backtrack*, ricerca locale e probabilismo. Come primo assaggio, sappiate che la ricerca binaria appartiene alla classe *divide-et-impera*.

In questo testo, svilupperemo tali argomenti in due filoni principali. La Parte II discuterà l'organizzazione di grandi quantità di dati nella memoria di un calcolatore affinché possano essere reperiti facilmente ed efficientemente. Questo è un problema fondamentale, per il quale tante soluzioni differenti (strutture di dati) sono state proposte, ognuna specializzata per un diverso utilizzo. La Parte III si concentrerà sulle principali tecniche di progettazione, mentre la Parte IV complementerà la Parte III, discutendo tecniche speciali per risolvere in modo subottimo problemi inerentemente intrattabili per i quali non si conoscono algoritmi efficienti.

1.5 REALITY CHECK

Il primo algoritmo di cui si abbia documentazione si trova nel papiro egizio di Ahmes (Fig. 1.1), risalente circa al 1650 a.C. (ma tratto per dichiarazione di Ahmes da un papiro ancora più antico) e conservato al British Museum di Londra. Il papiro contiene un algoritmo per la moltiplicazione. Siano a e b i due fattori (interi). Partendo da $p = 0$,

- si somma b a p se a è dispari;
- si dimezza a (prendendone la parte intera);
- si raddoppia b,

ripetendo finché non risulti $a = 0$. Al termine del procedimento p ha valore $a \cdot b$.

Per provare la correttezza, basta considerare le due seguenti proprietà: se a è un numero pari allora $a \cdot b = (a/2) \cdot 2b$, se viceversa a è dispari $a \cdot b = ((a-1)/2) \cdot 2b + b$. Verificando che l'algoritmo altro non fa se non passare per rappresentazioni equivalenti del problema in cui a decresce continuamente fino a giungere certamente a zero in un numero finito di passi, si prova la correttezza.

Fig. 1.1 Papiro di Ahmes (British Museum; foto Paul James Cowie).

Il fatto sorprendente è che tale algoritmo di moltiplicazione viene usato tuttoggi nei circuiti delle unità aritmetico/logiche dei calcolatori elettronici. Infatti, rappresentando gli interi in base 2, le operazioni di verifica della parità, dimezzamento e raddoppio sono elementari. Per verificare la parità basta controllare il valore del bit meno significativo, mentre per dimezzare (o raddoppiare) basta traslare a destra (o a sinistra) di una posizione tutti i bit.

ANALISI DI ALGORITMI

Per risolvere un problema, spesso sono disponibili molti algoritmi diversi. Come scegliere il "migliore"? Un criterio generalmente adottato consiste nel valutare la "bontà" di un algoritmo in base alla quantità di risorsa utilizzata per il calcolo. La risorsa di interesse dominante è il tempo richiesto per eseguire le azioni elementari, ma talvolta è considerato anche lo spazio necessario per immagazzinare (o memorizzare) e manipolare i dati.

2.1
TEMPO DI CALCOLO

In pratica, non è possibile misurare il tempo di calcolo di un algoritmo col numero di secondi richiesto per eseguire su un calcolatore elettronico una procedura che lo descriva. Infatti tale tempo dipende pesantemente dai dati di ingresso, dal linguaggio in cui la procedura è descritta, dalla qualità con cui viene automaticamente tradotta dal compilatore in una sequenza di bit e dalla natura e velocità del calcolatore elettronico. Occorre una misura "astratta" che tenga maggiormente conto del "metodo di risoluzione" con cui l'algoritmo effettua la computazione.

Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, viene spontaneo esprimere il tempo di calcolo come il costo complessivo delle operazioni elementari in funzione della dimensione n dei dati di ingresso. Sono considerate elementari le operazioni aritmetiche, logiche, di confronto e di assegnamento. Talvolta, è utile considerare non tutte le operazioni, ma solo quelle dominanti, cioè quelle che incidono maggiormente sul tempo di esecuzione. Il costo delle operazioni è principalmente valutato nel *caso pessimo*, cioè sul dato d'ingresso più sfavorevole, tra tutti quelli di dimensione n, ma talvolta è valutato anche nel *caso medio*, cioè mediando su tutti i possibili dati di dimensione n, tenendo conto della probabilità con cui ciascun dato può occorrere.

ESEMPIO – [Tempo di calcolo di min() iterativa]
Per valutare il tempo di calcolo della funzione iterativa min(), si osservi che la funzione min() è formata da un numero finito di istruzioni elementari. Ogni istruzione richiede un tempo costante di esecuzione, ma la costante può essere diversa da istruzione a istruzione. Indichiamo con c_h la costante richiesta per l'esecuzione dell'istruzione h-esima. Rivediamo la funzione min() indicando, per ogni istruzione, il costo di un'esecuzione ed il numero totale di volte che l'istruzione viene eseguita. Tenendo conto che la dimensione del problema è n si ottiene:

ITEM min(ITEM[] A, **integer** n)	Costo	#Volte
ITEM $min \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n **do**	c_2	n
if $A[i] < min$ **then**	c_3	$n-1$
$min \leftarrow A[i]$	c_4	$n-1$
return min	c_5	1

Si noti che l'incremento dell'indice i nel ciclo **for** è ripetuto n volte e non $n-1$. Infatti il **for**, come spiegato precedentemente, è un modo più compatto di scrivere un **while**, nel quale la condizione $i \leq n$ deve essere verificata una volta in più per poter uscire dal ciclo quando i supera n. Il tempo di calcolo $T(n)$ di min() si ottiene sommando il prodotto del costo di ciascuna istruzione per il numero di volte che tale istruzione è eseguita:

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

Pertanto, il tempo di calcolo di min() può essere espresso come

$$T(n) = an + b$$

con a e b costanti intere positive. Si noti che questo andamento di $T(n)$ lineare in n vale sia nel caso pessimo che in quello medio, poiché il corpo del **for** è sempre ripetuto esattamente $n-1$ volte, qualsiasi sia il dato A d'ingresso.

2.9 ESEMPIO – [Tempo di calcolo di binarySearch() ricorsiva]

Valutare il tempo di un algoritmo ricorsivo quale binarySearch() è un po' più complicato. Innanzitutto, l'elemento cercato potrebbe essere trovato "al primo colpo", ovvero subito nella posizione mediana; questo è il caso ottimo. Noi invece valuteremo il caso in cui non sia presente, che corrisponde al caso pessimo. Secondo, vengono eseguite parti diverse dell'algoritmo a seconda dei valori di i e j: se $i > j$, la ricorsione si ferma, altrimenti viene effettuata una chiamata ricorsiva. Nel codice della binarySearch() inseriamo quindi due colonne # che valutano il numero di volte che una particolare riga viene eseguita in ognuno dei possibili casi.

integer binarysearch(ITEM[] A, ITEM v, **integer** i, **integer** j)	Costo	$\#(i > j)$	$\#(i \leq j)$
if $i > j$ **then**	c_1	1	1
return 0	c_2	1	0
else			
integer $m \leftarrow \lfloor (i+j)/2 \rfloor$	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)	$c_7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
else			
return binarySearch(A, v, i, $m-1$)	$c_7 + T(\lfloor n/2 \rfloor)$	0	0/1

Si indichi ancora con $T(n)$ il tempo richiesto per un dato d'ingresso di dimensione n. Si ass... ma dapprima che $n = 0$ (cioè che $i > j$). In tal caso la funzione esegue direttamente la cond... zione di chiusura. Se non vi è alcun elemento, il tempo di calcolo è $c_1 + c_2$ e quindi:

$$T(0) = c$$

dove c è una costante.

Altrimenti, la funzione suddivide il vettore in due porzioni grandi $\lfloor (n-1)/2 \rfloor$ e $\lfloor n/2 \rfloor$ (se n è pari, queste hanno dimensione $n/2 - 1$ e $n/2$; se n è dispari, hanno entrambe dimensione $\lfloor n/2 \rfloor$). Per semplificare i calcoli, assumiamo che n sia una potenza di 2, ovvero $n = 2^k$ per un qualche k intero positivo. Ancora per il caso pessimo, supponiamo che si scelga sem... pre il lato più grande (quello di dimensione $n/2$, essendo n pari). Il costo della funzione è quindi:

$$\begin{aligned} T(n) &= T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 \\ &= T(n/2) + d \end{aligned}$$

dove d è una costante. Il tempo di calcolo $T(n)$ può essere dunque ricavato dalla soluzione delle seguenti relazioni, dette *relazioni di ricorrenza*:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n \geq 1 \end{cases}$$

Una tecnica utile per risolvere relazioni di ricorrenza consiste nel produrre una catena di uguaglianze ottenute per sostituzioni successive, applicando le relazioni per $n/2$, $n/4$, $n/8$, ..., 2, 1, 0. Si ottiene:

$$\begin{aligned} T(n) &= T(n/2) + d \\ &= T(n/4) + 2d \\ &= T(n/8) + 3d \\ &\ldots \\ &= T(1) + kd \\ &= T(0) + (k+1)d \\ &= kd + (c+d) \\ &= d \log n + e \end{aligned}$$

Quindi il caso pessimo di binarySearch() richiede un tempo di calcolo che, a meno dei valori delle costanti d e e, cresce come il logaritmo di n.

In pratica, non è necessario tener conto esplicitamente dei costi delle singole operazioni. Infatti, ha poca importanza che una moltiplicazione sia, mettiamo, dieci volte più lenta di una addizione, quando alla fin fine entrambe le operazioni richiedono tempi costanti, sep...

pur diversi. Di fatto, tutte le costanti delle singole operazioni possono venire "inglobate" in poche costanti, per esempio a e b, quando si conclude che il tempo di calcolo $T(n)$ cresce linearmente in n, cioè come $an + b$. Pertanto, il tempo di calcolo può essere valutato semplicemente contando il numero di operazioni elementari eseguite nel caso pessimo (o medio) su tutti i dati di ingresso di dimensione n.

Un'obiezione che può essere fatta alla valutazione del tempo di calcolo nel caso pessimo è che sia appunto troppo "pessimistica", perché il caso pessimo potrebbe verificarsi molto raramente. Tale valutazione presenta però l'indubbio vantaggio di garantire che l'algoritmo non richiederà mai, per nessun dato di ingresso di dimensione n, un tempo maggiore. Inoltre il caso pessimo, per certi problemi, occorre molto di frequente, come nel verificare se un dato elemento appartiene oppure no ad un insieme, il caso pessimo occorre tutte le volte che si inserisce un elemento e si verifica prima che non sia già incluso! Inoltre, benché la valutazione nel caso medio possa apparire più realistica, spesso non è chiaro sotto quali ipotesi di distribuzione di probabilità debba essere svolta. In genere viene assunta una distribuzione uniforme (in cui tutti i casi sono equiprobabili), che per molti problemi è irrealistica o porta allo stesso risultato, a meno di costanti, del caso pessimo. Per esempio, assumendo che un dato elemento appaia come i-esimo in un insieme di n elementi con la stessa probabilità $1/n$, per reperirlo occorre un tempo lineare in n sia nel caso medio che in quello pessimo, perché bisogna esaminare $n/2$ elementi in media ed n nel caso pessimo. Inoltre, la valutazione nel caso medio richiede l'impiego di formule matematiche molto complicate e di difficile risoluzione. A parte poche eccezioni, la valutazione del tempo di calcolo principalmente considerata è quella nel caso pessimo. Talvolta è anche considerato il cosiddetto *tempo ammortizzato*, dove il tempo $T(m)$ richiesto nel caso pessimo per eseguire una sequenza di m operazioni (ad esempio, ricerche, inserimenti e cancellazioni di elementi in un insieme) viene diviso per il numero di operazioni, ottenendo un tempo medio $T(m)/m$ per operazione, detto appunto ammortizzato. È bene notare che in questo caso, a differenza della valutazione nel caso medio, non sono affatto coinvolte probabilità, poiché il tempo ammortizzato è una sorta di tempo medio per ciascuna operazione valutato nel caso pessimo su tutte le sequenze di m operazioni.

ESEMPIO – [Criterio di costo uniforme e logaritmico]

La valutazione del tempo di calcolo basato sull'equipollenza delle operazioni primitive prende il nome di *criterio di costo uniforme*. Un modello alternativo è quello chiamato *criterio di costo logaritmico*, in cui il tempo richiesto da ciascuna istruzione elementare di un algoritmo dipende dal numero di bit necessari a rappresentare i suoi operandi. In generale, possiamo assumere che gli operandi siano rappresentati da un numero costante di bit, e che quindi le due misure coincidano a meno di una costante moltiplicativa. In alcuni casi, tuttavia, il criterio di costo uniforme può trarre in inganno sulla reale complessità di un algoritmo. Come esempio, si consideri il seguente algoritmo che calcola la funzione fattoriale:

```
fattoriale(integer n)
  integer tot ← 1
  for i ← 2 to n do
    | tot ← tot · i
  return tot
```

Dal punto di vista del criterio di costo uniforme, questo algoritmo richiede n moltiplicazioni, e quindi il suo tempo di calcolo cresce linearmente. Questo è certamente vero se consideriamo il tipo primitivo **integer** come un intero a 32 bit, per esempio; ma questo limita anche il valore massimo di cui può essere calcolato il fattoriale (12! è minore del più grande intero rappresentabile con 32 bit, mentre 13! è più grande di esso; quindi 12 è il valore massimo). Se invece consideriamo il tipo primitivo **integer** come interi ad arbitraria precisione (tipo BigInteger di Java e gli Integer di Haskell), a questo punto il numero di bit necessario per rappresentare n è pari a $d = \lfloor \log_2 n \rfloor + 1$, e il tempo di calcolo, dipendendo da n, è almeno esponenziale in d. Torneremo su questi argomenti quando parleremo degli algoritmi pseudo-polinomiali (Sezione 19.1).

2.2
ORDINE DI GRANDEZZA E COMPLESSITÀ

Nel valutare il tempo di calcolo $T(n)$ di una procedura, è in genere assai difficile quantificare con esattezza le costanti coinvolte. Questa difficoltà viene aggirata valutando il numero di operazioni in *ordine di grandezza*, cioè esprimendolo come limitazione della funzione $T(n)$ al tendere all'infinito della dimensione n, trascurando le costanti moltiplicative ed additive. Si parla così di *complessità computazionale asintotica* in ordine di grandezza (o, brevemente, *complessità*) di una procedura.

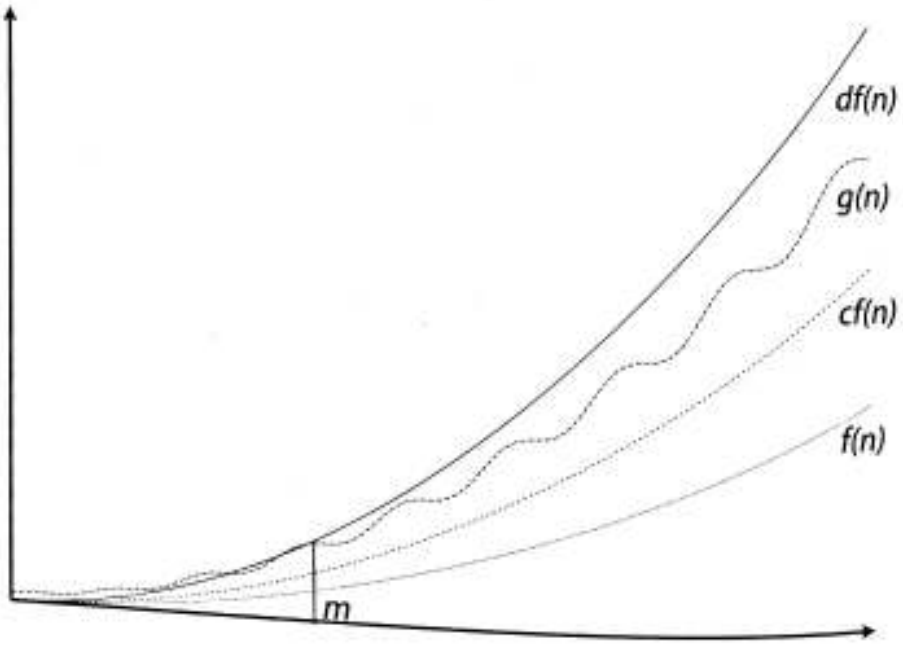


Fig. 2.1 Una funzione $g(n)$ che è $\Theta(f(n))$.

A tal fine, si usano le notazioni "O", "Ω" e "Θ" definite come segue:

- $O(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano due costanti positive c ed m per cui $g(n) \leq cf(n)$ per ogni $n \geq m$;
- $\Omega(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano due costanti positive c ed m per cui $cf(n) \leq g(n)$ per ogni $n \geq m$;

– $\Theta(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano tre costanti positive c_1, c_2 ed m per cui $c_1 f(n) \leq g(n) \leq c_2 f(n)$ per ogni $n \geq m$.

Si dice che "una funzione $g(n)$ è di ordine omicron di $f(n)$" (in breve: "$g(n)$ è $O(f(n))$") per indicare una limitazione superiore al comportamento asintotico di $g(n)$. Analogamente, si dice che "$g(n)$ è di ordine omega di $f(n)$" (in breve: "$g(n)$ è $\Omega(f(n))$") per indicarne una limitazione inferiore, e che "è di ordine theta di $f(n)$" (in breve: "$g(n)$ è $\Theta(f(n))$") per indicarne sia una limitazione inferiore che superiore[1].

2.4 ESEMPIO – [Ordini di grandezza]
La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché esistono le costanti $c = 5$ ed $m = 4$ per cui $g(n) \leq 5n^2$ per $n \geq 4$. $g(n)$ è anche $\Omega(n^2)$, perché esistono le costanti $c = 1$ ed $m = 1$ tali che $g(n) \geq n^2$ per $n \geq 1$. Si noti che, in accordo alla definizione, $g(n)$ è $O(n^k)$ per ogni $k \geq 2$, ed anche $\Omega(n^k)$ per ogni $k \leq 2$. In genere però si richiede l'ordine più stretto che è $\Theta(n^2)$, poiché $g(n)$ è sia $O(n^2)$ che $\Omega(n^2)$. Similmente, è facile verificare che la funzione $T(n)$ dell'Esempio 2.1 è $\Theta(n)$.

Per stimare gli ordini di grandezza, non è necessario ogni volta applicarne le definizioni e ricavare esplicitamente le costanti richieste. Si possono invece usare le seguenti regole, la cui dimostrazione è lasciata per esercizio, che permettono di semplificare la stima dell'ordine di grandezza di una funzione non negativa combinando insieme le stime delle sue parti.

(1) *Riflessività*: Per ogni costante c (inclusa quindi $c = 1$) ed ogni funzione f, $cf(n)$ è $O(f(n))$ (lo stesso vale per Ω e per Θ);

(2) *Transitività*: Se $g(n)$ è $O(f(n))$ ed $f(n)$ è $O(h(n))$, allora $g(n)$ è $O(h(n))$ (lo stesso vale per Ω e per Θ);

(3) *Simmetria*: $g(n)$ è $\Theta(f(n))$ se e solo se $f(n)$ è $\Theta(g(n))$;

(4) *Simmetria trasposta*: $g(n)$ è $O(f(n))$ se e solo se $f(n)$ è $\Omega(g(n))$;

(5) *Somma*: La funzione $f(n) + g(n)$ è $O(max\{f(n), g(n)\})$ (lo stesso vale per Ω e per Θ);

(6) *Prodotto*: Se $g(n)$ è $O(f(n))$ ed $h(n)$ è $O(q(n))$, allora la funzione $g(n)h(n)$ è $O(f(n)q(n))$ (lo stesso vale per Ω e per Θ).

2.4 ESEMPIO – [continua]
La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché $4n$ è $O(n)$ per la regola (1), $4n^2 = 4nn$ è $O(nn)$ per la regola (6), ed infine $4n^2 + 4n - 1$ è $O(\max\{n^2, n, 1\})$ applicando due volte la (5).

Si noti che le proprietà 1, 2, e 3 implicano che Θ definisce una relazione di equivalenza sulle funzioni, tale che ogni insieme $\Theta(f(n))$ è una classe di equivalenza (detta *classe di complessità*).

[1] Dato che $\Theta(f(n))$ è un insieme, alcuni autori, per indicare che $g(n)$ è $\Theta(f(n))$, utilizzano correttamente la notazione $g(n) \in \Theta(f(n))$, mentre altri usano impropriamente la notazione $g(n) = \Theta(f(n))$, intendendo esprimere con ciò che $\Theta(g(n)) = \Theta(f(n))$.

ESEMPIO – [Classificazione ordini di grandezza]
I seguenti ordini di grandezza sono via via crescenti: $\Theta(1)$ (ordine costante), $\Theta(\log n)$ (logaritmico), $\Theta(n)$ (lineare), $\Theta(n \log n)$ (pseudolineare), $\Theta(n^2)$ (quadratico), $\Theta(n^3)$ (cubico), $\Theta(2^n)$ (esponenziale in base 2), $\Theta(n^n)$ (esponenziale in base n). È possibile tracciare delle relazioni di inclusione anche più generali, per qualsiasi $r < s$, $h < k$ e $1 < a < b$:

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

In generale, un ordine $\Theta(n^k)$, con k costante positiva, viene detto *polinomiale*, mentre $\Theta(a^n)$, con a costante maggiore di 1, è detto *esponenziale*. Un ordine esponenziale o maggiore è detto *superpolinomiale*.

Le definizioni degli ordini di grandezza valgono per funzioni qualsiasi, ma nella valutazione della complessità di algoritmi sono applicate a funzioni come la $T(n)$ che contano il numero di operazioni nel caso pessimo (o medio) eseguite su tutti i possibili dati di ingresso di dimensione n. Poiché contano operazioni, tali funzioni $T(n)$ di complessità sono a valori non negativi (e quindi i loro ordini di grandezza verificano le proprietà (1)-(6) su menzionate).

Utilizzando gli ordini di grandezza, ogni operazione elementare costa $O(1)$ tempo. Le uniche istruzioni che non danno un contributo unitario sono ovviamente quelle condizionali, di iterazione (limitata o illimitata) e le chiamate di procedure e funzioni. Per esempio, la complessità della seguente istruzione condizionale è $O(f(n) + \max\{g(n), h(n)\})$ per la regola (5):

if condizione che richiede $O(f(n))$ tempo
  then qualcosa che richiede $O(g(n))$ tempo
  else qualcosa che richiede $O(h(n))$ tempo;

Invece, la complessità del seguente brano contenente tre costrutti iterativi, i primi due in sequenza, ma il terzo annidato nel secondo, è $O(\max\{ n \cdot f(n), n \cdot m \cdot g(m)\})$ per le regole (5) e (6):

for $i \leftarrow 1$ **to** n **do**
  qualcosa che richiede $O(f(n))$ tempo;
for $j \leftarrow 1$ **to** n **do**
  for $k \leftarrow 1$ **to** m **do**
    qualcosa che richiede $O(g(m))$ tempo;

A questo punto, può sorgere una domanda legittima: poiché si trascurano le costanti, è sempre lecito preferire, tra due algoritmi, quello avente complessità di ordine più basso? Ad esempio, un algoritmo di complessità $\Theta(n^2)$ è da preferire ad un altro di complessità $\Theta(n^3)$, anche se la costante moltiplicativa nel primo è 200 e quella del secondo è 4? La risposta dipende dalla dimensione attesa dell'input. Per $n < 50$ il secondo algoritmo è più veloce del primo; pertanto se l'algoritmo deve essere eseguito sempre su dati di piccola dimensione si

sceglierà quello con tempo $\Theta(n^3)$. Se invece sono previsti dati di dimensione grande, il rapporto tra i tempi di esecuzione, che è $4n^3/200n^2 = n/50$, cresce arbitrariamente con n e l'algoritmo di complessità $\Theta(n^2)$ risulta nettamente il migliore. Poiché in generale l'algoritmo deve poter essere eseguibile su dati di dimensione arbitraria, risulta sensato e conveniente scegliere sempre l'algoritmo la cui complessità è di ordine più basso possibile.

2.3 ALGORITMI EFFICIENTI E INEFFICIENTI

Per un singolo problema, è possibile progettare un gran numero di algoritmi diversi, ognuno caratterizzato dal proprio tempo di calcolo. Gli algoritmi con complessità superpolinomiale sono considerati inefficienti e non sono accettabili, a meno che non sia possibile dimostrare che il problema è inerentemente "difficile" [cfr. capitolo 18]. Una volta realizzato un algoritmo con complessità polinomiale, si cerca di migliorarne le prestazioni progettando algoritmi di complessità sempre inferiore. Per illustrare questi concetti, affrontiamo un problema molto importante, alla base della risoluzione di molti problemi che vedremo in seguito: l'ORDINAMENTO.

ORDINAMENTO (SORTING) Data una sequenza di valori $a_1, \ldots, a_n$, trovare una permutazione π di $\{1, \ldots, n\}$ tale che $a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n}$.

È possibile trarre ispirazione dalla definizione del problema, come già visto nel capitolo 1, anche se spesso l'algoritmo risultante non è molto efficiente. Supponiamo che la sequenza sia memorizzata in un vettore $A[1 \ldots n]$. Un semplice algoritmo è il seguente:

Genera tutte le permutazioni degli n elementi dell'array e verifica, per ciascuna permutazione, se per ogni coppia di elementi adiacenti $A[i]$ e $A[i+1]$ risulti $A[i] \leq A[i+1]$.

Poiché verificare che un array sia ordinato richiede $\Theta(n)$ tempo (basta un ciclo **for**) e il numero di permutazioni possibili è $n!$, la complessità di questo algoritmo è $\Theta(nn!)$, che è superpolinomiale.

Un semplice algoritmo polinomiale invece è il *Selection Sort*, che è basato sulla proprietà che in una sequenza ordinata il primo elemento ha valore minimo:

Trova il minimo elemento tra gli n iniziali e scambialo con quello che occupa la prima posizione; ripeti il procedimento sui restanti $n - 1$ elementi, poi sui rimanenti $n - 2$ e così via fino ad aver esaurito gli elementi.

Tale algoritmo è realizzato dalla procedura selectionSort(), la quale richiama una nuova versione iterativa di min() che restituisce la posizione dell'elemento minimo nella porzione di array $A[i \ldots n]$, per $i = 1, 2, \ldots, n$, invece che il suo valore:

```
selectionSort(ITEM[] A, integer n)

for i ← 1 to n do
    integer j ← min(A, i, n)
    A[i] ↔ A[j]

integer min(ITEM[] A, integer k, integer n)
    integer min ← k                          % Posizione del minimo parziale
    for h ← k + 1 to n do
        if A[h] < A[min] then min ← h        % Nuovo minimo parziale
    return min
```

La procedura selectionSort() esegue il ciclo **for** n volte. All'iterazione i viene chiamata la funzione min() che a sua volta esegue un altro **for** $n - i$ volte. Il tempo complessivo di calcolo della selectionSort() cresce, a meno di costanti, come

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

La complessità di selectionSort() è quindi $\Theta(n^2)$.

Vediamo ora l'algoritmo *Insertion Sort*, ispirato a quello utilizzato per ordinare una mano di ramino o scopone:

> Considera le carte una per volta consecutivamente, per esempio da sinistra verso destra: ogni volta che viene considerata una nuova carta, inseriscila nella posizione giusta rispetto alle altre carte già considerate e ordinate, traslando di una posizione verso destra tutte le carte maggiori.

```
insertionSort(ITEM[] A, integer n)

for i ← 2 to n do
    ITEM temp ← A[i]
    integer j ← i
    while j > 1 and A[j − 1] > temp do
        A[j] ← A[j − 1]
        j ← j − 1
    A[j] ← temp
```

La procedura insertionSort() considera, ad ogni iterazione, la "carta" $temp = A[i]$ da inserire, per $i = 2, 3, \ldots, n$. La porzione $A[1 \ldots i-1]$ del vettore è già ordinata ed è scandita a ritroso con l'indice j, traslando di una posizione a destra (da $A[j-1]$ ad $A[j]$) ogni elemento maggiore di $temp$, creando così uno spazio per esso. Al termine di ciascuna iterazione, $temp$ è inserito in $A[j]$ nello spazio che si è creato.

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

Fig. 2.2 Esecuzione della procedura insertionSort().

2.6 ESEMPIO – [Insertion sort]
La Fig. 2.2 mostra l'ordinamento della sequenza di elementi 7, 4, 2, 1, 8, 3, 5 con la procedura insertionSort().

La complessità della procedura dipende dalla disposizione iniziale degli elementi. Il caso peggiore si ha quando il vettore A è ordinato alla rovescia; in tal caso, ad ogni iterazione del **for**, l'elemento *temp* deve essere sempre trasferito in prima posizione, e questo richiederà i spostamenti e la complessità della procedura è $O(n^2)$. Per sequenze già ordinate, però, la procedura insertionSort() è $\Theta(n)$, poiché la condizione "$A[j-1] > temp$" è sempre falsa.

Possiamo fare di meglio, diminuendo la complessità in tutti i casi, e non solo nel caso ottimo? La risposta è positiva. L'algoritmo *Merge Sort* è basato sull'idea seguente (Fig. 2.3-2.4):

> Dividi il vettore in due sequenze di $n/2$ elementi ciascuno, ordina ciascuna sequenza, e poi "fondi" le due metà ordinate in un'unica sequenza ordinata.

La procedura ricorsiva MergeSort(), da chiamare con l'istruzione MergeSort(A, 1, n), realizza tale algoritmo.

```
MergeSort(ITEM[] A, integer primo, integer ultimo)
  if primo < ultimo then
    integer mezzo ← ⌊(primo + ultimo)/2⌋
    MergeSort(A, primo, mezzo)
    MergeSort(A, mezzo + 1, ultimo)
    Merge(A, primo, ultimo, mezzo)
```

Per fondere le due metà ordinate, la MergeSort() chiama la procedura Merge(), che si avvale di un vettore di appoggio B, utilizzato come parametro globale. La procedura Merge() usa tre indici i, j e k per scandire, rispettivamente, $A[primo \ldots mezzo]$, $A[mezzo + 1 \ldots ultimo]$ e $B[primo \ldots ultimo]$. Ad ogni passo, sono confrontati gli elementi $A[i]$ e $A[j]$, il minore dei due è copiato in $B[k]$, e sono incrementati k e l'indice dell'elemento minore. Il procedimento è iterato finché una delle due metà $A[primo \ldots mezzo]$ e $A[mezzo + 1 \ldots ultimo]$ è esaurita. Se la prima metà è stata esaurita (cioè $i > mezzo$), gli eventuali elementi non scanditi $A[j \ldots ultimo]$ della seconda metà si trovano già nelle posizioni che competono loro nell'ordinamento finale di $A[primo \ldots ultimo]$. Se invece la prima metà non è stata esaurita (cioè $i \leq mezzo$), gli elementi non scanditi $A[i \ldots mezzo]$ della prima metà vengono subito spostati nelle ultime posizioni $A[k \ldots ultimo]$ che competono loro nell'ordinamento finale. Infine, la porzione $B[primo \ldots k-1]$ è ricopiata in $A[primo \ldots k-1]$, ottenendo così $A[primo \ldots ultimo]$ ordinato.

```
Merge(A[], integer primo, integer ultimo, integer mezzo)
  integer i, j, k, h
  i ← primo; j ← mezzo + 1; k ← primo
  while i ≤ mezzo and j ≤ ultimo do
      if A[i] ≤ A[j] then
          B[k] ← A[i]
          i ← i + 1
      else
          B[k] ← A[j]
          j ← j + 1
      k ← k + 1
  j ← ultimo
  for h ← mezzo downto i do
      A[j] ← A[h]
      j ← j − 1
  for j ← primo to k − 1 do A[j] ← B[j]
```

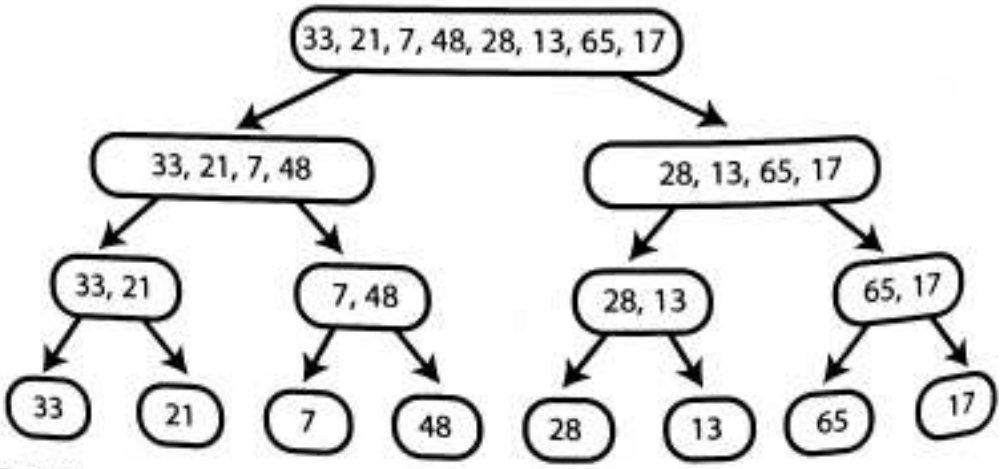


Fig. 2.3 Partizionamento dei dati con la procedura MergeSort.

ESEMPIO – [Partizione dei dati con Merge Sort]

Si supponga di eseguire la procedura Mergesort per i dati d'ingresso: $A[1] = 33$, $A[2] = 21$, $A[3] = 7$, $A[4] = 48$, $A[5] = 28$, $A[6] = 13$, $A[7] = 65$, $A[8] = 17$. Il partizionamento dei dati corrispondente è visualizzato nella Fig. 2.3.

ESEMPIO – [Computazione Merge Sort]

L'ordine in cui vengono ricostruiti gli insiemi è visualizzato nella Fig. 2.4. Al livello più basso, tutti i sottovettori costituiti da un unico elemento sono già ordinati. Una coppia di sottovettori viene fusa nel sottovettore superiore, indicato dalle frecce. Al termine, il vettore completo è ordinato.

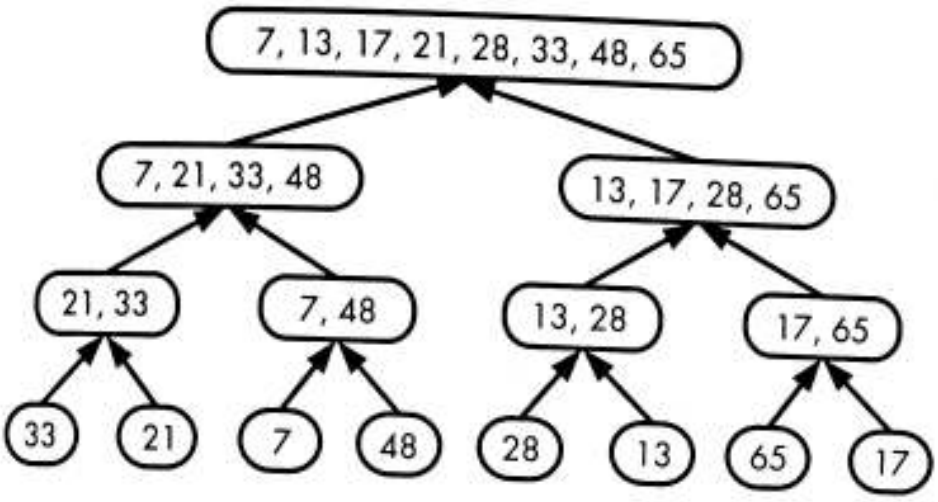


Fig. 2.4 Fusione dei dati con la procedura MergeSort.

Poiché la procedura Merge() è $O(n)$, la complessità $T(n)$ di Mergesort() è $O(n \log n)$. Infatti, assumendo per semplicità $n = 2^m$ e indicando con c e d due costanti positive, si ottiene:

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + dn & \text{se } n \geq 2 \end{cases}$$

Sostituendo successivamente, si ricava:

$$\begin{aligned} T(n) &\leq 2T(n/2) + dn \\ &\leq 2(2T(n/2^2) + dn/2) + dn \\ &\leq 2(2(2T(n/2^3) + dn/2^2) + dn/2) + dn \\ &\vdots \\ &\leq 2^m T(n/2^m) + 2^{m-1} dn/2^{m-1} + \ldots + 2^2 dn/2^2 + 2dn/2 + dn \\ &= 2^m c + dmn \end{aligned}$$

Essendo $m = \log n$, $T(n)$ risulta essere $O(n \log n)$.

A questo punto, abbiamo progettato un discreto numero di algoritmi per l'ordinamento

(ma non è finita qui: altri esempi verranno discussi in seguito). Per avere una visualizzazione tangibile della differenza di comportamento fra questi algoritmi, si osservi la Fig. 2.5. Assumendo che un'istruzione elementare sia eseguita ogni nanosecondo, come accade in un calcolatore elettronico attuale, il tempo necessario per ordinare un milione di interi per le procedure suggerite finora è il seguente:

- l'algoritmo banale basato su tutte le permutazioni richiede $\approx 2^{10^6}$ nanosecondi, un numero che ha oltre 300 000 cifre decimali. Come termine di paragone, si consideri che il numero di nanosecondi dal big-bang ad ora ha "solo" 27 cifre (il big-bang sarebbe avvenuto circa 15 miliardi di anni fa);
- selectionSort() e insertionSort() nel caso pessimo richiedono $\approx (10^6)^2$ nanosecondi, ovvero più di 16 minuti;
- Mergesort() richiede $\approx 10^6 \log_2 10^6$ nanosecondi, ovvero circa 2 centesimi di secondo;
- insertionSort() nel caso ottimo richiede 10^6 nanosecondi, ovvero un millesimo di secondo; ma questo avviene solo se il vettore è già ordinato.

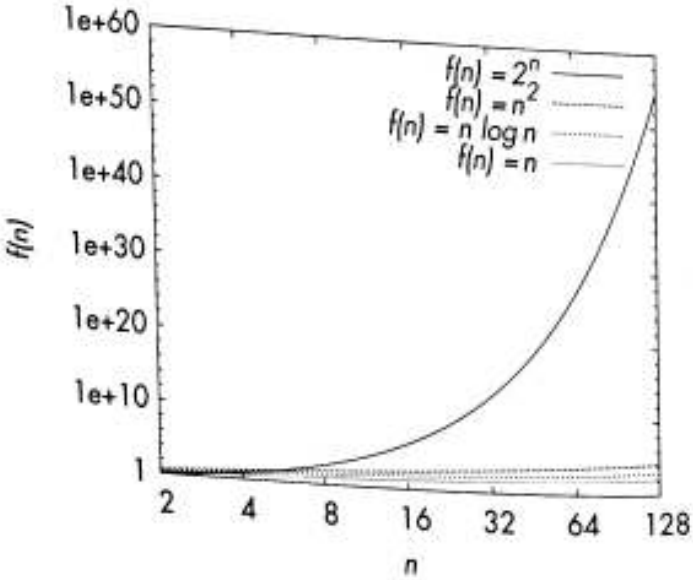


Fig. 2.5 Crescita di alcune funzioni (in scala logaritmica). Le tre rette scarsamente distanziate rappresentano le tre funzioni polinomiali n, $n \log n$, n^2 (in ordine, dal basso verso l'alto). La linea curva rappresenta la funzione esponenziale.

La lezione che ne possiamo trarre è che gli algoritmi superpolinomiali sono ovviamente da scartare e i polinomi di grado inferiore sono sempre da preferire.

La situazione non diverrebbe più rosea neanche se nei prossimi anni fossero realizzati calcolatori elettronici 100 o anche 1000 volte più veloci: per funzioni di complessità superpolinomiale la dimensione massima di un problema risolubile in un dato lasso di tempo non aumenterebbe che di pochissime unità. Per esempio, se D è la dimensione massima di un problema risolubile in un giorno con un algoritmo di complessità $\Theta(2^n)$, con un calcolatore 1000 volte più veloce la nuova dimensione massima D' non supera $D + 10$, perché da

$2^{D'} = 1000 \cdot 2^D$ segue passando ai logaritmi che $D' = D + \log 1000$. In altri termini, moltiplicando la funzione di complessità per una costante, la dimensione massima risolvibile praticamente non cambia (questo spiega intuitivamente perché si possono trascurare le costanti nel valutare la complessità). La conclusione che se ne trae è univoca: algoritmi superpolinomiali sono e resteranno sempre estremamente inefficienti e non utilizzabili in pratica. Naturalmente, un algoritmo con complessità polinomiale di grado elevato, per esempio uno che richieda tempo $\Theta(n^{1000})$, pur essendo più veloce di uno con complessità $\Theta(2^n)$ per n che tende all'infinito, sarebbe in pratica inefficiente quanto uno superpolinomiale. Con l'eccezione di appositi casi costruiti artificiosamente, però, i problemi risolvibili con algoritmi di complessità polinomiale richiedono un grado basso del polinomio (raramente più di 3) per cui è sensato considerare efficienti gli algoritmi polinomiali.

2.4 COMPLESSITÀ DI PROBLEMI E ALGORITMI OTTIMI

Per risolvere un problema, si cerca di scoprire algoritmi di complessità sempre più bassa. Fino a quale punto questa ricerca può essere spinta? In altri termini, esiste una limitazione inferiore alla complessità che dipende solo dal problema in esame, che stabilisca una soglia invalicabile alla complessità di ogni algoritmo, anche non noto, che risolva il problema? La risposta è sì. Se si riesce a dimostrare che qualunque algoritmo per il problema in esame deve avere complessità $\Omega(f(n))$, allora si è stabilita una limitazione inferiore alla complessità del problema. Se invece si trova un particolare algoritmo di complessità $O(g(n))$, allora si è stabilita una limitazione superiore alla complessità del problema. Se $f(n) = g(n)$, allora l'algoritmo è detto *ottimo*, perché la sua complessità è, in ordine di grandezza, la migliore possibile. In generale, trovare limitazioni inferiori significative è molto più difficile che trovare algoritmi efficienti. Questo perché non si deve progettare in modo costruttivo un particolare algoritmo che risolva il problema, ma occorre invece fornire una prova matematica generale (cioè dimostrare un teorema) che valga per qualsiasi algoritmo che possa mai venire ideato anche in futuro. Purtroppo, sono noti pochissimi metodi generali di dimostrazione per limitazioni inferiori, per lo più poco potenti. Vediamone tre molto semplici (altri due più potenti, basati su alberi di decisione e riduzioni, saranno discussi in seguito):

(1) *dimensione dei dati*: se un problema ha in ingresso n dati e richiede di esaminarli tutti, allora una limitazione inferiore della complessità è $\Omega(n)$;

(2) *eventi contabili*: se un problema richiede che un certo evento sia ripetuto almeno n volte, allora una limitazione inferiore della complessità è $\Omega(n)$;

(3) *oracolo*: se un oracolo (o avversario), utilizzando una certa regola che l'algoritmo non conosce e che vale solo per certi dati d'ingresso, "divina" ad ogni opportunità la situazione più sfavorevole e fa lavorare l'algoritmo il più possibile, allora combattendo contro di esso si può individuare una limitazione inferiore della complessità.

2.9 ESEMPIO – [Dimensione dei dati]
Si consideri il problema di sommare due interi, composti ciascuno da n bit e memorizzati in due vettori di lunghezza n. In questo caso, è necessario considerare tutti i $2n$ bit per effettuare la somma. Questa affermazione può essere dimostrata per assurdo. Supponiamo di possedere un algoritmo "magico" in grado di calcolare correttamente la somma senza esaminare tutti i bit. Si esegua l'algoritmo su una coppia di numeri x e y. Deve esistere qualche bit in posizione i che non viene esaminato in uno dei due numeri, diciamo x. Cambiamo il valore del bit i-esimo di x e diamo all'algoritmo la nuova coppia di numeri. Poiché il bit i-esimo non viene esaminato, la risposta sarà la stessa di prima, ma avendo cambiato gli addendi, o era sbagliata prima o è sbagliata adesso. Questo è assurdo perché abbiamo assunto che l'algoritmo fosse corretto.

2.10 ESEMPIO – [Eventi contabili]
Il problema della ricerca del minimo elemento in un insieme di n interi richiede almeno $n - 1$ confronti. Infatti, il minimo può essere uno qualsiasi tra gli n elementi. Ciascuno dei rimanenti $n - 1$ elementi deve essere scartato utilizzando almeno un confronto, poiché altrimenti non si potrebbe dire che tale elemento è il minimo. Una limitazione inferiore al numero di confronti necessari per trovare il minimo è quindi $\Omega(n)$. La procedura min(), che usa $\Theta(n)$ confronti, è dunque ottima.

2.11 ESEMPIO – [Oracolo]
Si consideri il problema di "fondere" due sequenze ordinate, ciascuna di n elementi interi, in un'unica sequenza ordinata di $2n$ elementi. Siano $X[1], X[2], \ldots, X[n]$ ed $Y[1], Y[2], \ldots, Y[n]$ le due sequenze, con $X[1] < X[2] < \cdots < X[n]$ ed $Y[1] < Y[2] < \cdots < Y[n]$, e siano i $2n$ elementi tutti distinti. Si consideri la seguente regola dell'oracolo:

$$X[i] < Y[j] \quad \text{se e solo se} \quad i < j$$

Tale regola ovviamente non vale per tutti i dati di ingresso, ma solo per alcuni. Un qualsiasi algoritmo che risolva il problema per tutti i possibili dati di ingresso deve ovviamente poterlo risolvere anche nel caso speciale nel quale valga la regola dell'oracolo. Se l'oracolo risolve il problema, la soluzione *deve* essere:

$$Y[1], X[1], Y[2], X[2], \ldots, Y[n], X[n].$$

Se un qualsiasi algoritmo, nel risolvere il problema, non eseguisse un confronto, ad es. quello tra $X[1]$ e $Y[2]$, allora potrebbe produrre la seguente sequenza, che sarebbe così indistinguibile da quella prodotta dall'oracolo:

$$Y[1], Y[2], X[1], X[2], Y[3], X[3], \ldots, Y[n], X[n].$$

Per produrre la sequenza dell'oracolo, l'algoritmo deve effettuare tutti i $2n - 1$ confronti tra elementi adiacenti della sequenza stessa. Pertanto $\Omega(n)$ è una limitazione inferiore al numero di confronti necessari per risolvere il problema, e quindi la procedura Merge() è ottima.

Le precedenti tecniche di dimostrazione, ancorché apparentemente semplici, nascondono sempre insidie e sottigliezze. Per esempio, con la dimensione dei dati occorre fare molta

attenzione, perché per molti problemi non è affatto necessario esaminare tutti i dati in ingresso. Inoltre, come sempre quando si dimostra un teorema, bisogna aver ben chiare quali siano le ipotesi sotto le quali si agisce. Chiariamo questa affermazione discutendo della limitazione inferiore del problema dell'ORDINAMENTO.

Per ordinare un vettore di n elementi, è ovviamente necessario esaminarli tutti. Infatti, anche se il vettore di ingresso fosse già ordinato, occorrerebbe comunque verificare l'ordinamento con un confronto tra ogni coppia di elementi adiacenti. Tale problema ha quindi complessità $\Omega(n)$.

Abbiamo quindi una limitazione inferiore $\Omega(n)$ e un algoritmo che opera in tempo $O(n \log n)$. Di fronte a questo scarto ci sono due possibilità: o esiste una limitazione inferiore più stretta, oppure esiste un algoritmo più efficiente. Nel capitolo 5, una volta introdotto il concetto di albero, saremo in grado di dimostrare che il problema dell'ordinamento è $\Omega(n \log n)$ se si utilizzano algoritmi che confrontano i valori, e quindi Mergesort() è ottimo. Potremmo allora interrompere qui la nostra ricerca del miglior algoritmo di ordinamento, perché fatto salvo per le costanti moltiplicative, lo abbiamo già trovato!

Tuttavia, supponiamo che gli elementi da ordinare, contenuti nel vettore A, siano interi compresi fra 1 e k. L'algoritmo *Counting Sort* inizialmente scandisce A e, avvalendosi di un vettore di appoggio $B[1 \ldots k]$, registra in $B[A[j]]$ il numero di occorrenze del valore $A[j]$, per $j = 1, 2, \ldots, n$. Successivamente, la procedura scandisce il vettore B, per $i = 1, 2, \ldots, k$, e scrive il valore i nel vettore A per $B[i]$ volte.

```
countingSort(integer[] A, integer n, integer k)
  integer[] B ← new integer[1 ... k]
  for i ← 1 to k do B[i] = 0
  for j ← 1 to n do B[A[j]] ← B[A[j]] + 1
  j ← 1
  for i ← 1 to k do
      while B[i] > 0 do
          A[j] ← i
          j ← j + 1
          B[i] ← B[i] − 1
```

Poiché la procedura scandisce vettori di k ed n elementi, il tempo richiesto è $O(n + k)$. Se k è $O(n)$, Counting Sort ha complessità $O(n)$ e risulta più vantaggioso di Merge Sort.

È bene notare che la limitazione superiore $O(n)$ non è in contrasto con la limitazione inferiore $\Omega(n \log n)$ sul numero di confronti tra elementi da ordinare. Infatti, Counting Sort non effettua confronti tra elementi di A. Inoltre, sono cambiate le ipotesi di base. Merge Sort è in grado di operare su tutti i possibili insiemi di valori, mentre Counting Sort è in grado di operare solo in un ristretto insieme di casi. Se k è $\Omega(n \log n)$, allora Counting Sort non è più conveniente degli algoritmi di ordinamento che hanno complessità $O(n \log n)$ ed anzi può essere molto svantaggioso: per esempio, se k è $\Theta(2^n)$ allora la complessità è superpolinomiale!

2.5 TECNICHE DI ANALISI

Analizzare la complessità di un algoritmo non sempre è un compito agevole. Le maggiori difficoltà si incontrano generalmente nell'analisi di algoritmi ricorsivi, nell'analisi del caso medio, e nell'analisi "ammortizzata" di una sequenza di operazioni. Fortunatamente, sono state individuate alcune tecniche che risultano di grandissima utilità nell'analisi di algoritmi.

Per analizzare la complessità di un algoritmo ricorsivo, occorre impostare relazioni di ricorrenza e individuarne l'ordine di grandezza. Ci sono tre metodi diversi per risolvere tali relazioni:

(1) espandere le relazioni stesse, sostituendo successivamente nella relazione di partenza scritta per $T(n)$ quella riscritta, per esempio, per $T(n-1)$, poi quella per $T(n-2)$, e così via finché, chiarita la struttura della somma così ottenuta, non resta che una formula "chiusa" per $T(n)$ che comprende termini in n e costanti;

(2) usare la soluzione generale di alcune relazioni di ricorrenza "comuni" che sono espresse in forma parametrica, e ricavare la soluzione di $T(n)$ semplicemente sostituendo al posto dei parametri i valori costanti che appaiono nella relazione per $T(n)$;

(3) "tentare" con una soluzione $f(n)$, lasciando qualche parametro non specificato, e dimostrare, generalmente per induzione su n, che $T(n)$ è $O(f(n))$ oppure $\Omega(f(n))$.

In mancanza di metodi più generali, si può sempre applicare la tecnica (1), che in linea di principio garantisce di trovare una soluzione, ma che in pratica può richiedere di risolvere o maggiorare sommatorie molto complicate. Nel seguito verrà trattata una versione particolare della tecnica (1), detta *per livelli*, e verrano introdotte le altre due.

Per l'analisi del caso medio, purtroppo, sono note poche tecniche generali e le formule che vengono impostate per analizzare la complessità sono in genere di difficilissima risoluzione. Vedremo due esempi di analisi "trattabili" introducendo una struttura di dati chiamata *tabella hash* [cfr. capitolo 7] ed un ulteriore algoritmo per l'ordinamento chiamato Quicksort [cfr. capitolo 12].

Infine, per l'analisi ammortizzata di una sequenza di operazioni su una struttura di dati, vedremo alcune tecniche molto semplici, quali il metodo di *aggregazione*, quello degli *accantonamenti* e quello del *potenziale*.

2.6 ANALISI PER LIVELLI

In questa tecnica di risoluzione di ricorrenze, l'espressione viene via via espansa per "livelli": al livello 0, abbiamo l'espressione originale; al livello 1, l'espressione a cui è stata applicata un'espansione; al livello 2, l'espressione a cui sono state applicate due espansioni e così via fino al caso base. Per illustrare questo procedimento, si consideri la ricorrenza seguente:

$$T(n)=\begin{cases} 1 & \text{per } n=1 \\ 4T(n/2)+n^2 & \text{per } n=2^h, h>0 \end{cases}$$

Ad ogni livello (escluso l'ultimo) il costo da pagare effettivamente è dovuto alla componente n^2 della formula ricorsiva; varia tuttavia la dimensione di n, che viene via via ridotta.

Al livello 0, il costo della ricorrenza è pari a n^2, dovuto al termine non ricorsivo. Al livello 1, la dimensione n è stata divisa in 4 parti di dimensione $n/2$; con la ricorsione, il termine n^2 applicato a $n/2$ dà origine a $(n/2)^2 = n^2/2^2$. Al livello 2, la dimensione n è stata divisa in 16 parti di dimensione $(n/4)^2$. In generale, al livello i-esimo la dimensione è divisa in 4^i parti di costo $(n/2^i)^2 = n^2/2^{2i}$. La ricorsione termina al livello $h = \log_2 n$, ovvero quando $n/2^h = 1$. Graficamente, l'espressione può essere visualizzata nel modo seguente:

Livello	Espressione	Costo
0	n^2	$4^0 n^2$
1	$\frac{n^2}{2^2}\ \frac{n^2}{2^2}\ \frac{n^2}{2^2}\ \frac{n^2}{2^2}$	$4^1 \frac{n^2}{2^2}$
2	$\frac{n^2}{2^4}\ \frac{n^2}{2^4}\ \cdots\ \frac{n^2}{2^4}\ \frac{n^2}{2^4}$	$4^2 \frac{n^2}{2^4}$
	$\cdots$	
i	$\frac{n^2}{2^{2i}}\ \frac{n^2}{2^{2i}}\ \cdots\ \frac{n^2}{2^{2i}}\ \frac{n^2}{2^{2i}}$	$4^i \frac{n^2}{2^{2i}}$
	$\cdots$	
h	$T(1)\ \ T(1)\ \cdots\ T(1)\ \ T(1)$	4^h

L'ultima riga costa $4^h\, T(1) = 4^{\log n}\, T(1) = \Theta(n^2)$. Sommando le righe precedenti si ottiene:

$$\sum_{i=0}^{\log n-1} n^2/2^{2i} \cdot 4^i = n^2 \sum_{i=0}^{\log n-1} \frac{2^{2i}}{2^{2i}} = n^2 \sum_{i=0}^{\log n-1} 1 = n^2 \log n$$

da cui si ricava:

$$T(n) = n^2 + n^2 \log n = O(n^2 \log n)$$

2.7 RISOLUZIONE DI RELAZIONI DI RICORRENZA "COMUNI"

In molti algoritmi ricorsivi accade spesso che l'algoritmo richiami se stesso un numero costante a_i di volte su dati di dimensione $n - i$, con i costante. Le chiamate ricorsive possono essere effettuate prima o dopo aver eseguito un numero polinomiale di operazioni. In tal caso, la funzione di complessità $T(n)$ dell'algoritmo può essere espressa in termini di $a_i T(n - i) + cn^\beta$. I parametri a_i e i sono costanti intere tali che $a_i \geq 1$, perché viene fat-

ta almeno una chiamata ricorsiva, e $i \geq 1$, perché la dimensione del problema deve essere diminuita di almeno uno. Invece, i parametri c e β sono costanti reali tali che $c > 0$ e $\beta \geq 0$, in quanto è eseguita almeno un'operazione oltre alle chiamate ricorsive.

Generalizzando, date $a_1, a_2, ..., a_h$ costanti intere non negative, con h costante positiva, si considerano relazioni di ricorrenza nelle quali $T(n)$ dipende da $\sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta$, con c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Una relazione di ricorrenza di questo tipo è detta *lineare* (perché n compare nei $T(n-i)$ con grado 1), a *coefficienti costanti* (essendo $a_1, a_2, ..., a_h$ costanti), di *ordine costante* (poiché anche h è una costante), e con *lavoro polinomiale di suddivisione e/o ricombinazione* (in quanto cn^β è un polinomio). Si ottiene il seguente risultato.

TEOREMA 1 – [Ricorrenze lineari di ordine costante]
Siano $a_1, a_2, ..., a_h$ costanti intere non negative, con h costante positiva, c e β costanti reali tali che $c > 0$ e $\beta \geq 0$, e sia $T(n)$ definita dalla relazione di ricorrenza:

$$T(n) = \begin{cases} \text{costante} & \text{per } n \leq m \leq h \\ \sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & \text{per } n > m \end{cases}$$

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

(1) $T(n)$ è $O(n^{\beta+1})$, se $a = 1$

(2) $T(n)$ è $O(a^n n^\beta)$, se $a \geq 2$

DIMOSTRAZIONE

Si supponga dapprima che ci sia un solo coefficiente non nullo, ovvero esista j, $1 \leq j \leq h$, tale che $a_j = a$, mentre $a_i = 0$, per $1 \leq i \leq h$ ed $i \neq j$. Si assuma per semplicità che n sia uguale ad $m + kj$, per un k intero. Sostituendo successivamente si ottiene:

$$\begin{aligned} T(n) &= aT(n-j) + cn^\beta \\ &= a\left(aT(n-2j) + c(n-j)^\beta\right) + cn^\beta \\ &\vdots \\ &= a\left(a \ldots a\left(a\left(T(m) + c(m+j)^\beta\right) + c(m+2j)^\beta\right) + \cdots + c(n-j)^\beta\right) + cn^\beta \\ &= a^k T(m) + \sum_{0 \leq v \leq k-1} a^v c(n-vj)^\beta \\ &\leq a^k T(m) + cn^\beta \sum_{0 \leq v \leq k-1} a^v \end{aligned}$$

Si individuano due casi possibili:

(1) Se $a = 1$, allora:

$$\begin{aligned} T(n) &\le T(m) + cn^{\beta}[1+1+\cdots+1] \\ &= T(m) + cn^{\beta}k \\ &= T(m) + cn^{\beta}(n-m)/j \end{aligned}$$

e quindi $T(n)$ è $O(n^{\beta+1})$.

(2) Se $a \ge 2$, allora:

$$\begin{aligned} T(n) &= a^k T(m) + cn^{\beta}\left[\left(a^k - 1\right)/(a-1)\right] \\ &\le a^k\left[T(m) + cn^{\beta}\right] \\ &= a^{(n-m)/j}\left[T(m) + cn^{\beta}\right] \end{aligned}$$

e quindi $T(n)$ è $O(a^n n^{\beta})$.

Per completare la dimostrazione, si consideri il caso più generale in cui $\prod_{i=1}^{h} a_i \neq 0$, cioè tutti gli h coefficienti sono non nulli, con $h > 1$. In tal caso risulta necessariamente $\sum_{i=1}^{h} a_i = a \ge 2$. Pertanto, si ha:

$$\begin{aligned} T(n) &= \sum_{1 \le i \le h} a_i T(n-i) + cn^{\beta} \\ &\le \sum_{1 \le i \le h} a_i T(n-1) + cn^{\beta} \\ &= aT(n-1) + cn^{\beta} \end{aligned}$$

e ci si riconduce al caso precedente con $j = h = 1$ e $a_j = a \ge 2$.
Infine, la dimostrazione è analoga anche per $n \neq m + kj$.

2.42 ESEMPIO – [Quicksort]
Per la procedura Quicksort(), che vedremo nel capitolo 12, le relazioni di ricorrenza, nel caso pessimo, sono del tipo: $T(n) = T(n-1) + cn$. Poiché $a = 1$ e $\beta = 1$, si ha che $T(n)$ è $O(n^2)$.

2.43 ESEMPIO – [Torri di Hanoi]
Per la procedura hanoi-ricorsiva(), che vedremo nel capitolo 4, si ottengono relazioni del tipo: $T(n) \le 2T(n-1) + c$. Poiché $a = 2$ e $\beta = 0$, ne segue che $T(n)$ è $O(2^n)$.

Per algoritmi di tipo *divide-et-impera* [capitolo 12] in cui si partizionino i dati in modo bilanciato, il problema originario di dimensione n è diviso in a sottoproblemi di dimensione n/b ciascuno. Se si dividono i dati e/o si ricombinano i risultati in tempo polinomiale, la funzione di complessità $T(n)$ può essere espressa in termini di $aT(n/b) + cn^{\beta}$. I parametri a e b sono costanti intere tali che $a \ge 1$, perché è fatta almeno una chiamata, e $b \ge 2$, in quanto il problema è suddiviso in almeno due sottoproblemi, mentre c e β sono, come prima, costanti reali tali che $c > 0$ e $\beta \ge 0$. Una relazione di ricor-

renza di questo tipo è ancora lineare, a coefficienti costanti, e con lavoro polinomiale di suddivisione e ricombinazione, ma non è più di ordine costante, bensì con *partizione bilanciata* (perché n è diviso per una costante b). In questo caso, si ottiene il seguente risultato.

TEOREMA 2 – [Ricorrenze lineari con partizione bilanciata]
Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$, e c, d e β costanti reali tali che $c > 0$, $d \geq 0$, e $\beta \geq 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{per } n = 1 \\ aT(n/b) + cn^{\beta} & \text{per } n > 1 \end{cases}$$

Posto $\alpha = \log a / \log b$, allora:

$$\begin{array}{lll} (1)\ T(n) \text{ è } O(n^{\alpha}) & & \text{se } \alpha > \beta \\ (2)\ T(n) \text{ è } O(n^{\alpha} \log n), & & \text{se } \alpha = \beta \\ (3)\ T(n) \text{ è } O(n^{\beta}) & & \text{se } \alpha < \beta \end{array}$$

DIMOSTRAZIONE

Si supponga dapprima per semplicità che n sia una potenza di b, cioè $n = b^k$ per un k intero. Sostituendo successivamente si ottiene:

$$\begin{aligned} T(n) &= aT\left(b^{k-1}\right) + cb^{k\beta} \\ &= a\left(aT\left(b^{k-2}\right) + cb^{(k-1)\beta}\right) + cb^{k\beta} \\ &= a\left(a\left(aT\left(b^{k-3}\right) + cb^{(k-2)\beta}\right) + cb^{(k-1)\beta}\right) + cb^{k\beta} \\ &\vdots \\ &= a\left(a\left(\cdots a\left(aT(1) + cb^{\beta}\right) + cb^{2\beta}\right) + \cdots + cb^{(k-1)\beta}\right) + cb^{k\beta} \\ &= a^k d + ca^{k-1}b^{\beta} + ca^{k-2}b^{2\beta} + \cdots + cab^{(k-1)\beta} + cb^{k\beta} \\ &= a^k d + cb^{k\beta}\left[\left(a/b^{\beta}\right)^{k-1} + \left(a/b^{\beta}\right)^{k-2} + \cdots + \left(a/b^{\beta}\right)^2 + a/b^{\beta} + 1\right] \end{aligned}$$

Si osservi che $a^k = a^{\log n / \log b} = 2^{\log a \log n / \log b} = n^{\log a / \log b} = n^{\alpha}$, e che $a = b^{\alpha}$, poiché $\alpha = \log a / \log b$. Ponendo $q = a/b^{\beta} = b^{\alpha}/b^{\beta} = b^{\alpha - \beta}$, la relazione può essere scritta come:

$$T(n) = n^{\alpha} d + cb^{k\beta}\left[q^{k-1} + q^{k-2} + \cdots + q^2 + q + 1\right]$$

Si individuano tre casi, a seconda che α sia maggiore, uguale, o minore di β, e che quindi q risulti, rispettivamente, maggiore, uguale, o minore di 1.

(1) Se $\alpha > \beta$, allora:

$$
\begin{aligned}
T(n) &= n^{\alpha} d + cb^{k\beta}\left[\left(q^{k}-1\right)/(q-1)\right] \\
&\le n^{\alpha} d + cb^{k\beta} q^{k}/(q-1) \\
&= n^{\alpha} d + ca^{k}/(q-1) \\
&= n^{\alpha}\left[d + c/(q-1)\right]
\end{aligned}
$$

e quindi $T(n)$ è $O(n^{\alpha})$;

(2) Se $\alpha = \beta$, allora:

$$
\begin{aligned}
T(n) &= n^{\alpha} d + cb^{k\beta}[1+1+\cdots+1] \\
&= n^{\alpha} d + cb^{\beta} k \\
& n^{\alpha}\left[d + c\log n/\log b\right]
\end{aligned}
$$

e pertanto $T(n)$ è $O(n^{\alpha} \log n)$;

(3) Se $\alpha < \beta$, allora:

$$
\begin{aligned}
T(n) &= n^{\alpha} d + cb^{k\beta}\left[\left(1-q^{k}\right)/(1-q)\right] \\
&= n^{\alpha} d + cb^{k\beta}\left[1/(1-q)\right] \\
&= n^{\alpha} d + cn^{\beta}/(1-q)
\end{aligned}
$$

e quindi $T(n)$ è $O(n^{\beta})$.

Nel caso in cui n non sia una potenza di b, si può considerare un problema di dimensione più grande, per es. n', tale che n' sia la più piccola potenza di b maggiore di n. Poiché $n' \le bn$ e b è una costante, la dimensione aumenta solo di un fattore costante e la soluzione rimane valida in ordine di grandezza. Inoltre, poiché si è interessati all'analisi del caso pessimo, la soluzione della relazione di ricorrenza si applica anche al caso in cui valga il "$\le$" anziché l'"$=$". Infine, la soluzione resta valida anche se $T(n)$ è uguale ad una costante per $n \le h$, con h costante intera non necessariamente uguale ad 1, oppure se al posto di n/b si ha $\lfloor n/b \rfloor$, oppure $\lceil n/b \rceil$.

2.44 ESEMPIO – [Merge Sort]
Per la procedura Mergesort() le relazioni di ricorrenza sono del tipo: $T(n) \le 2T(n/2) + cn$. Poiché $a = b = 2$ ed $\alpha = \log 2/\log 2 = 1 = \beta$, si ha che $T(n)$ è $O(n \log n)$.

2.45 ESEMPIO – [Ricerca binaria]
Per la procedura binarySearch() si ha: $T(n) \le T(n/2) + c$. Poiché $a = 1$, $b = 2$, $\alpha = \log 1/\log 2 = 0 = \beta$, ne segue che $T(n)$ è $O(\log n)$.

2.8 ANALISI PER TENTATIVI

Consideriamo adesso un metodo per individuare il comportamento asintotico della soluzione di una relazione di ricorrenza qualsiasi, che non sia necessariamente né lineare, né

a coefficienti costanti, né di ordine costante. Tale metodo procede "per tentativi": si suppone che la soluzione sia di un certo tipo, per esempio $T(n) \leq cn^{\beta}$ con c e β costanti, la si sostituisce nella relazione di ricorrenza e si cerca di dimostrare per induzione su n se è effettivamente una soluzione della ricorrenza. Purtroppo, non c'è una regola generale per individuare la soluzione corretta al primo tentativo; in questi casi occorre un po' di esperienza. Spesso si inizia tentando con soluzioni che sono sicuramente limitazioni superiori oppure inferiori "lasche", restringendo successivamente con tentativi di limitazioni "più strette". Per dimostrare correttamente un ordine di grandezza tramite induzione, però, bisogna prestare particolare attenzione. Infatti, per non cadere miseramente in errori banali, occorre provare che la relazione vale per ogni $n \geq m$ (con m costante) ma sempre per le medesime costanti m e c.

2.16 ESEMPIO – [Tentativi]
Si consideri la relazione di ricorrenza

$$T(n) = \begin{cases} 1 & \text{per } n = 1 \\ 2T(n/2) + n & \text{per } n = 2^h, h > 0 \end{cases}$$

Si tenti di dimostrare per induzione che $T(n) \leq cn$, per qualche costante c. Chiaramente, $T(1) = 1$ verifica la base dell'induzione per $c \geq 1$. Assumendo per ipotesi induttiva che valga $T(n/2) \leq cn/2$, si ricava che $T(n) \leq 2(cn/2) + n \leq cn + n = cn(1 + 1/c)$. Ciò non autorizza a concludere che, poiché $(1 + 1/c)$ è $O(1)$, allora $T(n)$ è $O(n)$. Infatti, $(1 + 1/c) > 1$ e quindi non si è trovato che $T(n) \leq cn$, ma che $T(n) \leq (c + 1)n$, dove $c + 1$ è una costante più grande di c. Pertanto, non è vero che $T(n/2) \leq cn/2$ e $T(n) \leq cn$ per la stessa costante c, e quindi $T(n)$ non è $O(n)$. Si tenti adesso con $T(n) \leq cn^2$. Dopo passaggi analoghi si ottiene che $T(n) \leq 2c(n/2)^2 + n = cn^2(1/2 + 1/(cn))$. Poiché la quantità $(1/2 + 1/(cn))$ è minore di 1 per n sufficientemente grande, la relazione $T(n) \leq cn^2$ vale per ogni $n \geq m$ per le stesse costanti c ed m, e quindi $T(n)$ è $O(n^2)$. Con qualche altro tentativo si può dimostrare che $T(n)$ è $O(n \log n)$, risultato che d'altronde sapevamo già dimostrare per altra via (sia per sostituzione che applicando il teorema sulle ricorrenze lineari con partizione bilanciata).

2.9 ANALISI AMMORTIZZATA

Quando si analizza il costo medio dell'esecuzione di m operazioni su una struttura di dati, un approccio superficiale può portare a limitazioni superiori corrette, ma molto lontane dalla realtà.

2.17 ESEMPIO – [Contatore binario]
Si consideri un contatore binario di k bit che conta "circolarmente" a partire da 0, realizzato con un vettore A di k valori interi pari a 0 o 1, dove il bit meno significativo è memorizzato in $A[0]$ e il bit più significativo è memorizzato in $A[k-1]$. Si consideri un'operazione di incremento add() del vettore realizzata secondo il ben noto schema di addizione di 1 (modulo 2^k) con propagazione del riporto. Qual è il costo di n operazioni di incremento sul contatore?

```
add(integer[] A, integer k)
integer i ← 0
while i < k and A[i] = 1 do
    A[i] ← 0
    i ← i + 1
if i < k then A[i] ← 1
```

Una risposta superficiale si basa sull'osservazione che un'operazione di incremento costa $O(k)$ nel caso pessimo (se tutti i bit sono pari a 1, sommare 1 li porta tutti a 0). Quindi n operazioni vengono eseguite in $O(kn)$ tempo, e il costo medio è $O(k)$.

Tuttavia, osserviamo cosa succede al contatore per $k = 8$ e $n = 16$:

n	A	Costo	n	A	Costo
0	00000000	-			
1	0000000<u>1</u>	1	9	0000100<u>1</u>	1
2	000000<u>10</u>	2	10	000010<u>10</u>	2
3	0000001<u>1</u>	1	11	00001011	1
4	0000<u>100</u>	3	12	0000<u>1100</u>	3
5	00000101	1	13	00001101	1
6	000001<u>10</u>	2	14	000011<u>10</u>	2
7	00000111	1	15	00001111	1
8	0000<u>1000</u>	4	16	000<u>10000</u>	5

I bit che sono cambiati ad ogni chiamata di add() sono sottolineati. Come si può vedere, il numero totale di bit cambiati è sicuramente più basso di nk. Cerchiamo quindi di calcolare esattamente quanti sono, osservando che il bit $A[0]$ cambia ad ogni incremento; il bit $A[1]$ cambia ogni 2 incrementi; il bit $A[2]$ ogni 4, e così via. In altre parole, il bit i-esimo cambia ogni 2^i incrementi; il numero totale di cambiamenti è quindi:

$$\sum_{i=0}^{k-1}\left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1}\frac{n}{2^i} = n\sum_{i=0}^{k-1}\frac{1}{2^i} \leq n\sum_{i=0}^{\infty}\frac{1}{2^i} = 2n$$

Pertanto, per n operazioni, il numero di bit cambiati è limitato superiormente da $2n$; il costo ammortizzato di ogni operazione è quindi $O(1)$. Questo approccio all'analisi ammortizzata viene detto *metodo dell'aggregazione*, in quanto si calcola il costo aggregato di tutte le n operazioni, per poi dividere per n.

Esistono approcci alternativi al metodo dell'aggregazione. Nel *metodo degli accantonamenti*, ad ogni operazione i si assegna un costo ammortizzato a_i che può essere superiore o inferiore al costo effettivo c_i. In questo modo, le operazioni meno costose vengono cariate di un costo aggiuntivo detto *credito*; i crediti accumulati vengono utilizzati per pagare le operazioni più costose. Lo scopo è quello di dimostrare che la somma dei costi effet-

tivi per n operazioni è sempre maggiorata dalla somma dei costi ammortizzati, che quindi costituisce una limitazione superiore alla somma:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$$

Riconsideriamo quindi il nostro contatore. Il costo effettivo di un'operazione di incremento è $d + 1$, dove d è il numero di bit a uno da cambiare (che devono essere sostituiti con zeri) e 1 è il costo per cambiare un bit da zero ad uno (tranne nel caso di overflow, quando tutti i bit sono a uno). L'idea è porre il costo ammortizzato di un incremento pari a $2 = 1 + 1$; 1 per cambiare un bit da zero ad uno, ed uno per il futuro cambio da uno a zero. In questo modo, il credito disponibile $\left(\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} c_i\right)$ è esattamente pari al numero di bit a uno presenti in tutto il vettore, che è maggiore o uguale a 0, e quindi il costo effettivo di n operazioni è sicuramente limitato superiormente da $2n$.

L'ultima tecnica usata nell'analisi ammortizzata è il *metodo del potenziale* perché impiega una funzione potenziale, che in un certo senso mantiene l'"energia potenziale" della struttura di dati.

Formalmente, una funzione potenziale Φ associa ciascuna configurazione della struttura di dati ad un numero reale. Sia dunque Φ_i il valore della funzione potenziale dopo che è stata eseguita l'i-esima operazione, per $i = 1, 2, ..., n$.

Il costo ammortizzato a_i dell'i-esima operazione è il costo effettivo dell'operazione c_i più l'incremento di potenziale dovuto all'operazione:

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

Il costo ammortizzato della sequenza di n operazioni è pertanto:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0$$

dove Φ_n e Φ_0 sono il potenziale della configurazione finale ed iniziale, rispettivamente. Di solito, è opportuno che $\Phi_0 = 0$. Se $\Phi_i - \Phi_{i-1} > 0$, allora il costo ammortizzato a_i dell'i-esima operazione è più del costo effettivo c_i, e l'eccedenza va ad accrescere il potenziale. Se invece $\Phi_i - \Phi_{i-1} < 0$, il costo ammortizzato è meno di quello effettivo ed il potenziale accumulato in precedenza viene usato per compensare il debito. Se $\Phi_n - \Phi_0 \geq 0$, allora il costo ammortizzato della sequenza di operazioni rappresenta una limitazione superiore al costo effettivo della sequenza. Si osservi che, se $\Phi_i - \Phi_0 \geq 0$ per ogni i, allora si ha la garanzia di pagare sempre in anticipo il costo di ciascuna operazione.

Nel caso del nostro contatore, la funzione potenziale Φ è definita come il numero di bit a uno presenti nel vettore.

2.10 REALITY CHECK

Il numero di marzo 2006 della rivista *Nature* si intitolava *2020 – Future of Computing*. L'intero numero è una lettura importante per uno studente di Informatica; ma uno degli articoli, *Science in an exponential world* di Alexander Szalay e Jim Gray è particolarmente interessante per chi ha appena iniziato a valutare l'efficienza degli algoritmi.

L'articolo si apre con una breve introduzione storica al lavoro dello scienziato naturalista: fino a non molto tempo fa, le osservazioni scientifiche venivano registrate nei "registri di laboratorio", su carta. Oggi lo scenario è cambiato: la scienza moderna si basa sugli elaboratori e la quantità di dati che è possibile ottenere e registrare cresce in maniera esponenziale. Già oggi *dataset* misurabili in terabyte (mille gigabyte) sono comuni; ma già si vedono all'orizzonte *dataset* misurabili in petabyte (un milione di gigabyte). Ecco alcuni esempi:

- GenBank è un database aperto che ospita sequenze di DNA (per una breve introduzione ad un algoritmo di bio-informatica si veda il capitolo 13). Al momento in cui scriviamo, ospita il DNA di oltre 100 000 organismi, per un totale di 110 miliardi di basi;
- il Large Synoptic Survey Telescope, che verrà completato nel 2015, produrrà 1.28 petabyte di dati all'anno;
- il Large Hadron Collider del Cern, che ha iniziato a funzionare alla fine del 2009, produrrà 15 petabyte di dati all'anno quando la sua operatività sarà completa.

Grandi sfide si aprono quindi per lo studente di Informatica che voglia abbracciare una doppia carriera, nell'Informatica e nelle Scienze naturali. È facile capire che per simili quantità di dati non c'è spazio per algoritmi esponenziali e anche molti degli algoritmi polinomiali più complessi dovranno essere evitati.

2.1 Si ordinino le seguenti funzioni per ordine di grandezza crescente:

$$\log^2 n,\ 3^{n-2},\ \pi^n,\ n^5 - 5n^2,\ n^4 - 7n^3,\ n^{\log n},\ \log^n n,\ n/\log n,\ n^{1/2},\ 19.$$

2.2 Per dimostrare che l'ordine di grandezza di una funzione $g(n)$ è $O(f(n))$ si può calcolare $\lim_{n\to\infty} g(n)/f(n)$. Se tale limite esiste ed è una costante c (reale), allora $g(n)$ è $O(f(n))$, mentre se il limite è ∞ allora $g(n)$ cresce più velocemente di $f(n)$. Si dimostri che $n \log n$ è $O(n^a)$ per ogni $a > 1$.

2.3 Si mostri che, per stabilire se una funzione $g(n)$ è $\Omega(f(n))$, si può calcolare $\lim_{n\to\infty} g(n)/f(n)$ ed affermare che se il limite esiste ed è una costante $c > 0$ oppure ∞, allora $g(n)$ è $\Omega(f(n))$. Come si può inoltre stabilire se $g(n)$ è $\Theta(f(n))$?

2.4 Si dimostri che tutti i polinomi crescono meno di tutti gli esponenziali, cioè che n^k è $O(b^n)$, per qualsiasi $k > 0$ e $b > 1$.

2.5 Si provi se le funzioni 2^{n+1}, 2^{2n} e 4^n sono $O(2^n)$.

2.6 Si dimostri che $\log n!$ è $\Theta(n \log n)$.

2.7 Si dimostri, per induzione, che $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$.

2.8 Si considerino le funzioni $f(n) = n$ e $g(n) = n^{1+\operatorname{sen} n}$. Si dimostri che le due funzioni non sono confrontabili in ordine di grandezza, cioè che non vale né che $f(n)$ è $O(g(n))$, né che $f(n)$ è $\Omega(g(n))$.

2.9 [Duplicati] Dato un vettore di n interi, si vuole decidere se esiste un elemento che compare due volte. Determinare una limitazione inferiore e una superiore alla complessità di questo problema.

2.10 Sia dato un vettore di n elementi che possono assumere solo i tre valori VERDE, BIANCO e ROSSO. Si vuole ordinare il vettore in $\Theta(n)$ tempo in modo che tutti i "verdi" precedano tutti i "bianchi" e tutti i "bianchi" precedano tutti i "rossi". Le uniche operazioni ammesse sono l'esame di un elemento e lo scambio di due elementi, dati i loro indici.

2.11 Dato un vettore V di n interi positivi distinti, si vuole decidere se esistono due elementi che danno per somma 17. Determinare una limitazione inferiore $\Omega(n)$ e una superiore $O(n)$ alla complessità di questo problema.

2.12 [Analisi di algoritmi ricorsivi] Per ciascuna delle seguenti funzioni si valuti l'ordine di grandezza del tempo $T(n)$.

```
integer pippo(integer n)
  if n ≤ 2 then
  |  return 1
  else if n > 99 then
  |  integer i ← ⌊n/2⌋
  |  return pippo(i) + n · i
  else
  |  return pippo(n − 2) + 5
```

```
integer follia(integer n)
  if n > 10 then
  |  for i ← n downto n − 3 do
  |  |  i ← ⌊n/2⌋
  |  return follia(⌊n/2⌋) · follia(⌊n/2⌋) + n · 5
  else if n > 20 then
  |  return follia(n + 1)
  else
  |  return n
```

2.13 Scrivere un algoritmo ricorsivo che riceve come parametro un vettore di n elementi tale che il suo tempo di esecuzione $T(n)$ verifichi la relazione

$$T(n)=\begin{cases} d & \text{per } n \le k \\ 5T(n/2)+2T(n-1)+cn^2 & \text{per } n > k \end{cases}$$

con c, d e k costanti opportune. L'ordine di grandezza di $T(n)$ è polinomiale oppure no?

2.14 [Analisi con cambio di variabile] Si dimostri che la ricorrenza

$$T(n)=\begin{cases} d & \text{per } n \le k \\ 2T(\sqrt{n})+\log n & \text{per } n > k \end{cases}$$

con d e k costanti opportune, ha soluzione $O(\log n \log \log n)$.

2.15 [Analisi per livelli di ricorsione] Si dimostri che la ricorrenza

$$T(n)=\begin{cases} d & \text{per } n \le k \\ T(n/3)+T(2n/3)+cn & \text{per } n > k \end{cases}$$

con d, k e c costanti opportune, ha soluzione $O(n \log n)$.

2.16 [Analisi per tentativi] Si dimostri per tentativi che la ricorrenza

$$T(n)=\begin{cases} d & \text{per } n \le k \\ 3T(n/4)+n\log n & \text{per } n > k \end{cases}$$

con d e k costanti opportune, ha soluzione $O(n \log n)$.

2.1 L'ordine è il seguente:

$$19, \log^2 n, n^{1/2}, n/\log n, n^4 - 7n^3, n^5 - 5n^2, n^{\log n}, 3^{n-2}, \pi^n, \log^n n$$

2.2

$$\lim_{n\to\infty} \frac{n \log n}{n^a} = \lim_{n\to\infty} \frac{\log n}{n^{a-1}} = \lim_{n\to\infty} \frac{n^{-1}}{(a-1)n^{a-2}} = \lim_{n\to\infty} \frac{1}{(a-1)n^{a-1}} = 0$$

dove si è applicata la regola de L'Hôpital, derivando sia il numeratore che il denominatore, poiché sia il limite di $\log n$ che quello di n^{a-1} sono ∞. Pertanto si ha che $n \log n$ è $O(n^a)$ per ogni $a > 1$ (questo spiega perché l'ordine $O(n \log n)$ sia detto pseudolineare).

2.5 Essendo $2^{n+1} = 2 \cdot 2^n$, si ha che 2^{n+1} è $O(2^n)$. Poiché invece $2^{2n} = 2^{n+n} = 2^n 2^n$, non è possibile che $2^{2n} < c2^n$ per ogni $n \geq m$, per nessuna costante c, e quindi 2^{2n} non è $O(2^n)$. Lo stesso vale per 4^n, perché $4^n = (2^2)^n = 2^{2n}$.

2.6 Dato che $(n/2)^{n/2} < n! < n^n$, si ha che $\frac{n}{2} \log \frac{n}{2} < \log n! < n \log n$. Pertanto $\log n!$ è $\Theta(n \log n)$. Da questo segue anche che $\log n!$ è $O(n^a)$ per ogni $a > 1$ (si veda l'Esercizio 2).

2.7 Procediamo per induzione su h. La base dell'induzione è facilmente verificabile per $h = 0$, perché $\sum_{i=1}^{n} i^0 = \sum_{i=1}^{n} 1 = n$. Si assuma che la limitazione sia vera per un generico h, e si consideri la sommatoria per $h + 1$.

$$\sum_{i=1}^{n} i^{h+1} = \sum_{i=1}^{n} i^h i \leq \sum_{i=1}^{n} i^h n = n \sum_{i=1}^{n} i^h$$

Poiché per ipotesi induttiva $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$, segue che $\sum_{i=1}^{n} i^{h+1}$ è $O(n^{h+2})$.

2.10 Una limitazione inferiore alla complessità del problema è $\Omega(n)$, poiché un qualsiasi algoritmo deve almeno esaminare tutti gli elementi. La seguente procedura ordinaBandiera() inizialmente scandisce il vettore $B[1 \ldots n]$ in avanti alla ricerca del primo elemento $B[k]$ non "verde", ed a ritroso alla ricerca dell'ultimo elemento $B[j]$ non "rosso". Successivamente, viene scandita in avanti (con l'indice i) la porzione $B[k \ldots j]$ del vettore. Ogni elemento $B[i]$ "verde" incontrato è spostato indietro, scambiandolo con $B[k]$ e incrementando poi l'indice k, mentre ogni elemento $B[i]$ "rosso" è spostato in avanti scambiandolo con $B[j]$ e decrementando poi l'indice j. La scansione termina quando l'indice i scavalca j. La complessità della procedura, da richiamarsi con l'istruzione ordinaBandiera(B, n), è ovviamente $\Theta(n)$.

```
ordinaBandiera(integer [] B, integer n)
integer k ← 1
integer j ← n
while k ≤ n and B[k] = VERDE do k ← k + 1
while j ≥ 1 and B[j] = ROSSO do j ← j − 1
integer i ← k
while i ≤ j do
    if B[i] = ROSSO then
        B[i] ↔ B[j]
        j ← j − 1
    else if B[i] = VERDE then
        B[i] ↔ B[k]
        k ← k + 1
        i ← i + 1
    if B[i] = BIANCO then
        i ← i + 1
```

2.11 Una limitazione inferiore $\Omega(n)$ si determina con la tecnica della dimensione dei dati. Infatti V potrebbe contenere una sola coppia di elementi la cui somma dà 17 e se non si esaminasse un elemento di tale coppia, che può trovarsi in una posizione qualsiasi di V, si fornirebbe la risposta sbagliata. Per ottenere una limitazione superiore $O(n)$, scandiamo tutti gli elementi di V e memorizziamo in un secondo vettore $P[1 \ldots 16]$ tutti gli elementi di V minori o uguali a 16. Quindi controlliamo se in P esistono due elementi che danno per somma 17. Dato che la dimensione di P è costante, ci sono $O(1)$ coppie di elementi da controllare. Quindi il tempo richiesto è $O(n)$ ed è dovuto alla scansione di V.

2.12 Per la funzione pippo() si ha:

$$T(n) = \begin{cases} d & \text{per } n \le 99 \\ T(n/2) + c & \text{per } n > 99 \end{cases}$$

con c e d costanti. Applicando il teorema delle ricorrenze lineari con partizione bilanciata, si ha: $a = 1$, $b = 2$, $\alpha = 0$, $\beta = 0$, ed essendo $\alpha = \beta = 0$, $T(n)$ risulta essere $O(\log n)$.

Per la funzione follia() si ricava:

$$T(n) = \begin{cases} d & \text{per } n \le 10 \\ 2T(n/2) + c & \text{per } n > 10 \end{cases}$$

con c e d costanti, poiché "**then return** follia(n + 1)" non è mai eseguito. Per il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene: $a = 2$, $b = 2$, $\alpha = 1$, $\beta = 0$, ed essendo $\alpha > \beta$, $T(n)$ è $O(n)$.

2.44 Effettuiamo il cambio di variabile $m = \log n$, ovvero $n = 2^m$. Allora $\sqrt{n} = 2^{m/2}$ e quindi si ottiene $T(2^m) = 2T(2^{m/2}) + m$. Ponendo adesso $T(2^m) = S(m)$ si ricava la relazione di ricorrenza $S(m) = 2S(m/2) + m$ la cui soluzione è $O(m \log m)$. Dato che $m = \log n$, si ottiene infine che $T(n)$ è $O(\log n \log \log n)$.

2.45 Si noti che non è possibile applicare i teoremi sulle ricorrenze lineari e che il metodo di sostituzione porta a calcoli molto complicati. Allora vediamo cosa accade in ciascun livello della ricorsione sul valore complessivo della funzione. Al livello 0 della ricorsione, $T(n)$ contribuisce con cn al valore della funzione, al quale vanno aggiunti i contributi delle due chiamate ricorsive. Al livello 1 della ricorsione, $T(n/3)$ contribuisce con $cn/3$ e $T(2n/3)$ con $2cn/3$, per un totale ancora di cn, ai quali vanno aggiunti i contributi delle quattro chiamate ricorsive. Al livello 2, le quattro chiamate ricorsive $T(n/9)$, $T(2n/9)$, $T(2n/9)$ e $T(4n/9)$ contribuiscono ancora con $cn/9 + 2cn/9 + 2cn/9 + 4cn/9 = cn$. In sostanza, per ogni livello di ricorsione abbiamo un contributo di al più cn. Il numero di livelli di ricorsione è massimo seguendo le chiamate del tipo 2/3 e cresce come $O(\log_{3/2} n)$. Pertanto, dato che ad ogni livello di ricorsione si paga $O(n)$, la funzione $T(n)$ cresce come $O(n \log n)$.

STRUTTURE DI DATI

PAOLO BOSCO

TIPI DI DATO E STRUTTURE DI DATI

Capitolo 3

I termini dato e tipo di dato, precedentemente introdotti, benché possano sembrare simili, hanno significato diverso:

- in un linguaggio di programmazione, il *dato* è un valore che una variabile può assumere;
- un *tipo di dato* è un modello matematico, consistente in una collezione di valori sui quali sono ammesse certe operazioni.

ESEMPIO – [Interi]
Il numero sette è un dato, che può essere assegnato ad una variabile di tipo intero. La collezione dei numeri interi, con le usuali operazioni aritmetiche ivi definite, è un tipo di dato.

Ogni linguaggio di programmazione fornisce direttamente alcuni tipi di dato, detti tipi di dato primitivi. Come si è visto precedentemente, può tuttavia risultare utile definirne di nuovi. Nel trattare i tipi di dato, comunque, è bene distinguere tra i dati stessi e la loro rappresentazione. Ad esempio, un numero naturale è un'entità astratta, che può essere rappresentata in vari modi: in base 10, come nelle istruzioni Java, in base 2, come nei circuiti di un calcolatore, o addirittura con la notazione degli antichi romani, così che il numero "tredici" sarà rappresentato, rispettivamente, con "13", "1101", o "XIII". In ogni caso, le proprietà dei naturali sono specifiche di tali numeri e non dipendono da come i loro valori sono rappresentati.

In generale, le proprietà che caratterizzano un tipo di dato dipendono esclusivamente dalla specifica del tipo di dato stesso mentre devono essere indipendenti dal modo in cui i dati sono rappresentati. Si parla allora di un *tipo di dato astratto*, dove astratto va inteso rispetto alla rappresentazione del tipo di dato. In altri termini, la specifica del tipo di dato fornisce le caratteristiche dei valori e delle operazioni per mezzo di definizioni e nozioni puramente matematiche. Una corretta specifica equivale ad un buon "manuale d'uso"

chiaro, conciso e non ambiguo. Essa permette ad un progettista di realizzare il tipo di dato, cioè di scegliere come rappresentarne i valori e di scrivere il codice di procedure e funzioni che corrispondono alle operazioni indicate nella specifica. Inoltre, permette anche ad un utente di scrivere procedure e funzioni che utilizzano il tipo di dato senza conoscerne la rappresentazione.

3.1 STRUTTURE DI DATI: DEFINIZIONE

Accade spesso che i dati da elaborare siano riuniti in agglomerati, detti *strutture di dati* (o strutture informative). Due aspetti caratterizzano una struttura di dati:

- la modalità con cui è possibile accedere ai dati stessi; ovvero un insieme di operatori che permettono di manipolare elementi della struttura o di aggregare elementi per costruire altri agglomerati;
- l'organizzazione in memoria dei dati stessi, progettata per ottimizzare il tempo di esecuzione degli operatori ammessi.

3.2 ESEMPIO – [Vettore]

Un vettore è una sequenza di elementi dello stesso tipo. Sugli elementi si possono effettuare operazioni di lettura e scrittura. La lettura consiste nel reperire il valore di un elemento, mentre la scrittura nel sostituire il valore di un elemento con un nuovo valore. In entrambi i casi si accede direttamente (in tempo O(1)) al generico elemento specificandone la posizione (indice) all'interno della sequenza.

È comodo fornire una classificazione delle strutture di dati in base alle caratteristiche presentate dalla disposizione dei dati, dal loro numero e dal loro tipo. Di solito si usa distinguerle in:

- *lineari*, in cui gli agglomerati sono formati da dati disposti in sequenza, tra i quali si individua un primo elemento, un secondo, eccetera;
- *non lineari*, in cui non è individuata una sequenza;
- *statiche*, in cui il numero di elementi dell'agglomerato rimane sempre costante nel tempo;
- *dinamiche*, in cui il numero di elementi può aumentare o diminuire nel tempo;
- *omogenee*, in cui i dati sono tutti dello stesso tipo;
- *non omogenee*, in cui i dati non sono tutti dello stesso tipo.

3.2 ESEMPIO

Il vettore è una struttura lineare, omogenea e statica.

In questo capitolo, descriveremo la specifica astratta delle principali strutture di dati; l'organizzazione in memoria e la realizzazione di tali strutture saranno argomenti dei prossimi capitoli.

In particolare, discuteremo di due aspetti relativi alla specifica: la notazione con cui sono indicati sia i tipi di dato che le operazioni (specifica sintattica), e il significato associato ad essi (specifica semantica). La specifica sintattica fornisce l'elenco delle operazioni peculiari della struttura stessa. Inoltre descrive i domini di partenza e di arrivo, cioè i tipi degli operandi e del risultato, per ogni operazione. La specifica semantica invece descrive, quanto più formalmente possibile, l'effetto che l'operazione ha sulla struttura di dati. In molte delle librerie standard disponibili nei moderni linguaggi di programmazione la specifica semantica è scritta in linguaggio naturale; una specifica formale richiederebbe invece una descrizione matematica dei domini di partenza e arrivo e della relazione che li lega.

La realizzazione riconduce la specifica ai tipi di dato e alle operazioni già disponibili. In particolare, le operazioni della struttura che modificano la struttura stessa sono di solito simulate con procedure, mentre quelle che restituiscono un dato primitivo sono simulate con funzioni. È bene sottolineare che alla stessa specifica possono corrispondere diverse realizzazioni, più o meno efficienti, a seconda dei dati e delle procedure utilizzati. Tali realizzazioni sono "nascoste" all'utente che è interessato soltanto a quali dati e operatori utilizzare, ma non si deve preoccupare di come questi sono realizzati. Al limite, le realizzazioni possono essere addirittura cambiate senza che l'utente neanche se ne accorga!

Per eseguire un'operazione su una struttura di dati, si utilizzerà la notazione a "punto", già utilizzata per accedere ai campi di un record: ad esempio, l'istruzione S.insert(x) inserisce l'elemento x nell'insieme S. All'interno del codice degli operatori di una struttura di dati, è possibile nominare direttamente i campi di una struttura, senza la notazione a "punto"; è inoltre possibile riferirsi alla struttura stessa utilizzando l'operatore **this**, che restituisce un puntatore alla struttura stessa.

3.2 STRUTTURE DI DATI: SPECIFICA

Le principali strutture di dati di cui discuteremo in questo libro "imitano" alcuni dei principali costrutti matematici di uso comune. Abbiamo già trattato di vettori e matrici introducendo lo pseudocodice; discutiamo ora dei concetti di sequenze, insiemi, dizionari, alberi e grafi.

3.2.1 SEQUENZE

Una *sequenza* (*sequence*, in inglese) è una struttura di dati dinamica e lineare, contenente generici elementi di tipo ITEM, potenzialmente anche duplicati. È quindi possibile aggiungere o togliere elementi, specificando la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o dalla quale il vecchio elemento va tolto.

ESEMPIO – [Schedario]
Uno schedario contenente informazioni sugli impiegati di un'azienda, in cui è possibile ricercare, aggiungere e togliere schede, è un esempio di sequenza.

L'ordine in cui gli elementi vengono organizzati nella sequenza è quindi importante: esiste un primo elemento, un secondo elemento, e così via fino all'ultimo. Normalmente, è possibile accedere direttamente solo ad un ristretto sottoinsieme di elementi (di solito il primo o l'ultimo); per accedere ad un generico elemento, occorre scandire sequenzialmente gli elementi della sequenza: partendo da un elemento accedibile direttamente, ci si sposta via via da un elemento ad uno adiacente nella sequenza, fino a raggiungere l'elemento desiderato.

Sia $S = s_1, s_2, ..., s_n$ una sequenza. La lunghezza di S è pari ad n, il numero dei suoi elementi. Ogni elemento è caratterizzato da una posizione in S, indicata con pos_i, e da un valore s_i. Anche se viene immediato immaginare che pos_i sia uguale ad i, si considera la posizione come un tipo di dato a sé stante la cui realizzazione varia a seconda della realizzazione adottata per il tipo di dato sequenza e può non essere uguale all'intero i. Utilizzeremo Pos come identificatore per questo generico tipo. Per comodità, si assume inoltre l'esistenza di una posizione pos_0 che precede quella del primo elemento e di una posizione pos_{n+1} che segue quella dell'ultimo elemento.

Un tipico insieme di operazioni per il tipo di dato sequenza comprende, a parte l'operazione di creazione di una sequenza vuota, quelle di selezione head() e tail(), per accedere direttamente al primo o all'ultimo elemento della sequenza, e next() e prev(), per passare da un elemento al successivo o al precedente nella sequenza; quelle di interrogazione empty() e finished(), per verificare che la sequenza sia vuota o che sia stato oltrepassato un estremo della sequenza; quella di lettura read() per leggere il valore di un elemento; e infine, quelle di modifica write() per cambiare il valore di un elemento, insert() per inserire un nuovo elemento nella sequenza, e remove() per cancellare un vecchio elemento della sequenza.

SEQUENCE

% Restituisce **true** se la sequenza è vuota
boolean empty()

% Restituisce **true** se p è uguale a pos_0 oppure a pos_{n+1}
boolean finished(Pos p)

% Restituisce la posizione del primo elemento
Pos head()

% Restituisce la posizione dell'ultimo elemento
Pos tail()

SEQUENCE (continua)

℞ Restituisce la posizione dell'elemento che segue p
Pos next(Pos p)

℞ Restituisce la posizione dell'elemento che precede p
Pos prev(Pos p)

℞ Inserisce l'elemento v di tipo ITEM nella posizione p.
℞ Restituisce la posizione del nuovo elemento, che diviene il predecessore di p
Pos insert(Pos p, ITEM v)

℞ Rimuove l'elemento contenuto nella posizione p.
℞ Restituisce la posizione del successore di p, che diviene successore del predecessore di p
Pos remove(Pos p)

℞ Legge l'elemento di tipo ITEM contenuto nella posizione p
ITEM read(Pos p)

℞ Scrive l'elemento v di tipo ITEM nella posizione p
write(Pos p, ITEM v)

Dalla specifica emerge che per accedere ad un elemento occorre conoscerne la posizione. Ciò è possibile solo conoscendo a priori la posizione dell'elemento precedente (o seguente) ed applicando quindi l'operazione next() o prev(). Le uniche operazioni che danno per risultato una posizione, senza che gliene venga fornita una in ingresso, sono le operazioni head() e tail(). Per accedere ad un generico elemento occorre pertanto scandire la sequenza in avanti a partire dal primo elemento, o a ritroso partendo dall'ultimo. Se viene oltrepassato un estremo della sequenza, next() e prev() restituiscono, rispettivamente, pos_{n+1} e pos_0. È possibile accorgersi di questo fatto con l'interrogazione finished(), che in tal caso restituirà il valore booleano **true**.

Poiché inserzioni e cancellazioni "allungano" ed "accorciano" la sequenza, esse provocano la modifica di tutte le posizioni degli elementi che seguono quello inserito o cancellato. In particolare, insert(p, v) con $p = pos_i$ inserisce il nuovo elemento di valore v come nuovo i-esimo elemento della sequenza, mentre il vecchio i-esimo elemento diventa l'$(i + 1)$-esimo, il vecchio $(i + 1)$-esimo diventa l'$(i + 2)$-esimo ecc. Se $p = pos_{n+1}$, insert() inserisce il nuovo elemento in fondo alla sequenza, lasciando inalterate le posizioni dei vecchi elementi. Se la sequenza è vuota e $p = pos_0 = pos_{n+1}$, il nuovo elemento diviene il primo (ed unico) elemento della sequenza. Invece, remove(p) con $p = pos_i$ cancella il vecchio i-esimo elemento della sequenza, mentre il vecchio $(i + 1)$-esimo elemento diventa l'i-esimo, il vecchio$(i + 2)$-esimo diventa l'$(i + 1)$-esimo ecc. Sia insert() che remove() restituiscono la posizione del nuovo elemento i-esimo nella sequenza.

3.2.2
INSIEMI

Un *insieme* (*set*, in inglese) è una collezione (o famiglia) di elementi distinti (detti anche componenti o membri) dello stesso tipo. L'insieme è una delle strutture matematiche più importanti, e può essere descritto o elencandone tutti gli elementi o stabilendo una proprietà che ne caratterizzi gli elementi. A differenza delle sequenze, gli elementi non sono caratterizzati da una posizione, né sono ammesse più copie dello stesso elemento nel medesimo insieme.

Il numero di elementi che compongono un insieme A è detto *cardinalità* (o dimensione) di A e si indica con $|A|$. Se tale numero è finito, allora l'insieme ha cardinalità finita, altrimenti ha cardinalità infinita. Un insieme che non contiene alcun elemento è detto *vuoto*, e si indica con $\emptyset$.

ESEMPIO – [Definizione di insiemi e cardinalità]
$A = \{1, 5, 13, 7\}$, B = {Studenti dei Corsi di Laurea in Informatica in Italia immatricolati nel 2009}, C = {numeri reali compresi tra 0 e 1}. La cardinalità di A è 4, quella di B è finita, mentre quella di C è infinita. La cardinalità di $\emptyset$ è zero.

La relazione fondamentale per gli insiemi è l'appartenenza di un elemento x ad un insieme A, indicata formalmente con $x \in A$. Da questa si deriva l'inclusione (o contenimento) tra due insiemi A e B: indicata con $A \subseteq B$, significa che ogni elemento x che appartiene ad A appartiene anche a B. Se $A \subseteq B$ e $B \subseteq A$, allora $A = B$, cioè A e B sono lo stesso insieme. Viceversa, le negazioni dell'appartenenza, dell'inclusione e dell'eguaglianza si indicano, rispettivamente, con $x \notin A$, $A \nsubseteq B$ e $A \neq B$. Le operazioni principali tra due insiemi A e B sono l'unione, l'intersezione e la differenza, indicate, rispettivamente, con $A \cup B$, $A \cap B$ e $A - B$. $A \cup B$ denota l'insieme contenente tutti gli elementi di A e tutti gli elementi di B. $A \cap B$ denota l'insieme contenente tutti e soli gli elementi che appartengono sia ad A che a B. Infine, $A - B$ denota l'insieme che contiene tutti e soli gli elementi che appartengono ad A ma non appartengono a B.

Una specifica ragionevole per il tipo di dato insieme prevede una serie di operatori fondamentali per la modifica di insiemi, come la creazione, l'inserimento e la cancellazione; operatori per verificare se un certo elemento è presente, e per ottenere il numero di elementi contenuti nell'insieme (questa operazione può essere utilizzata anche per scoprire se un insieme è vuoto). Una volta realizzati questi operatori di base, è possibile costruire operatori insiemistici quali l'unione, l'intersezione e la differenza di insiemi.

SET

% Restituisce la cardinalità dell'insieme
integer size()

% Restituisce **true** se x è contenuto nell'insieme
boolean contains(ITEM x)

% Inserisce x nell'insieme, se non già presente
insert(ITEM x)

% Rimuove x dall'insieme, se presente
remove(ITEM x)

% Restituisce un nuovo insieme che è l'unione di A e B
SET union(SET A, SET B)

% Restituisce un nuovo insieme che è l'intersezione di A e B
SET intersection(SET A, SET B)

% Restituisce un nuovo insieme che è la differenza di A e B
SET difference(SET A, SET B)

3.2.3
DIZIONARI

Il termine informatico *dizionario* (in inglese, *dictionary* o *map*) rappresenta il concetto matematico di relazione univoca $R: D \rightarrow C$ fra gli elementi di un insieme D (il *dominio*) e gli elementi di un insieme C (il *codominio*). Gli elementi del dominio prendono spesso il nome di *chiavi* e gli elementi del codominio *valori*: si parla così di associazioni chiave-valore.

Le operazioni ammesse su questo tipo di dato permettono di ottenere il valore associato ad una particolare chiave (se presente), o **nil** altrimenti; di inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti; di rimuovere un'associazione chiave-valore esistente.

DICTIONARY

% Restituisce il valore associato alla chiave k se presente, **nil** altrimenti
ITEM lookup(ITEM k)

% Associa il valore v alla chiave k
insert(ITEM k, ITEM v)

% Rimuove l'associazione della chiave k
remove(ITEM k)

3.2.4
ALBERI E GRAFI

Le strutture matematiche adottate dalla programmazione non si esauriscono con sequenze, insiemi e dizionari; particolare importanza rivestono gli alberi e i grafi.

Un *albero ordinato* (o più semplicemente *albero*) è formato da un insieme finito di elementi, detti *nodi*. Se tale insieme non è vuoto, allora un particolare nodo è designato come *radice*, ed i rimanenti nodi, se esistono, sono a loro volta partizionati in insiemi ordinati disgiunti, ciascuno dei quali è un albero ordinato.

3.6 ESEMPIO – [Albero genealogico]

L'albero genealogico di una dinastia, a partire dal progenitore, divide gerarchicamente i suoi membri in successive generazioni, fino ad interrompersi in attesa della nascita di nuovi discendenti. La Fig. 3.1 mostra la dinastia dei re merovingi, dove per semplicità ne sono stati indicati solo i principali. Verifichiamo come la struttura di Fig. 3.1 sia costruita in accordo alla definizione ricorsiva di albero ordinato appena introdotta. I nodi sono 16. Meroveo è la radice dell'albero, ed i rimanenti 15 nodi sono ripartiti in un unico albero con radice Childerico I. Analogamente, sistemato Childerico I, i rimanenti 14 nodi sono ripartiti nell'unico albero con radice Clodoveo I. Si noti che, una volta sistemati i primi tre nodi, gli altri 13 sono ripartiti in 5 alberi, dei quali se ne individua il primo (quello con radice Thierry I), il secondo (con radice Clodomiro), e così via. Ciò spiega perché l'albero sia chiamato "ordinato".

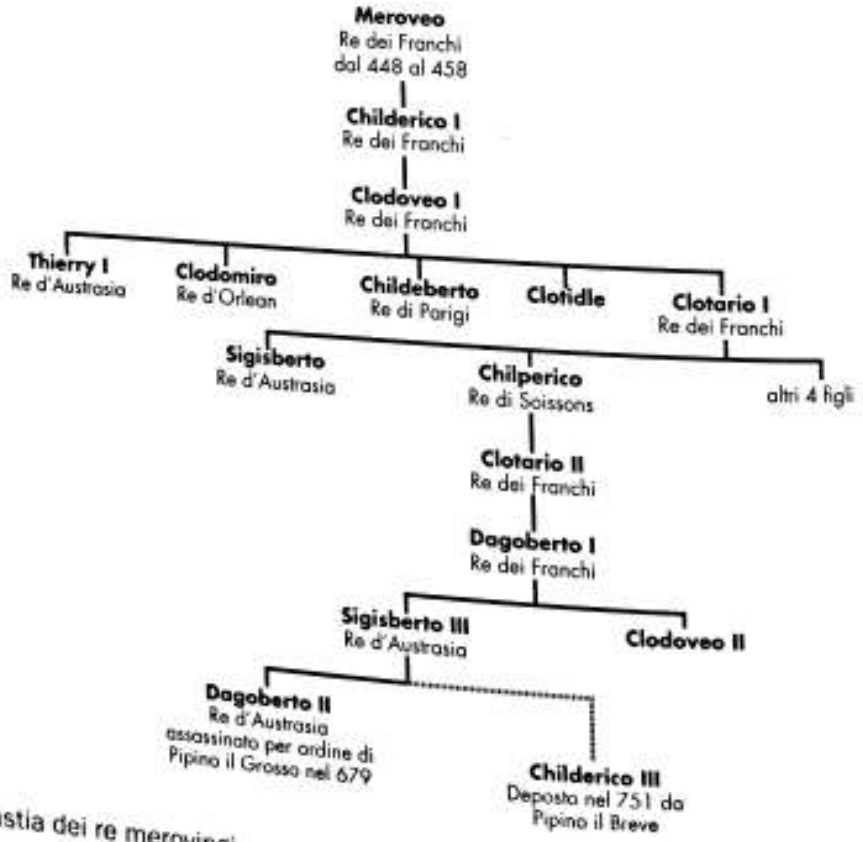


Fig. 3.1 La dinastia dei re merovingi.

Anche i grafi sono strutture non lineari, organizzate da un insieme di *nodi* e da un insieme di relazioni tra i nodi dette *archi*. Tipicamente, vengono rappresentati in forma grafica come un insieme di cerchietti (i nodi) collegati da linee rette (gli archi). In memoria, posso-

no essere rappresentati da un insieme di record collegati da puntatori (ma esistono anche rappresentazioni più efficienti, soprattutto quando tali strutture sono statiche).

ESEMPIO – [Linee aeree]
I nodi possono rappresentare aeroporti e gli archi collegamenti aerei tra aeroporti, come illustrato nella Fig. 3.2.

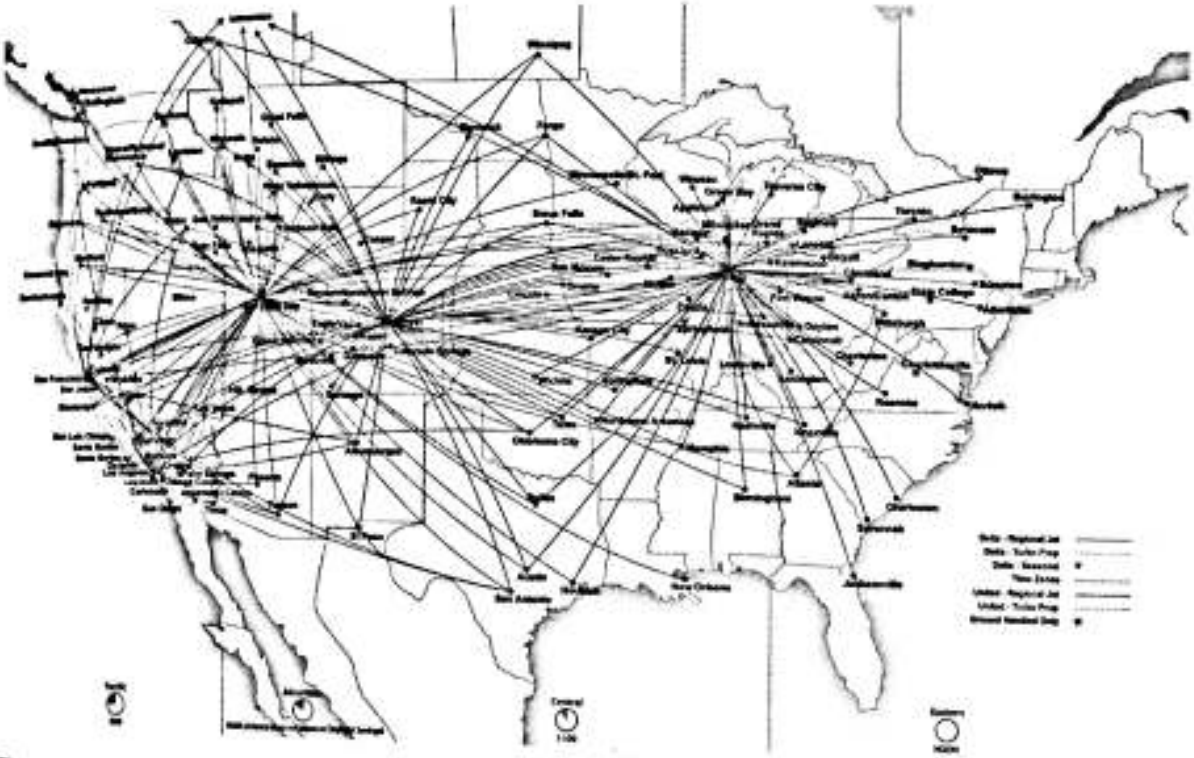


Fig. 3.2 Alcuni collegamenti aerei negli Stati Uniti.

Fra le operazioni fondamentali che possono essere effettuate su alberi e grafi, sottolineiamo il concetto di *visita*: l'ispezione completa di tutti i nodi dell'albero o del grafo. Per effettuare una visita, è necessario disporre di un insieme di operatori che permettano di muoversi fra i nodi, seguendo i collegamenti fra di essi. È ovviamente importante anche avere la possibilità di modificare queste strutture, aggiungendo e rimuovendo nodi e collegamenti. Rimandiamo tuttavia la discussione di questi operatori ai relativi capitoli 5 e 9, in quanto è necessario introdurre numerosi altri concetti prima di arrivare alla specifica di tali strutture di dati.

3.2.5 DISCUSSIONE

Le specifiche dei tipi di dato astratto offerte in questo capitolo sono solo alcuni esempi; è possibile arricchirle con nuove operazioni, se necessario, o combinare insieme specifiche diverse. Per esempio:

- si possono aggiungere le operazioni min() e max() al tipo di dati astratto SET, se sui valori dell'insieme è definito un ordinamento totale;

- se gli elementi di un insieme sono ordinati internamente in qualche modo, potrebbe essere utile usare le operazioni per scorrere gli elementi di una sequenza (head() e next()), ottenendo un controllo più raffinato del costrutto **foreach** $s \in S$ introdotto per scandire tutti gli elementi di un insieme;
- i concetti di insieme e dizionario sono intimamente collegati. Ad esempio, le chiavi contenute in un dizionario sono un insieme; come tale, potrebbe essere necessario scorrere tutte le chiavi del dizionario, per esempio per stamparne il contenuto; oppure potrebbe essere utile conoscere la chiave minima ecc.

È anche possibile restringere l'insieme di operazioni ammesse, per poter progettare una struttura di dati particolarmente efficiente nel risolvere un compito specifico. Ad esempio, una coda con priorità è un insieme in cui oltre all'operazione di inserimento, sono presenti solo operazioni per la lettura e la cancellazione del minimo [capitolo 10.1]; per via del ristretto e particolare numero di operatori, le operazioni di inserimento e cancellazione possono essere completate in tempo $O(\log n)$, mentre la lettura viene completata in tempo $O(1)$; tale struttura è particolarmente adatta per compiti quali la simulazione ad eventi e l'ordinamento in base alla priorità.

3.3 STRUTTURE DI DATI: REALIZZAZIONE

Esiste un collegamento molto stretto fra alcuni dei tipi di dato astratto introdotti nella sezione precedente e le principali strutture di dati che possono essere utilizzate per realizzare questi tipi di dato. Per esempio, una sequenza è normalmente realizzata tramite liste con puntatori, dove gli elementi vengono memorizzati in record in cui alcuni dei campi puntano all'elemento successivo e/o all'elemento precedente, tanto che spesso lista e sequenza sono considerati sinonimi.

Ma esiste anche la possibilità di fornire realizzazioni diverse degli stessi tipi di dato astratto, con differenti caratteristiche per quanto riguarda l'efficienza e le funzionalità offerte. Per esempio, un insieme può essere realizzato tramite un albero o tramite un vettore organizzato come tabella hash [capitolo 7]; nel primo caso, molte delle operazioni ammesse richiederanno tempo $O(\log n)$, ma sarà possibile scorrere gli elementi dell'insieme in maniera ordinata; nel secondo caso, le operazioni richiederanno tempo $O(1)$ ma l'ordine è completamente perso. Analoghe considerazioni si applicano alle chiavi di un dizionario.

Seguendo queste considerazioni, nei prossimi capitoli discuteremo sia di strutture di dati basate su puntatori per le quali specifica e realizzazione sono fortemente legate, come liste [capitolo 4] e alberi [capitolo 5], sia di strutture di dati basate su vettori, come code, pile e tabelle hash [capitoli 4 e 7], utili a realizzare diversi tipi di dato astratto. Forti di queste conoscenze, vedremo vari modi di realizzare insiemi e dizionari [capitolo 8]. Concluderemo introducendo i grafi [capitolo 9], per poi vedere alcune strutture di dati speciali progettate per compiti particolari [capitolo 10].

3.4 REALITY CHECK

Il linguaggio Java fornisce un insieme di interfacce e classi per i principali tipi astratti di dato che prende il nome di *Java Collections*. La Fig. 3.3 fornisce un quadro di insieme delle interfacce incluse nelle Collections; oltre a sequenze (`List`), insiemi (`Set`) e dizionari (`Map`), esistono anche le interfacce `SortedSet` e `SortedMap`, un'estensione di `Set` e `Map` che supportano l'ordinamento degli oggetti e delle chiavi (aggiungendo metodi per ottenere il minimo e il massimo, e per ottenere sottoinsiemi definiti da un estremo inferiore e un estremo superiore).

Per quanto riguarda le realizzazioni, è interessante notare che sono presenti sia implementazioni basate su tabelle hash (`HashSet`, `HashMap`) sia implementazioni basate su alberi (`TreeSet`, `TreeMap`).

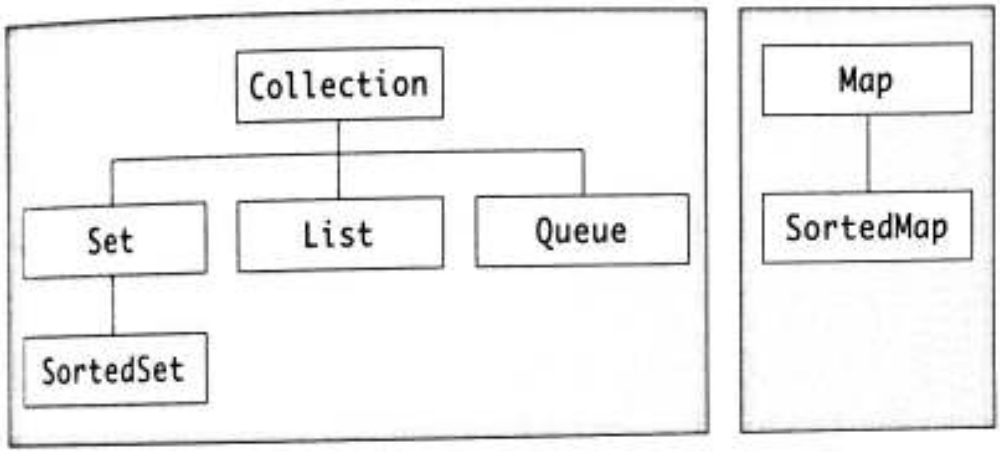


Fig. 3.3 Le interfacce di Java Collections.

Le operazioni di scorrimento degli elementi di una sequenza, insieme o dizionario sono fornite tramite l'interfaccia `Iterator`, che fornisce metodi quali `next()`, che restituisce il prossimo elemento, e `hasNext()` che restituisce **true** se esiste un prossimo elemento. È interessante notare che a partire da Java 1.5 è possibile sostituire il codice seguente, che stampa tutti gli elementi di una generica `Collection`:

```
void printAll(Collection<Object> C)  {
  for (Iterator<Object> i = C.iterator(); i.hasNext(); )
    System.out.println(i.next());
}
```

con il codice seguente, più elegante, in cui si utilizza il costrutto denominato "for-each" in Java (si legge come **foreach** $o \in C$):

```
void printAll(Collection<Object> C)  {
  for (Object o : C)
    System.out.println(o);
}
```

Per concludere, vogliamo sottolineare che la *Standard Template Library* (STL) è una libreria che contiene non solo l'equivalente C++ delle Collections Java, ma anche molti altri algoritmi di utilità generale. Anche i nomi sono simili, prevedendo strutture quali `List`, `Set` e `Map`.

3.1 [Specifica numeri complessi] Si fornisca una specifica del tipo di dato numero complesso, che includa i seguenti operatori: parte reale, parte immaginaria, argomento, modulo, costruzione a partire dalla rappresentazione cartesiana, costruzione a partire dalla rappresentazione trigonometrica, addizione, sottrazione, moltiplicazione, coniugato, divisione.

3.2 [Realizzazione numeri complessi] Si fornisca una realizzazione in pseudocodice, o nel vostro linguaggio preferito, del tipo di dato numero complesso specificato nell'Esercizio 3.1, utilizzando record e puntatori.

3.3 [Memorizzazione matrice] Una matrice w a tre dimensioni può essere memorizzata "per colonne" in un vettore v, facendo variare più in fretta il primo indice, indi il secondo, ed infine il terzo. Si scriva la formula che dà la corrispondenza tra $w[i, j, h]$ e $v[p]$.

3.4 [Memorizzazione matrice] Si riscriva la formula che dà la corrispondenza tra un generico elemento $w[i_1, i_2, ..., i_k]$ di una matrice a k dimensioni e $v[p]$, assumendo di memorizzare w nel vettore v "per righe" e che ciascun indice i_j possa variare nell'intervallo min_j e max_j, per $1 \leq j \leq k$.

3.5 [Polinomi] Si fornisca una specifica e una realizzazione per il tipo di dato astratto polinomio, che permetta di creare nuovi polinomi di grado k, di variare i coefficienti, di sommare, sottrarre e moltiplicare polinomi e infine di valutare il polinomio per uno specifico valore dell'incognita.

3.6 Si scriva una procedura che, data una sequenza L di interi, stampi tutti gli elementi di L i cui valori sono uguali alla differenza tra il valore dell'elemento immediatamente precedente e quello dell'elemento immediatamente seguente (per esempio, se $L = 5, 2, 3, 1, 2$, allora stampa 2, 1). Si assuma per semplicità che L contenga almeno tre elementi.

3.7 Si scriva una procedura di complessità ottima che, data una sequenza L di interi, riordina L in modo che tutti gli elementi dispari precedano, nello stesso ordine che avevano inizialmente in L, tutti gli elementi pari (per esempio, se $L = 3, 7, 8, 1, 4$, allora si ottiene $L = 3, 7, 1, 8, 4$).

3.8 [Rango] Il rango di un elemento di una sequenza di interi è definito come la somma del suo valore e dei valori degli elementi che lo seguono. Scrivere una procedura ricorsiva di complessità ottima che, data una sequenza L, modifica L in modo che ogni elemento contenga il proprio rango (per esempio, se $L = 3, 2, 5$, allora si vuole ottenere $L = 10, 7, 5$).

3.3 Si assuma che la matrice w abbia dimensioni $n \times m \times k$ con numerazione delle linee a partire da 1. Memorizziamo la matrice nel vettore v di dimensione nmk facendo variare l'indice i più in "fretta", poi facendo variare l'indice j e infine l'indice h. Allora il generico elemento $w[i,j,h]$ della matrice corrisponde all'elemento $v[p]$ del vettore, dove $p = i + (j-1)n + (h-1)nm$.

3.6 La procedura stampaDifferenze() scorre la sequenza mantenendo tre posizioni, confrontando il contenuto della seconda con il contenuto della prima e della terza, e aggiornando opportunamente i puntatori.

```
stampaDifferenze(SEQUENCE L)
  POS p1 ← L.head()
  POS p2 ← L.next(p1)
  POS p3 ← L.next(p2)
  while not finished(p3) do
      if L.read(p2) = L.read(p1) – L.read(p3) then print L.read(p2)
      p1 ← p2
      p2 ← p3
      p3 ← L.next(p2)
```

3.8 Una limitazione inferiore alla complessità del problema è data dalla lunghezza n della sequenza L, ed è quindi $\Omega(n)$. La soluzione ricorsiva si basa sulla seguente proprietà:

$$rango(a_i) = \begin{cases} rango(a_{i+1}) + a_i & \text{se } i < n \\ a_n & \text{se } i = n \end{cases}$$

```
rango(SEQUENCE L, POS p)
  POS q ← L.next(p)
  if not L.finished(q) then
      rango(L, q)
      integer a ← L.read(p) + L.read(q)
      L.write(p, a)
```

Per chiamare la procedura, si deve verificare prima che la sequenza L non sia vuota e chiamare rango(L.head()). Il tempo $T(n)$ richiesto dalla procedura verifica le seguenti relazioni di ricorrenza, con c e d costanti,

$$T(n) = \begin{cases} c & \text{per } n = 1 \\ T(n-1) + d & \text{per } n > 1 \end{cases}$$

la cui soluzione è $\Theta(n)$. Pertanto la procedura ha complessità ottima.

STRUTTURE DI DATI ELEMENTARI

Capitolo 4

Come discusso nel capitolo precedente, le sequenze sono uno dei tipi di dato astratto più importanti nella programmazione e nella progettazione di algoritmi. Molto spesso i dati di cui un programma ha bisogno vengono analizzati in sequenza, uno dopo l'altro.

44 ESEMPIO – [Il catalogo di Don Giovanni]

Il famoso catalogo delle conquiste di Don Giovanni comprende italiane, tedesche, spagnole, francesi e turche. Tra queste ci sono contadine, cameriere, cittadine, duchesse, baronesse, marchesine e principesse. Il fedele servitore Leporello ha strutturato il catalogo come una sequenza ordinata per data di conquista; vorrebbe ora cancellare dal catalogo tutte le duchesse e copiare in un'altra sequenza *L* tutte le spagnole, mantenendone lo stesso ordine che avevano nel catalogo.

Nella procedura donGiovanni() sono utilizzate due posizioni *p* e *q* per scandire, rispettivamente, le sequenze *L* e *Catalogo*. *L* è inizializzata alla sequenza vuota, *q* alla prima posizione del catalogo, e *p* alla prima posizione di *L*. Poiché all'inizio *L* è vuota, $p = pos_0 = pos_{|L|+1}$, la posizione che segue l'ultimo elemento. All'interno del ciclo **while**, la posizione *p* non viene mai modificata; viene quindi mantenuta sempre uguale a quella che segue l'ultimo elemento di *L*, in modo da inserire un nuovo elemento in fondo alla sequenza. In questo modo, gli elementi inseriti in *L* mantengono lo stesso ordine che avevano nel catalogo. Nel ciclo **while**, *prima* è verificata la nazionalità e *poi* è controllato il rango. Così, se una conquista è al tempo stesso sia duchessa che spagnola, viene prima inserita nella sequenza *L* e poi cancellata dal catalogo. L'operazione remove() restituisce la posizione del successore di *q*, come previsto dalla specifica; questa posizione viene assegnata a *q*, proseguendo la ricerca.

Per valutare la complessità della procedura donGiovanni(), occorre prima conoscere la complessità di ogni operazione impiegata. In altri termini, occorre conoscere la realizzazione della struttura astratta *sequenza*. Questo capitolo introduce le principali strutture di dati per la realizzazione di sequenze. Nella prima parte discuteremo di strutture basate su puntatori, che permettono di realizzare sequenze dinamiche. Nella seconda discuteremo di strutture basate su vettori, che se da un lato limitano il dinamismo, dal-

l'altro si dimostrano estremamente semplici ed efficienti dal punto di vista dell'occupazione di memoria.

```
donGiovanni(SEQUENCE catalogo)
  SEQUENCE L ← Sequence()
  POS q ← catalogo.head()
  while not catalogo.finished(q) do
      ITEM v ← catalogo.read(q)
      if v.nazione = spagnola then L.insert(L.next(L.tail()), v)
      if v.rango = duchessa then
          q ← catalogo.remove(q)
      else
          q ← catalogo.next(q)
```

4.1 LISTE CON PUNTATORI

La prima realizzazione che analizziamo prende il nome di *lista* (*list*, in inglese) ed è basata sull'uso di puntatori, dove alla contiguità tra elementi della sequenza non corrisponde necessariamente una contiguità di memorizzazione. Questa realizzazione permette sia di ottenere complessità $O(1)$ per tutte le operazioni, sia di non porre limiti alla dimensione massima della sequenza.

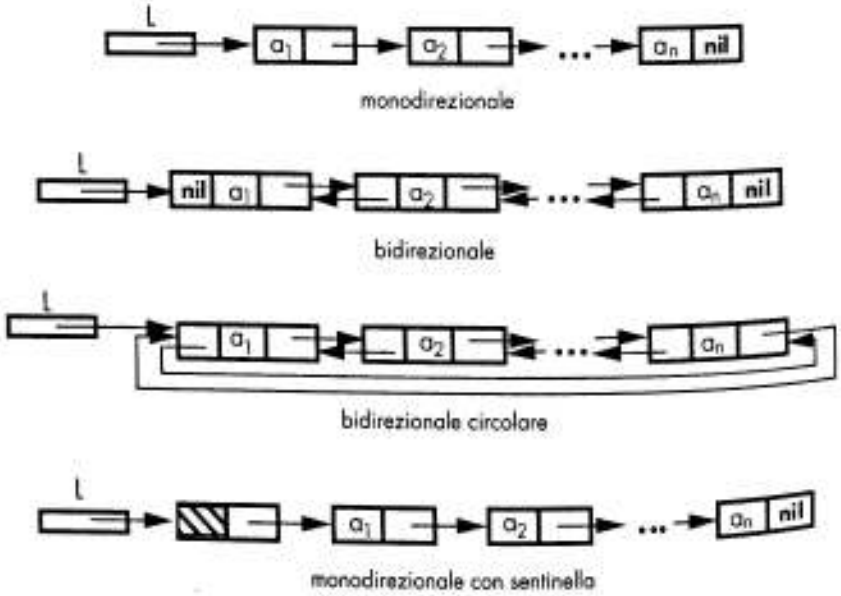


Fig. 4.1 Alcune realizzazioni di una lista con puntatori.

Come mostrato nella Fig. 4.1, ci sono vari modi possibili di realizzare le liste mediante puntatori. L'idea di base è di memorizzare una lista di n elementi in n record, usualmente detti *celle*, tali che l'i-esima cella contenga il valore dell'i-esimo elemento della lista e l'indirizzo (puntatore) della cella contenente l'elemento successivo (realizzazione monodirezionale) o gli indirizzi sia della cella successiva che precedente (realizzazione bidirezionale).

La prima cella è indirizzata da una variabile L di tipo puntatore, mentre l'ultima cella contiene il valore convenzionale **nil** come indirizzo della cella successiva (ovviamente, nel caso bidirezionale, la prima cella contiene **nil** come indirizzo della cella precedente). Ricordiamo che gli indirizzi delle celle sono noti soltanto alla macchina, non al programmatore.

Entrambe le realizzazioni possono essere rese circolari, facendo sì che l'ultima cella contenga l'indirizzo della prima cella e, nel caso bidirezionale, che la prima cella contenga l'indirizzo dell'ultima. Per rendere più leggibile lo pseudocodice delle operazioni insert() e remove() quando operano sugli elementi estremi della lista, può essere conveniente introdurre una cella in più, detta *sentinella*, che è direttamente indirizzata da L, ma non contiene il valore di alcun elemento. Si hanno così otto realizzazioni possibili, a seconda che ci siano o no bidirezionalità, circolarità, e sentinella. La Fig. 4.1 illustra quattro di queste otto possibilità. Tutte le otto realizzazioni richiedono $\Theta(n)$ spazio di memoria.

In questa sezione, descriveremo la realizzazione bidirezionale circolare con sentinella che permette di ottenere complessità $O(1)$ per ciascuna operazione semplificandone nel contempo lo pseudocodice, a scapito di un lieve spreco di memoria. Va però notato che è possibile ottenere complessità $O(1)$ per quasi tutte le operazioni (con l'esclusione dei soli tail() e prev() che hanno complessità $O(n)$) anche con una semplice realizzazione monodirezionale, senza circolarità né sentinelle, usando quindi minore quantità di memoria ma complicando lo pseudocodice delle operazioni.

Con la realizzazione bidirezionale circolare con sentinella, la posizione pos_i è uguale al puntatore della cella contenente a_i, per $1 \leq i \leq n$, mentre pos_0 e pos_{n+1} sono uguali all'indirizzo della sentinella (si veda la Fig. 4.2). In questo caso, i tipi Pos e List coincidono. Si noti che una lista di n elementi è formata da $n + 1$ celle (che contengono gli n elementi più la sentinella) e che quindi una lista vuota contiene esattamente una cella (la sentinella) che ha il proprio indirizzo sia come indirizzo della cella precedente che della cella successiva.

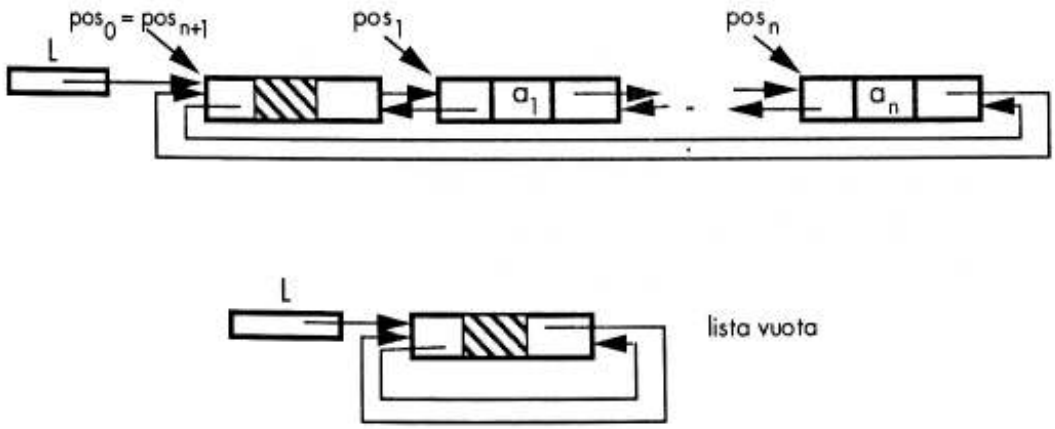


Fig. 4.2 Realizzazione bidirezionale circolare con sentinella.

Si noti che nelle procedure insert() e remove() la posizione $p = pos_i$ non viene modificata all'interno della procedura stessa, mentre viene restituita al chiamante la posizione corrispondente al nuovo elemento i-esimo. In accordo con la specifica, nel caso di insert() il nuovo elemento inserito diventa l'i-esimo, mentre quello che era il vecchio i-esimo diventa l'$(i + 1)$-esimo ecc. Analogamente, nel caso di remove(), il vecchio $(i + 1)$-esimo elemento diventa l'i-esimo ecc. (si veda la Fig. 4.3).

```
LIST
LIST pred                    %Predecessore
LIST succ                    %Successore
ITEM value                   %Elemento

LIST List()
    LIST t ← new List
    t.pred ← t
    t.succ ← t
    return t

boolean empty()
    return pred = succ = this

POS head()
    return succ

POS tail()
    return pred

POS next(POS p)
    return p.succ

POS prev(POS p)
    return p.pred

boolean finished(POS p)
    return (p = this)

ITEM read(POS p)
    return p.value

write(POS p, ITEM v)
    p.value ← v

POS insert(POS p, ITEM v)
    LIST t ← List()
    t.value ← v
    t.pred ← p.pred
    t.pred.succ ← t
    t.succ ← p
    p.pred ← t
    return t

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

Utilizzando i puntatori è molto gravoso, se non addirittura impossibile, verificare che gli input di ogni operazione siano corretti. Per esempio, la specifica di read(p) dovrebbe prevedere che p sia una posizione di un elemento della lista; essendo p realizzato con un puntatore, la verifica di questa precondizione richiederebbe la scansione della lista! Per ragioni di efficienza, la condizione non viene verificata, ed il corretto uso delle operazioni è quindi piena responsabilità del programmatore.

4.2 REALIZZAZIONE CON VETTORI

Una realizzazione alternativa consiste nel memorizzare gli elementi della sequenza in un vettore. In tal caso, la posizione di un elemento è un indice del vettore e *pos*, è proprio uguale ad i, per $1 \leq i \leq n$. In altri termini, alla contiguità tra elementi della sequenza corrisponde una contiguità di memorizzazione. Questa realizzazione permette agevolmente, cioè in tempo $O(1)$, di passare da un elemento al successivo o al precedente, di accorgersi se si tenta di superare un estremo della sequenza, di leggere o modificare il valore di un elemento, anche tramite accesso diretto (non sequenziale).

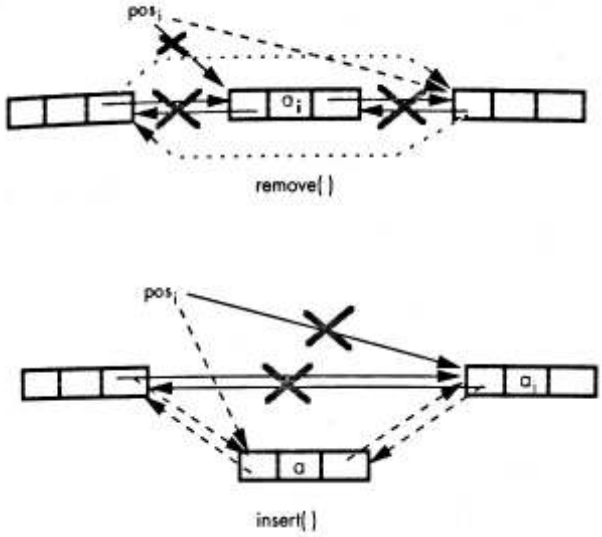


Fig. 4.3 Aggiornamento dei puntatori nelle procedure remove() e insert().

Lo svantaggio principale è che l'inserzione di un nuovo elemento o la cancellazione di uno vecchio richiedono un inaccettabile tempo $O(n)$, poiché è necessario scalare di una posizione a destra, in caso di inserzione, o a sinistra, in caso di cancellazione, tutti gli elementi che seguono quello inserito o cancellato.

Se gli inserimenti avvengono solo in fondo alla sequenza (con costo $O(1)$), questo approccio è ancora accettabile; tuttavia, si presenta un ulteriore problema: il vettore ha dimensione fissa, e il numero di elementi inseriti può crescere oltre la dimensione del vettore. Quando il vettore si esaurisce, una possibilità è quella di allocare un nuovo vettore di dimensione doppia, copiare tutti gli elementi dal vecchio al nuovo vettore, e infine liberare la memoria utilizzata dal vecchio vettore.

Chiaramente il costo di tale operazione è $O(n)$ nel caso pessimo, ma è possibile dimostrare che il costo "ammortizzato" degli inserimenti resta $O(1)$. Per semplificare i calcoli, assumiamo che il vettore abbia inizialmente dimensione 1 e consideriamo il costo di inserire n elementi. Sia c_i il costo dell'i-esimo inserimento:

$$c_i = \begin{cases} i & \text{se } \exists k : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

In altre parole, il costo è normalmente 1; ma un inserimento eseguito immediatamente dopo aver riempito completamente il vettore (ovvero quando il numero di elementi è pari ad una potenza di 2) costa i, perché dobbiamo copiare il vettore vecchio in quello nuovo. Il costo totale $C(n)$ per n inserimenti è quindi:

$$C(n) = \sum_{i=1}^{n} c_i \leq n + \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k$$

La prima componente n è data dagli n inserimenti "puri", mentre la seconda è data dal costo delle copiature. Semplificando l'espressione otteniamo:

$$C(n) \leq n + 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leq n + 2n$$

Quindi il costo totale per n inserimenti è limitato superiormente da $3n$, che significa che il costo ammortizzato per ogni inserimento è $O(1)$.

4.3 PILE E CODE

Se è noto a priori in che modo gli elementi di una sequenza possono essere inseriti o rimossi, è possibile pensare a realizzazioni efficienti basate su vettore.

Una *pila* (in inglese, *stack*) è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi soltanto ad un estremo (la "testa") della sequenza. Può essere quindi vista come un caso speciale di sequenza in cui l'ultimo elemento inserito è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa. Tale meccanismo di accesso è detto Lifo (dall'inglese "Last In First Out", ovvero "ultimo arrivato, primo servito").

ESEMPIO – [Pila]

Una catasta di pratiche burocratiche che si accumulano sulla scrivania di un funzionario ministeriale è spesso (ingiustamente) trattata col meccanismo Lifo, in cui l'ultima pratica inserita in testa alla catasta sarà esaurita per prima. Un tubetto di pastiglie ha un meccanismo di accesso analogo, poiché l'ultima pastiglia inserita nel tubetto dal fabbricante è anche la prima ad essere estratta dall'acquirente (si veda la Fig. 4.4).

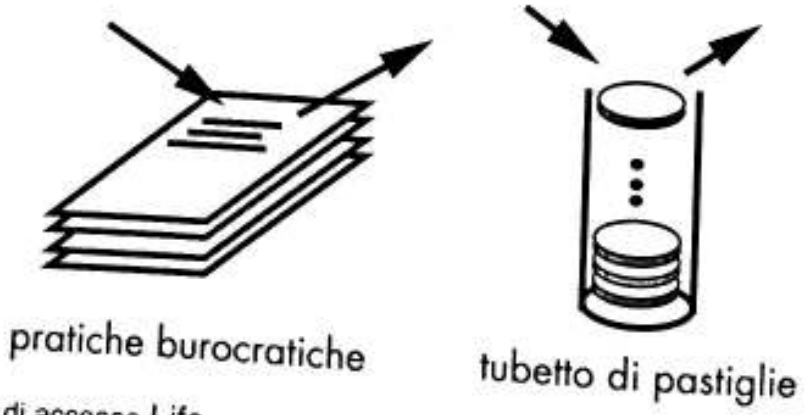


Fig. 4.4 Meccanismo di accesso Lifo.

Analogamente, una *coda* (in inglese, *queue*) è una sequenza di elementi di un certo tipo, in cui è possibile aggiungere elementi ad un estremo (il "fondo") e togliere elementi dall'altro estremo (la "testa"). Può essere considerata come una particolare sequenza in cui il primo elemento inserito è anche il primo ad essere rimosso e non è possibile accedere ad alcun elemento che non sia quello in testa o quello in fondo. Tale meccanismo di accesso è detto Fifo (dall'inglese "First In First Out", ovvero "primo arrivato, primo servito").

ESEMPIO – [Coda, o Fila d'attesa]
In Inghilterra, una fila di persone ad uno sportello postale è gestita con la disciplina Fifo (si veda la Fig. 4.5).

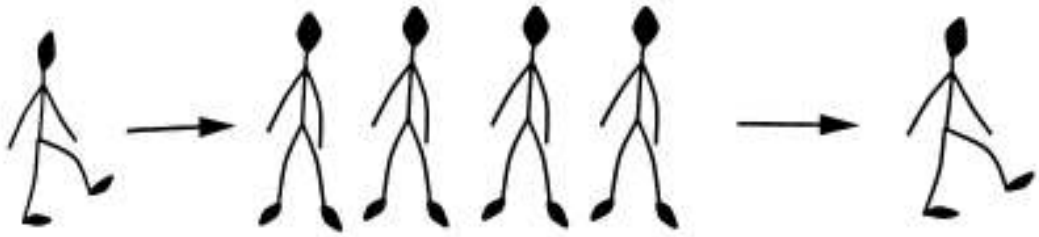


Fig. 4.5 Meccanismo di accesso Fifo.

4.3.1
SPECIFICA

Sebbene le operazioni delle pile e delle code possano essere simulate con le operazioni di una lista, è prassi comune utilizzare nomi leggermente diversi.

Le operazioni per il tipo di dato pila comprendono, oltre alla creazione di una pila vuota, l'interrogazione isEmpty(), per verificare che la pila sia vuota oppure no, l'operazione di lettura top(), per leggere il valore dell'elemento in testa alla pila, e quelle di modifica pop(), per eliminare l'elemento che si trova in testa alla pila, e push(), per aggiungere un elemento in testa alla pila. La specifica sintattica del tipo di dato è la seguente:

STACK

% Restituisce **true** se la pila è vuota
boolean isEmpty()

% Inserisce *v* in cima alla pila
push(ITEM *v*)

% Estrae l'elemento in cima alla pila e lo restituisce al chiamante
ITEM pop()

% Legge l'elemento in cima alla pila
ITEM top()

Le operazioni per il tipo di dato coda comprendono isEmpty() e top() con la stessa specifica delle rispettive operazioni della pila, dequeue() che rimuove un elemento dalla testa della coda, e enqueue() che inserisce un elemento in fondo alla coda.

QUEUE

% Restituisce **true** se la coda è vuota
boolean isEmpty()

% Inserisce *v* in fondo alla coda
enqueue(ITEM *v*)

% Estrae l'elemento in testa alla coda e lo restituisce al chiamante
ITEM dequeue()

% Legge l'elemento in testa alla coda
ITEM top()

Poiché pile e code sono casi particolari di sequenze, ogni realizzazione descritta per le liste funziona anche per le pile e per le code. Per esempio, le operazioni per le pile possono essere simulate con quelle delle liste nel modo seguente:

p.top() = *p*.read(*p*.head())
p.pop() = *p*.remove(*p*.head())
p.push(*v*) = *p*.insert(*p*.head(), *v*).

Usando le realizzazioni delle liste con puntatori viste nel paragrafo precedente, la complessità di ciascuna operazione per le pile è $O(1)$.

È possibile realizzare in modo simile le operazioni della coda; ma è importante notare che la complessità di enqueue() è $O(1)$ se viene usata una realizzazione con liste bidirezionali, mentre diviene $\Theta(n)$ usando le altre realizzazioni.

Oltre alle realizzazioni per pile e code che si basano sulle liste, ci sono due realizzazioni specifiche, che utilizzano vettori, in cui tutte le operazioni delle pile e delle code continuano ad avere complessità $O(1)$. Tali realizzazioni sono utilizzabili quando si è in grado di limitare superiormente con un opportuno valore il numero massimo di elementi che possono essere presenti nella pila o nella coda.

4.3.2
REALIZZAZIONE DI UNA PILA CON VETTORE

Come mostrato nella Fig. 4.6, una realizzazione con vettore del tipo di dato pila consiste nel memorizzare gli *n* elementi della pila, in ordine inverso, nelle prime *n* posizioni di un vettore di $m \geq n$ elementi, mantenendo un cursore alla testa della pila. La pila vuota viene individuata dal valore 0 contenuto nel cursore, mentre la pila satura, cioè che contiene il massimo numero consentito di elementi, è individuata dal valore *m*. pop() è realizzata decrementando il valore del cursore, mentre push() è realizzata incrementando il valore del cursore ed inserendo il nuovo elemento nella posizione del vettore individuata dal cursore. Eventuali tentativi di inserzione in una pila satura o di estrazione da

una pila vuota possono sorgere in caso di errori di programmazione e non sono ammessi. In particolare, il tentativo di inserzione in una pila satura è segnalato con un opportuno messaggio di errore. Con questa realizzazione, ogni operazione della pila richiede tempo costante.

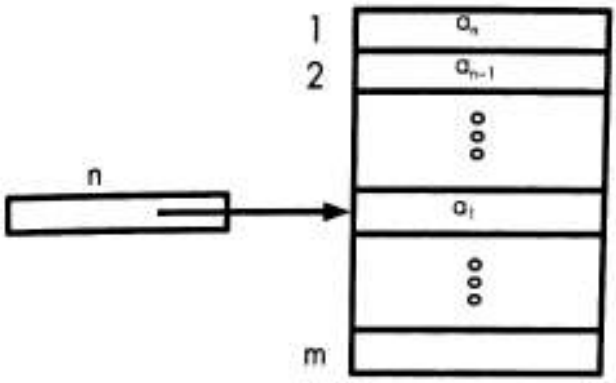


Fig. 4.6 Realizzazione di una pila con un vettore.

```
STACK
ITEM[] A                    %Elementi
integer n                   %Cursore
integer m                   %Dim. Massima

STACK Stack(integer dim)
    STACK t ← new STACK
    t.A ← new integer [1...dim]
    t.m ← dim
    t.n ← 0
    return t

ITEM top()
    precondition: n > 0
    return A[n]

boolean isEmpty()
    return n = 0

ITEM pop()
    precondition: n > 0
    ITEM t ← A[n]
    n ← n − 1
    return t

push(ITEM v)
    precondition: n < m
    n ← n + 1
    A[n] ← v
```

4.3.3
REALIZZAZIONE DI UNA CODA CON VETTORE CIRCOLARE

Si supponga di avere un vettore di m elementi, indicati da 0 a $m - 1$, in cui l'elemento di indice 0 è considerato come successore di quello di indice $m - 1$. La coda è contenuta in n posizioni consecutive del vettore, per esempio in verso orario, e i suoi elementi estremi sono individuati da due cursori. Per cancellare o per inserire un elemento, viene incrementato di uno (modulo m) il cursore di testa o di fondo, rispettivamente. In questo modo la coda "migra" lungo il vettore circolare e, ad ogni istante, si trova in posizioni consecutive del vettore secondo il verso orario.

Anziché utilizzare due cursori, per individuare la coda è sufficiente conoscere l'indice

del primo elemento e la lunghezza della coda stessa, come mostrato nella Fig. 4.7. Infatti, se i è la posizione dell'elemento in testa alla coda e ci sono n elementi in essa, allora quella dell'ultimo elemento della coda è $(i + n - 1)$ modulo m. La complessità di ogni operazione è $O(1)$.

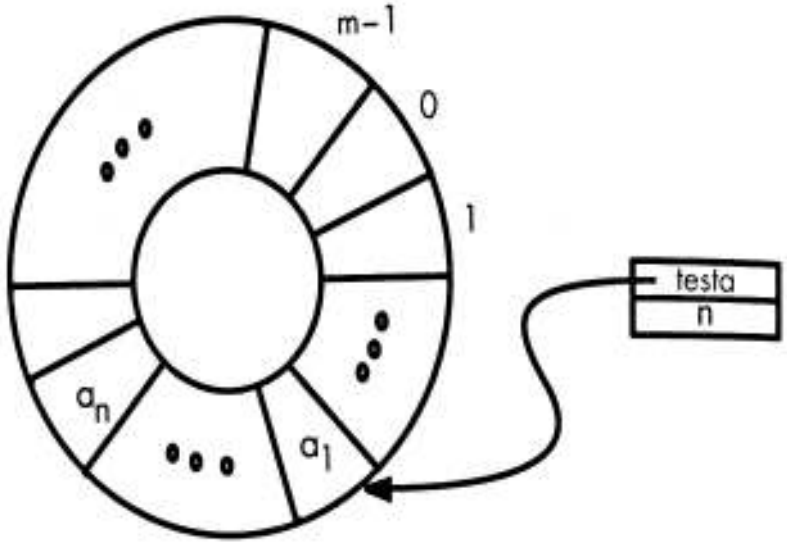


Fig. 4.7 Realizzazione di una coda con un vettore circolare.

```
QUEUE
ITEM[] A                    %Elementi
integer n                   %Dim. attuale
integer testa               %Testa
integer m                   %Dim. Massima

QUEUE Queue(integer dim)
    QUEUE t ← new QUEUE
    t.A ← new integer [0...dim − 1]
    t.m ← dim
    t.testa ← 0
    t.n ← 0
    return t

ITEM top()
    precondition: n > 0
    return A[testa]

boolean isEmpty()
    return n = 0

ITEM dequeue()
    precondition: n > 0
    ITEM t ← A[testa]
    testa ← (testa + 1) mod m
    n ← n − 1
    return t

enqueue(ITEM v)
    precondition: n < m
    A[(testa + n) mod m] ← v
    n ← n + 1
```

4.4
PILE E PROCEDURE RICORSIVE

Una importante applicazione delle pile consiste nella gestione dell'esecuzione di procedure ricorsive. Infatti, l'esecuzione di una procedura ricorsiva prevede il "salvataggio" dei dati sui quali lavora la procedura al momento di una nuova chiamata ricorsiva interna. Tali

dati sono successivamente "riesumati" quando la computazione interna termina e viene ripresa la computazione che era stata sospesa. Questo meccanismo è ovviamente di tipo Lifo, poiché la computazione "più interna" di una procedura ricorsiva, cioè quella iniziata per ultima, è sempre la prima a terminare. Per chiarire come procede il calcolo in una procedura ricorsiva, si consideri un problema "leggendario", divenuto ormai un classico della programmazione ricorsiva:

TORRI DI HANOI In un monastero tibetano ci sono n dischi d'oro, forati al centro, tutti di diametro diverso. I dischi sono infilati in un piolo verticale, accatastati per diametro decrescente a partire dal basso. I monaci devono spostare tutti i dischi dal piolo in cui si trovano in un altro piolo, formando una catasta identica a quella di partenza. È possibile usare un terzo piolo di appoggio per effettuare i trasferimenti. Si può sfilare da un piolo soltanto un disco per volta, e il disco sfilato deve essere subito infilato in un altro piolo evitando di sovrapporlo ad un disco di diametro più piccolo.[1]

Questo rompicapo si presta ad una semplice risoluzione ricorsiva. Indichiamo i tre pioli coi nomi *origine*, *destinazione* e *intermedio*. Se c'è un solo disco, la soluzione è banale: si trasferisce direttamente il disco dal piolo *origine* al piolo *destinazione*. Se ci sono n dischi, con $n > 1$, basta spostare i primi $n - 1$ dischi della catasta dal piolo *origine* al piolo *intermedio*, indi trasferire direttamente il disco rimasto dal piolo *origine* al piolo *destinazione*, e infine spostare gli $n - 1$ dischi dal piolo *intermedio* al piolo *destinazione*. Supponendo che *origine*, *intermedio* e *destinazione* siano tre interi distinti (p.e. 1, 2 e 3) si ottiene la seguente procedura:

```
hanoi-ricorsiva(integer n, integer origine, integer destinazione, integer intermedio)
  if n = 1 then
  |   % Trasferisci un disco da origine a destinazione
  else
(1) |   hanoi-ricorsiva(n – 1, origine, intermedio, destinazione)
    |   % Trasferisci un disco da origine a destinazione
(2) |   hanoi-ricorsiva(n – 1, intermedio, destinazione, origine)
```

Benché sia facile dimostrare la correttezza di questa procedura usando il principio d'induzione, è molto difficile capire quale sia l'effettiva sequenza di trasferimenti eseguiti dalla procedura stessa.

4.4 ESEMPIO – [Torri di Hanoi]
Si assuma $n = 3$, *origine* = 1, *intermedio* = 2 e *destinazione* = 3. L'esecuzione della procedura hanoi-ricorsiva() con questi dati provoca la seguente sequenza di chiamate ricorsive e di trasferimenti di dischi (si veda la Fig. 4.8):

[1] Narra la leggenda che il mondo finirà prima che i monaci abbiano spostato tutti i dischi!

1. hanoi-ricorsiva(3,1,3,2)
    1.1. hanoi-ricorsiva(2,1,2,3)
        1.1.1. hanoi-ricorsiva(1,1,3,2)
            1.1.1.1. trasferisci un disco da 1 a 3
        1.1.2. trasferisci un disco da 1 a 2
        1.1.3. hanoi-ricorsiva(1,3,2,1)
            1.1.3.1. trasferisci un disco da 3 a 2
    1.2. trasferisci un disco da 1 a 3
    1.3. hanoi-ricorsiva(2,2,3,1)
        1.3.1. hanoi-ricorsiva(1,2,1,3)
            1.3.1.1. trasferisci un disco da 2 a 1
        1.3.2. trasferisci un disco da 2 a 3
        1.3.3. hanoi-ricorsiva(1,1,3,2)
            1.3.3.1. trasferisci un disco da 1 a 3

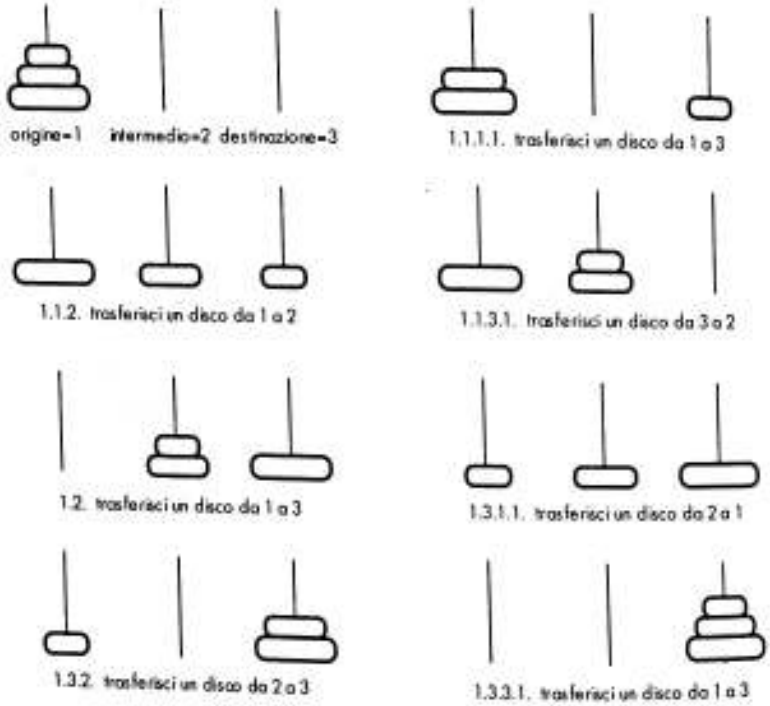


Fig. 4.8 Sequenza di trasferimenti di dischi eseguiti dalla procedura ricorsiva hanoi-ricorsiva() per $n = 3$.

La complessità della procedura hanoi-ricorsiva() è esponenziale. Infatti, il numero $T(n)$ di trasferimenti di dischi è dato da:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n-1)+1 & \text{se } n \geq 2 \end{cases}$$

Sostituendo successivamente si ottiene:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 = 4T(n-2)+3 \\ &\vdots \\ &= 2^{n-1}T(1)+2^{n-1}-1 = 2^n-1 \end{aligned}$$

Assumendo un trasferimento ogni microsecondo, per $n = 100$ occorrono ben 400.000 miliardi di secoli: nessun dubbio che il mondo finirà prima!

Vediamo come sia possibile eliminare manualmente la ricorsione dalla procedura hanoi-ricorsiva(), trasformandola in una procedura iterativa equivalente chiamata hanoi-iterativa(), utilizzando una pila. La procedura è organizzata attorno ad un ciclo **while** principale la cui esecuzione "simula" una singola chiamata di hanoi-ricorsiva(). All'inizio di ogni esecuzione del ciclo principale, le variabili *n*, *origine*, *destinazione* e *intermedio* rappresentano i parametri formali della chiamata simulata. Ogni chiamata ricorsiva "aperta" ma non ancora terminata viene simulata da una tupla contenente i parametri formali inserita nella pila. Per snellire il codice, la tupla è letta e scritta tramite parentesi angolari ⟨⟩.

L'invocazione ricorsiva (1) viene simulata dal ciclo **while** interno, che inserisce nella pila le successive chiamate per $n - 1$, $n - 2$, ..., 3, 2. In ognuna di queste chiamate, vengono scambiate le variabili *destinazione* e *intermedio*. Quando $n = 1$ (caso base), il disco nel piolo *origine* viene spostato nel piolo *destinazione*. A questo punto, si estraggono dalla pila i parametri formali dell'ultima chiamata lasciata aperta, si sposta un disco da *origine* a *destinazione*, e si simula l'invocazione ricorsiva (2) decrementando *n* di 1 e scambiando i pioli *origine* e *intermedio* (i pioli *intermedio* e *destinazione* erano già stati scambiati) e ricominciando il ciclo, che continua fino a quando la pila non è vuota.

```
hanoi-iterativa(integer n, integer origine, integer destinazione, integer intermedio
  STACK S ← Stack()
  boolean finedelmondo ← false
  while not finedelmondo do
      while n ≠ 1
          S.push(⟨n, origine, destinazione, intermedio⟩)
          n ← n − 1
          destinazione ↔ intermedio
      % Trasferisci un disco da origine a destinazione
      if S.isEmpty() then
          finedelmondo ← true
      else
          ⟨n, origine, destinazione, intermedio⟩ ← S.pop()
          % Trasferisci un disco da origine a destinazione
          n ← n − 1
          origine ↔ intermedio
```

Anche se questo è un caso semplice, con solo due chiamate, l'approccio seguito per la trasformazione non è intuitivo. Nei casi più complessi, seguire il filo logico della ricorsione è ancora più difficile. Esiste tuttavia un approccio sistematico per la trasformazione di *ogni* procedura ricorsiva in una iterativa equivalente, che consiste nel salvare, oltre ai parametri attuali delle chiamate (e le eventuali variabili locali), anche il "punto di ritorno", ovvero il punto del codice da cui riprendere l'esecuzione. Le regole da seguire sono le seguenti:

(1) Creare una pila vuota e iniziare un ciclo **while** principale con i valori dei parametri attuali ricevuti dalla chiamata.

(2) Creare una variabile *pos* che rappresenta il particolare spezzone di codice da eseguire nel ciclo **while**. Per questa variabile, utilizziamo un'etichetta INIZIO per indicare l'inizio di una chiamata; un'etichetta per indicare tutti gli spezzoni di codice che seguono il rientro da una chiamata (nel nostro caso, DOPO1 dopo la chiamata (1) e DOPO2 dopo la chiamata (2)); FINE per indicare il codice da eseguire al termine della chiamata (che qui coincide con DOPO2); e STOP per indicare che l'esecuzione è terminata.

(3) Sostituire ogni chiamata ricorsiva con un'istruzione che salva nella pila i valori dei parametri, delle variabili locali e l'etichetta da cui ripartire dopo il completamento della chiamata ricorsiva, seguita da istruzioni che modificano i parametri assegnando loro i valori che devono assumere nella chiamata ricorsiva.

(4) Scrivere il codice per FINE, che controlla se la pila è vuota e, in caso affermativo, termina il ciclo mettendo *pos* = STOP; altrimenti, estrae dalla pila i valori salvati dei parametri, delle eventuali variabili locali e della prossima etichetta da eseguire.

Tali regole sono valide se tutti i parametri sono passati per valore (si consideri per esercizio cosa accade quando ci sono parametri passati per reference). Applicando queste regole ad hanoi-ricorsiva(), si ottiene la seconda versione di hanoi-iterativa() presentata di seguito.

4.5 REALITY CHECK

Nelle librerie Java, sono presenti due realizzazioni del tipo di dati astratto "sequenza" basate su vettori: `Vector` e `ArrayList`, entrambe nel package `java.util`. `Vector` è presente sin dalle prime versioni di Java (JDK 1.0), mentre `ArrayList` è stato aggiunto con la libreria Java Collections (JDK 1.2). `Vector` d'altronde è stato riscritto per realizzare l'interfaccia `List` delle Collections, quindi `Vector` e `ArrayList` sembrano del tutto simili. Perché dunque ve ne sono due?

Leggendo fra le righe della documentazione di `ArrayList`, si scoprono alcuni dettagli interessanti: *The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. [...] The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.* Questi commenti non sono presenti nella documentazione di `Vector`; invece, si fa specifico riferimento al fatto che `Vector` non cresce raddoppiando, ma allocando un nuovo vettore più lungo di un numero costante d di elementi.

Proviamo a calcolare il costo ammortizzato di n inserimenti quando l'incremento è costante e non si raddoppia. Si assuma che la dimensione iniziale del vettore sia d e denotiamo con c_i il costo dell'i-esimo inserimento:

$$c_i = \begin{cases} i & \text{se } i \bmod d = 1 \\ 1 & \text{altrimenti} \end{cases}$$

```
hanoi-iterativa(integer n, integer origine, integer destinazione, integer intermedio)
STACK S ← Stack()
integer pos ← INIZIO
while pos ≠ STOP do
    if pos = INIZIO then
        if n = 1 then
            % Trasferisci un disco da origine a destinazione
            pos ← FINE
        else
            S.push(⟨n, origine, destinazione, intermedio, DOPO1⟩)
            n ← n − 1
            destinazione ↔ intermedio
    else if pos = DOPO1 then
        % Trasferisci un disco da origine a destinazione
        S.push(⟨n, origine, destinazione, intermedio, FINE⟩)
        n ← n − 1
        origine ↔ intermedio
        pos ← INIZIO
    else if pos = FINE then
        if S.isEmpty() then
            pos ← STOP
        else
            ⟨n, origine, destinazione, intermedio, pos⟩ ← S.pop()
```

Il costo totale $C(n)$ di n inserimenti è quindi $O(n^2)$, come illustrato qui sotto, e il costo ammortizzato per singolo inserimento è $O(n)$:

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n} c_i \\
&\leq n + \sum_{j=1}^{\lfloor n/d \rfloor} d \cdot j \\
&= n + d\frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\
&\leq n + \frac{(n/d + 1)n}{2} = O(n^2)
\end{aligned}
$$

Che lezione trarre da questa lettura della documentazione? Una scelta a prima vista banale (meglio incrementare o raddoppiare?) si rivela di estrema importanza; e la scelta corretta potrebbe essere controintuitiva (meglio raddoppiare!). Chi ha scritto `Vector` non aveva seguito un buon corso di Algoritmi e Strutture Dati!

4.6 [I primi n interi] Si scriva una procedura ricorsiva di complessità ottima che, dato un intero $n \geq 1$, costruisca due liste L ed M tali che $L = 1, 2, ..., n-1, n$ ed $M = n, n-1, ..., 2, 1$.

4.7 [Epurazione] Si scriva una procedura che, data una lista L di interi, restituisca un'altra lista M che contenga solo gli elementi di L i cui valori non compaiono esattamente due volte (p.e., se $L = 5, 7, 3, 2, 2, 1, 2, 3$, allora $M = 5, 7, 2, 2, 1, 2$).

4.8 Si supponga di rappresentare un numero decimale d mediante una lista L tale che il valore dell'elemento i-esimo della lista sia uguale a quello dell'i-esima cifra del numero. Si scriva una funzione che effettua la somma di due numeri. (p.e.: se $d = 190$ e $d' = 17$ allora $L = 1, 9, 0$, $L' = 1, 7$, e il risultato $d'' = d + d'$ è restituito nella lista $L'' = 2, 0, 7$).

4.9 Dopo essere stati inquisiti per tangentopoli, n politici hanno deciso di suicidarsi disponendosi in cerchio e uccidendo via via una persona ogni k (... ma l'ultimo rimasto si suiciderà poi per davvero?). Si scriva una procedura che, data la lista L degli aspiranti suicidi, e dato k, $1 \leq k \leq n$, stampi l'ordine delle vittime (p.e. se L = Caio, Sempronio, Tizio, Tullio e $k = 3$, allora l'ordine delle vittime è: Tizio, Sempronio, Tullio, Caio). Si assuma di poter cancellare gli elementi da L.

4.10 Si scriva una procedura ricorsiva che data una lista L di interi, la modifichi eliminando ogni elemento pari e replicando ogni elemento dispari tante volte quanti sono gli elementi pari che lo precedono (p.e., $L = 4, 6, 7, 3, 2, 5$, allora si ha $L = 7, 7, 7, 3, 3, 3, 5, 5, 5, 5$).

4.11 Si scrivano le procedure per le operazioni delle liste usando la realizzazione con puntatori monodirezionale non circolare e senza sentinella. Tutte le operazioni (tranne tail() e prev()) devono avere complessità $O(1)$. (Suggerimento: si definisca pos_i uguale all'indirizzo della cella contenente a_{i-1}, se $i > 0$, e pos_i = **nil**, se $i = 1$).

4.12 È possibile memorizzare due pile in un unico vettore comune, qualora se ne facciano "crescere" una a partire dal primo elemento del vettore e l'altra a partire dall'ultimo. Si assuma di avere in memoria un solo vettore disponibile per memorizzare le pile. In tal caso, non potranno essere create più di due pile. Si riscrivano le procedure relative alle usuali operazioni sulle pile, includendo le opportune condizioni di errore qualora si cerchi di creare più di due pile.

C Utilizzando le tecniche di trasformazione illustrate in questo capitolo, si fornisca una versione iterativa della procedura ricorsiva primiN(), che computa la lista dei primi n interi, mostrata nell'Esercizio 1.

C Il tipo di dato DEQUEUE (double ended queue) è una sequenza modificabile ad entrambi gli estremi, in cui è possibile inserire un elemento in testa o inserirlo in fondo, e cancellare un elemento dalla testa o cancellarlo dal fondo. Si fornisca una specifica ed una realizzazione mediante puntatori.

C Si realizzino le operazioni della coda, modificando la realizzazione monodirezionale non circolare di una lista, ottenendo complessità $O(1)$ anche per enqueue(). Si usino n celle in modo che la prima cella sia indirizzata da un puntatore *testa* e l'ultima da un puntatore *fondo*, come mostrato nella Fig. 4.9.

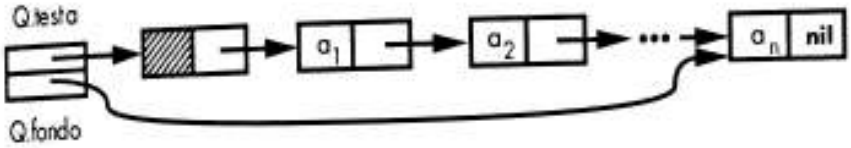


Fig. 4.9 Realizzazione monodirezionale con sentinella di una coda.

C Il *Pigeonhole Sort* ("ordinamento della piccionaia") è un'estensione del Counting Sort che permette di ordinare in tempo lineare coppie (chiave, valore) invece che singoli interi. Le chiavi devono essere valori interi compresi fra 1 e k. Il funzionamento è simile al Counting Sort; tuttavia, durante la scansione iniziale, se si incontra una coppia chiave-valore (i, v) si aggiunge v in fondo ad una lista $B[i]$ (invece di incrementare il contatore $B[i]$). Durante la scansione del vettore B, si trascrivono nuovamente tutte le coppie chiavi-valore, in ordine di chiave e mantenendo lo stesso ordinamento per i valori con la stessa chiave. Realizzate il Pigeonhole Sort nel vostro linguaggio preferito.

4.4 Una limitazione inferiore alla complessità del problema è $\Omega(n)$ perché un qualsiasi algoritmo deve almeno generare gli n elementi della lista L e della lista M. La procedura primiN() effettua un inserimento in testa alla lista L ed uno in fondo alla lista M ad ogni chiamata ricorsiva, in modo da memorizzare gli interi in senso crescente in L e decrescente in M.

```
primiN(integer n, LIST L, LIST M)

  if n ≥ 1 then
      L.insert(L.head(), n)
      M.insert(M.next(M.tail()), n)
      primiN(n − 1, L, M)
```

4.5 Si effettua una doppia scansione della lista L per selezionare un elemento da L, verificare che non compaia esattamente 2 volte, e inserirlo in M. La posizione p è quella dell'elemento di L da verificare, la posizione q è utilizzata per scandire L alla ricerca delle occorrenze dell'elemento di posizione p, mentre *conta* serve a contare quante volte tale elemento si ripete. La posizione r è quella che segue l'ultimo elemento di M. La complessità è $O(n^2)$.

```
epurazione(LIST L, LIST M)

  LIST M ← List()
  POS p ← L.head()
  while not L.finished(p) do
      conta ← 0
      POS q ← L.head()
      while not L.finished(q) do
          if L.read(q) = L.read(p) then
              conta ← conta + 1
          q ← L.next(q)
      if conta ≠ 2 then
          M.insert(M.next(M.tail()), L.read(p))
      p ← L.next(p)
```

4.4 Il problema può essere risolto semplicemente scorrendo la lista, ripartendo dalla testa della lista tutte le volte che si arriva in fondo.

```
tangentopoli(LIST L)
  POS p ← L.head()
  while not L.empty() do
      for i ← 1 to k − 1 do
          p ← L.next(p)
          if L.finished(p) then p ← L.head()
      print L.read(p)
      p ← L.remove(p)
```

4.11 Illustriamo un algoritmo di PigeonholeSort in pseudocodice, che ordina un vettore di RECORD in base al campo numerico *key* associato ad ognuno di essi.

```
PigeonholeSort(RECORD[] A, integer n, integer min, integer max)
  integer size ← max − min +1
  LIST[] L ← new LIST [0 ... size − 1]
  for j ← 1 to size do
      L[j] ← List()
  for i ← 1 to n do
      LIST M ← L[A[i].key − min]
      M.insert(M.tail(), A[i])
  i ← 1
  for j ← 0 to size − 1 do
      POS p ← L[j].head()
      while not L[j].finished(p) do
          A[i] ← L[j].read(p)
          i ← i +1
          p ← L[j].next()
```

Capitolo 5

ALBERI

L'albero ordinato (spesso chiamato più semplicemente "albero") è una struttura informativa fondamentale, che si presta a rappresentare svariate situazioni, quali:

- partizioni successive di un insieme in sottoinsiemi disgiunti;
- organizzazioni gerarchiche di dati;
- procedimenti enumerativi o decisionali.

ESEMPIO – [Classificazioni botaniche e zoologiche]
Le classificazioni delle specie vegetali e animali usate nelle scienze naturali rappresentano un ben noto esempio di strutturazione gerarchica. La Fig. 5.1 mostra una delle tante classificazioni del Regno Vegetale.

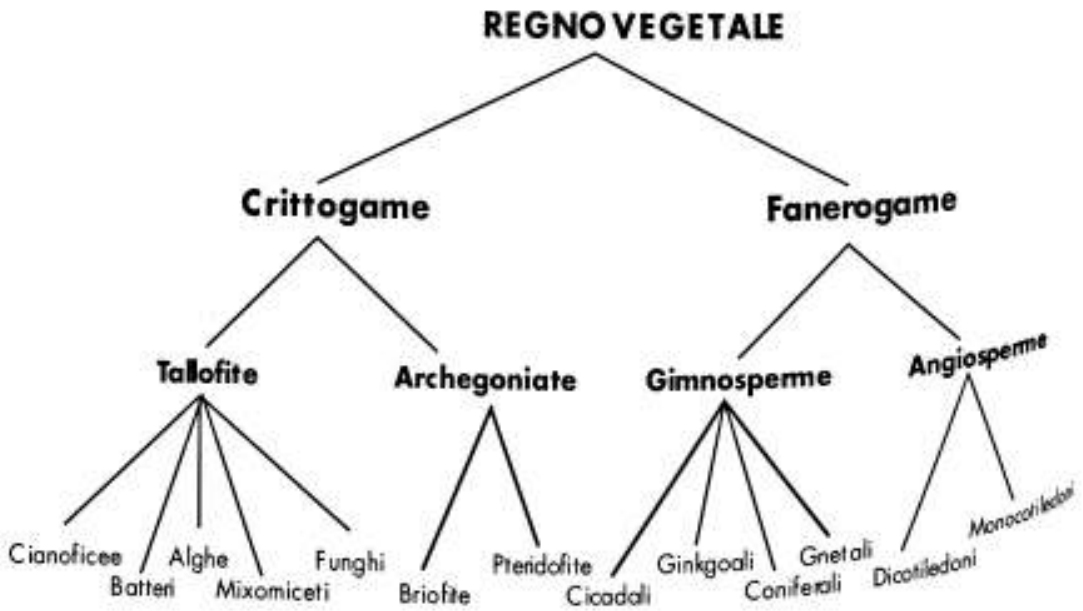


Fig. 5.1 Classificazione del regno vegetale.

ESEMPIO – [Chiamate ricorsive]
Le chiamate di un algoritmo ricorsivo possono essere rappresentate con una struttura ad albero. Ad esempio, le chiamate dell'algoritmo hanoi-ricorsiva() per $n = 3$ discusse nell'Esempio 4.3 possono essere raffigurate come in Fig. 5.2.

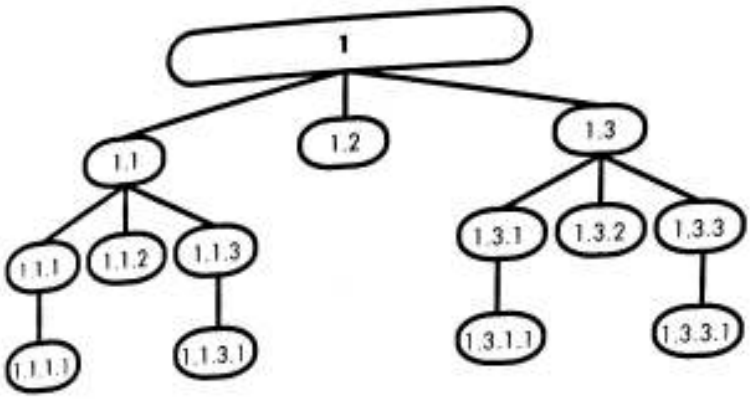


Fig. 5.2 Struttura delle chiamate ricorsive di hanoi-ricorsiva() per $n = 3$.

La terminologia usata nel trattare gli alberi è uno strano misto di termini matematici, botanici e di parentela. Sia T un albero ordinato di n nodi di radice r. Siano $T_1, ..., T_k$ gli insiemi disgiunti non vuoti in cui sono partizionati i rimanenti $n - 1$ nodi di T, aventi come radici, rispettivamente, i nodi $r_1, ..., r_k$. Ciascun T_i è detto *sottoalbero* (*subtree*, in inglese) di T radicato in r_i, mentre ciascun r_i è detto *figlio* (*child*) di r. I nodi $r_1, ..., r_k$ sono tra loro *fratelli* (*sibling*), ed r è il loro *padre* (*parent*). Un *discendente* di un nodo u è un figlio di u oppure un figlio di un discendente di u; analogamente, un *antenato* di u è suo padre oppure il padre di un suo antenato. Un nodo senza figli è detto *foglia* (*leaf*), mentre la *radice* dell'albero (*root*) è l'unico nodo senza padre. In pratica, in un albero ordinato è stabilito un doppio ordinamento tra i nodi. In primo luogo c'è un ordinamento "verticale" per *livelli*, cioè in base alla distanza dei nodi dalla radice. La radice r è a livello 0, tutti i figli di r stanno al livello 1, i figli dei figli di r stanno al livello 2, e così via fino all'esaurimento dei nodi. L'*altezza* di un albero è il massimo livello delle sue foglie. In secondo luogo, è stabilito un ordinamento "orizzontale" tra i nodi che stanno allo stesso livello, distinguendo così tra il primo figlio di un nodo (il figlio maggiore), il suo secondo figlio, e così via fino all'ultimo (il figlio minore).

Come visto negli esempi precedenti, i nodi si disegnano spesso con cerchietti, mentre coppie padre-figlio si visualizzano con linee che collegano coppie di cerchietti. La terminologia italiana pare di origine patriarcale (si usano i termini padre, figlio e fratello, anziché madre, figlia e sorella), mentre quella inglese è più "politicamente corretta". Inoltre gli alberi sono usualmente rappresentati rovesciati, con la radice in alto e le foglie in basso (si veda la Fig. 5.3).

ESEMPIO – [Alberi ordinati]
La Fig. 5.4 mostra tre alberi ordinati T_1, T_2, e T_3. Pur essendo formati dagli stessi 5 nodi, i tre alberi sono diversi tra loro. Infatti, la radice di T_1 (il nodo 1) è diversa da quella di T_2 e T_3 (il nodo 2). Inoltre, considerando i nodi allo stesso livello ordinati da sinistra verso destra, T_2 e T_3 sono diversi tra loro. Infatti, pur avendo la stessa radice (il nodo 2), gli stessi figli della radice (i nodi 1, 3, 4), e lo stesso figlio del nodo 3 (il nodo 5), l'ordine dei figli di 2 non è lo stesso in T_2 e in T_3.

Fig. 5.3 L'albero nel mondo informatico.

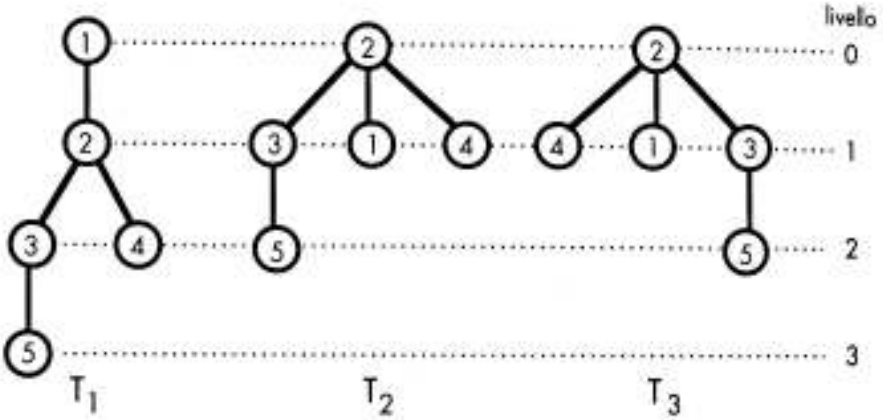


Fig. 5.4 Alberi ordinati.

5.1
SPECIFICA

L'albero ordinato può essere interpretato come una generalizzazione di una sequenza lineare, in cui ciascun elemento ha ancora un solo predecessore (il padre), ma in generale può avere più di un successore (i figli). Infatti, una sequenza è un particolare albero in cui ogni nodo ha al più un figlio e c'è una sola foglia.

Analogamente alle operazioni sulle sequenze, quelle definite sugli alberi ordinati prevedono la possibilità di interrogare un nodo per ottenere informazioni sui suoi parenti, quali il padre, i figli o i fratelli. A seconda della particolare utilizzazione della struttura di dati albero, alcune di queste "parentele" possono rivelarsi non necessarie; per lo stesso motivo, anche la modalità con cui l'insieme dei figli viene ispezionato può dipendere dall'applicazione. La specifica presentata in questa sezione è generica e permette di accedere a tutte le informazioni di parentela. Nel seguito, vedremo come versioni ridotte di questa specifica possono essere realizzate in maniera efficiente, occupando una quantità limitata di memoria.

Nella nostra specifica, i concetti di nodo e di radice di sottoalbero sono considerati sinonimi: le operazione che dovrebbero restituire dei nodi (come per esempio quelle che permettono di ispezionare i figli) restituiscono elementi di tipo TREE, che rappresenta la radice di un sottoalbero. La creazione di un nuovo albero corrisponde alla creazione di un singolo nodo isolato, senza padre né figli. Così come nelle liste, esistono operazioni quali read() e write() che permettono di leggere e scrivere il contenuto dei nodi.

```
TREE
% Costruisce un nuovo albero, costituito da un solo nodo contenente v
Tree(ITEM v)

% Legge il valore
ITEM read()

% Scrive v nel nodo
write(ITEM v)

% Restituisce il padre; nil se questo nodo è radice
TREE parent()

% Restituisce il primo figlio; nil se questo nodo è foglia
TREE leftmostchild()

% Restituisce il prossimo fratello del nodo a cui è applicato; nil se assente
TREE rightSibling()

% Inserisce il sottoalbero t come primo figlio di questo nodo
insertChild(TREE t)
  precondition: t.parent() = nil

% Inserisce il sottoalbero t come successivo fratello di questo nodo
insertSibling(TREE t)
  precondition: t.parent() = nil

% Distrugge il sottoalbero radicato nel primo figlio di questo nodo
deleteChild()

% Distrugge il sottoalbero radicato nel prossimo fratello di questo nodo
deleteSibling()
```

Sono presenti le operazioni parent(), leftmostChild() e rightSibling(), che restituiscono rispettivamente il padre, il primo figlio e il fratello successivo di un nodo. Le ultime due possono essere utilizzate per scorrere la lista dei figli. Se parent() restituisce **nil**, il nodo è radice; se leftmostChild() restituisce **nil**, il nodo è foglia (ovviamente, un nodo può essere radice e foglia contemporaneamente). Se rightSibling() restituisce **nil**, è stata ispezionata l'intera lista di figli.

La creazione di nuovi alberi sfrutta la ricorsività della definizione di questa struttura di dati; oltre al caso base, rappresentato dalla costruzione di alberi costituiti da singoli nodi isolati (descritta in precedenza), è possibile "aggiungere" un sottoalbero come figlio (operazione insertChild()) o come fratello successivo (operazione insertSibling()) di un particolare nodo. La precondizione per applicare questi metodi con successo è che l'albero da aggiungere non sia già sottoalbero di un altro albero (parent()= **nil**). Infine, sono presenti le operazioni deleteChild() e deleteSibling() che distruggono il sottoalbero radicato nel primo figlio e nel fratello successivo del nodo su cui si applica l'operazione. Gli effetti delle operazioni insertChild(), insertSibling(), deleteChild() sono illustrati nella Fig. 5.5.

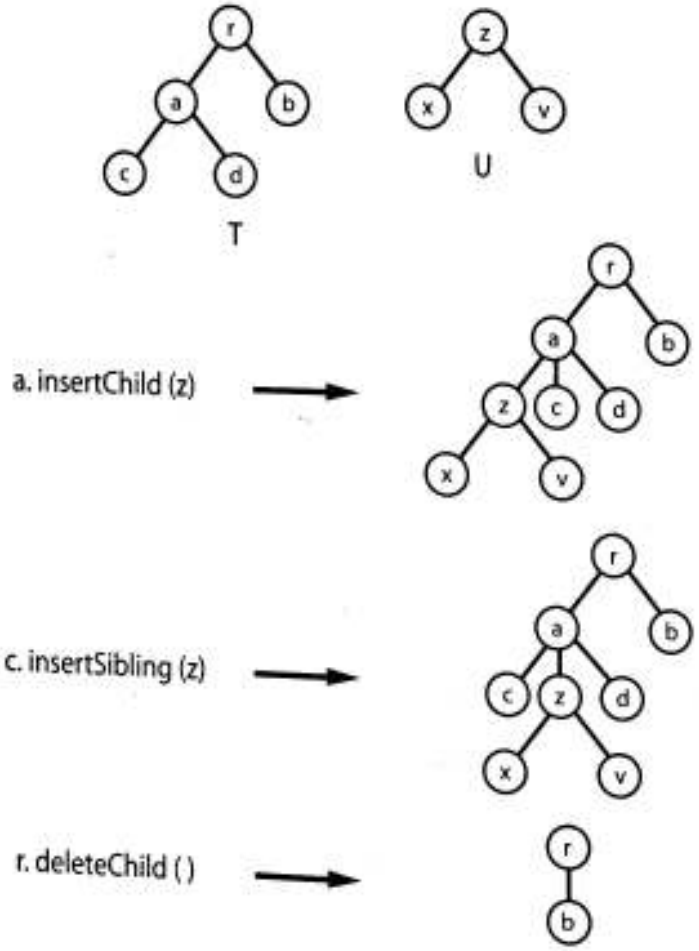


Fig. 5.5 Inserzioni e cancellazioni di sottoalberi.

5.2 VISITE

La visita di un albero ordinato consiste nel seguire una "rotta" di viaggio che consenta di esaminare ogni nodo dell'albero esattamente una volta. La visita può essere eseguita in vari modi, detti "ordini", tra cui i quattro più importanti sono: ordine anticipato (o *previsita*), ordine posticipato (o *postvisita*), ordine simmetrico (o *invisita*), ordine per livelli. I primi tre metodi sono *visite in profondità*, perché si segue un percorso radice-foglia, mentre l'ultima è detta anche *visita in ampiezza*, perché vengono visitati i nodi livello per livello, "orizzontalmente", fino a raggiungere il livello massimo dell'albero.

Sia T un albero non vuoto di radice r. Se r non è una foglia ma ha $k > 0$ figli, indichiamo con $T_1, ..., T_k$ i k sottoalberi di T radicati nei k figli di r. Gli ordini di visita in profondità sono definiti ricorsivamente come segue:

- la *previsita* di T consiste nell'esaminare r e poi nell'effettuare, nell'ordine, la previsita di $T_1, ..., T_k$;
- la *postvisita* di T consiste nell'effettuare, nell'ordine, la postvisita di $T_1, ..., T_k$ e poi nell'esaminare r;
- Fissato $i \geq 1$, la *invisita* di T consiste nell'effettuare la invisita di $T_1, ..., T_i$, nell'esaminare r, e poi nell'effettuare la invisita di $T_{i+1}, ..., T_k$.

ESEMPIO – [Visita di alberi ordinati]

Gli ordini di visita dell'albero della Fig. 5.1 sono:

- **Previsita**: Regno vegetale, Crittogame, Tallofite, Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Archegoniate, Briofite, Pteridofite, Fanerogame, Gimnosperme, Cicadali, Ginkgoali, Coniferali, Gnetali, Angiosperme, Dicotiledoni, Monocotiledoni;
- **Postvisita**: Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Tallofite, Briofite, Pteridofite, Archegoniate, Crittogame, Cicadali, Ginkgoali, Coniferali, Gnetali, Gimnosperme, Dicotiledoni, Monocotiledoni, Angiosperme, Fanerogame, Regno vegetale;
- **Invisita** ($i = 1$): Cianoficee, Tallofite, Batteri, Alghe, Mixomiceti, Funghi, Crittogame, Briofite, Archegoniate, Pteridofite, Regno vegetale, Cicadali, Gimnosperme, Ginkgoali, Coniferali, Gnetali, Fanerogame, Dicotiledoni, Angiosperme, Monocotiledoni;
- **Per livelli**: Regno vegetale, Crittogame, Fanerogame, Taliofite, Archegoniate, Gimnosperme, Angiosperme, Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Briofite, Pteridofite, Cicadali, Ginkgoali, Coniferali, Gnetali, Dicotiledoni, Monocotiledoni.

```
visitaProfondità(TREE t)
  precondition t ≠ nil
(1) esame "anticipato" del nodo radice di t
    TREE u ← t.leftmostChild()
    while u ≠ nil do
        visitaProfondità(u)
        u ← u.rightSibling()
(2) esame "posticipato" del nodo radice di t
```

In pratica, l'ordine di visita anticipato coincide con quello del diritto di successione al trono nell'albero genealogico di una dinastia regnante. Per illustrare la differenza tra gli ordini di visita anticipato e posticipato, c'è un modo pittoresco che non tira in ballo la ricorsione. Si immagini che l'albero sia una sorta di isola fatta a fiordo: i cerchietti rappresentano fari e le linee che collegano coppie di fari rappresentano alte scogliere. Una nave fa la circumnavigazione dell'isola, partendo dalla radice e costeggiando sempre le scogliere. Il capitano segna sul libro di bordo il nome di ciascun faro che viene incontrato nella navigazione, ma solo quando lo incontra per la prima volta. È facile verificare che se la circumnavigazione procede in senso antiorario, allora si ottiene proprio l'ordine di visita anticipa-

to, mentre se procede in senso orario si ottiene l'ordine di visita posticipato inverso (cioè letto da destra verso sinistra, ovvero dall'ultimo nodo visitato al primo).

Lo schema di procedura visitaProfondità() serve per effettuare sia la visita anticipata che quella posticipata. Se l'esame del nodo t viene effettuato solo nella riga (1), ignorando la riga (2), si ottiene la procedura previsita(), mentre se viene effettuato nella riga (2), ignorando la riga (1), si ottiene la procedura postvisita().

La invisita richiede codice leggermente più complesso; quando invisita() viene eseguito sul nodo t, la stessa procedura viene richiamata ricorsivamente sui primi i figli di t, se presenti; viene effettuata la visita simmetrica di t, e quindi invisita() viene richiamata sui restanti figli.

```
invisita(TREE t, integer i)
    precondition: t ≠ nil
    TREE u ← t.leftmostChild()
    integer k ← 0
    while u ≠ nil and k < i do
        k ← k + 1
        invisita(u, i)
        u ← u.rightSibling()
    esame "simmetrico" del nodo t
    while u ≠ nil do
        invisita(u, i)
        u ← u.rightSibling()
```

```
visitaAmpiezza(TREE t)
    precondition: t ≠ nil
    QUEUE Q ← Queue()
    Q.enqueue(t)
    while not Q.isEmpty() do
        TREE u ← Q.dequeue()
        esame "per livelli" del nodo u
        u ← u.leftmostChild()
        while u ≠ nil do
            Q.enqueue(u)
            u ← u.rightSibling()
```

La visita per livelli illustrata nell'algoritmo visitaAmpiezza() utilizza un approccio completamente diverso. Invece di essere basata su una visita ricorsiva (che implicitamente utilizza una pila), è una procedura iterativa basata su una coda. All'inizio, si inserisce la radice nella coda. Quando un nodo viene estratto, vengono inseriti tutti i suoi figli nella coda, in ordine. È facile dimostrare per induzione che tutti i nodi del livello i-esimo vengono esaminati prima dei nodi del livello $(i + 1)$-esimo.

Nelle tre procedure di visita in profondità descritte viene effettuata esattamente una chiamata ricorsiva per ciascun nodo, per un totale di n chiamate, dove n è il numero di nodi dell'albero. Nella visita in ampiezza, ogni nodo è inserito in coda esattamente una volta. Pertanto, la complessità delle quattro procedure di visita è $\Theta(n)$, purché si fornisca una realizzazione in cui le operazioni utilizzate nelle procedure di visita richiedano tempo $O(1)$.

5.3 REALIZZAZIONE CON PUNTATORI PADRE/PRIMO-FIGLIO/FRATELLO

La realizzazione più naturale, detta a puntatori *padre/primo-figlio/fratello*, segue fedelmente la specifica, memorizzando per ogni nodo il valore, il puntatore al padre, il puntatore al primo figlio e il puntatore al fratello successivo. La Fig. 5.6 illustra un albero realizzato in questo modo. I nodi sono memorizzati direttamente in strutture di tipo TREE, che vengono create come nodi isolati contenenti un singolo valore.

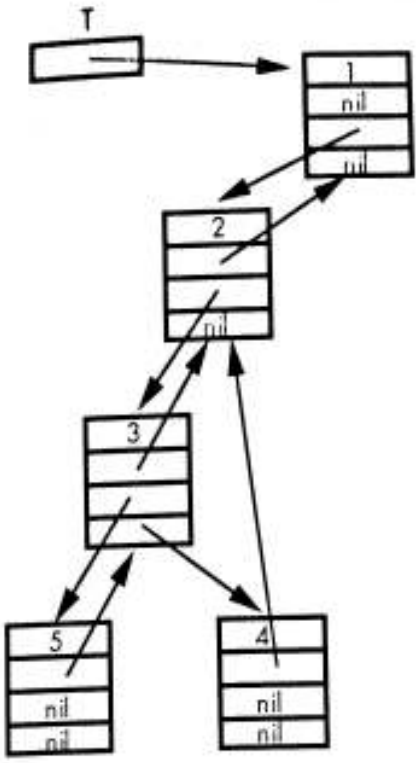


Fig. 5.6 Realizzazione con puntatori padre/primo-figlio/fratello dell'albero T_1 di Fig. 5.4.

Le operazioni per spostarsi lungo l'albero (parent(), leftmostChild() e rightSibling()), così come le operazioni di lettura e scrittura del valore del nodo (read() e write()) vengono realizzate in tempo $O(1)$, perché semplicemente leggono o scrivono il valore corrispondente. Non vengono mostrate per via della loro semplicità.

Più interessanti sono le operazioni che realizzano le funzionalità di modifica dell'albero. Le operazioni di inserimento di alberi esistenti sono simili alle corrispondenti operazioni nelle liste; insertChild() inserisce un elemento in testa alla lista dei figli, mentre insertSibling() lo inserisce dopo un particolare figlio. In entrambi i casi, il costo è $O(1)$, in quanto si tratta semplicemente di "agganciare" l'albero t nella posizione selezionata. Le operazioni di cancellazione, invece, hanno un costo $O(n)$ perché operano ricorsivamente sull'albero, usando la procedura ricorsiva delete().

5.4
REALIZZAZIONE CON VETTORE DEI PADRI

In certi casi, non sono richieste tutte le operazioni precedentemente introdotte. Ad esempio, in alcuni problemi sui grafi l'output è rappresentato da un albero, che viene costruito e navigato "al contrario", seguendo i puntatori ai padri. In questo caso, i riferimenti a figli e fratelli sono inutili, e l'albero può essere rappresentato tramite un *vettore dei padri*, con un consumo minimo di memoria.

Si supponga che i nodi dell'albero T siano numerati da 1 ad n. Il vettore dei padri p contiene, per ogni nodo i, $1 \leq i \leq n$, l'indice del padre: $p[i] = j$, se j è il padre di i; $p[i] = 0$, se i è la radice.

```
TREE
    NODE parent                                   % Puntatore al padre
    NODE child                                    % Puntatore al primo figlio
    NODE sibling                                  % Puntatore al successivo fratello
    ITEM value                                    % Valore del nodo

    Tree(ITEM v)                                  % Crea un nuovo nodo
    |   TREE t ← new TREE
    |   t.value ← v
    |   t.parent ← t.child ← t.sibling ← nil
    |   return t

    insertChild(TREE t)
    |   t.parent ← this
    |   t.sibling ← child                         % Inserisce t prima dell'attuale primo figlio
    |   child ← t

    insertSibling(TREE t)
    |   t.parent ← parent
    |   t.sibling ← sibling                       % Inserisce t prima dell'attuale fratello
    |   sibling ← t

    deleteChild()
    |   NODE newChild ← child.rightSibling()
    |   delete(child)
    |   child ← newChild

    deleteSibling()
    |   NODE newBrother ← sibling.rightSibling()
    |   delete(sibling)
    |   sibling ← newBrother

    delete(TREE t)
    |   NODE u ← t.leftmostChild()
    |   while u ≠ nil do
    |   |   TREE next ← u.rightSibling()
    |   |   delete(u)
    |   |   u ← next
    |   delete t
```

ESEMPIO – [Vettore dei padri]
Il contenuto del vettore dei padri per l'albero T_1 della Fig. 5.4 è: $p[1] = 0$, $p[2] = 1$, $p[3] = 2$, $p[4] = 2$, $p[5] = 3$.

In questa realizzazione, l'albero non è ordinato: non è infatti chiara la relazione tra fratelli. Normalmente, questo non è un problema; altrimenti, è possibile assumere una relazione implicita, per esempio imponendo che un figlio abbia sempre un numero maggiore del padre e che nodi fratelli siano numerati in modo crescente da sinistra verso destra (ciò implica che la radice è nella prima posizione del vettore).

Ovviamente la specifica deve essere adattata, tenendo conto del fatto che la struttura albero e il nodo non coincidono più, eliminando le operazioni inutili e/o ridondanti e aggiungendo l'operazione setParent(), che "aggiunge" un sottoalbero ad un nodo esistente. Mostriamo qui di seguito una semplice realizzazione, tralasciando per semplicità di memorizzare i valori dei nodi.

```
TREE
integer[] p                                   % Vettore dei padri
% Costruisce una "foresta" con n nodi isolati
Tree(integer n)
|   p ← new integer[1 ... n]
|   for i ← 1 to n do p[i] ← 0
% Restituisce il padre del nodo i; restituisce 0 se i è radice
integer parent(integer i)
|   return p[i]
% Rende il nodo i un figlio del nodo j
setParent(integer i, integer j)
|   p[i] ← j
```

5.5 ALBERI BINARI

Un albero binario è un particolare albero ordinato in cui ogni nodo ha al più due figli e si fa distinzione tra il figlio *sinistro* ed il figlio *destro* di un nodo. Questa proprietà è assai sottile, poiché impone che due alberi T e U aventi gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, siano distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T e come figlio destro del medesimo nodo in U.

ESEMPIO – [Alberi binari]
La Fig. 5.7 mostra tre alberi $T_1, \ldots, T_3$. Visti come alberi ordinati, essi sono tutti identici tra loro. T_1 non è binario, perché non è chiara la convenzione per distinguere i figli sinistri dai destri. Considerando i figli disegnati a sinistra (destra) come figli sinistri (destri), allora T_2 e T_3 sono binari; inoltre essi sono tra loro distinti, poiché non hanno gli stessi figli destri e sinistri.

L'importanza degli alberi binari risiede sia sulla semplicità della loro struttura che sulle innumerevoli applicazioni in cui possono essere impiegati.

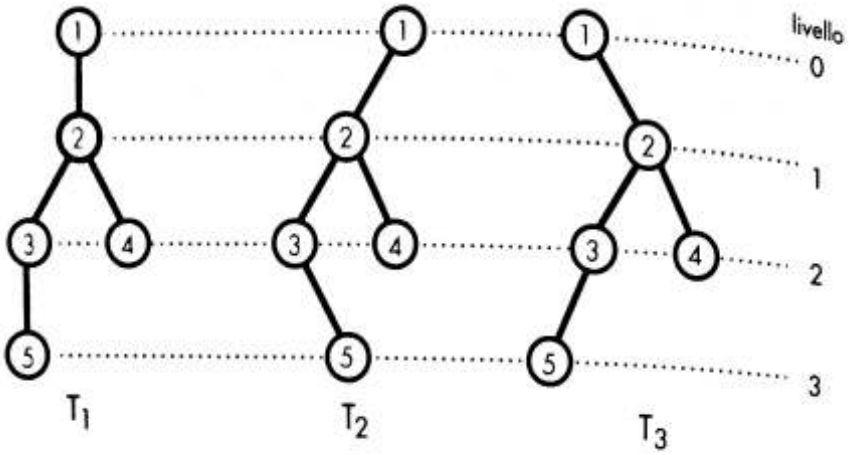


Fig. 5.7 Alberi ordinati e alberi binari.

5.2 ESEMPIO – [Albero delle scelte]

Le possibili sequenze di confronti (a due alternative) effettuate da un algoritmo per risolvere un problema possono essere visualizzate con un albero binario, detto albero delle scelte. In tale albero, i nodi con figli rappresentano confronti tra dati del problema, le coppie padri-figli i risultati dei confronti, e le foglie soluzioni possibili del problema. Un percorso radice-foglia rappresenta la sequenza di confronti effettuati per individuare una soluzione e il livello massimo di una foglia dà il numero di confronti effettuati nel caso pessimo. Un albero delle scelte che minimizzi il livello massimo delle foglie fornisce con tale valore una limitazione inferiore al numero di decisioni che ogni algoritmo che risolva il problema deve effettuare nel caso peggiore.

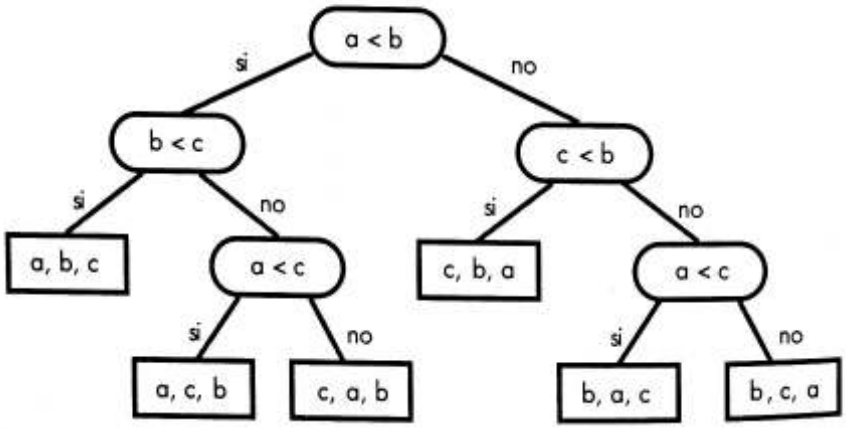


Fig. 5.8 Albero delle scelte per l'ordinamento di tre numeri.

La Fig. 5.8 illustra un albero delle scelte per il problema di ordinare tre numeri a, b, c. Poiché un qualsiasi algoritmo di ordinamento di n numeri deve fare una sequenza di scelte che permetta di individuare la soluzione corretta tra le $n!$ permutazioni possibili dei dati di ingresso, l'albero delle scelte per un qualsiasi algoritmo di ordinamento ha almeno $n!$ foglie. Poiché ad ogni livello dell'albero il numero di nodi al più raddoppia, la lunghezza del percorso radice-foglia più lungo non può essere inferiore a $\log_2 n!$ Essendo $n! = n \cdot (n-1) \cdot 3 \cdot 2 \cdot 1 \geq n(n-1) \cdot (n/2) \geq (n/2)^{n/2}$, si ha $\log_2 n! \geq (n/2) \log_2(n/2)$. Pertanto una *limitazione inferiore* alla complessità del problema dell'ordinamento per algoritmi basati su confronti è $\Omega(n \log n)$.

Una possibile specifica per gli alberi binari include le operazioni per leggere e scrivere i figli destro e sinistro, che sostituiscono le operazioni per leggere e scrivere il primo figlio e i fratelli successivi.

```
TREE
  % Restituisce il figlio sinistro (destro) di questo nodo; restituisce nil se assente
  TREE left()
  TREE right()

  % Inserisce il sottoalbero t come figlio sinistro (destro) di questo nodo
  insertLeft(TREE t)
    precondition: t.parent = nil
  insertRight(TREE t)
    precondition: t.parent = nil

  % Distrugge il sottoalbero sinistro (destro) di questo nodo
  deleteLeft()
  deleteRight()
```

Una realizzazione ragionevole fa uso di puntatori, come nel caso degli alberi ordinati, i cui campi *child* e *sibling* sono sostituiti da *left* e *right*. La complessità di tutte le operazioni è $O(1)$, con esclusione di quella di cancellazione, che agendo ricorsivamente può avere complessità $O(n)$. Per motivi di spazio, le operazioni parent(), left(), right(), read() e write() non sono mostrate; semplicemente, restituiscono il valore della variabile corrispondente.

```
TREE

Tree(ITEM v)
    TREE t ← new TREE
    t.parent ← nil
    t.left ← t.right ← nil
    t.value ← v
    return t

insertLeft(TREE T)
    T.parent ← this
    left ← T

insertRight(TREE T)
    T.parent ← this
    right ← T

deleteLeft()
    if left ≠ nil then
        left.deleteLeft()
        left.deleteRight()
        delete left
        left ← nil

deleteRight()
    if right ≠ nil then
        right.deleteLeft()
        right.deleteRight()
        delete right
        right ← nil
```

La sostanziale equivalenza di queste due realizzazioni non deve sorprendere, poiché nel proporre quella per gli alberi ordinati si è implicitamente trasformato l'albero ordinato in uno binario! Come illustrato nella Fig. 5.9, è sempre possibile rappresentare un albero or-

dinato T con un albero binario B avente gli stessi nodi e la stessa radice, in cui ogni nodo di B ha come figlio sinistro il primo figlio che ha in T e come figlio destro il fratello successivo che ha in T. È facile verificare che le sequenze dei nodi esaminati su T e su B coincidono se T e B sono entrambi visitati in ordine anticipato, oppure se T è visitato in ordine posticipato e B è visitato in ordine simmetrico.

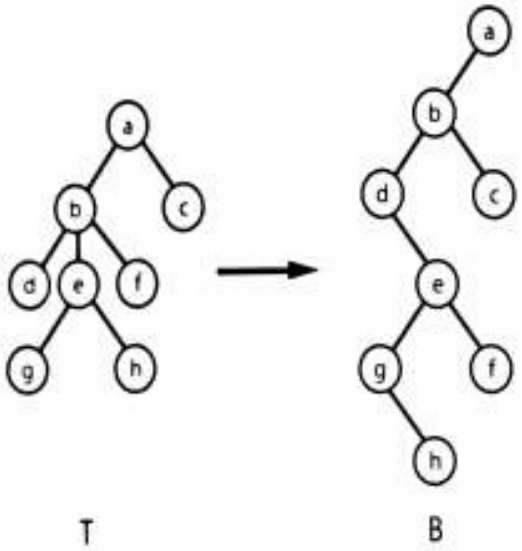


Fig. 5.9 Rappresentazione di un albero ordinato T con un albero binario B.

Utilizzando le operazioni introdotte, la visita di un albero binario viene effettuata molto semplicemente da visitaProfondità(). La *previsita* si ottiene esaminando il nodo t soltanto nella riga (1) e ignorando le righe (2) e (3). Analogamente, la *invisita* o la *postvisita* si hanno esaminando il nodo t solo nella riga (2) o in (3) rispettivamente.

```
visitaProfondità(TREE t)

    if t ≠ nil then
(1)     esame "anticipato" di t
        visitaProfondità(t.left())
(2)     esame "simmetrico" di t
        visitaProfondità(t.right())
(3)     esame "posticipato" di t
```

5.6 ALTRE REALIZZAZIONI

Se gli alberi sono dotati di caratteristiche particolari, è possibile progettare realizzazioni alternative per essi. Per esempio, un *albero d-ario completo* è un albero in cui ogni nodo interno ha esattamente d figli. Tale albero può essere memorizzato in un vettore, inserendo la radice nell'indice 1, mentre i figli di un nodo i sono memorizzati negli indici $d \cdot (i-1) + 2, d \cdot (i-1) + 3, \ldots, d \cdot i + 1$. Vedremo un esempio di questo metodo di memorizzazione quando introdurremo gli alberi binari heap [capitolo 10]. Alternativamente, se il numero massimo di figli di un nodo è limitato superiormente da un intero d, è possibile

sostituire la lista primo figlio/fratello con un vettore di puntatori di dimensione *d*; gli eventuali figli mancanti sono sostituiti con puntatori **nil**.

5.7 REALITY CHECK

L'elenco delle applicazioni degli alberi è così vasto che è difficile sceglierne una. La struttura base di un file system, ad esempio, è un albero; le directory sono nodi interni, mentre i file sono foglie. Tuttavia, poiché alcuni file system permettono di elencare lo stesso file in più directory (tramite hard link o symbolic link), sarebbe più corretto descrivere tali file system come grafi diretti aciclici [capitolo 9.6].

ESERCIZI

[Altezza] L'altezza di un albero ordinato è il massimo livello delle sue foglie. Si fornisca una funzione che calcoli in tempo ottimo l'altezza di un albero ordinato T di n nodi.

[Cancella le foglie con somma k] Dato un albero ordinato i cui nodi contengono valori interi, se ne vogliono cancellare tutte le foglie per le quali il percorso radice-foglia ha somma complessiva dei valori uguale a k. Fornire una procedura di complessità ottima.

[Larghezza] La larghezza di un albero ordinato è il numero massimo di nodi che stanno tutti al medesimo livello. Si fornisca una funzione che calcoli in tempo ottimo la larghezza di un albero ordinato T di n nodi.

[Visite] Gli ordini di visita di un albero binario di 9 nodi sono i seguenti: A, E, B, F, G, C, D, I, H (anticipato), B, G, C, F, E, H, I, D, A (posticipato) e B, E, G, F, C, A, D, H, I (simmetrico). Si ricostruisca l'albero binario.

[Numero nodi per sottoalbero] Dato un albero binario t non vuoto, si vuole memorizzare, in ciascun nodo u, il numero di nodi che si trovano nel sottoalbero radicato in u. Si fornisca un algoritmo lineare nel numero di nodi.

[Cancella foglie] Dato un albero binario i cui nodi contengono interi, si vuole cancellare ogni foglia che sia un figlio sinistro e contenga lo stesso intero del padre. Si scriva una procedura ricorsiva di complessità ottima.

[Inserisci foglie] Dato un albero binario, si vuole aggiungere ad ogni foglia un figlio sinistro contenente il valore 0. Si scriva una procedura ricorsiva di complessità ottima.

[Raggruppa le foglie con 0 e 1] Dato un albero binario le cui foglie contengono 0 od 1 e i cui nodi interni contengono solo 0, si vuole cambiare il contenuto delle foglie in modo che, visitandole da sinistra verso destra, si incontrino prima tutti gli 0 e poi tutti gli 1. Si scriva una procedura ricorsiva di complessità ottima.

Dato un albero binario, i cui nodi contengono elementi interi, si scriva una procedura di complessità ottima per ottenere l'albero inverso, ovvero un albero in cui il figlio destro (con relativo sottoalbero) è scambiato con il figlio sinistro (con relativo sottoalbero).

5.40 Dato un albero binario i cui nodi contengono interi, si vuole aggiungere ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia. Si scriva una procedura ricorsiva di complessità ottima.

5.41 Si dimostri per induzione che in un albero binario il numero delle foglie è uguale al numero dei nodi con due figli più uno.

5.42 [Albero binario completo] Un albero binario completo di altezza k è un albero binario in cui tutti i nodi, tranne le foglie, hanno esattamente due figli, e tutte le foglie si trovano al livello k. Si dimostri per induzione che, in un albero binario completo di altezza k, il numero dei nodi è $2^{k+1} - 1$ ed il numero delle foglie è 2^k.

5.43 [Visite iterative] Utilizzando le pile, si scrivano tre procedure iterative di complessità ottima per effettuare, rispettivamente, le visite anticipata, differita e simmetrica di un albero binario.

Poiché la limitazione inferiore è $\Omega(n)$ per la dimensione dei dati, la funzione deve richiedere tempo $O(n)$. Si utilizza lo schema della visita differita nella quale si calcola l'altezza di ogni sottoalbero di T. L'altezza di un sottoalbero radicato in un generico nodo u è zero, se u è una foglia, mentre si ottiene sommando uno all'altezza massima dei sottoalberi radicati nei figli di u, altrimenti. Il tempo richiesto è $\Theta(n)$ e quindi ottimo. La chiamata della funzione è depth(T).

```
depth(TREE t)
  integer max ← 0
  TREE u ← t.leftmostChild()
  while u ≠ nil do
      integer t ← depth(u) + 1
      if t > max then max ← t
      u ← u.rightSibling()
  return max
```

Una limitazione inferiore alla complessità del problema è $\Omega(n)$ poiché è necessario esaminare tutte le foglie dell'albero. L'algoritmo risolutivo si basa su una visita anticipata. Durante la discesa calcola per ogni nodo u la somma dei valori contenuti nei nodi incontrati a partire dalla radice fino ad t. Quando giunge ad una foglia, la cancella se tale somma è uguale a k. Si assume una realizzazione con puntatori padre/primo figlio/fratello. Per cancellare la foglia usa le operazioni deleteChild() e deleteSibling(), che richiedeno tempo $O(1)$ per cancellare un sottoalbero di un solo nodo. La complessità della procedura è $\Theta(n)$ e la chiamata iniziale è leaftk(T, k, 0). Se la chiamata iniziale restituisce **true**, l'intero albero (costituito da una sola foglia di valore k) deve essere cancellato dal chiamante.

```
boolean leafk(TREE t, integer k, integer somma)
  somma ← somma + t.read()
  if somma = k and t.leftmostChild() = nil then
      return true                          % Foglia con valore k; deve essere cancellata
  else
      while t.leftmostChild() ≠ nil and leafk(t.leftmostChild(), k, somma) do
          t.deleteChild()
      TREE u ← t.leftmostChild()                          % Nodo da cui cancellare
      TREE v ← iif(u ≠ nil, u.rightSibling(), nil)        % Potenziale nodo da cancellare
      while v ≠ nil do
          if leafk(v, k, somma) then
              u.deleteSibling()            % u rimane fisso e un fratello viene eliminato
          else
              u ← v                                       % u avanza
          v ← u.rightSibling()                            % Prossimo fratello
      return false
```

Si procede con una postvisita, memorizzando in ciascun nodo u il valore 1, se u è una foglia, o 1 più la somma dei valori contenuti nel figlio sinistro e nel figlio destro, altrimenti.

```
integer count(TREE t)
  if t = nil then return 0
  else
      integer somma ← 1 + count(t.left()) + count(t.right())
      t.write(somma)
      return somma
```

Tutte le foglie e quindi tutti i nodi dell'albero devono essere visitati nel caso pessimo. L'algoritmo proposto è basato su una previsita e ha complessità lineare nel numero di nodi e quindi ottima. Quando un nodo t è visitato, si verifica se ha un figlio sinistro ($u \neq$ **nil**), se è foglia (u.left() = u.right() = **nil**) e se ha lo stesso valore del padre (t.read() = u.read()), nel qual caso lo si cancella.

```
deleteLeaf(TREE t)
  if t ≠ nil then
      TREE u ← t.left()
      if u ≠ nil and u.left() = u.right() = nil and t.read() = u.read() then
      | t.deleteLeft()
      deleteLeaf(t.left())
      deleteLeaf(t.right())
```

Valgono le stesse considerazioni dell'esercizio precedente riguardo alla limitazione inferiore e all'uso dei puntatori, ma stavolta l'algoritmo proposto è basato su una postvisita.

```
inserisciFoglia(TREE t)
  if t ≠ nil then
      inserisciFoglia(t.left())
      inserisciFoglia(t.right())
      if t.left() = t.right() = nil then
      | TREE new ← Tree(0)
      | t.insertLeft(new)
```

SOLUZIONI

5.41 Sia $T(n)$ un generico albero binario con $n \geq 1$ nodi e si indichi con $n_2(T(n))$ il numero di nodi con due figli di $T(n)$ e con $F(T(n))$ il numero di foglie di $T(n)$. Si vuole dimostrare per induzione che $F(T(n)) = n_2(T(n)) + 1$.

La base dell'induzione è vera per $n = 1$, perché in tal caso $F(T(1)) = 1$ e $n_2(T(1)) = 0$. Si assuma la proprietà vera per $T(n)$. $T(n + 1)$ può essere ottenuto da $T(n)$ aggiungendo un figlio o ad una foglia, o ad un nodo con un figlio. Nel primo caso, sia il numero delle foglie sia quello dei nodi con due figli non cambiano e quindi: $F(T(n + 1)) = F(T(n)) = n_2(T(n)) + 1 = n_2(T(n + 1)) + 1$. Nel secondo caso, sia il numero dei nodi con due figli sia quello delle foglie aumenta di uno e quindi: $F(T(n + 1)) = F(T(n)) + 1 = n_2(T(n)) + 1 + 1 = n_2(T(n + 1)) + 1$.

ALBERI BILANCIATI DI RICERCA

Capitolo 6

Gli alberi sono una delle strutture di dati più efficienti per realizzare dizionari, in modo particolare quando si vuole mantenere la possibilità di scorrere le chiavi presenti nella struttura secondo un qualche ordinamento. Affinché questo possa accadere, i nodi dell'albero devono essere dotati di informazioni di chiave e di valore, e la struttura dell'albero deve permettere di effettuare le ricerche. Vedremo quindi le caratteristiche minime necessarie per la ricerca, e studieremo poi come rendere efficienti ricerche, inserzioni e cancellazioni.

6.1
ALBERI BINARI DI RICERCA

Le associazioni chiave-valore di un dizionario D possono essere contenute nei nodi di un albero binario B, detto *albero binario di ricerca* o ABR, che verifica le seguenti proprietà:

(1) per ogni nodo u, tutte le chiavi contenute nel sottoalbero radicato nel figlio sinistro di u sono minori della chiave contenuta in u;

(2) per ogni nodo u, tutte le chiavi contenute nel sottoalbero radicato nel figlio destro di u sono maggiori della chiave contenuta in u.

6.1 ESEMPIO – [Albero binario di ricerca]
Un ABR è mostrato nella Fig. 6.1. Si noti che con una visita simmetrica dell'albero si esaminano le chiavi in ordine crescente.

Le proprietà (1) e (2) hanno lo scopo di agevolare la ricerca di un elemento. È sufficiente confrontare l'elemento cercato con quello contenuto nella radice e, se sono diversi, riapplicare il procedimento al sottoalbero radicato nel figlio sinistro o destro, secondo l'esito del confronto. Questo procedimento è molto simile a quello effettuato nella ricerca binaria.

Come il nome suggerisce, tale ricerca genera infatti partizioni successive degli elementi secondo uno schema coincidente con un albero binario.

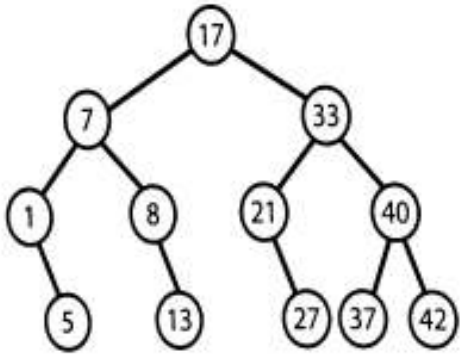


Fig. 6.1 Un albero binario di ricerca.

ESEMPIO – [Ricerca binaria]

L'ABR di Fig. 6.1 corrisponde ad una ricerca binaria effettuata sui dati dell'Esempio 1.2.

Assumendo che l'albero binario sia realizzato con puntatori [capitolo 5], le procedure lookupNode(), insertNode() e removeNode() cercano, inseriscono e rimuovono nodi da alberi binari di ricerca, e possono essere utilizzate per realizzare le relative operazioni degli insiemi e dei dizionari. Per illustrare un approccio complementare rispetto a quello utilizzato introducendo gli alberi binari, proponiamo una versione iterativa delle procedure suddette.

La procedura lookupNode() cerca il nodo contenente la chiave x nell'albero binario di ricerca radicato in T; restituisce **nil**, nel caso che la chiave non sia presente, il puntatore al nodo contenente x, altrimenti. La complessità della procedura è $O(h)$, dove h è l'altezza dell'albero.

```
TREE lookupNode(TREE T, ITEM x)
  while T ≠ nil and T.key ≠ x do
  |   T ← iif(x < T.key, T.left, T.right)
  return T
```

Per inserire associazioni chiavi-elemento, la procedura insertNode() effettua innanzitutto una ricerca, in modo simile a quanto visto per lookupNode(), per localizzare il nodo u contenente l'elemento (in caso di modifica dell'associazione corrente) o il nodo p che diventerà il padre del nuovo nodo contenente l'elemento (in caso di inserzione). Nel caso il nodo sia trovato, viene aggiornato solo il campo *value*; altrimenti viene creato un nuovo nodo, e questo viene agganciato come figlio sinistro o destro di p, a seconda del valore della chiave, dalla procedura di appoggio link(). insertNode() restituisce un puntatore all'albero, che corrisponde all'albero esistente, se questo conteneva già almeno un nodo, o corrisponde al nodo creato, se questo è il primo a popolare l'albero (ovvero se p = **nil**). Come prima la complessità è $O(h)$, ma si noti che l'altezza può crescere di uno.

```
link(TREE p, TREE u, ITEM x)
if u ≠ nil then u.parent ← p
if p ≠ nil then
    if x < p.key then p.left ← u
                 else p.right ← u
```

La procedura link(), utilizzata anche dalla procedura removeNode(), gestisce le due azioni necessarie per collegare un nodo padre p ad un nodo figlio u: modificare il puntatore al padre di u, e inserire u come figlio sinistro o destro di p, a seconda del valore del parametro x che viene confrontato con il contenuto del padre. Come casi particolari, p può essere **nil**, che significa che u è la nuova radice; oppure u può essere **nil**, che significa che un nodo è stato cancellato e **nil** sostituisce il figlio corrispondente nel padre. link() ha costo $O(1)$.

```
TREE insertNode(TREE T, ITEM x, ITEM v)
TREE p ← nil                                          % Padre
TREE u ← T
while u ≠ nil and u.key ≠ x do                        % Cerca posizione inserimento
    p ← u
    u ← iif(x < u.key, u.left, u.right)
if u ≠ nil and u.key = x then
    u.value ← v                                       % Chiave già presente
else
    TREE n ← Tree(x, v)                               % Crea un nodo contenente coppia chiave-valore
    link(p, n, x)
    if p = nil then return n                          % Primo nodo ad essere inserito
return T                                              % Restituisce albero non modificato
```

Per realizzare removeNode() si procede in modo analogo all'inserzione, con una complicazione in più data dal fatto che l'elemento x da cancellare si trova in un nodo u che non necessariamente è una foglia. Possono sorgere tre casi, illustrati nella Fig. 6.2.

(1) Se u non ha figli, allora u è una foglia che viene semplicemente rimossa. C'è da gestire tuttavia il caso particolare in cui u sia l'unico nodo dell'albero.

(2) Se u ha un solo figlio f, allora u è rimosso rendendo f figlio del padre di u.

(3) Se u ha due figli, ci si può ridurre al caso (1) o (2) sostituendo l'elemento da cancellare x con il più piccolo elemento y tale che $y > x$, che si trova nel nodo s raggiungibile scendendo sempre a sinistra nel sottoalbero radicato nel figlio destro di u. L'elemento y prende il posto di x nel nodo u mentre viene rimosso il nodo s che, per definizione, non può avere un figlio sinistro.

La procedura removeNode() restituisce il puntatore alla radice dell'albero, che può cambiare se il nodo cancellato è proprio la radice. La procedura inizia cercando tramite

lookupNode() il nodo da cancellare u; la radice viene mantenuta nel puntatore T. Se u non è stato trovato, la procedura termina restituendo la radice inalterata.

```
TREE removeNode(TREE T, ITEM x)
  TREE u ← lookupNode(T, x)
  if u ≠ nil then
      if u.left ≠ nil and u.right ≠ nil then                    % Caso (3)
          TREE s ← u.right
          while s.left ≠ nil do s ← s.left
          u.key ← s.key
          u.value ← s.value
          u ← s
          x ← s.key
      TREE t
      if u.left ≠ nil and u.right = nil then                    % Caso (2) - Solo figlio sx
          t ← u.left
      else                                                      % Caso (2) - Solo figlio dx / Caso (1)
          t ← u.right
      link(u.parent, t, x)
      if u.parent = nil then T ← t
      delete u
  return T
```

A questo punto, la procedura è organizzata nel modo seguente:

- il caso (3) appare per primo, in quanto si riduce al caso (1) o (2); innanzitutto viene identificato il nodo s e la sua associazione chiave-valore viene ricopiata nel nodo u; a questo punto, il puntatore u assume il valore di s e si passa a gestire la cancellazione di s tramite i casi (1) e (2).
- il caso (2) con solo un figlio sinistro è semplice; il figlio sinistro viene collegato al padre di u; se il padre è assente, u era radice e il suo figlio sinistro diventa la nuova radice.
- il caso (2) con solo un figlio destro è simmetrico; il figlio destro viene collegato al padre di u; se il padre è assente, u era radice e il suo figlio destro diventa la nuova radice. Si noti che in pratica questo codice gestisce anche il caso (1), dove però il figlio destro di u è **nil**. Se entrambi, il padre e il figlio u, sono **nil**, l'albero è ora vuoto e la nuova radice diventa per l'appunto **nil**.

```
TREE min(TREE T)
  while T.left ≠ nil do
      T ← T.left
  return T
```

```
TREE max(TREE T)
  while T.right ≠ nil do
      T ← T.right
  return T
```

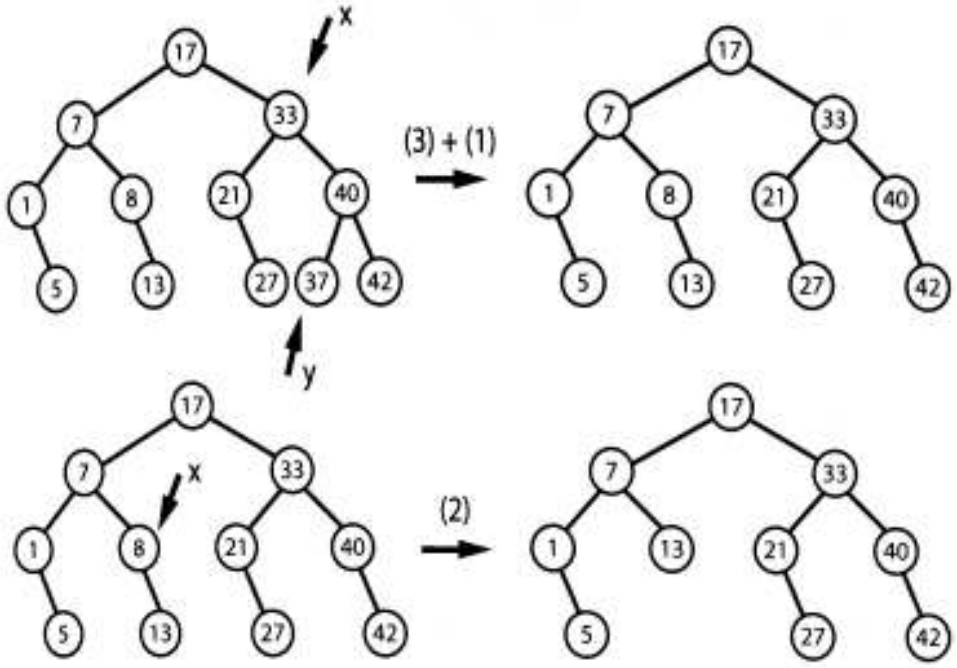


Fig. 6.2 I tre casi di cancellazione di un elemento *x* da un albero binario di ricerca.

La memoria associata al nodo u viene rimossa e la radice (potenzialmente modificata) viene restituita al chiamante. Richiedendo al massimo una discesa radice-foglia, anche questa procedura ha costo $O(h)$.

```
TREE successorNode(TREE t)
  if t = nil then
  |   return t
  if t.right ≠ nil then
  |   return min(t.right)
  TREE p ← t.parent
  while p ≠ nil and t = p.right do
  |   t ← p
  |   p ← p.parent
  return p
```

```
TREE predecessorNode(TREE t)
  if t = nil then
  |   return t
  if t.left ≠ nil then
  |   return max(t.left)
  TREE p ← t.parent
  while p ≠ nil and t = p.left do
  |   t ← p
  |   p ← p.parent
  return p
```

A differenza delle tabelle hash che verranno presentate nel prossimo capitolo, gli alberi di ricerca mantengono l'ordinamento fra i valori. Un operatore min() che restituisca il nodo contenente il minimo può essere realizzato facilmente in tempo $O(h)$. Basta infatti effettuare una ricerca in cui si segue sempre il figlio sinistro, fino a trovare un nodo che non ha figlio sinistro. L'operatore max() è simmetrico, basta seguire sempre il figlio destro.

Oltre a questi operatori, dato un nodo t (non necessariamente la radice), può essere interessante ottenere il successore/predecessore di t nell'albero, ovvero il nodo la cui chiave segue/precede immediatamente $t.key$ nell'ordinamento. Per trovare il successore di t, sono dati due casi:

- se t ha un figlio destro, il successore è il minimo u del sottoalbero destro;

- altrimenti, il successore è il primo antenato u di t per cui t sta nel sottoalbero sinistro di u.

Il codice di successorNode() segue semplicemente i due casi; il codice di predecessorNode() è simmetrico.

Con l'operatore min() e la procedura successorNode() è possibile scorrere tutte le chiavi e/o i valori presenti nell'albero T secondo l'ordinamento utilizzando codice come questo:

```
TREE u ← min(T)
while u ≠ nil do
    {Elabora u}
    u ← successorNode(u)
```

Questo codice ha complessità ottima $O(n)$, in quanto tutti i nodi vengono visitati al massimo due volte: una volta scendendo lungo la direzione radice-foglia, alla ricerca di un minimo; e una volta salendo lungo la direzione foglia-radice, alla ricerca di un antenato.

Usando un ABR, la complessità delle singole operazioni lookupNode(), insertNode(), removeNode() è sempre $O(h)$. Purtroppo, h è $O(n)$ nel caso pessimo, perché l'albero può "allungarsi" inserendo e cancellando nodi. Per fortuna, è possibile modificare la struttura introducendo opportuni metodi di "bilanciamento" che permettono di contenere h entro $O(\log n)$. Esistono diversi approcci di questo tipo, che qui semplicemente elenchiamo:

- Alberi AVL (Adelson-Velskii e Landis, 1962)
- B-alberi (Bayer, McCreight, 1972)
- Alberi Red-Black (Bayer, 1972)
- Alberi 2-3 (Hopcroft, 1983)
- B-alberi binari (Andersson, 1993)

Tutte queste strutture sono caratterizzate dalla stessa limitazione superiore $O(\log n)$ per l'altezza dell'albero. Nel seguito presenteremo solo gli alberi Red-Black, che sono fra quelli più utilizzati in pratica.

6.2 ALBERI RED-BLACK

Un albero è *perfettamente bilanciato* se tutte le foglie si trovano allo stesso livello. Fissato un numero n di nodi e un numero massimo d di figli per nodo (ad esempio $d = 2$ per gli alberi binari), l'albero perfettamente bilanciato è quello con altezza minima tra tutti gli alberi con le stesse caratteristiche. Esempi di alberi perfettamente bilanciati sono i B-alberi e gli Alberi 2-3. Un albero è invece *k-bilanciato* se, per ogni nodo u, la massima differenza di altezza fra tutti i sottoalberi di u è pari a k. Gli alberi AVL sono un esempio di alberi 1-bilanciati.

Tutte queste strutture di dati cercano di minimizzare l'altezza dell'albero, al fine di limitare il numero di livelli che devono essere esplorati per trovare una chiave. Ma la rigidità del-

le loro strutture rende più costose le operazioni di inserimento e cancellazione, che richiedono complesse ristrutturazioni dell'albero per evitare sbilanciamenti. Gli alberi Red-Black in questa situazione rilassano i vincoli di bilanciamento, riducendo le operazioni necessarie di ristrutturazione, senza tuttavia rinunciare ad un'altezza limitata superiormente da $O(\log n)$.

Un albero Red-Black è un ABR in cui ogni nodo ha un attributo aggiuntivo *color*, che può essere rosso (RED) o nero (BLACK). Oltre ai requisiti soliti degli alberi binari di ricerca, per gli alberi Red-Black devono valere i seguenti vincoli:

(v_1) la radice è nera;

(v_2) tutte le foglie sono nere;

(v_3) entrambi i figli di un nodo rosso sono neri;

(v_4) tutti i cammini semplici da ogni nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri.

Le foglie di un albero Red-Black non contengono alcun dato. In realtà, non è necessario memorizzare esplicitamente queste foglie nella memoria del computer – un puntatore **nil** scritto in *left* (*right*) sarebbe sufficiente a codificare che il figlio sinistro (destro) è una foglia. Tuttavia, gli algoritmi e le dimostrazioni risultano semplificate se le foglie sono memorizzate esplicitamente. Per risparmiare memoria, a volte viene utilizzato un nodo sentinella che assume il ruolo di tutte le foglie; tutti i puntatori a foglie puntano quindi allo stesso nodo. La Fig. 6.3 mostra un albero Red-Black con 10 nodi interni e 11 foglie[1].

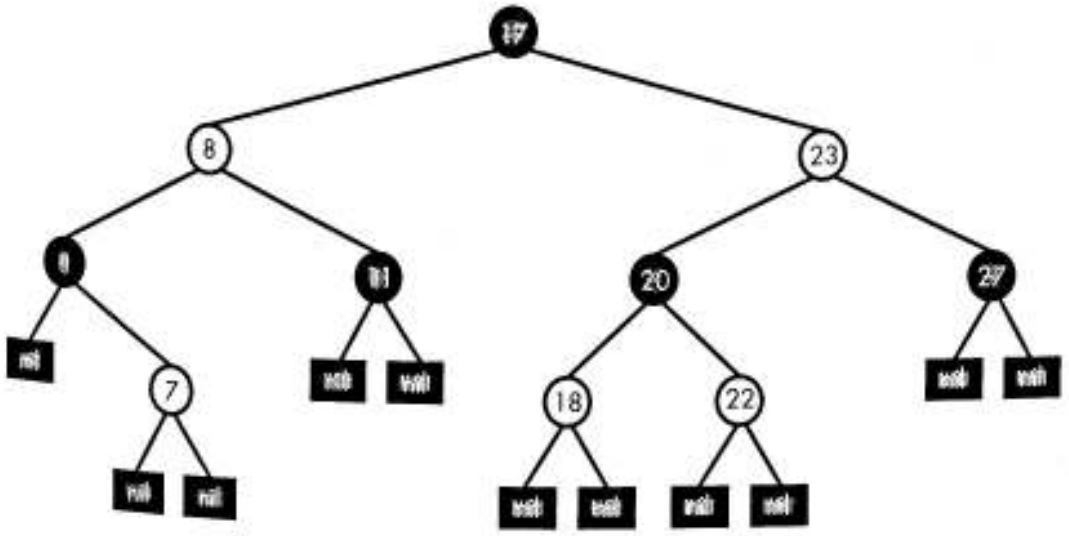


Fig. 6.3 Un albero Red-Black con 10 nodi interni e 11 foglie.

Vediamo ora come i vincoli Red-Black limitino il grado di sbilanciamento dell'albero e garantiscano un'altezza limitata superiormente. Il numero di nodi *neri* lungo ogni percorso da un nodo u (escluso) ad una qualsiasi foglia è detto *altezza nera* di u ed è indicato con $b(u)$. Questo valore è ben definito in quanto tutti i percorsi di questo tipo hanno lo stesso numero di nodi neri per il vincolo (v_4). L'altezza nera della radice è detta altezza nera dell'albero.

[1] Nelle figure del capitolo il colore rosso è indicato in grigio chiaro.

TEOREMA 1

Il sottoalbero con radice u contiene almeno $2^{b(u)} - 1$ nodi interni.

DIMOSTRAZIONE

Si procede per induzione sull'altezza h dell'albero (non sull'altezza nera).

Nel *caso base*, $h = 0$. Allora u deve essere una foglia **nil** e il sottoalbero con radice in u contiene esattamente almeno $2^{b(u)} - 1 = 2^0 - 1 = 0$ nodi interni.

Nel *passo induttivo*, $h > 1$. Allora u è un nodo interno con due figli. Ogni figlio v ha un'altezza nera $b(v)$ pari a $b(u)$ o a $b(u) - 1$, a seconda del suo colore (se rosso: $b(u)$, se nero: $b(u) - 1$). Usando l'ipotesi induttiva, possiamo dire che ogni figlio ha almeno $2^{b(u)-1} - 1$ nodi interni. Quindi, il sottoalbero con radice in u ha almeno:

$$2^{b(u)-1} - 1 + 2^{b(u)-1} - 1 + 1 = 2^{b(u)} - 1$$

nodi interni. Questo completa la dimostrazione.

TEOREMA 2

La lunghezza del più lungo cammino semplice radice-foglia non supera il doppio della lunghezza del più corto cammino semplice radice-foglia. Più formalmente, detto $l(u)$ il livello di un nodo u, vale la seguente proprietà:

$$max\ \{l(u) : u.left = u.right = \mathbf{nil}\} \leq 2 \cdot min\{l(v) : v.left = v.right = \mathbf{nil}\}$$

DIMOSTRAZIONE

Si noti innanzitutto che nessun percorso può avere due nodi rossi in fila, a causa del vincolo (v_3). Tutti i cammini semplici radice-foglia hanno lo stesso numero di nodi neri per il vincolo (v_4); il più corto di essi è composto da soli nodi neri, mentre il più lungo alterna nodi rossi e neri, e quindi è lungo al massimo il doppio.

L'albero risultante non è quindi perfettamente bilanciato, tuttavia l'altezza dell'albero resta comunque limitata:

TEOREMA 3

L'altezza di un albero rosso-nero con n nodi interni è al più $2 \log(n + 1)$.

DIMOSTRAZIONE

Per il Teorema 1, l'albero ha almeno $n \geq 2^{b(r)} - 1$, dove r è la radice. L'altezza dell'albero è al più il doppio dell'altezza nera della radice (come detto sopra, ogni nodo rosso deve essere seguito da uno nero), e quindi $n \geq 2^{h/2} - 1$. Semplificando, si ottiene $n + 1 \geq 2^{h/2}$ e applicando il logaritmo ad entrambi i lati della disequazione si ottiene $h \leq 2 \log(n + 1)$.

Questo garantisce che tutte le operazioni di ricerca, inserimento e cancellazione possono essere completate in tempo $O(\log n)$, dove n è il numero di nodi.

Le operazioni lookupNode(), successorNode(), predecessorNode(), min() e max() non richiedono alcuna modifica rispetto al normale comportamento degli alberi binari di ricerca. Tuttavia, gli inserimenti o le cancellazioni possono violare i vincoli Red-Black; ripristinare tali vincoli può richiedere una quantità $O(\log n)$ di modifiche all'albero, ognuna delle quali con costo $O(1)$. Nel seguito illustreremo come effettuare le modifiche, illustrando solo informalmente la loro correttezza: per i dettagli precisi e le dimostrazioni formali (molto lunghe e macchinose) si faccia riferimento agli articoli originali.

Le modifiche possono essere di due tipi:

- *cambi di colore*, in cui la variabile *color* viene mutata;
- *rotazioni*, in cui la struttura dell'albero binario di ricerca viene cambiata, senza tuttavia violare le proprietà di ordinamento di tali alberi.

Le rotazioni muovono un nodo "verso l'alto" e un nodo "verso il basso"; lo scopo è ridurre lo sbilanciamento dell'albero, scegliendo una coppia di nodi tali per cui il nodo spostato in alto abbia un sottoalbero con altezza maggiore del sottoalbero del nodo spostato in basso.

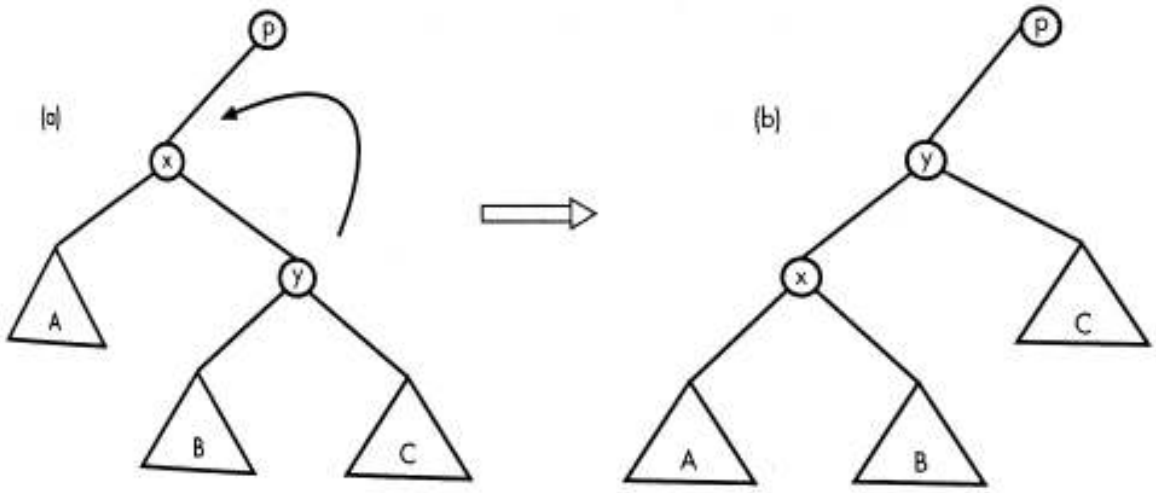


Fig. 6.4 Rotazione a sinistra a partire dal nodo x. (a) Situazione iniziale. (b) Situazione finale.

Si parla di rotazione sinistra o destra, a seconda della direzione in cui i nodi vengono spostati. La Fig. 6.4 illustra una rotazione sinistra a partire dal nodo x; la rotazione destra è speculare. L'idea è spostare x verso il basso e y verso l'alto; per farlo, bisogna:

(1) spostare il sottoalbero B come figlio destro di x;

(2) far diventare x il figlio sinistro di y;

(3) far diventare y figlio di p, il vecchio padre di x.

Si noti che le proprietà dell'ABR non vengono violate. Infatti, i sottoalberi A e C restano figli sinistro di x e destro di y, rispettivamente; x diventa figlio sinistro di y, senza causare problemi in quanto la chiave in x è minore della chiave in y; infine, le chiavi contenute in B sono maggiori della chiave in x, e vengono correttamente spostate nel sottoalbero destro di x.

Il codice associato a queste operazioni è illustrato nella procedura rotateLeft(); le righe sono marcate con (1), (2) e (3) per identificare le corrispondenti operazioni. La procedura restituisce il nodo che ha sostituito x, ovvero y. Il costo di una rotazione è ovviamente $O(1)$; la rotazione destra rotateRight() può essere ottenuta semplicemente scambiando fra loro ogni occorrenza delle variabili *left* e *right*.

```
TREE rotateLeft(TREE x)
  TREE y ← x.right
  TREE p ← x.parent
(1) x.right ← y.left                          % Il sottoalbero B diventa figlio destro di x
(1) if y.left ≠ nil then y.left.parent ← x
(2) y.left ← x                                % x diventa figlio sinistro di y
(2) x.parent ← y
(3) y.parent ← p                              % y diventa figlio di p
(3) if p ≠ nil then
      if p.left = x then p.left ← y else p.right ← y
  return y
```

La procedura di inserimento insertNode() va modificata come segue. Nel caso il nuovo elemento inserito causi la creazione di un nuovo nodo, una procedura di bilanciamento balanceInsert() viene chiamata su di esso. Al termine del bilanciamento, è possibile che sia stata creata una nuova radice, che va restituita al chiamante. Il modo piu semplice per identificare la nuova radice è risalire l'albero a partire dal nuovo nodo, seguendo i puntatori ai padri. Tali modifiche sono evidenziate in nero nell'algoritmo seguente, dove la parte inalterata viene mostrata in grigio.

```
TREE insertNode(TREE T, ITEM x, ITEM v)
  TREE p ← nil
  TREE u ← T                                  % Padre
  while u ≠ nil and u.key = x do
      p ← u                                   % Cerca posizione inserimento
      u ← iif(x < u.key, u.left, u.right)
  if u ≠ nil and u.key = x then
      u.value ← v
  else                                        % Chiave già presente
      TREE n ← Tree(x, v)
      link(p, n, x)                           % Crea un nodo contenente coppia chiave-valore
      balanceInsert(n)
  while n.parent ≠ nil do
      n ← n.parent
  return n                                    % Restituisce radice corrente dell'albero
```

Il nodo appena inserito (da qui in poi chiamato t) viene colorato di rosso; questo garantisce che il vincolo (v_4) sia comunque rispettato, perché le altezze nere delle for

glie non cambiano; ma è possibile che il vincolo (v_3) non sia più vero, in quanto il padre di t potrebbe essere rosso anch'esso; oppure è possibile che la radice sia rossa, se t è il primo nodo ad essere inserito (ovvero la radice); in questo caso viene violato il vincolo (v_1).

La procedura balanceInsert() è organizzata in sette diversi casi, illustrati in Fig. 6.5, ognuno dei quali corregge un possibile problema. I casi (1), (2), (5a), (5b) sono terminali, ovvero al completamento della loro esecuzione l'albero rispetta i vincoli Red-Black; il caso (3) risolve il problema localmente, ma il mancato rispetto dei vincoli si può ripresentare più in alto nell'albero, con uno qualunque dei 7 casi; infine, i casi (4a) e (4b) portano ai casi (5a) e (5b), rispettivamente. La procedura balanceInsert() è quindi organizzata tramite un ciclo **while**, che viene ripetuto finché $t \neq$ **nil**, ovvero fino a quando non si finisce in uno dei casi (1), (2), (5a) e (5b).

Per comodità di esposizione, all'inizio di ogni iterazione del ciclo **while** vengono inizializzate tre variabili p, n e z, che corrispondono al padre, al "nonno" (padre del padre) e allo "zio" (fratello del padre). Per i casi (1) e (2) è possibile che siano pari a **nil**, ovvero che tali nodi non esistano. Vediamo quindi i casi:

(1) Se il nuovo nodo t non ha padre, è il primo ad essere inserito e quindi è la radice. È sufficiente colorare questo nodo di nero e terminare.

(2) Se il padre di t esiste ed è nero, non c'è da fare alcun ulteriore aggiornamento: né il vincolo (v_1) né il vincolo (v_3) sono violati e si può terminare.

Nei casi seguenti il padre è rosso. Possiamo quindi dedurre che esiste il nonno, in quanto un nodo rosso non può essere radice per il vincolo (v_1); per il vincolo (v_4), esiste anche lo zio.

(3) Se lo zio è rosso, è possibile colorare di nero il padre e lo zio, e di rosso il nonno. Poiché tutti i cammini che passano per lo zio e il padre devono passare necessariamente per il nonno, la lunghezza dei cammini neri non è cambiata. Ma il problema può essersi spostato sul nonno: può essere violato il vincolo (v_1), ovvero il nonno può essere una radice rossa; oppure il vincolo (v_3), ovvero il nonno rosso può avere un padre rosso. Poniamo $t = n$, e il ciclo continua.

Nei quattro casi seguenti, lo zio è nero. I quattro casi si distinguono per la struttura dei rapporti destro/sinistro delle coppie (p, t) e (n, p).

(4a),(4b) Si assuma che t sia figlio destro di p e che p sia figlio sinistro di n. Una rotazione a sinistra a partire dal nodo p scambia i ruoli di t e p, ottenendo una situazione simile a (5a), dove i nodi rossi in conflitto sul vincolo (v_3) sono entrambi figli sinistri dei loro padri. Dobbiamo quindi porre $t = p$ ed eseguire un'ulteriore (ultima) iterazione del ciclo. Si noti che alcuni cammini radice-foglia sono ora instradati diversamente; ma i nodi coinvolti nel cambiamento sono p e t, entrambi rossi, quindi la lunghezza dei cammini neri non cambia. Il caso (4b) è speculare a (4a), con t figlio sinistro di p e p figlio destro di n; in questo caso è necessaria una rotazione a destra.

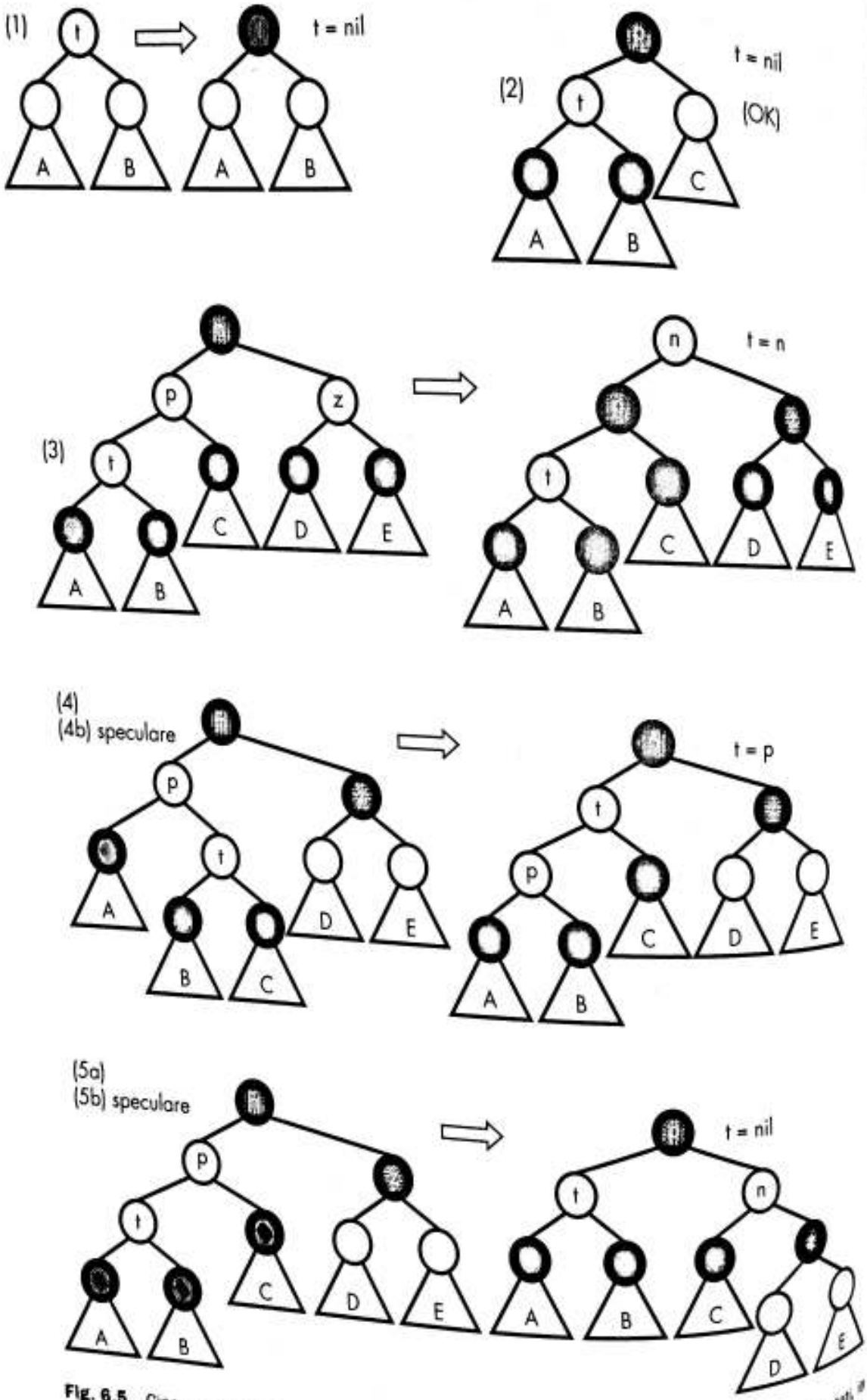


Fig. 6.5 Cinque dei sette possibili casi di inserimento. I casi (4b) e (5b) non sono rappresentati, in quanto speculari ai casi (4a) e (5a). Il caso (2) non richiede alcuna modifica, mentre tutti gli altri casi collegano la situazione iniziale e la situazione finale con una freccia.

(5a),(5b) Si assuma infine che t sia figlio sinistro di p e p sia figlio sinistro di n. Una rotazione a destra a partire da n ci porta ad una situazione in cui t e n sono figli di p; colorando n di rosso e p di nero ci troviamo in una situazione in cui tutti i vincoli Red-Black sono rispettati; in particolare, la lunghezza di tutti i cammini neri che passano per la radice è uguale alla situazione iniziale. Il caso (5b) è speculare a (5a), con t figlio destro di p e p figlio destro di n; in questo caso è necessaria una rotazione a sinistra.

La complessità della procedura balanceInsert() è $O(h)$, dove h è l'altezza dell'albero. Infatti, l'esecuzione di ognuno dei singoli casi è $O(1)$. Il caso (3) sposta il problema a livello del nonno, e quindi il suo codice può essere ripetuto al massimo $h/2$ volte. Tutti gli altri casi possono essere eseguiti al massimo una volta. Poiché h è $O(\log n)$, l'operazione di inserimento ha costo logaritmico nel numero di nodi.

```
balanceInsert(TREE t)
t.color ← RED
while t ≠ nil do
    TREE p ← t.parent                                          % Padre
    TREE n ← iif(p ≠ nil, p.parent, nil)                       % Nonno
    TREE z ← iif(n = nil, nil, iif(n.left = p, n.right, n.left))   % Zio
    if p = nil then                                            % Caso (1)
        t.color ← BLACK
        t ← nil
    else if p.color = BLACK then                               % Caso (2)
        t ← nil
    else if z.color = RED then                                 % Caso (3)
        p.color ← z.color ← BLACK
        n.color ← RED
        t ← n
    else
        if (t = p.right) and (p = n.left) then                 % Caso (4.a)
            rotateLeft(p)
            t ← p
        else if (t = p.left) and (p = n.right) then            % Caso (4.b)
            rotateRight(p)
            t ← p
        else
            if (t = p.left) and (p = n.left) then              % Caso (5.a)
                rotateRight(n)
            else if (t = p.right) and (p = n.right) then       % Caso (5.b)
                rotateLeft(n)
            p.color ← BLACK
            n.color ← RED
            t ← nil
```

Per quanto riguarda la cancellazione, ricordiamo che la procedura removeNode() è suddivisa in tre casi. Quando si cancella un nodo u con due figli (nel caso degli alberi Red-Black, con due figli non-foglie), si cerca l'elemento minimo s del sottoalbero destro di u (il successore) e si copia la coppia chiave-valore da s in u. Il colore del nodo u può rimanere inalterato, e quindi i vincoli Red-Black vengono rispettati, ma il problema si sposta nel nodo s, che deve essere rimosso e può avere un figlio destro. Si ricade quindi negli altri due casi.

Da questo punto in avanti, possiamo quindi assumere che stiamo cancellando un nodo u con al massimo un figlio t non foglia. La chiamata balanceDelete(T,t), dove T e la radice dell'albero, avviene subito dopo la chiamata a link() nella procedura removeNode(). Al termine della chiamata, poiché la radice può essere cambiata, è necessario identificare la nuova radice, così come avevamo fatto con la balanceInsert() e restituirla così al chiamante.

```
TREE removeNode(TREE T, ITEM x)
  TREE u ← lookupNode(T,x)
  if u ≠ nil then
      if u.left ≠ nil and u.right ≠ nil then                    % Caso 3
          TREE s ← u.right
          while s.left ≠ nil do s ← s.left
          u.key ← s.key
          u.value ← s.value
          x ← s.key
          u ← s
      TREE t
      if u.left ≠ nil and u.right = nil then                    % Caso (2) -Solo figlio sx
          t ← u.left
      else                                                      % Caso (2) -Solo figlio dx / Caso (1)
          t ← u.right
      link(u.parent, t, x)
      if u.color = BLACK then
          balanceDelete(T,t)
      if u.parent = nil then T ← t
      delete u
  while T.parent ≠ nil do
      T ← T.parent
  return T
```

La chiamata ha luogo se e solo se il nodo cancellato (e sostituito da t) ha colore nero. In questo caso, possiamo avere violato uno dei seguenti vincoli:

- la radice può ora essere rossa, violando il vincolo (v_1), in quanto il nodo t che ha sostituito u può avere questo colore;
- se il padre di u e t sono rossi, il fatto che ora siano uno padre dell'altro viola il vincolo (v_3);
- rimuovendo un nodo nero, abbiamo ridotto di uno l'altezza nera di tutte le foglie i cui cammini verso la radice passavano per u (e ora passano per t), violando quindi il vincolo (v_4).

Se invece il nodo u era rosso, la sua rimozione non cambia l'altezza nera delle foglie, non può creare due nodi rossi consecutivi, e non può mutare il colore della radice in rosso: in altre parole, non c'è alcun aggiornamento da fare.

La procedura balanceDelete() è organizzata in otto possibili casi; i primi quattro, illustrati in Fig. 6.6, assumono che il nodo su cui opera la balanceDelete() sia figlio sinistro di suo padre; gli altri casi sono speculari, con t figlio destro di suo padre e tutti i riferimenti sinistro-destro scambiati.

Come nel caso della balanceInsert(), il passaggio fra i vari casi è gestito tramite un ciclo **while**. Ad ogni iterazione del ciclo, partendo dal nodo t si ottengono i riferimenti al padre p, al fratello f e ai nipoti destro e sinistro nd e ns. Il ciclo termina quando il nodo t è la radice, oppure se è colorato di rosso. Al termine del ciclo, il nodo t viene colorato di nero. Questa operazione garantisce il rispetto del vincolo (v_1) (se il nodo colorato di nero è la radice) o del vincolo (v_3) (colorando un nodo nero, "spezziamo" il caso in cui il nodo e uno dei suoi figli siano entrambi rossi). Vedremo nella discussione dei casi come venga ripristinato il vincolo (v_4).

```
balanceDelete(TREE T, TREE t)
while t ≠ T and t.color = BLACK do
    TREE p ← t.parent                                   % Padre
    if t = p.left() then
        TREE f ← p.right                                % Fratello
        TREE ns ← f.left                                % Nipote sinistro
        TREE nd ← f.right                               % Nipote destro
        if f.color = RED then                           % (1)
            p.color ← RED
            f.color ← BLACK
            rotateLeft(p)
            % t viene lasciato inalterato, quindi si ricade nei casi (2), (3), (4)
        else
            if ns.color = nd.color = BLACK then         % (2)
                f.color ← RED
                t ← p
            else if ns.color = RED and nd.color = BLACK then   % (3)
                ns.color ← BLACK
                f.color ← RED
                rotateRight(f)
                % t viene lasciato inalterato, quindi si ricade nel caso (4)
            else if nd.color = RED then                 % (4)
                f.color ← p.color
                p.color ← BLACK
                nd.color ← BLACK
                rotateLeft(p)
                t ← T
    else
        % Casi (5)-(8) speculari a (1)-(4)
if t ≠ nil then t.color ← BLACK
```

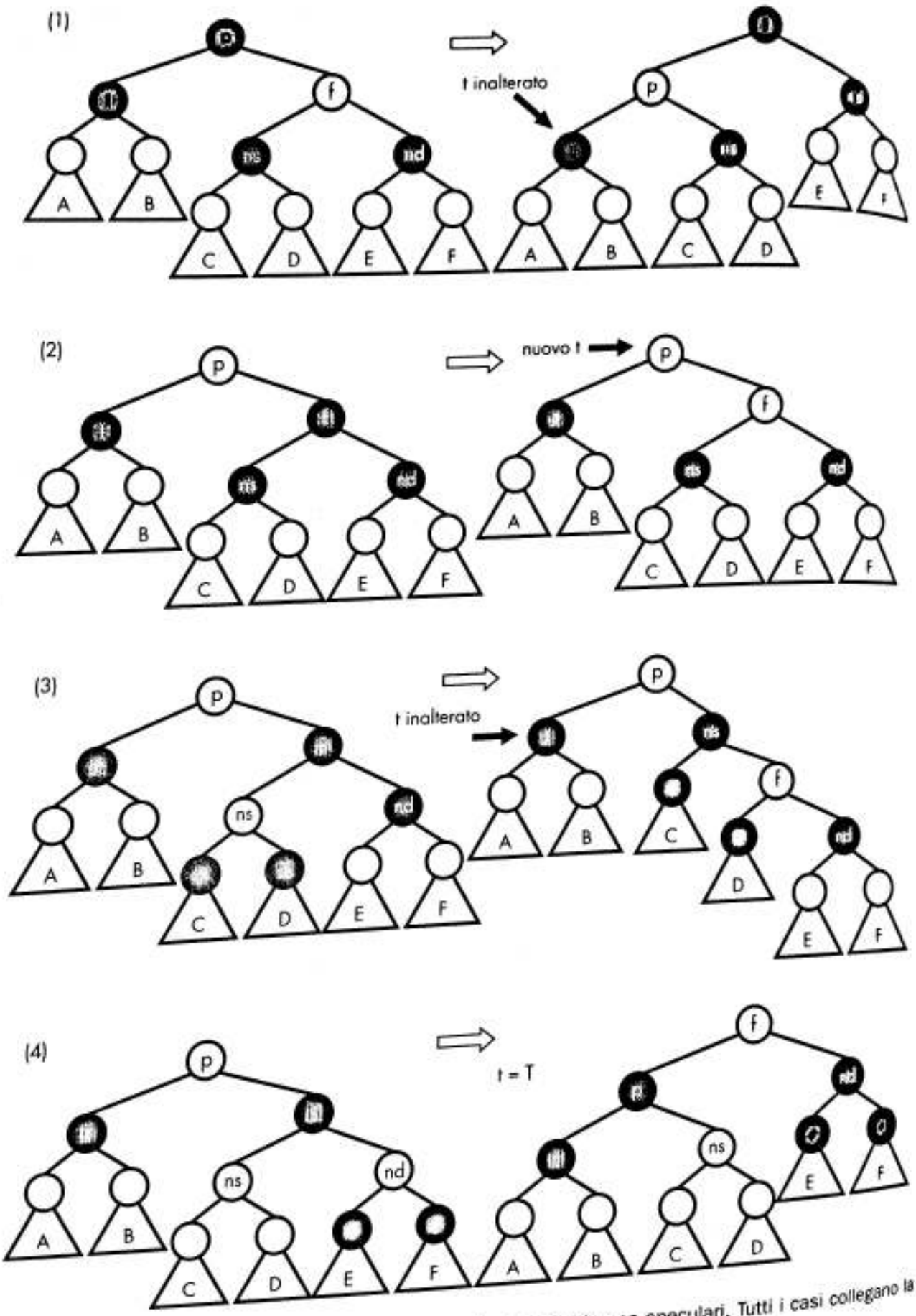


Fig. 6.6 Quatto degli otto possibili casi di cancellazione, gli altri sono speculari. Tutti i casi collegano la situazione iniziale e la situazione finale con una freccia.

(1) Se il fratello f è rosso, i suoi figli ns e nd e il padre p devono essere necessariamente neri. I percorsi radice-foglia che passano per t hanno perso un nodo nero, quindi il vincolo (v_4) è stato violato. Scambiando i colori di f e p ed applicando una rotazione sinistra, lasciando invariato il puntatore t, ci ritroviamo in uno dei casi (2), (3), (4), in quanto i percorsi radice-foglia che passano per t hanno ancora un nodo nero mancante, ma ora il fratello di t non è più rosso. Gli altri percorsi radice-foglia soggetti alla rotazione ed ai cambi di colore non subiscono variazioni nella lunghezza nera.

Nei casi che seguono, il colore di f è sempre nero.

(2) ns e nd sono entrambi neri, mentre p può assumere un qualunque colore. Come prima, i percorsi radice-foglia che passano per t hanno perso un nodo nero, quindi il vincolo (v_4) è stato violato; colorando di rosso il nodo f, anche tutti i percorsi radice-foglia che passano per esso perdono un nodo nero. Questo significa che tutti i percorsi che passano per p (padre di t e f) sono nelle stesse condizioni. Il puntatore t assume quindi il valore p, e il ciclo ricomincia. A questo punto, se il nuovo nodo t ha colore rosso, il ciclo termina e t viene colorato di nero; questo aggiunge un nodo nero a tutti i percorsi radice-foglia che passano per t, ripristinando il vincolo (v_4). Se invece t è già nero, il ciclo continua.

(3) Nel caso ns sia rosso e nd sia nero, si colora il nodo f di rosso, il nodo ns di nero e si applica una rotazione destra a f. Lasciando t inalterato, ci siamo spostati nel caso (4). Nel sottoalbero radicato in f non si sono violate le regole (v_3) e (v_4); il problema resta nel nodo t.

(4) Nel caso nd sia rosso, applichiamo una rotazione sinistra a p; coloriamo f con il colore del padre, coloriamo nd di nero e coloriamo p di nero. In questo modo, abbiamo aggiunto un nodo nero ai cammini radice-foglia che passano per t, senza modificare la lunghezza nera dei cammini che passavano per f. Ponendo $t = T$, terminiamo l'algoritmo e ci salvaguardiamo anche dal caso in cui f sia diventato radice e sia rosso.

Per riassumere, dal caso (1) si passa ad uno dei casi (2), (3), (4); dal caso (2) si torna ad uno qualunque dei casi, ma risalendo di un livello l'albero. Dal caso (3) si passa al caso (4), mentre nel caso (4) si termina. In altre parole, è possibile visitare al massimo un numero $O(\log n)$ di casi, ognuno dei quali è gestito in tempo $O(1)$. In altre parole, la complessità di balanceDelete() e quindi di delete() è pari a $O(\log n)$.

6.3 REALITY CHECK

Fra gli alberi bilanciati, l'approccio Red-Black è quello più diffuso: lo troviamo, ad esempio, nelle strutture di dati `TreeMap` e `TreeSet` delle Java Collections, così come nei

container `std::set<Value>` e `std::map<Key, Value>` della Standard Template Library del C++.

Un approccio diverso al bilanciamento degli alberi è quello dei *B-alberi* ed in particolare della variante B⁺. Il loro nome deriva forse dall'iniziale di Bayer che li ha inventati insieme a McCreight negli anni '70, oppure da Boeing, l'azienda presso cui Bayer e McCreight lavoravano, o piuttosto dall'iniziale del termine "balanced".

A differenza degli alberi Red-Black, i B-alberi sono ottimizzati per sistemi di memorizzazione orientata alle pagine, quali database e file system. In questi sistemi, i dati vengono memorizzati in memoria secondaria, divisi in pagine di grandi dimensioni (e.g., 4KB). Il tempo di accesso a tali pagine è misurato in millisecondi e non in nanosecondi, come avviene in memoria principale; inoltre, ogni lettura comporta l'acquisizione di un'intera pagina, anche se è necessario accedere ad un singolo dato. Una struttura di dati ottimizzata per questi sistemi deve quindi minimizzare il numero di letture di pagine diverse, cercando di sfruttare il più possibile tutti i dati contenuti in una pagina. I B⁺-alberi permettono non solo di eseguire ricerche, inserimenti e cancellazioni in tempo logaritmico, ma anche di accedere sequenzialmente ai dati in modo facile ed efficiente.

I B⁺-alberi sono implementati in un discreto numero di file system, fra i quali i più famosi sono ReiserFS, XFS, JFS e NTFS, e nei database relazionali quali IBM DB2, Oracle 8, e Microsoft SQL Server. Vediamo rapidamente i principi del loro funzionamento, senza entrare nei loro complessi dettagli implementativi.

In un B⁺-albero, ogni nodo è memorizzato in una pagina di memoria secondaria. Ogni nodo contiene una sequenza di chiavi e puntatori, assieme ad un contatore che indica il numero di chiavi presenti e un flag booleano che dice se il nodo è interno oppure una foglia. I puntatori dei nodi interni puntano ad altri nodi memorizzati in memoria secondaria; nei nodi foglia, invece, puntano ai dati associati alla chiave, come ad esempio un file oppure una riga di database. In altre parole, i nodi interni non contengono valori, ma permettono solo l'esplorazione dell'albero verso le foglie, dove i dati sono memorizzati. Inoltre, i B⁺-alberi si contraddistinguono dal fatto che le foglie si trovano tutte allo stesso livello e sono collegate in una lista doppia, per permettere interrogazioni su intervalli di dati in maniera efficiente.

Ogni B⁺-albero è caratterizzato da un parametro $m > 2$, detto anche *ordine* del B⁺-albero, che rappresenta il numero massimo di figli che un nodo interno può avere; questo valore può essere molto elevato, ben superiore al valore 2 che si osserva negli alberi binari. Ogni nodo interno deve essere pieno almeno per metà; ovvero, detto f il numero di figli di un nodo, $\lceil m/2 \rceil \leq f \leq m$. Fa eccezione la radice, che può avere anche due figli (o essere una foglia).

La struttura di un B⁺-albero è illustrata in Fig. 6.7. I nodi interni contengono fino ad $m - 1$ chiavi $k[1, \ldots m - 1]$ e fino ad m puntatori $p[1, \ldots, m]$. Ogni puntatore $p[i]$, con $1 < i < m$, punta al sottoalbero contenente le chiavi comprese fra $k[i - 1]$ (estremo escluso) e $k[i]$ (estremo incluso). Il puntatore $p[1]$ punta al sottoalbero contenente le chiavi minori o uguali a $k[1]$, mentre il puntatore $p[m]$ contiene le chiavi maggiori di $k[m - 1]$. Nei nodi foglia, invece, sono presenti fino ad m coppie chiavi-puntatori, dove il puntatore punta al dato indicizzato, come un file in un file system o una riga in un database.

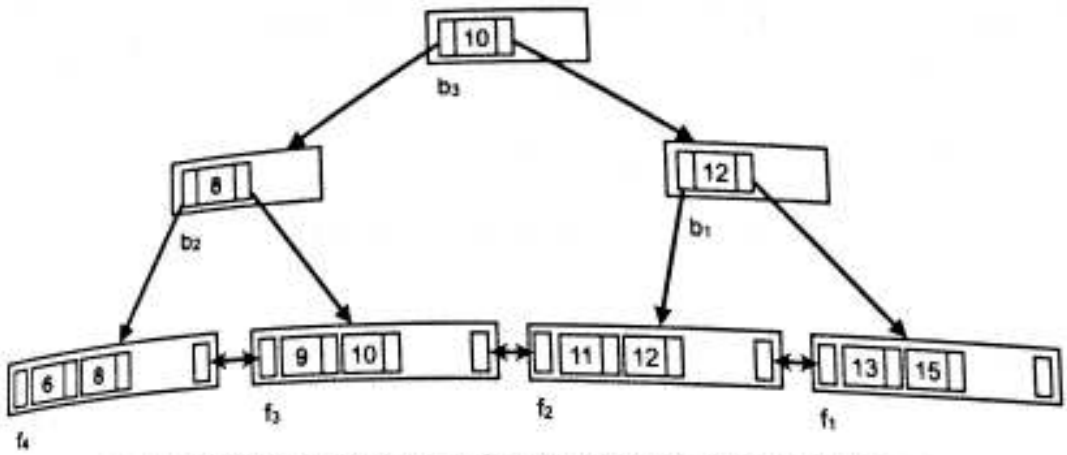


Fig. 6.7 Un B+-albero di ordine m =3 contenente 3 nodi interni e 5 foglie con 11 chiavi.

In base a queste considerazioni, l'algoritmo per la ricerca di una coppia chiave-valore parte dalla radice e seleziona il primo puntatore la cui corrispondente chiave è maggiore o uguale alla chiave cercata, oppure $p[m]$ se nessuna chiave memorizzata nel nodo soddisfa questo requisito. Si sposta quindi nel sottolbero indicato dal puntatore e continua così ricorsivamente nei sottoalberi analizzati. Una volta raggiunta una foglia, la chiave viene cercata fra quelle memorizzate all'interno ed eventualmente viene restituito il valore associato.

ESEMPIO – [Ricerca in un B⁺-albero]

Volendo cercare la chiave 12 nell'albero di Fig. 6.7, si caricherà innanzitutto il nodo radice b_3 dalla memoria secondaria in memoria principale. Poiché 12 è maggiore di 10, si utilizzerà il secondo puntatore di b_3 e si caricherà in memoria principale la pagina b_1. Poiché 12 è minore o uguale a 12, si utilizzerà il primo puntatore e si caricherà in memoria la pagina f_2. Poiché f_2 è una foglia e contiene 12, si utilizzerà il puntatore associato a 12 per raggiungere il valore cercato.

La procedura di inserimento in un B⁺-albero è logicamente semplice, ma complessa da programmare. Ne vediamo quindi solo a grandi linee il funzionamento. Per comodità di presentazione, assumiamo che m sia dispari, ovvero sia uguale ad un valore $2d+1$, e assumiamo che non siano ammesse chiavi ripetute.

Innanzitutto, si sfrutta il meccanismo di ricerca per individuare la foglia f_i in cui la chiave dovrebbe essere collocata. Se f_i non è ancora piena (il numero di elementi è inferiore a m), la chiave e il relativo puntatore vengono aggiunti ad essa. Per costruzione della procedura di ricerca, le chiavi e i puntatori del nodo padre della foglia non richiedono di essere modificati e la procedura di inserimento termina.

Altrimenti, se f_i è piena (contiene già $m = 2d+1$ chiavi), si crea una nuova foglia f_j e si dividono equamente le $2d+2$ chiavi (quelle contenute in f_i e quella da inserire) fra f_j (le prime $d+1$) e f_j (le seconde $d+1$). Il valore $d+1$ è pari a $\lfloor m/2 \rfloor$, il numero minimo di figli che devono essere contenuti in un nodo.

A questo punto, la chiave massima contenuta in f_j e il puntatore a f_j devono essere inseriti nel nodo interno b_k padre di f_i. Se b_k ha meno di $m = 2d+1$ figli, la coppia chiave-

puntatore viene semplicemente inserita (in ordine) nella pagina e la procedura di inserimento termina.

Altrimenti, se b_k ha $m = 2d + 1$ figli, si genera un'ulteriore pagina b_l ei $2d + 2$ puntatori vengono divisi fra b_l e b_k, usando lo stesso principio utilizzato per suddividere le foglie. Questo genera un nuovo inserimento al livello superiore, che può arrivare fino alla radice. Nel caso venga suddivisa la radice, si genera una nuova radice che inizialmente ha due figli.

Come ultimo caso particolare, si consideri il B⁺-albero quando non è presente alcuna chiave. I primi m inserimenti vengono effettuati direttamente nella radice, che inizialmente è una foglia. Non appena questa foglia viene divisa in due foglie, viene generato un nodo interno che agisce come radice e ha le due foglie come figli.

ESEMPIO – [Inserimenti in un B⁺-albero]

Vediamo una sequenza di inserimento di chiavi che genera il B⁺-albero di Fig. 1.7. Si assuma che il parametro m sia uguale a 3, un valore basso ma necessario per motivi grafici. Si osservi la Fig. 6.8. Inizialmente (parte (a)), vengono inserite le chiavi 10,12,13, che vengono inserite nella foglia che costituisce la radice. Nella parte (b), viene inserita la chiave 15, che forza la creazione di f_2 e la suddivisione delle chiavi fra f_1 e f_2, oltre che la generazione del primo nodo interno, la radice b_1. L'inserimento della chiave 8 nella parte (c) non crea alcuna suddivisione, ma riempe la foglia f_2. Nella parte (d), l'inserimento della chiave 11 comporta la creazione di f_3 e la relativa suddivisione di chiavi; la chiave più alta inserita in f_3 viene inserita in b_1, che quindi ora è completo. Nella parte (e), la chiave 9 viene inserita in f_3, senza modifiche ai nodi interni. Infine, l'inserimento della chiave 6 forza la suddivisione di f_3 in f_3 ed f_4; l'inserimento della chiave 8 in b_1 comporta la suddivisione anche di questa pagina, con la conseguente creazione di una nuova radice, mostrata in Fig. 6.7.

La procedura di cancellazione comporta una serie di casi simili, che possono portare all'eliminazione della radice e alla riduzione dell'altezza dell'albero.

A questo punto, è necessario fare alcune considerazioni sull'efficienza di questo approccio. Il valore $m = 3$ mostrato qui non è realistico; se si considerano pagine di 4KB. e coppie chiave-puntatore di 16 byte (due interi a 64 bit), è possibile ottenere un valore $m =$ 256. Il fatto che le pagine debbano avere almeno $\lfloor m/2 \rfloor$ figli, indica che nel caso pessimo tutti i nodi sono riempiti con almeno 128 puntatori. Questo significa che l'albero risultante avrà altezza pari a $O(\log_{128} n)$, dove n è il numero di chiavi inserite. Sebbene questo valore sia comunque logaritmico, la base molto alta fa sì che il numero di nodi da caricare in memoria principale sia molto basso, ottimizzando così il tempo di accesso alla memoria secondaria. Dal punto di vista dell'occupazione di memoria, poiché i nodi possono essere mezzi pieni, questo comporta un utilizzo al più doppio della memoria necessaria per memorizzare tale albero. Questo "spreco" è giustificato dalla maggiore efficienza della struttura.

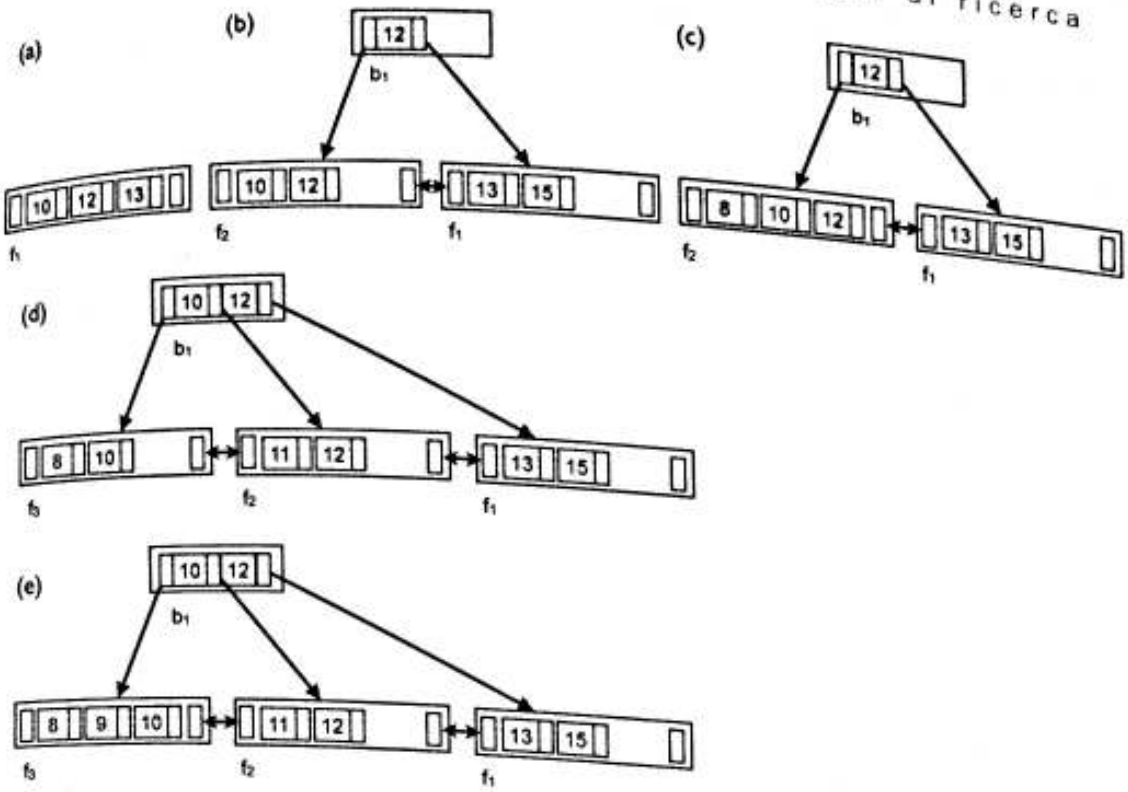


Fig. 6.8 Inserimento di chiavi in un B⁺-albero di ordine $m = 3$. (a) Vengono inserite le chiave 10, 12, 13; (b) viene inserita la chiave 15; (c) viene inserita la chiave 8; (d) viene inserita la chiave 11; (e) viene inserita la chiave 9. Infine, con l'inserimento della chiave 6, si genera il B⁺-albero di Fig. 1.7.

ESERCIZI

6.1 Dimostrare che, poiché l'ordinamento di n elementi basato su confronti richiede tempo $\Omega(n \log n)$, allora qualunque algoritmo basato su confronti per costruire un ABR contenente n valori è $\Omega(n \log n)$.

6.2 Si scriva una versione ricorsiva della procedura insertNode().

6.3 Si scriva una funzione che verifichi se un albero binario è un albero di ricerca usando gli operatori degli alberi binari.

6.4 [Stampa intervallo] Dato un albero binario di ricerca i cui nodi contengono interi positivi, si scriva una procedura efficiente che usi gli operatori degli alberi binari e stampi, in ordine crescente, tutte le chiavi che appartengono ad un dato intervallo $[A, B]$.

6.5 [Invisita iterativa] Dato un albero binario di ricerca realizzato con puntatori e i cui nodi contengono interi, si scriva una versione iterativa della visita simmetrica senza usare una pila.

6.6 [Concatenazione] Dati due alberi binari di ricerca T_1 e T_2 tali che le chiavi in T_1 sono tutte minori delle chiavi in T_2, scrivere una procedura che restituisce un albero di ricerca contenente tutte le chiavi in tempo $O(h)$, dove h è l'altezza massima dei due alberi.

6.7 [Realizzazione di insiemi con ABR] Si assuma di rappresentare gli insiemi con ABR. Quali sono le complessità delle operazioni union(), difference() e intersection()?

6.8 [Realizzazione di insiemi con alberi Red-Black] Si assuma di rappresentare gli insiemi dell'esercizio precedente con alberi Red-Black. Vi sono dei cambiamenti da apportare alla vostra soluzione? La complessità cambia?

6.9 [Colorazione Red-Black] Quali degli alberi in Fig. 6.7 possono essere colorati rispettando i vincoli degli alberi Red-Black e quali no?

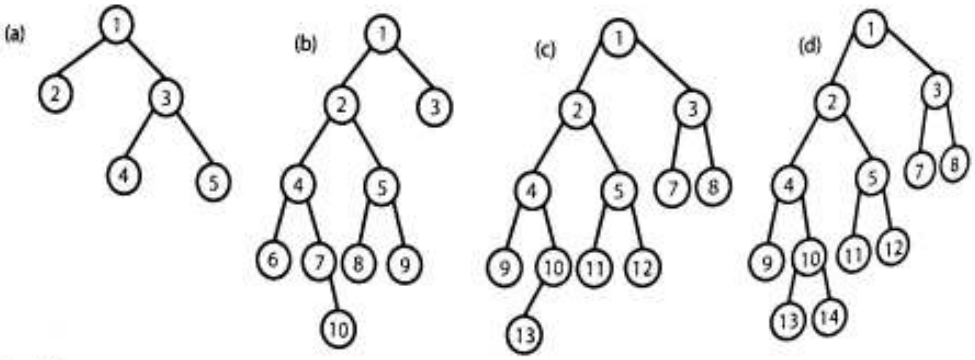


Fig. 6.7 Alberi da colorare.

6.10 [Dimensioni degli alberi Red-Black] Qual è il numero minimo di nodi, non foglie, di un albero Red-Black di altezza nera b? Qual è il numero massimo?

63 Supponiamo per assurdo che esista un algoritmo basato su confronti che permetta di costruire l'albero in tempo $\Theta(g(n))$, e che $g(n)$ non sia $\Omega(n \log n)$. Allora sarebbe possibile ordinare n numeri costruendo l'albero, e poi visitando lo stesso in ordine simmetrico (in tempo $\Theta(n)$). Questo algoritmo di ordinamento avrebbe costo $\Theta(g(n) + n)$ che non è $\Omega(n \log n)$, il che è impossibile vista la dimostrazione della limitazione inferiore per l'ordinamento.

64 L'algoritmo si basa su una visita simmetrica in cui vengono scartati i sottoalberi con chiavi non comprese nell'intervallo $[A, B]$. La sua complessità è $O(h + k)$, dove h è l'altezza dell'albero e k è il numero di chiavi che appartengono ad $[A, B]$. Infatti, occorre tempo $O(h)$ per individuare la chiave più piccola e quella più grande in $[A, B]$ e tempo $O(k)$ per visitare tutte le chiavi in $[A, B]$. La procedura assume che l'albero non sia vuoto.

```
stampaNodi(TREE t, integer A, integer B)
  integer v ← t.read()
  if t.left() ≠ nil and A < v then stampaNodi(t.left(), A, B)
  if A ≤ v and v ≤ B then print v
  if t.right() ≠ nil and B > v then stampaNodi(t.right(), A, B)
```

65 Si usano le funzioni min() e successorNode(), poiché una visita simmetrica di un ABR T visita le chiavi in ordine crescente. Una versione iterativa si ottiene immediatamente visitando per primo il nodo restituito da min() e poi richiamando iterativamente la successorNode() finché non si giunge al nodo che contiene la chiave massima (e che ha **nil** come successore).

```
invisita (TREE t)
  TREE u ← t.min()
  while u ≠ nil do
      esamina il nodo u
      u ← u.successorNode()
```

Nonostante invisita() chiami per ogni nodo una procedura di complessità $O(h)$, la complessità della visita è $\Theta(n)$ e non $O(nh)$. Infatti la funzione invisita() attraversa il minimo numero di nodi necessari a raggiungere il successore di u. Pertanto, la sequenza di n chiamate nella invisita() attraversa ciascuna coppia padre-figlio esattamente una volta scendendo dal padre al figlio e ciascuna coppia figlio-padre esattamente una seconda volta risalendo dal figlio al padre.

6.7 Siano A e B, con $|A| = n_A$ e $|B| = n_B$, gli insiemi in ingresso e C l'insieme da fornire come risultato, e denotiamo con T_X un albero bilanciato di ricerca che memorizza un generico insieme X. Per realizzare union() e difference(), occorre innanzitutto costruirsi T_C facendosi in tempo $O(n_A)$ una copia di T_A. Per l'union(), si visita T_B e si aggiunge a T_C ogni chiave k di T_B che non appartiene a T_A. Dato che la visita richiede tempo $O(n_B)$, mentre A.contains(k) e C.insert(k) ne consumano $O(\log n_A)$ e $O(\log (n_A + n_B))$, rispettivamente, la complessità della union() è $O(n_A + n_B \log (n_A + n_B))$. Un tempo $O(n_A + n_B \log n_A)$ è invece richiesto dalla Difference(), dato che occorre cancellare da T_C ogni chiave k di T_B che appartiene a T_A e C.remove(k) consuma anch'essa tempo $O(\log n_A)$. Per Intersection() invece non è necessario ricopiarsi T_A in T_C, ma si parte con T_C inizialmente vuoto, e vi si aggiunge ogni chiave k di T_B che appartiene a T_A, per un totale di $O(n_B \log n_A)$ tempo. Si noti che in quest'ultimo caso si possono eventualmente scambiare i ruoli di T_A e T_B in modo da effettuare la visita sull'albero col minor numero di chiavi. Si può quindi risparmiare tempo, rispetto alla realizzazione con liste ordinate, nel caso particolare che la cardinalità di un insieme sia molto minore della cardinalità dell'altro. Analoghe considerazioni valgono anche per la union() ove conviene che $n_A > n_B$. Infine, è interessante il caso in cui si voglia modificare direttamente uno dei due insiemi in input, per esempio A, perché in questa eventualità, essendo $C = A$, non occorre ricopiarsi T_A in T_C, e le complessità scendono a $O(n_B \log n_A)$ per difference() e intersection() e a $O(n_B \log (n_A + n_B))$ per union(); quindi si potrebbe di nuovo risparmiare tempo rispetto alla realizzazione con liste ordinate nel caso particolare che le cardinalità dei due insiemi siano molto diverse tra loro.

6.10 L'albero Red-Black di altezza nera b con il minimo numero di nodi è l'albero in cui in tutti i cammini radice-foglia non si incontrano nodi rossi, e tutte le foglie hanno lo stesso livello $h + 1$. L'altezza è quindi $h = b$. Il numero totale di nodi interni è allora $2^b - 1$.

L'albero Red-Black di altezza nera b con il massimo numero di nodi è l'albero perfetto che alterna livelli rossi a livelli neri. Se l'altezza nera è b, l'altezza dell'albero è $h = 2b$. Il numero totale di nodi interni è quindi $2^{2b} - 1$.

TABELLE HASH

Capitolo 7

Le *tabelle hash* sono un metodo per la realizzazione di dizionari e insiemi, alternativo agli alberi bilanciati visti nel capitolo precedente. In questa struttura di dati, l'idea è ricavare direttamente dal valore della chiave la posizione che la chiave stessa dovrebbe occupare in un vettore. Con questa realizzazione, ciascuna operazione, pur avendo complessità $O(n)$ nel caso pessimo, richiede tempo $O(1)$ nel caso medio.

Sia U l'universo delle chiavi, ovvero l'insieme di tutte le possibili chiavi distinte e si memorizzi il dizionario in un vettore A di dimensione m, con indici compresi fra 0 e $m-1$. La soluzione ideale sarebbe quella di possedere una funzione iniettiva

$$H : U \rightarrow \{0, ..., m-1\}$$

che permetta di ricavare la posizione $H(k)$ della chiave k nel vettore A in modo che per ogni $k_1 \in U$ e $k_2 \in U$, con $k_1 \neq k_2$, risulti $H(k_1) \neq H(k_2)$. Per garantire l'iniettività, ovviamente, m non può essere più piccolo di $|U|$.

Una possibilità è quella di utilizzare un vettore con $m = |U|$ posizioni, in cui si accede direttamente alla posizione contenente la chiave in tempo $O(1)$. Questo meccanismo prende il nome di *tabella ad accesso diretto*.

ESEMPIO – [Ritardatreno]

In una non precisata nazione, le ferrovie statali sono note per i loro ritardi. Invece di risolvere il problema, le ferrovie hanno deciso di mettere a disposizione degli utenti il sito `www.ritardatreno.it`, che dato un numero di treno, risponde con la sua posizione e il ritardo stimato. In questa nazione, i treni passeggeri sono numerati da 1 a 36 999, e ogni giorno viaggiano circa 10 000 treni. Visto che il sito è molto popolare, i progettisti hanno deciso di ottimizzare le ricerche mantenendo in memoria una tabella di dimensione $m = 36\,999$, le cui posizioni sono riempite per poco meno di un terzo.

Questo metodo funziona se U non è troppo grande, oppure se la dimensione dell'insieme

K delle chiavi effettivamente inserite nella tabella non è dissimile dalla dimensione di U. In caso contrario, questo metodo è improponibile perché comporta un enorme spreco di memoria.

ESEMPIO – [Tabella dei simboli]

I compilatori utilizzano un dizionario, detto tabella dei simboli, per memorizzare tutti gli identificatori (nomi di variabili, costanti, procedure ecc.) definiti dall'utente all'interno del proprio codice. Questi identificatori sono inseriti nel dizionario quando sono compilate le "dichiarazioni", mentre la loro appartenenza è verificata quando sono compilate le "istruzioni eseguibili". Nel passato, molti compilatori ammettevano come simboli stringhe di al più 8 caratteri alfanumerici senza distinzione di minuscole e maiuscole, di cui il primo alfabetico; in questo caso, $|U| = 26 \cdot 36^7 > 2 \cdot 10^{12}$. È chiaramente impensabile utilizzare un vettore con oltre duemila miliardi di posizioni per un programma che potrà contenere, al più, qualche centinaio di identificatori! Oggigiorno, linguaggi come Java permettono identificatori di lunghezza illimitata, quindi utilizzare una tabella di dimensione U perde completamente di senso.

Per evitare inutili sprechi di memoria, la dimensione m del vettore va scelta in base al numero di chiavi attese $|K|$, cioè quelle che si crede possano essere effettivamente memorizzate contemporaneamente, e non in base al numero di tutte le chiavi possibili, la maggior parte delle quali potrebbe non apparire mai nel dizionario. Occorre quindi rinunciare alla iniettività della funzione H ammettendo che chiavi diverse possano avere lo stesso indirizzo nel vettore, cioè che $H(k_1) = H(k_2)$ anche se $k_1 \neq k_2$.

In tal caso, la soluzione estrema che minimizza l'uso della memoria sembra quella in cui ci sia un unico indirizzo $H(k)$ per ogni possibile chiave k, per esempio mantenendo tutte le chiavi in un'unica lista. In questo modo, la memoria richiesta è linearmente proporzionale al numero n di chiavi presenti nel dizionario, ma anche il tempo per le operazioni cresce con un inaccettabile $O(n)$. Occorre pertanto una soluzione di compromesso, con m maggiore di 1 ma molto minore di $|U|$, che mantenga bassi sia il tempo di esecuzione delle operazioni (possibilmente $O(1)$, come nel caso $m = |U|$), sia l'occupazione di memoria (possibilmente $O(n)$, come nel caso $m = 1$).

Rinunciando alla biunivocità di H, sorge il nuovo problema delle *collisioni*, ovvero come scegliere le posizioni per memorizzare due chiavi k_1 e k_2 che hanno lo stesso indirizzo $H(k_1) = H(k_2)$, qualora siano entrambe presenti nel vettore. Un esempio è utile per illustrare le principali problematiche che si presentano.

ESEMPIO – [Musicisti]

Sia U = {cognomi di musicisti} e si assuma $m = 26$. Si consideri la funzione $H(k) = h$, $0 \leq h \leq 25$, se il carattere iniziale di k è l'$(h + 1)$-esima lettera dell'alfabeto inglese. La funzione H non è biunivoca, poiché, p.e., H(`Bach`) = H(`Beethoven`) = H(`Brahms`) = 1.

Sia A inizialmente vuoto e si supponga di inserire le chiavi `Albinoni`, `Offenbach` e `Palestrina`, in quest'ordine. Poiché H(`Albinoni`) = 0, H(`Offenbach`) = 14 e H(`Palestrina`) = 15, le tre chiavi saranno inserite, rispettivamente, nelle posizioni 0, 14, e 15. Se successivamente si presenta `Prokofev`, poiché H(`Prokofev`) = 15 e la posizione 15 è occupata da `Palestrina`, nasce una collisione tra le due chiavi: dove sistemare `Prokofev`? Una scelta ragionevole è quella di cercare la prima posizione libera dopo la 15, in questo caso la 16, e lì memorizzare `Prokofev`. Se poi arriva `Puccini`, un trattamento analogo porta a riservargli la posizione 17. Un successivo arrivo di `Rossini`, per il quale H(`Rossini`) = 17,

provoca la memorizzazione del nuovo venuto nella posizione 18 (si veda la Fig. 7.1). La situazione è qui aggravata dal fatto che `Rossini` ha trovato la sua posizione abusivamente occupata da `Puccini`, cioè da una chiave per la quale la funzione H dà un risultato diverso. Inoltre, `Rossini` stesso va ad occupare una posizione che non gli compete.

Se ora si vuole verificare l'appartenenza di una chiave già presente nel dizionario, occorre procedere attraverso le stesse posizioni per le quali si è passati durante la sua inserzione: così se si vuole ricercare `Puccini` occorrono ben tre accessi al vettore A, mentre per ricercare `Palestrina` ne basta uno. Questa non uniformità nel numero di accessi necessari a reperire una chiave è dovuta alla presenza di un agglomerato: benché A contenga solo 6 chiavi su 26 memorizzabili, ben 5 di queste si trovano in posizioni contigue e chiavi k diverse con indirizzo $H(k)$ compreso tra 14 e 18 devono procedere attraverso posizioni comuni per essere reperite. Ovviamente, dopo aver riempito le 26 posizioni disponibili, una eventuale 27esima chiave non potrà più essere inserita.

Sequenza di arrivo:	**Indirizzi hash:**
ALBINONI	H(ALBINONI) = 0
OFFENBACH	H(OFFENBACH) = 14
PALESTRINA	H(PALESTRINA) = 15
PROKOFEV	H(PROKOFEV) = 15
PUCCINI	H(PUCCINI) = 15
ROSSINI	H(ROSSINI) = 17

Memorizzazione

0	ALBINONI
	⋮
14	OFFENBACH
15	PALESTRINA
16	PROKOFEV
17	PUCCINI
18	ROSSINI
	⋮
25	

Fig. 7.1 Un (cattivo) esempio di tabella hash.

L'Esempio 7.3 è molto istruttivo e focalizza le problematiche che sorgono qualora si voglia realizzare un dizionario con una tabella hash.

Innanzitutto, la funzione H dell'Esempio 7.3 non è una buona funzione di accesso (anzi, è proprio da sconsigliare), sia perché l'indirizzo $H(k)$ dipende pesantemente da configurazioni di caratteri (in questo caso il primo) che appaiono nella chiave, sia perché non vi è alcuna garanzia che distribuisca uniformemente le chiavi nel vettore. Infatti, ognuna delle $|U|$ possibili chiavi dovrebbe avere la stessa probabilità $1/m$ che il suo indirizzo $H(k)$ sia una qualsiasi delle m posizioni del vettore, ed è ben noto che la distribuzione della lettera iniziale dei cognomi non è affatto uniforme.

In secondo luogo, per inserire chiavi che hanno trovato la loro posizione occupata, occorre assolutamente evitare il formarsi di agglomerati. Nell'Esempio 7.3, 5 delle 6 chiavi complessivamente presenti nel dizionario stanno in posizioni contigue. Ciò è dovuto alla inadeguatezza sia della funzione H, sia del metodo usato per esplorare il vettore (cioè tentare nella posizione successiva, poi in quella ancora successiva, e così via). Se le 6 chiavi fossero state inserite seguendo un criterio uniforme, essendo $\lfloor 26/6 \rfloor = 4$, ci si aspetterebbe in media di trovare nel vettore all'incirca una posizione occupata seguita da tre libere.

Purtroppo, la presenza di agglomerati tende a farli espandere ancora di più. Infatti, se una posizione vuota è preceduta da j occupate, allora la probabilità che tale posizione vuota sia la prossima ad essere riempita è $(j + 1)/m$, contro una probabilità di $1/m$ nel caso che la posizione precedente fosse anch'essa libera! In particolare, quando il vettore è quasi completamente occupato, diviene pressoché impossibile evitare il formarsi di agglomerati ed il tempo per inserimenti e ricerche di chiavi tende a crescere linearmente con la dimensione m del vettore. Inoltre, se il vettore si riempie completamente, non è ovviamente più possibile alcuna ulteriore inserzione.

Ricapitolando, per realizzare efficientemente un dizionario con una tabella hash:

(1) occorre una funzione H, detta appunto *funzione hash*, che sia calcolabile velocemente (in tempo $O(1)$) e che distribuisca le chiavi uniformemente nel vettore A, al fine di ridurre il numero di collisioni tra chiavi diverse che hanno lo stesso indirizzo hash;

(2) occorre un metodo di scansione, per reperire chiavi che hanno trovato la loro posizione occupata, che permetta di esplorare tutto il vettore A e non provochi la formazione di agglomerati di chiavi;

(3) la dimensione m del vettore A deve essere una sovrastima del numero di chiavi attese, onde evitare di riempire A completamente.

7.1 FUNZIONI HASH

Trovare una buona funzione hash non è semplice; essa deve ricavare l'output (un numero intero) che sia scorrelato dalla struttura dell'input (la chiave stessa). Da qui il termine "hash", che in inglese significa "tritare" o "polpetta": si sminuzza la chiave in pezzettini che sono rimescolati in modo da formare un intero che sia il più "casuale" possibile. Ovviamente l'algoritmo che calcola la funzione hash non è casuale, in quanto se si ricalcola più volte $H(k)$ per la stessa chiave k si ottiene sempre lo stesso valore, ma l'indirizzo hash si comporta da un punto di vista statistico come se fosse stato davvero prodotto con uno o più lanci casuali di una moneta.

Una buona funzione hash deve minimizzare il numero di collisioni presenti nel sistema; uno dei possibili criteri per definire questa proprietà è detto *uniformità semplice*: detta $P(k)$ la probabilità che una chiave k sia inserita nella tabella, sia

$$Q(i) = \sum_{k:H(k)=i} P(k)$$

la probabilità che una chiave qualsiasi finisca nella cella i. Una funzione H gode della proprietà di *uniformità semplice* se $\forall i \in \{0, ..., m-1\}$: $Q(i) = 1/m$.

Per definire funzioni hash, è conveniente considerare la rappresentazione binaria $bin(k)$ della chiave k. Se la chiave k non è numerica, $bin(k)$ è data dalla concatenazione della rappresentazione binaria di ciascun carattere che la compone. In tutti i linguaggi di programmazione esiste una funzione primitiva (chiamiamola *ord*) che applicata ad un ca-

rattere, restituisce un intero che è il numero ordinale del carattere nell'insieme dei caratteri ammessi. È altresì utile denotare con $int(b)$ il numero intero rappresentato da una stringa binaria b. In questo modo, oltre a poter agevolmente passare da una chiave alfanumerica k alla sua rappresentazione binaria $bin(k)$, si può passare da una qualsiasi stringa binaria b, inclusa $b = bin(k)$, ad un numero intero $int(b)$.

Quattro metodi di generazione di indirizzi hash sono i seguenti. I primi due presuppongono $m = 2^p$, per qualche p; il terzo lavora bene con tutti i valori di m; il quarto presuppone m dispari (meglio se primo):

(1) *Estrazione*: $H(k) = int(b)$, dove b è un sottoinsieme di p bit di $bin(k)$, solitamente estratti nelle posizioni centrali;

(2) *Xor*: $H(k) = int(b)$, dove b è dato dalla somma modulo 2, effettuata bit a bit, di diversi sottoinsiemi di p bit di $bin(k)$;

(3) *Moltiplicazione*: $H(k) = \lfloor m\,(iC - \lfloor iC \rfloor) \rfloor$, dove m è un numero qualsiasi (le potenze di 2 vanno bene, anzi sono consigliate), i è $int(bin(k))$ e C è un numero reale, $0 < C < 1$;

(4) *Divisione*: $H(k)$ è uguale al resto della divisione di $int(bin(k))$ per m.

7.4 ESEMPIO – [Funzioni hash]

Si supponga che le chiavi siano di 6 caratteri alfanumerici. Chiavi più lunghe sono troncate, mentre chiavi più corte sono espanse a destra con spazi. Si assuma $ord(A) = 1$, $ord(B) = 2$, ..., $ord(Z) = 26$ e $ord(\underline{b}) = 32$, dove $\underline{b}$ indica lo spazio, con rappresentazione di ogni ordinale su 6 bit. Per le chiavi `Weber`$\underline{b}$ e `Webern` si ottengono le seguenti rappresentazioni su 36 bit, dove si sono evidenziati per chiarezza i 6 gruppi di 6 bit che rappresentano i caratteri della chiave:

$bin(\texttt{Weber}\underline{b})$ = 010111 000101 000010 000101 010010 100000
$bin(\texttt{Webern})$ = 010111 000101 000010 000101 010010 001110

Calcoliamo gli indirizzi hash ottenuti con le funzioni definite sopra

(1) Sia $m = 256 = 2^8$. Estraendo gli 8 bit dalla posizione 15 alla 22 di $bin(k)$, $H(\texttt{Weber}) = H(\texttt{Webern}) = int(00100001) = 33$. Le due chiavi danno quindi luogo ad una collisione.

(2) Sia ancora $m = 256 = 2^8$ e si calcoli b raggruppando $bin(k)$ in cinque gruppi di 8 bit, dopo aver espanso a destra $bin(k)$ con quattro zeri in modo che $bin(k)$ sia formato da 40 bit. $H(\texttt{Weber}\underline{b}) = int(11000011) = 195$ mentre $H(\texttt{Webern}) = int(00100001) = 33$. Infatti, indicando con $\oplus$ la somma bit a bit modulo 2, si ottiene:

$$11000011 = 01011100 \oplus 01010000 \oplus 10000101 \oplus 01001010 \oplus 00000000$$
$$00100001 = 01011100 \oplus 01010000 \oplus 10000101 \oplus 01001000 \oplus 11100000$$

Benché $H(\texttt{Webern})$ sia uguale a 33, come nel caso (1), la collisione è stata eliminata.

(3) Poiché si usano 6 bit per rappresentare ciascun carattere della chiave, $int(bin(\texttt{Webern}))$ è interpretato come un numero in base $2^6 = 64$.

$$\begin{aligned} i &= int(bin(\texttt{Webern})) \\ &= 23 \cdot 64^5 + 5 \cdot 64^4 + 2 \cdot 64^3 + 5 \cdot 64^2 + 18 \cdot 64^1 + 14 \cdot 64^0 \\ &= 64(64(64(64(23 \cdot 64 + 5) + 2) + 5) + 18) + 14. \end{aligned}$$

Sia $C = (\sqrt{5} - 1)/2$ e $m = 256$. Allora

$$H(\texttt{Webern}) = \lfloor m\,(iC - \lfloor iC \rfloor) \rfloor = \lfloor 256 \cdot 0.9996833801\ldots \rfloor = 255.$$

Il calcolo di $H(\texttt{Weberb})$ è lasciato per esercizio.

(4) Sia $m = 383$. $H(\texttt{Webern})=242$ è ottenuto prendendo il resto della divisione per 383 di $int(bin(\texttt{Webern}))$. Il calcolo di $H(\texttt{Weberb})$ è lasciato per esercizio. □

La scelta in (1) di estrarre i p bit dalle posizioni centrali è dovuta al fatto che molte chiavi alfanumeriche potrebbero iniziare con la stessa sequenza di caratteri, così come molte chiavi potrebbero terminare con degli spazi perché espanse a destra in quanto inizialmente troppo brevi (come visto nell'Esempio 7.4).

Il metodo della moltiplicazione ha il vantaggio di essere indipendente dal valore di m; tuttavia la scelta del valore C è delicata. Il valore utilizzato nell'esempio è stato proposto da Knuth, e dovrebbe funzionare in gran parte dei casi. Un ulteriore svantaggio di questo metodo è il costo della moltiplicazione; esistono tuttavia realizzazione efficienti basate su semplici operazioni di shift e di moltiplicazione di interi.

Il metodo della divisione presuppone che m sia dispari, perché se m fosse uguale a 2^p allora due numeri con gli stessi p bit finali darebbero sempre luogo ad una collisione. Questa funzione è la migliore dal punto di vista probabilistico e fornisce un'eccellente distribuzione degli indirizzi $H(k)$ nell'intervallo $[0, m-1]$, specie se m è un numero primo non troppo vicino ad una potenza di 2 (o a qualsiasi altra base con la quale il calcolatore utilizzato rappresenti i numeri). Se il calcolatore rappresenta numeri usando la base 2, allora alcuni valori adeguati per m sono: 13, 23, 47, 97, 193, 383, 769, 1531, 3067, 6143, 12289, 24571.

Nel metodo della divisione è conveniente applicare la cosiddetta "regola di Horner" per la valutazione di un polinomio in un punto, come già anticipato implicitamente nell'Esempio 7.4. Infatti, un qualsiasi polinomio

$$p(x) = a_q x^q + a_{q-1} x^{q-1} + \cdots + a_1 x + a_0$$

può essere riscritto come

$$p(x) = x(\cdots(x((a_q x) + a_{q-1})\cdots) + a_1) + a_0$$

Per valutare p nel punto x, si procede secondo tale parentesizzazione, dalla parentesi più interna verso quella più esterna. In questo modo il polinomio di grado q è valutato in x utilizzando soltanto q prodotti e q addizioni. La seguente procedura calcola la funzione hash per una chiave di ℓ caratteri utilizzando la regola di Horner ed effettuando l'operazione mod ad ogni iterazione, per evitare che b divenga troppo grande e provochi "overflow" (superamento della capacità di una parola di memoria).

Ovviamente, per chiavi definite su alfabeti di dimensione diversa si userà una base diversa da 64. La complessità della funzione H è $O(1)$, ma la costante dipende dalla lunghezza ℓ della chiave. Per chiavi molto lunghe è opportuno usare una combinazione dei metodi (2) e (4), al fine di ridurre il tempo richiesto dal calcolo della funzione hash.

```
integer H(ITEM[] k, integer ℓ)
  integer b ← ord(k[1])
  for j ← 2 to ℓ do
    b ← ((b · 64) + ord(k[j])) mod m
  return b
```

7.2 METODI DI SCANSIONE

I metodi di scansione si distinguono in *esterni* e *interni*, a seconda che le chiavi siano memorizzate all'esterno o all'interno del vettore A. Il metodo esterno per eccellenza si basa sulle liste di trabocco, mentre quelli interni più comuni sono la scansione lineare, la scansione quadratica, la scansione pseudocasuale, e l'hashing doppio. Il metodo esterno è il più semplice e, come vedremo, anche il più efficiente. Il vettore A contiene, in ogni posizione, l'identificatore di una lista, detta appunto "lista di trabocco", e tutte le chiavi che collidono sullo stesso indirizzo hash vengono inserite nella stessa lista.

ESEMPIO – [Liste di trabocco]
La Fig. 7.2 illustra la memorizzazione delle chiavi dell'Esempio 7.3 utilizzando liste di trabocco (monodirezionali senza sentinella).

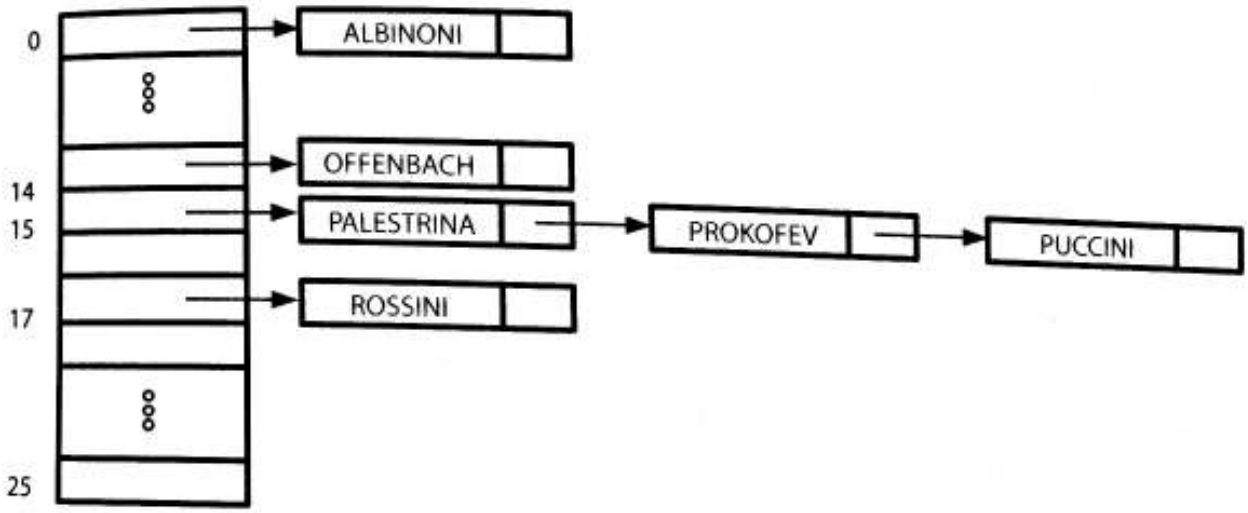


Fig. 7.2 Liste di trabocco.

Il metodo di scansione con liste di trabocco presenta il duplice vantaggio di non imporre alcun limite alla capacità del dizionario e di evitare completamente il formarsi di agglomerati. Le operazioni lookup(), insert() e remove() per una chiave k si riducono, rispettivamente, a procedure di ricerca, inserzione in testa e cancellazione di k nella lista $A[H(k)]$ e richiedono tempo lineare nella lunghezza della lista, purché sia scelta una realizzazione efficiente per le liste. Per esempio, una realizzazione adeguata si può basare su un vettore di puntatori a liste bidirezionali. Ovviamente la dimensione m di A, cioè il numero di liste, va scelta in modo oculato, così da evitare sia troppe liste, la

maggior parte delle quali sarebbero vuote, sia poche liste, che tenderebbero ad essere troppo lunghe.

Se non è possibile l'utilizzo di puntatori, allora si ricorre ad un metodo di scansione interna. In questo caso non ci sono liste esterne ad A che vengono scandite seguendo puntatori, ma piuttosto si calcola esplicitamente la sequenza di scansione all'interno di A (come si è visto nell'Esempio 7.3).

La funzione hash viene quindi estesa con un secondo parametro, un indice i compreso fra 0 e $m - 1$:

$$H : U \times \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$$

$H(k, i)$ rappresenta la posizione della chiave k dopo i ispezioni "fallite" (durante un inserimento, dopo i volte che si è trovata una casella occupata; durante una ricerca, dopo i volte che si è trovata una casella occupata da un valore diverso da k). La funzione $H(k, i)$ va scelta in modo che si tocchino tutte le m posizioni del vettore A, una ed una sola volta. In altre parole, $\{H(k, 0), ..., H(k, m-1)\}$ deve essere una permutazione degli indici $\{0, ..., m-1\}$. Così, se il vettore A è completamente occupato e la chiave k da ricercare non vi è contenuta, la ricerca si può arrestare non appena verrà toccata per la seconda volta la posizione di partenza $H(k)$.

SCANSIONE LINEARE: $H(k, i) = (H(k) + h \cdot i) \bmod m$

In questo metodo, si fa uso di un parametro h primo con m, che rappresenta la distanza tra due successive posizioni esaminate nella scansione (se $h = 1$, si ottiene la scansione a passo unitario vista nell'Esempio 7.3). La divisione in modulo rende la scansione "circolare", così da continuare dall'inizio di A se si tenta di uscire oltre la fine di A. L'ipotesi di primalità tra m e h garantisce che la scansione tocchi tutte le posizioni di A prima di riconsiderare posizioni già esaminate. Questo metodo di scansione ha lo svantaggio di non ridurre la possibilità di formazione di agglomerati di chiavi. Infatti, se due chiavi k e k' hanno indirizzi hash $H(k') = H(k) + h$, le due sequenze di scansione $H(k)$, $H(k) + h$, $H(k) + 2h$, $H(k) + 3h$, ... e $H(k')$, $H(k') + h$, $H(k') + 2h$, ... collidono a partire dalla posizione $H(k) + h$. In questo caso, il vantaggio di avere due indirizzi hash distinti per k e k' è di fatto annullato. Questo tipo di problema prende il nome di *agglomerazione primaria* (*primary clustering*).

SCANSIONE QUADRATICA: $H(k, i) = (H(k) + h \cdot i^2) \bmod m$

In questo metodo, m è un numero primo; l'elevazione al quadrato di i riduce l'effetto di agglomerazione dovuto alla scansione lineare poiché la distanza tra due successive posizioni nella sequenza di scansione non è più costante ma variabile. Due chiavi diverse k e k' con indirizzi hash iniziali distinti ($H(k) \neq H(k')$) non collidono più lungo tutta la sequenza di ispezione; tuttavia, il problema rimane per due chiavi con indirizzi hash iniziali uguali ($H(k) = H(k')$). Questo problema prende il nome di *agglomerazione secondaria*. Un problema a parte è che la sequenza così generata non è una permutazione di $\{0, ..., m-1\}$. Può non essere dunque possibile l'inserimento di k perché tutte le po-

sizioni considerate nella sequenza di scansione sono occupate, mentre possono esistere altre posizioni libere non considerate nella sequenza. Questo svantaggio, comunque, è talvolta trascurabile, specie per m non troppo piccolo.

SCANSIONE PSEUDOCASUALE: $H(k, i) = (H(k) + r_i) \bmod m$

In questo metodo, $r_0 = 0$ e r_i è l'i-esimo numero generato da un generatore di numeri pseudo-casuali [capitolo 17], tale per cui $\{r_0, ..., r_{m-1}\}$ è una permutazione di $\{0, ..., m-1\}$. Questo risolve il problema della mancanza di copertura di tutte le posizioni di A, ma resta il problema dell'agglomerazione secondaria.

HASHING DOPPIO: $H(k, i) = (H(k) + i \cdot H'(k)) \bmod m$

In questo metodo, si utilizza un'ulteriore funzione hash H' diversa da H. In questo modo, anche se per due chiavi k e k' si ha $H(k) = H(k')$, vi è un'alta probabilità che le due ampiezze del passo di scansione $H'(k)$ ed $H'(k')$ siano diverse. Ciò evita, in pratica, la formazione di agglomerati, sia primari che secondari. Affinché la sequenza di scansione possa includere tutte le posizioni, occorre che il passo di scansione $H'(k)$ sia primo rispetto ad m. Questa condizione può essere soddisfatta imponendo che H' generi solo numeri dispari, qualora m sia una potenza di 2, oppure che H' generi interi minori di m, qualora m sia un numero primo.

Usando i metodi di scansione interna e permettendo la cancellazione di chiavi, però, sorge un nuovo problema. Una volta raggiunta una posizione "vuota" nella ricerca di una chiave k, non si è mai sicuri che la chiave stessa non si trovi da qualche altra parte del vettore, perché la posizione attualmente vuota poteva essere occupata quando k è stata inserita.

7.6 ESEMPIO – [Cancellazione]

Cancellando `Prokofev` dalla tabella della Fig. 7.1 ed utilizzando la scansione lineare a passo unitario, la successiva ricerca di `Puccini` fallisce. Infatti tale ricerca si arresta sulla prima posizione vuota dopo la posizione 15, cioè sulla 16, mentre `Puccini` è memorizzato nella posizione 17.

Con i metodi di scansione interna, una posizione in cui è avvenuta una cancellazione, pur essendo disponibile per una eventuale inserzione, non può essere considerata nella scansione come una posizione mai riempita! Occorre pertanto scandire anche le posizioni in cui si è cancellato ed arrestarsi o sulla prima posizione mai riempita o dopo essere ritornati su una posizione già scandita. Poiché ciò comporta un aumento nel tempo di ricerca, se sono previste molte cancellazioni è più conveniente utilizzare il metodo di scansione esterna.

Siamo quindi pronti per presentare il codice di esempio di una tabella hash basata su hashing doppio. La tabella è memorizzata in due vettori A e V di dimensione m, contenenti generici oggetti di tipo ITEM, che contengono rispettivamente l'insieme di chiavi e l'insieme di valori ad essi associati. Assumiamo l'esistenza di due valori speciali, **nil** e **deleted**, che indicano rispettivamente una posizione vuota e una posizione cancellata.

HASH

```
ITEM[] A                                   % Tabella delle chiavi
ITEM[] V                                   % Tabella dei valori
integer m                                  % Dimensione della tabella

HASH Hash(integer capacità)
    HASH t ← new HASH
    t.m ← capacità
    t.A ← new ITEM[0 ... t.m − 1]
    t.V ← new ITEM[0 ... t.m − 1]
    for i ← 0 to t.m − 1 do t.A[i] ← nil
    return t

integer scan(ITEM k, boolean insert)
    integer c ← m                          % Prima posizione deleted
    integer i ← 0                          % Numero di ispezione
    integer j ← H(k)                       % Posizione attuale
    while A[j] ≠ k and A[j] ≠ nil and i < m do
        if A[j] = deleted and c = m then c ← j
        j ← (j + H'(k)) mod m
        i ← i + 1
    if insert and A[j] ≠ k and c < m then j ← c
    return j

ITEM lookup(ITEM k)
    integer i ← scan(k, false)
    if A[i] = k then
        return V[i]
    else
        return nil

insert(ITEM k, ITEM v)
    integer i ← scan(k, true)
    if A[i] = nil or A[i] = deleted or A[i] = k then
        A[i] ← k
        V[i] ← v
    else
        % Errore: tabella hash piena

remove(ITEM k)
    integer i ← scan(k, false)
    if A[i] = k then
        A[i] ← deleted
```

Le operazioni del dizionario si basano sulla funzione scan(k, *insert*), che scandisce il vettore A alla ricerca della chiave k e si arresta quando una di queste condizioni è vera: (i) è stata trovata una casella contenente la chiave k; (ii) è stata trovata una casella vuota; (iii) sono state scandite tutte le caselle. L'ultima condizione viene verificata contando il numero di ispezioni tramite la variabile i e controllando che sia inferiore a m. Nel caso scan() sia chiamata da lookup() e remove() (*insert* = **false**), viene restituito l'indice in cui la scansione si è arrestata; nel caso la funzione sia chiamata da insert() (*insert* = **true**), vi sono varie possibilità. Se la chiave è stata trovata, si restituisce l'indice corrispondente. Altrimenti, se durante la ricerca sono state trovate una o più posizioni cancellate, si restituisce l'indice della prima di tali posizioni. Altrimenti, si restituisce l'indice in cui la ricerca è terminata (o perché si è trovata una posizione **nil**, o perché tutte le posizioni sono state ispezionate). L'informazione sulla prima posizione cancellata è contenuta nella variabile c, che viene inizializzata ad m per indicare che non è ancora stata trovata una posizione cancellata; al primo incontro di **deleted**, e solo al primo incontro, la variabile c viene modificata opportunamente.

lookup() chiama quindi scan() e restituisce il valore associato alla chiave, se essa è presente, **nil** altrimenti. remove() lavora allo stesso modo, cancellando il valore trovato, se presente. Anche insert() chiama scan(); se viene restituito l'indice di una casella vuota, cancellata, o contenente k, la chiave e il valore vengono memorizzati nei vettori corrispondenti; altrimenti, la casella identificata non contiene k, la tabella è piena e viene segnalato un errore.

7.3 COMPLESSITÀ MEDIA

Benché nel caso pessimo la ricerca di un elemento possa richiedere la scansione di tutta la tabella, nel caso medio tale ricerca è drasticamente più veloce. È stato infatti dimostrato che il tempo medio di ricerca dipende soltanto dal cosiddetto *fattore di carico* della tabella, cioè dal rapporto tra il numero di chiavi memorizzate nella tabella e la dimensione della tabella stessa. In altri termini, tabelle di dimensioni diverse presentano lo stesso tempo di ricerca qualora la percentuale di posizioni occupate sia la stessa. Siano:

n = numero di chiavi memorizzate nella tabella hash
m = dimensione della tabella hash
α = n/m (fattore di carico)
$I(\alpha)$ = numero medio di accessi alla tabella per la ricerca di una chiave non presente nella tabella (*ricerca con insuccesso*)
$S(\alpha)$ = numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (*ricerca con successo*)

Le grandezze $I(\alpha)$ ed $S(\alpha)$ possono adeguatamente rappresentare la complessità media delle operazioni lookup(), insert() e remove(). Infatti, per inserire un elemento non presente nella tabella occorre innanzitutto una ricerca con insuccesso, seguita dall'inserimento vero e proprio, mentre per cancellare un elemento presente nella tabella occorre

una ricerca con successo, seguita dalla cancellazione vera e propria. Pertanto, i tempi medi di inserzione e cancellazione dipendono, a meno di costanti, da $I(\alpha)$ ed $S(\alpha)$.

Le grandezze $I(\alpha)$ ed $S(\alpha)$ sono state valutate matematicamente, assumendo una distribuzione uniforme delle chiavi. Le funzioni ottenute al variare di α per i vari metodi di scansione sono, con buona approssimazione, le seguenti:

SCANSIONE	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \le \alpha < 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\alpha/2}{1-\alpha}$
Hashing doppio	$0 \le \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Liste di trabocco	$\alpha \ge 0$	$1+\alpha$	$1+\alpha/2$

Il risultato per la scansione lineare è alquanto difficile da dimostrare e non può essere qui riportato. In contrasto, il risultato per le liste di trabocco si dimostra facilmente nel modo seguente. Il fattore di carico α rappresenta anche la lunghezza media di ciascuna lista di trabocco. In caso di successo, mediamente viene scandita metà lista, mentre in caso di insuccesso viene scandita un'intera lista. In entrambi i casi, è necessario un accesso al vettore A che contiene il puntatore alla lista.

Per l'hashing doppio (e indirettamente, anche la scansione quadratica e l'hashing pseudocasuale), la difficoltà principale consiste nel dimostrare che non si creano agglomerati. Per semplicità, si assuma che ciò sia stato dimostrato e si consideri una ricerca con insuccesso. Se si denota con p_i la probabilità di avere esattamente i collisioni, il costo di una ricerca con insuccesso è

$$1+\sum_{i=1}^{\infty} i \cdot p_i$$

dove il termine 1 è il costo per l'accesso all'unica posizione vuota di A che non dà collisione, e $p_i = 0$ per $i > n$, essendoci n chiavi memorizzate. Per valutare la serie, si usa la relazione

$$\sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} q_i$$

dove q_i denota la probabilità di avere almeno i collisioni. Per ricavare le probabilità q_i, si osservi che $q_1 = n/m = \alpha$, perché si ha almeno una collisione accedendo ad una delle n posizioni occupate sulle m totali. Invece, $q_2 = (n/m)((n-1)/(m-1))$ perché, dopo aver trovato la prima posizione occupata, si ha almeno un'altra collisione accedendo ad una delle rimanenti $n-1$ posizioni occupate tra le $m-1$ posizioni non ancora accedute. Iterando il procedimento, si ricava

$$q_i = (n/m)((n-1)/(m-1)) \cdots ((n-i+1)/(m-i+1))$$

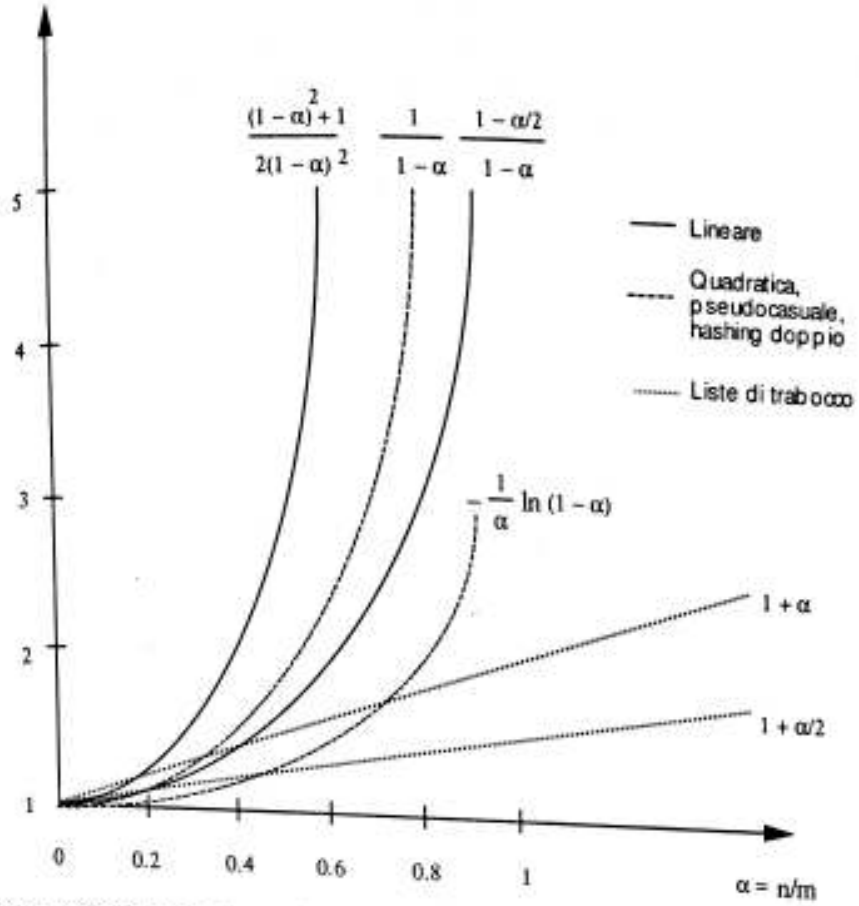


Fig. 7.3 Numero medio di accessi per ricerche in tabelle hash.

Per m ed n grandi, quest'ultima quantità si può maggiorare con $(n/m)^i = \alpha^i$. Pertanto il costo di una ricerca con insuccesso è, per $0 \leq \alpha < 1$:

$$I(\alpha) \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

D'altro canto, il costo di una ricerca con successo è pari al costo di una ricerca con insuccesso quando la chiave è stata inserita, poiché in entrambi i casi viene effettuata la stessa sequenza di scansione. Al momento dell'inserzione, però il fattore di carico era diverso da quello che si ha al momento della ricerca con successo.

Sia β il fattore di carico all'atto dell'inserzione ed α quello all'atto della ricerca con successo. Facendo la media sui possibili valori di β, che sono compresi tra 0 ed α, si ottiene:

$$S(\alpha) = \frac{1}{\alpha}\int_0^{\alpha} \frac{1}{1-\beta}\,d\beta = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} = -\frac{1}{\alpha}\ln(1-\alpha)$$

Un accurato confronto delle funzioni riportate nella Fig. 7.3 mostra la netta superiorità del metodo delle liste di trabocco, per $\alpha < 1$. Per $\alpha \geq 1$, questo metodo non è confrontabile con gli altri, che sono utilizzabili soltanto per $\alpha < 1$.

Tra i metodi di scansione interna, quello lineare è il più lento, specie per valori di α vicini ad 1. Per valori contenuti di α, anche la scansione lineare può talvolta risultare

conveniente (p.e. imponendo $\alpha \leq 0.5$, allora $I(\alpha) \leq 4$ e $S(\alpha) \leq 2$). È chiaro, comunque, che anche per gli altri metodi interni conviene evitare di spingere a livelli elevati la percentuale di occupazione della tabella. Infatti il tempo di ricerca cresce rapidamente all'aumentare di α. Inoltre, le cancellazioni di chiavi non fanno diminuire il tempo di ricerca, poiché nella scansione occorre passare anche sulle posizioni che contengono la marca **deleted**. In ogni caso, qualora siano previste molte cancellazioni, è conveniente utilizzare le liste di trabocco.

Comunque, vi è un caso in cui la scansione lineare a passo unitario risulta più vantaggiosa degli altri metodi. Ciò accade quando la tabella non risieda in memoria principale, ma su dispositivi di memoria secondaria, siano essi ad accesso sequenziale, o anche ad accesso casuale ma con chiavi raggruppate in pagine. In quest'ultimo caso, poiché il trasferimento di una pagina dalla memoria secondaria a quella principale è lento, la scansione lineare risulta conveniente. Infatti permette di effettuare la scansione in memoria principale all'interno della stessa pagina, riducendo così il numero di trasferimenti di pagine dalla memoria secondaria a quella principale.

7.4 RISTRUTTURAZIONE

Usando la scansione interna, non è conveniente che la tabella si riempia troppo. In pratica, quando a causa di un'inserzione α diviene maggiore di una prefissata soglia (di solito 0.5), è vantaggioso ristrutturare la tabella utilizzando le tecniche viste in precedenza.

La tabella viene ridimensionata con m raddoppiato e sono reinserite nella nuova tabella tutte le chiavi presenti in quella vecchia (cambiando funzione hash, ovviamente, visto che m è raddoppiato). In questo modo, la nuova tabella ha fattore di carico al più dimezzato (cioè non superiore a 0.25) e non ci sono più posizioni "cancellate" (cioè sono tutte o occupate o mai usate).

Chiaramente il costo di tale ristrutturazione è $O(m)$ nel caso pessimo, ma è possibile dimostrare che il costo "ammortizzato" delle operazioni lookup(), insert() e remove() resta $O(1)$. Considerando per semplicità solo le inserzioni e trascurando le cancellazioni, è possibile vedere che il costo totale per eseguire n inserimenti è limitato superiormente da $3n$, e quindi il costo ammortizzato è costante. La dimostrazione è identica a quella vista nel capitolo 4.2.

Il risultato precedente si può estendere al caso in cui la sequenza di operazioni includa sia insert() sia remove() e si usi ristrutturare la tabella sia espandendola quando è troppo piena (come visto precedentemente), sia contraendola quando è quasi vuota (ridimensionandola con m dimezzato quando α scende sotto una certa soglia, per esempio 0.125).

Analoghe ristrutturazioni in espansione o contrazione sono convenienti anche per le liste di trabocco, nel qual caso è opportuno espandere se α supera 2.

7.5 REALITY CHECK

Realizzazioni efficienti di tabelle hash si trovano nelle librerie standard di tutti i più diffusi linguaggi di programmazione, come la `HashMap` di Java Collections o la `hash_map` della Standard Template Library di C++.

Una interessante estensione del concetto di *tabella hash* (*hash table* in inglese) è quello di *distributed hash table*, o DHT. Queste strutture di dati distribuite sono utilizzate nei sistemi peer-to-peer per realizzare le funzionalità di un dizionario in maniera decentralizzata. In questo caso, le "posizioni" della tabella hash non sono contenute in una singola macchina, ma sono distribuite fra le macchine partecipanti. Il problema di identificare la macchina che contiene una particolare chiave diventa un problema di ricerca in un sistema distribuito.

2.4 Si indichi il contenuto di una tabella hash di dimensione 11, inizialmente vuota, dopo l'inserzione, nell'ordine, delle chiavi: C, A, L, C, I, O, P, U, L, I, T, O. Si usi la funzione hash $H(k) = k \bmod 11$ per la k-esima lettera dell'alfabeto italiano e il metodo di scansione lineare a passo unitario. Si indichi successivamente il contenuto della tabella dopo avervi cancellato, nell'ordine: G, A, L, L, I, A, N, I, e poi inserito, nell'ordine: R, I, V, E, R, A.

2.5 Si consideri una tabella hash di dimensione $m = 11$ inizialmente vuota. Si mostri il contenuto della tabella dopo aver inserito nell'ordine, i valori 33, 10, 24, 14, 16, 13, 23, 31, 18, 11, 7. Si assuma che le collisioni siano gestite mediante indirizzamento aperto utilizzando come funzione di hash $h(k)$, definita nel modo seguente:

$$h(k) = (h'(k) + 3i + i^2) \bmod m$$
$$h'(k) = k \bmod m$$

Mostrare il contenuto della tabella al termine degli inserimenti e calcolare il numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella.

2.6 Si definisca la realizzazione di un dizionario contenente in media 256 elementi, in cui la chiave richiede 8 bit, in modo da minimizzare il tempo di accesso.

2.7 Si forniscano procedure per realizzare i metodi di scansione lineare e quadratico.

2.8 Un dizionario è realizzato mediante una tabella hash con liste di trabocco. Alcune chiavi sono ricercate molto più frequentemente delle altre, ma non si sa a priori quali siano tali chiavi. Quale accorgimento si può adottare per sperare di abbassare il tempo di accesso a queste chiavi?

2.9 Supponete di avere un albero di directory, contenente un numero n di file, alcuni dei quali potrebbero essere replicati una o più volte, sotto nomi e percorsi diversi. Il vostro compito è elencare tutti i file replicati (ovvero, con lo stesso contenuto e la stessa lunghezza). Proponente un metodo efficiente ($O(n)$) per risolvere questo problema.

2.10 Supponete di cercare una chiave k^* in una lista concatenata di lunghezza n, dove ogni elemento contiene una chiave k ed un valore hash $h(k)$. Ogni chiave è una lunga stringa di caratteri. Come potreste trarre vantaggio dai valori hash quando ricercate nella lista un elemento con una data chiave? Proporre una procedura che effettui la ricerca di una chiave in maniera efficiente.

7.8 Utilizzando il metodo della divisione, si suggerisce di evitare valori vicini a potenze di 2. Si consideri una stringa di caratteri scelti da un alfabeto di dimensione 2^p, e si consideri una dimensione m per la divisione pari a $2^p - 1$. Il motivo è che, con questo alfabeto, due stringhe ottenute una da una permutazione dell'altra danno origine ad una collisione. Dimostrare come questo avvenga considerando lo scambio di due caratteri in una stringa; dallo scambio è poi possibile generalizzare al concetto di permutazione.

7.9 Progettare un algoritmo efficiente per identificare ed eliminare tutti gli elementi duplicati di un vettore non ordinato.

7.10 Supponiamo di mantenere ordinate le liste di trabocco. Quali sono i vantaggi e/o svantaggi di tale scelta in termini di efficienza?

7.1

0			
1	A	-	A
2	O	O	O
3	C	C	C
4	P	P	P
5			R
6			E
7	T	T	T
8	U	U	U
9	I	-	I
10	L	-	V

7.2 Si osservi che con 8 bit si possono rappresentare in base 2 tutti i numeri compresi tra 0 e $2^8 - 1 = 255$. Ma allora ci si aspetta che nel dizionario siano presenti al massimo 256 chiavi. Utilizziamo quindi un vettore $A[0 \ldots 255]$ ad accesso diretto, dove la generica chiave k viene memorizzata direttamente in $A[int(bin(k))]$.

7.3 Quando si accede ad una chiave, si può scambiarla di posto con quella che la precede nella lista di trabocco. In questo modo, le chiavi ricercate più di frequente tendono a portarsi verso la testa delle liste di trabocco, richiedendo un minor tempo di accesso.

7.4 Il valore hash può essere utilizzato per evitare di dover confrontare tutte le chiavi con k^*, un'operazione pesante per via della lunghezza delle stringhe. Si confronta invece il valore hash di k^* con il valore hash di ogni chiave k incontrata; se $h(k) \neq h(k^*)$, k e k^* sono sicuramente diverse. Altrimenti, k è uguale a k^* oppure siamo in presenza di una collisione; è necessario quindi confrontare le chiavi.

7.5 Si considerino le stringhe `ab` e `ba`; mostriamo che assegnando alle variabili a e b il valore numerico dei due caratteri, si dà origine ad una collisione.

$$
\begin{aligned}
& (2^p a + b) \bmod 2^p - 1 \\
= & ((2^p - 1)a + a + b) \bmod 2^p - 1 \\
= & (a + b) \bmod 2^p - 1 \\
= & ((2^p - 1)b + b + a) \bmod 2^p - 1 \\
= & (2^p b + a) \bmod 2^p - 1
\end{aligned}
$$

INSIEMI E DIZIONARI

Capitolo 8

I tipi di dato insieme e dizionario possono essere realizzati con tutte le strutture di dati viste finora: liste e vettori, alberi bilanciati e tabelle hash. Ognuna di queste strutture ha vantaggi e svantaggi, per quanto riguarda l'occupazione di memoria e l'efficienza delle operazioni. Compito del progettista di algoritmi è sapersi orientare fra le varie opzioni e scegliere quella più appropriata. La tabella seguente sintetizza le opzioni disponibili e i costi di esecuzione di alcune delle operazioni previste. I dettagli delle realizzazioni degli insiemi sono descritti nelle sezioni seguenti; la realizzazione dei dizionari è simile.

	contains lookup	insert	remove	min	SCANSIONE	ORDINE
Vettore booleano	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$	Sì
Lista non ordinata	$O(n)$	$O(1)^{\dagger}$	$O(n)$	$O(n)$	$O(n)$	No
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sì
Hash (caso medio) memorizzazione interna	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	No
Hash (caso medio) memorizzazione esterna	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$	$O(m+n)$	No
Alberi bilanciati di ricerca	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Sì

† Sapendo a priori che l'elemento da inserire non è già presente, altrimenti la complessità è $O(n)$

8.1 INSIEMI REALIZZATI CON VETTORI BOOLEANI

Si supponga di dover elaborare insiemi i cui elementi possono essere interi compresi tra 1 ed N. Un insieme A può essere rappresentato con un vettore booleano di N bit, in cui il k-esimo

SET (vettore booleano)

```
boolean[] V
integer dim
integer capacità

SET Set(integer N)
    SET t ← new SET
    t.dim ← 0
    t.capacità ← N
    t.V ← new boolean[1 ... N]
    for i ← 1 to N do V[i] ← false
    return t

integer size()
    return dim

boolean contains(integer x)
    return V[x]

insert(integer x)
    if not V[x] then
        dim ← dim + 1
    V[x] ← true

remove(integer x)
    if V[x] then
        dim ← dim − 1
    V[x] ← false

SET union(SET A, SET B)
    SET C ← Set(max(A.capacità, B.capacità))
    for i ← 1 to A.capacità do
        if A.contains(i) then C.insert(i)
    for i ← 1 to B.capacità do
        if B.contains(i) then C.insert(i)

SET intersection(SET A, SET B)
    SET C ← Set(min(A.capacità, B.capacità))
    for i ← 1 to min(A.capacità, B.capacità)
        if A.contains(i) and B.contains(i) then C.insert(i)

SET difference(SET A, SET B)
    SET C ← Set(A.capacità)
    for i ← 1 to A.capacità
        if A.contains(i) and (i > B.capacità or not B.contains(i)) then
            C.insert(i)
```

bit è vero se e solo se $k \in A$. In questo modo, è estremamente semplice verificare se un elemento k appartiene all'insieme, oppure inserire o cancellare un elemento k: basta accedere direttamente al k-esimo bit del vettore! Anche le operazioni di unione, intersezione e differenza di due insiemi si realizzano facilmente con operazioni logiche. Infatti, per l'unione basta porre a vero tutti i bit che sono veri in almeno uno dei due vettori (tramite **or**), per l'intersezione tutti quelli veri in entrambi i vettori (tramite **and**) e per la differenza tutti quelli veri nel primo vettore e falsi nel secondo.

La complessità di isEmpty(), contains(), insert() e remove() è $O(1)$, mentre quella di Set(), union(), Intersection() e difference() è $\Theta(N)$, perché occorre scandire l'intero vettore.

Questa realizzazione, pur avendo il pregio di essere estremamente semplice, presenta molti svantaggi. Innanzitutto vi è un notevole spreco di memoria, poiché ogni insieme richiede sempre $\Theta(N)$ spazio per la sua memorizzazione, indipendentemente dal numero di elementi effettivamente presenti. Inoltre, alcune operazioni sono troppo lente, perché richiedono $O(N)$ tempo, anche per insiemi con pochi elementi, inclusa l'operazione di scorrere tutti gli elementi nel costrutto **foreach**. Infine, la cardinalità dell'insieme non può superare la costante N.

8.2 INSIEMI REALIZZATI CON LISTE NON ORDINATE

Si può rappresentare un insieme con una lista i cui elementi sono quelli dell'insieme. In questo modo, l'insieme non deve necessariamente essere composto da interi compresi tra 1 ed N. Inoltre, l'occupazione di memoria dell'insieme, anziché essere sempre $\Theta(N)$, cresce linearmente col numero degli elementi effettivamente presenti nell'insieme.

Le operazioni su insiemi si riducono così a procedure che scandiscono liste, inserendovi o cancellandovi elementi. Per esempio, la procedura contains() è stata implicitamente già descritta nel capitolo 4.1 utilizzando le operazioni delle liste. La procedura remove() è analoga alla contains() dove, dopo aver localizzato l'elemento scandendo la lista, si procede alla sua eliminazione. La procedura insert() è anch'essa semplice: dopo aver verificato con la contains() che l'elemento non sia già presente, si può inserirlo direttamente in testa alla lista come fosse una pila. La complessità di Set() e isEmpty() è $O(1)$, mentre quella di contains(), insert() e remove() è $O(n)$, dove $n = |A|$, cioè il numero di elementi presenti nell'insieme. La complessità di insert() è dovuta alla verifica preliminare che l'elemento da inserire non sia già presente nell'insieme. Qualora la non appartenenza all'insieme sia nota a priori, la complessità della insert() può essere abbassata ad $O(1)$.

Le funzioni che realizzano union(), intersection() e difference() sono abbastanza simili tra loro. Occorrono infatti due variabili per scandire le liste che rappresentano l'insieme A in input e l'insieme C che viene restituito dalla funzione. Per union(), si inseriscono dapprima tutti gli elementi di B in C, mentre per intersection() e difference() si parte con C vuoto. Successivamente, per ogni elemento di A, si verifica tramite la contains() se tale elemento compare anche in B; nel caso di Intersection(), lo si inserisce se compare; nel caso di union() e difference(), lo si inserisce se non compare. Per esempio, qui di seguito è riportata la procedura per difference().

```
SET difference(SET A, SET B)
  SET C ← Set()
  POS p ← A.head()
  POS r ← C.head()
  while not A.finished(p) do
      if not B.contains(A.read(p)) then
          C.insert(A.read(p), r)
      p ← A.next(p)
  return C
```

Poiché contains() richiede una scansione di B ed è annidata con la scansione più esterna di A, le procedure union(), intersection() e difference() richiedono tempo $O(nm)$, dove $n = |A|$ ed $m = |B|$, purché le liste siano realizzate con puntatori.

Infine, il costo del costrutto **foreach** $s \in A$ è $O(n)$, e ovviamente l'insieme viene analizzato in maniera non ordinata.

8.3
INSIEMI REALIZZATI CON LISTE ORDINATE

Se sugli elementi dell'insieme è definita una relazione "$\leq$" di ordinamento totale, l'insieme può essere rappresentato con una lista ordinata per valori crescenti degli elementi. In questo modo, la complessità delle operazioni union(), intersection() e difference() può essere abbassata a $O(n + m)$. La complessità di insert(), invece, non può scendere sotto $O(n)$, perché quando si inserisce un elemento bisogna rispettare l'ordinamento. Il tempo richiesto da contains() e remove() resta ovviamente $O(n)$.

A titolo di esempio, l'operazione intersection() può essere scritta nel modo seguente. Si usano questa volta tre variabili di tipo posizione per scandire A, B e C, ma grazie all'ordinamento si può evitare di utilizzare contains(). Infatti, si considerano il primo elemento di A ed il primo elemento di B. Se sono uguali, si inserisce l'elemento in C e ci si posiziona sugli elementi successivi sia in A che in B. Se invece i due elementi sono diversi, ci si posiziona sull'elemento successivo solo nella lista del minore, senza avanzare nell'altra lista. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Si noti che, per mantenere anche C ordinata, l'inserzione in C va sempre effettuata in coda.

Nella procedura Intersection(), sia ciascun elemento di A sia ciascun elemento di B è considerato al più una volta. La complessità di intersection() è pertanto $O(n + m)$, purché le liste siano realizzate con puntatori. Per quanto mostrato nell'Esempio 2.10, la fusione di sequenze ordinate richiede $\Omega(n + m)$ tempo. La complessità di union(), intersection() e difference() è pertanto ottima, relativamente alla realizzazione scelta.

Come nella realizzazione con liste non ordinate, il costrutto **foreach** può essere realizzato in tempo ottimale $O(n)$, ma ovviamente questa volta gli elementi vengono scanditi in maniera ordinata.

Un ulteriore vantaggio della realizzazione con liste ordinate è quello di permettere di realizzare con complessità $O(1)$ l'operazione min() per la ricerca del minimo.

Realizzando le liste ordinate in modo da accedere direttamente anche all'ultimo elemento (come nel caso bidirezionale circolare) anche l'analoga operazione max() di ricerca del massimo può essere realizzata con la stessa complessità. Si noti che, realizzando l'insieme A con un vettore booleano o con una lista non ordinata, le operazioni min() e max() richiedono nel caso pessimo una scansione completa del vettore o della lista e la complessità risultante è, rispettivamente, $O(N)$ e $O(n)$.

```
SET intersection(SET A, SET B)
  SET C ← Set()
  POS p ← A.head()
  POS q ← B.head()
  POS r ← C.head()
  while not A.finished(p) and not B.finished(q) do
      if A.read(p) = B.read(q) then
          C.insert(A.read(p), r)
          p ← A.next(p)
          q ← B.next(q)
      else if A.read(p) < B.read(q) then
          p ← A.next(p)
      else
          q ← B.next(q)
  return C
```

8.4 REALIZZAZIONE CON TABELLE HASH

Abbiamo già visto che le tabelle hash possono essere utilmente impiegate per realizzare dizionari. Qualora l'obiettivo sia ottenere realizzazioni efficienti delle operazioni di base, e non sia necessario implementare le operazioni insiemistiche di unione, intersezione, e differenza, le tabelle hash sono un'opzione efficiente anche per la realizzazione di insiemi. A differenza dei dizionari, non vi è un'associazione chiave-valore, ma semplicemente un elenco di chiavi. Assumendo che la funzione hash disperda uniformemente le chiavi nella tabella, il costo atteso per le operazioni di insert(), remove() e contains() diventa quindi $O(1)$ ed è pertanto ottimo.

Il costrutto **foreach** ha costo $O(m)$ nel caso di memorizzazione interna (dove m è il numero di caselle nella tabella), dato che bisogna passare per tutte le caselle, anche vuote, della tabella hash; il costo $O(m + n)$ è in caso di memorizzazione esterna, perché si devono scandire tutte le m liste di trabocco contenenti complessivamente n chiavi. La scansione non è ovviamente ordinata.

8.5 REALIZZAZIONE CON ALBERI BILANCIATI

Infine, se il costrutto **foreach** viene utilizzato spesso, l'ordine di scansione è importante, e si vuole comunque un'implementazione più efficiente delle operazioni di base rispetto alle liste ordinate, è possibile utilizzare alberi di ricerca bilanciati, come per esempio gli alberi Red-Black. In questo caso, le operazioni insert(), remove() e contains() hanno costo $O(\log n)$, mentre il costrutto **foreach** ha costo $O(n)$.

8.6 REALITY CHECK

Gli insiemi e i dizionari sono fra le principali strutture di dati contenute nelle Java Collections. Le interfacce `Set` e `Map` contengono le operazioni di base viste in questo capitolo, con poche differenze: negli insiemi insert() è chiamato `add()`; nei dizionari insert() è chiamato `put()` e lookup() è chiamato `get()`. Il metodo Java `isEmpty()` corrisponde allo pseudocodice size() = 0. Le operazioni insiemistiche non operano su due insiemi restituendone un terzo, ma modificano l'insieme su cui viene chiamata l'operazione a partire dal contenuto di un secondo insieme; ad esempio, `addAll(S)` aggiunge gli elementi dell'insieme S (unione), `retainAll(S)` rimuove tutti gli elementi non contenuti in S (intersezione), `removeall(S)` rimuove tutti gli elementi contenuti in S (differenza).

Molte delle realizzazioni della struttura di dati insieme viste in questo capitolo sono presenti in Java, incluse le liste ordinate, le tabelle hash e gli alberi bilanciati (Red-Black).

8.1 [Intersezione di liste non ordinate] Si scriva la procedura di complessità ottima per intersection(), assumendo che gli insiemi siano realizzati con liste non ordinate.

8.2 [Unione di liste non ordinate] Si scriva la procedura di complessità ottima per union(), assumendo che gli insiemi siano realizzati con liste non ordinate.

8.3 [Unione di liste ordinate] Si scriva la procedura di complessità ottima per union(), assumendo che gli insiemi siano realizzati con liste ordinate.

8.4 [Differenza con liste ordinate] Si scriva la procedura di complessità ottima per difference(), assumendo che gli insiemi siano realizzati con liste ordinate.

8.5 [Differenza simmetrica] La differenza simmetrica $A \Delta B$ di due insiemi A e B è data da tutti gli elementi che stanno nell'unione $A \cup B$ ma non nell'intersezione $A \cap B$. Si scrivano tre procedure per calcolare la differenza simmetrica, assumendo realizzazioni con vettori booleani, liste non ordinate, e liste ordinate, rispettivamente.

8.6 [Realizzazione insieme con tabella hash] Si assuma di rappresentare gli insiemi con tabelle hash. Quali sono le complessità nel caso medio delle operazioni union(), intersection() e difference()?

8.7 [Unione di insiemi con alberi Red-Black] Si scriva la procedura union() utilizzando un albero bilanciato Red-Black.

8.8 [Codominio di un dizionario] Si scriva una funzione che restituisca l'insieme codominio di un dizionario, ovvero l'insieme di tutti i valori contenuti nelle coppie chiave-valore registrati nel dizionario.

8.9 [`LinkedHashSet`] Si realizzi una struttura di dati insieme basata su tabella hash il cui ordine di iterazione (l'ordine in cui vengono analizzati gli elementi in caso di utilizzo del costrutto **foreach**) corrisponda a quello di inserimento. Si tragga ispirazione dalla specifica della classe `LinkedHashSet` contenuta nelle Java Collections.

8.10 [Multinsieme] Si discuta come realizzare un *multinsieme*, una generalizzazione del concetto di insieme che ammette elementi ripetuti, tramite tabelle hash.

B3 Nella procedura union(), si usano tre variabili di tipo posizione per scandire A, B e C. Come nella procedura intersection(), si considerano il primo elemento di A ed il primo elemento di B. Se sono uguali, si inserisce in C uno dei due e ci si posiziona sugli elementi successivi sia in A che in B. Se invece i due elementi sono diversi, si inserisce in C il minore e ci si posiziona sull'elemento successivo solo nella lista del minore, senza avanzare nell'altra lista. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Se una lista tra A e B non è stata esaurita, si prosegue inserendo in C tutti gli elementi rimasti nella lista che non era stata esaurita. Per mantenere C ordinata, l'inserzione in C è sempre effettuata in fondo. La complessità è $O(m + n)$, come per intersection().

```
SET union(SET A, SET B)
C ← Set()
p ← A.head()
q ← B.head()
r ← C.head()
while not A.finished(p) and not B.finished(q) do
    if A.read(p) = B.read(q) then
        C.insert(A.read(p), r)
        p ← A.next(p)
        q ← B.next(q)
    else if A.read(p) < B.read(q) then
        C.insert(A.read(p), r)
        p ← A.next(p)
    else
        C.insert(B.read(q), r)
        q ← B.next(q)
while not A.finished(p) do
    C.insert(A.read(p), r)
    p ← A.next(p)
while not B.finished(q) do
    C.insert(B.read(q), r)
    q ← B.next(q)
return C
```

B4 Come nella procedura intersection(), si considerano il primo elemento di A ed il primo elemento di B. Se quello di A è minore, allora lo si inserisce in C e ci si posiziona sull'elemento successivo di A. Se quello di B è minore, allora si avanza di una posizione solo in B, mentre se i due elementi sono uguali, si avanza di una posizione sia in A che in B. In entrambi i casi non si inserisce alcun elemento in C. Il procedimento viene iterato finché non è esaurita almeno una delle due liste. Se la lista A non è stata esaurita, si prosegue inserendo in C tutti gli elementi rimasti in A. La complessità è $O(m + n)$, come per intersection().

```
SET difference(SET A, SET B)
C ← Set()
p ← A.head()
q ← B.head()
r ← C.head()
while not A.finished(p) and not B.finished(q) do
    if A.read(p) < B.read(q) then
        C.insert(A.read(p), r)
        p ← A.next(p)
    else
        if A.read(p) = B.read(q) then p ← A.next(p)
        q ← B.next(q)
while not A.finished(p) do
    C.insert(A.read(p), r)
    p ← A.next(p)
return C
```

8.6 Ricordiamo che per considerare con un'unica scansione tutte le chiavi contenute in una tabella hash occorre tempo $O(m)$ in caso di memorizzazione interna (dato che bisogna passare per tutte le caselle, anche vuote, della tabella hash) e tempo $O(m + n)$ in caso di memorizzazione esterna (perché si devono scandire tutte le m liste di trabocco contenenti complessivamente n chiavi). Essendo n minore o uguale ad m nel primo caso e minore, uguale o maggiore di m nel secondo, il tempo richiesto per considerare tutte le chiavi in un'unica scansione si può unificare nella formula $O(\max\{m, n\})$ comprendente entrambi i metodi di memorizzazione.

Siano A e B gli insiemi in ingresso e C l'insieme da fornire come risultato, e denotiamo con n_X ed m_X il numero di chiavi e la dimensione della tabella hash T_X che memorizza il generico insieme X. Per le proprietà delle operazioni degli insiemi, occorre dimensionare la tabella T_C in modo che $m_C \geq \min\{m_A, m_B\}$ se si vuole eseguire intersection(), $m_C \geq m_A$ per difference(), ed $m_C \geq m_A + m_B$ per union().

Per effettuare intersection(), si considerano tutte le chiavi della tabella più piccola tra T_A e T_B e si inserisce in T_C ogni chiave della tabella più piccola che non compare nella tabella più grande. Dato che insert() e contains() richiedono tempo medio costante, il tempo medio di intersection() è $O(\max\{m, n\})$, dove m ed n sono i parametri della tabella più piccola. Per union(), si considerano prima tutte le chiavi di una tabella, per esempio T_A, inserendole tutte in T_C, e poi si considerano tutte le chiavi dell'altra tabella T_B, ma inserendo in T_C solo le chiavi di T_B che non compaiono in T_A, con un tempo medio di $O(\max\{m_A, m_A\} + \max\{m_B, m_B\})$. Infine, per difference(), si considerano tutte le chiavi di T_A, inserendo in T_C solo quelle che non compaiono in T_B, con un tempo medio di $O(\max\{m_A, m_A\})$.

8.10 È possibile associare, ad ogni elemento inserito nella tabella hash, un contatore di occorrenze. Tutte le volte che viene inserito un elemento non ancora presente nell'insieme, il contatore viene inizializzato a uno. Tutte le volte che viene inserito un elemento già presente, il contatore viene incrementato di uno. La cancellazione decrementa il contatore, mentre la cancellazione effettiva avviene quando il contatore raggiunge il valore zero.

GRAFI

Capitolo 9

I grafi sono utilizzati per rappresentare insiemi di oggetti e le relazioni che intercorrono fra essi. Molti problemi di calcolo sono equivalenti a problemi su grafi, e possono essere risolti con tecniche sperimentate.

9.1 DEFINIZIONI

Per poter ragionare su una struttura di dati così fondamentale, è importante definirne precisamente la terminologia.

9.1.1 GRAFI ORIENTATI

Un *grafo orientato* (o *grafo diretto*, *directed graph* in inglese) è una coppia $G = (V, E)$, con V insieme finito di elementi, detti *nodi* (o vertici, *vertex* in inglese), ed E insieme finito di coppie ordinate di nodi, detti *archi* (o spigoli, *edge* in inglese). I grafi orientati possono essere rappresentati graficamente disegnando ogni nodo con un cerchietto e ogni arco (u, v) con una freccia che esce dal nodo u ed entra nel nodo v. Il numero di nodi e il numero di archi sono di solito indicati, rispettivamente, con n ed m, oppure con $|V|$ ed $|E|$. Poiché possono esserci al più $n - 1$ archi uscenti da ciascun nodo, risulta $0 \leq m \leq n(n - 1)$.

9.1 ESEMPIO – [Grafo orientato]
$G = (V, E)$ con $V = \{1, 2, 3, 4\}$ ed $E = \{(1,2), (1,3), (2, 4), (3, 2), (4, 2), (4, 3)\}$ è un grafo orientato, ove $n = |V| = 4$ ed $m = |E| = 6$. G può essere rappresentato come mostrato nella Fig. 9.1.

In un grafo orientato, un nodo v si dice *adiacente* ad un nodo u se esiste un arco $(u,v) \in E$. Tale arco si dice *incidente* da u in v. Il *grado entrante* (*uscente*) di un nodo u è il numero di archi incidenti in (da) u.

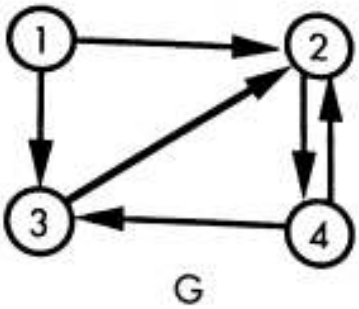


Fig. 9.1 Un grafo orientato.

In un grafo orientato G, un *cammino* è una sequenza di nodi $u_0, u_1, ..., u_k$ tali che $(u_i, u_{i+1}) \in E$, per $i = 0, ..., k-1$. Il cammino parte dal nodo u_0, attraversa i nodi interni $u_1, ..., u_{k-1}$, arriva al nodo u_k, ed ha *lunghezza* uguale al numero k di archi. Se non ci sono nodi ripetuti, cioè $u_i \neq u_j$ per $0 \leq i < j \leq k$, allora il cammino è semplice, mentre se $u_0 = u_k$, allora è chiuso. Un cammino sia semplice che chiuso, dove l'unica ripetizione è $u_0 = u_k$, è detto *ciclo*.

ESEMPIO – [Cammini]

Nel grafo G di Fig. 9.1, la sequenza 1-2-4-3-2 è un cammino, 1-2-4 è un cammino semplice, e 2-4-3-2 è un ciclo.

9.1.2
GRAFI NON ORIENTATI

Talvolta, gli archi di un grafo G sono formati da coppie non ordinate $[u, v]$. In tal caso, G è un *grafo non orientato*. Mentre in un grafo orientato (u, v) e (v, u) indicano due archi distinti, in uno non orientato $[u, v]$ e $[v, u]$ indicano lo stesso arco. Grafi non orientati sono usati per rappresentare relazioni simmetriche tra oggetti.

ESEMPIO – [Grafo non orientato]

Il grafo non orientato $G = (V, E)$ con $V = \{1, 2, 3, 4\}$ e $E = \{[1,2], [1,3], [2,4], [3,2], [4,3]\}$ è mostrato nella Fig. 9.2. I nodi 1 e 3 sono adiacenti, ma 1 e 4 non lo sono.

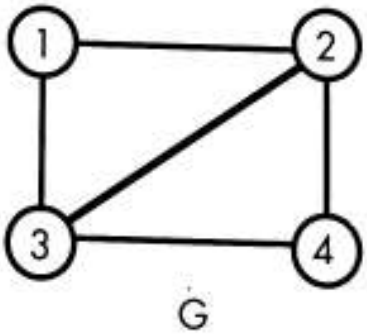


Fig. 9.2 Un grafo non orientato.

ESEMPIO – [Cabala]

Il biblico "Legno (o Albero) della Vita", con i suoi 10 Sefiroth e i 22 Sentieri, è rappresentato nella tradizione cabalistica con il grafo non orientato mostrato prima del testo.

Anche nei grafi non orientati ci sono nozioni analoghe a quelle di cammino e ciclo. In un grafo non orientato G, una *catena* è una sequenza di nodi $u_0, u_1, ..., u_k$ tali che $[u_i, u_j] \in E$, per $i = 0, ..., k-1$. La catena collega u_0 con u_k, o viceversa, ed ha lunghezza uguale a k. Se non ci sono nodi ripetuti, allora la catena è semplice, mentre se $u_0 = u_k$, allora è chiusa. Una catena sia semplice che chiusa i cui archi sono tutti distinti è un *circuito*. Si noti che nei grafi orientati la mancanza di ripetizioni di nodi implica la mancanza di ripetizione degli archi, ma questo non è vero per i grafi non orientati.

9.5 ESEMPIO – [Catene e circuiti]
Nel grafo G di Fig. 9.2, la sequenza 1-2-4-2-3-1 è una catena chiusa e 2-4-3-2 è un circuito. La sequenza 1-2-1 è una catena chiusa ma non un circuito, benché non ci siano nodi ripetuti, perché è ripetuto l'arco [1,2].

9.1.3
CONNESSIONE E CONNESSIONE FORTE

Sia $G = (V, E)$ un grafo orientato. G è *fortemente connesso* se per ogni coppia di nodi u e v esiste almeno un cammino da u a v ed almeno un cammino da v ad u. Un grafo $G' = (V', E')$ è una *componente fortemente connessa* di G se e solo se è un sottografo di G fortemente connesso e massimale:

- G' è un *sottografo* di G ($G' \subseteq G$) se e solo se $V' \subseteq V$ e $E' \subseteq E$
- G' è massimale se non è possibile trovare un sottografo G'' di G che sia fortemente connesso e "più grande" di G', ovvero tale per cui $G' \subseteq G'' \subseteq G$.

9.6 ESEMPIO – [Connessione]
Il grafo G di Fig. 9.1 non è fortemente connesso. Infatti esiste un cammino da 1 a 4, ma non da 4 a 1. Il grafo è infatti composto da due componenti fortemente connesse: {1} e {2, 3, 4}.

Sia $G = (V, E)$ un grafo non orientato. G è *connesso* se e solo se per ogni coppia di nodi distinti u e v esiste una catena tra u e v. Un grafo $G' = (V', E')$ è una *componente connessa* di G se e solo se è un sottografo di G connesso e massimale.

9.7 ESEMPIO
Il grafo G di Fig. 9.2 è connesso. Il grafo di Fig. 9.3 è formato da tre componenti connesse.

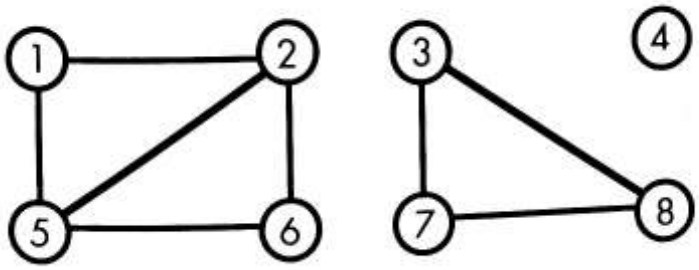


Fig. 9.3 Un grafo non orientato con tre componenti connesse.

9.1.4
RELAZIONI FRA ALBERI E GRAFI

Un grafo non orientato è un *albero libero* se è connesso e per ogni coppia di nodi esiste una e una sola catena semplice. È facile verificare che una definizione equivalente di albero libero richiede, oltre alla connessione, che il numero di archi sia minimo (cioè uguale al numero di nodi meno uno) oppure che non esista alcun circuito. Un *albero radicato* è ottenuto da uno libero stabilendo un ordinamento "verticale" tra i nodi: un nodo r è designato arbitrariamente come radice e i nodi rimanenti sono ordinati per livelli, cioè in base alla loro distanza dalla radice. Dato un albero libero di n nodi, da questo si possono ricavare pertanto n alberi radicati tra loro distinti. Un albero ordinato è ottenuto da uno radicato stabilendo un ordinamento "orizzontale" tra i nodi che stanno allo stesso livello [capitolo 5].

9.8 ESEMPIO – [Alberi liberi e radicati]

Il grafo G di Fig. 9.2 non è un albero libero perché ha 4 nodi e più di 3 archi. Invece, la Fig. 9.4 mostra quattro alberi $T_1, \ldots, T_4$. Visti come alberi liberi, essi sono tutti identici tra loro. T_1 non è radicato, mentre gli altri lo sono. Visti come alberi radicati, T_2 e T_3, avendo entrambi il nodo 2 come radice, sono identici tra loro, ma sono diversi da T_4 che è radicato in 1. La Fig. 9.4 mostra anche un grafo F che non è un albero perché ha 5 nodi, ma meno di 4 archi e quindi non è connesso. Essendo formato da un insieme di alberi, F è una *foresta*.

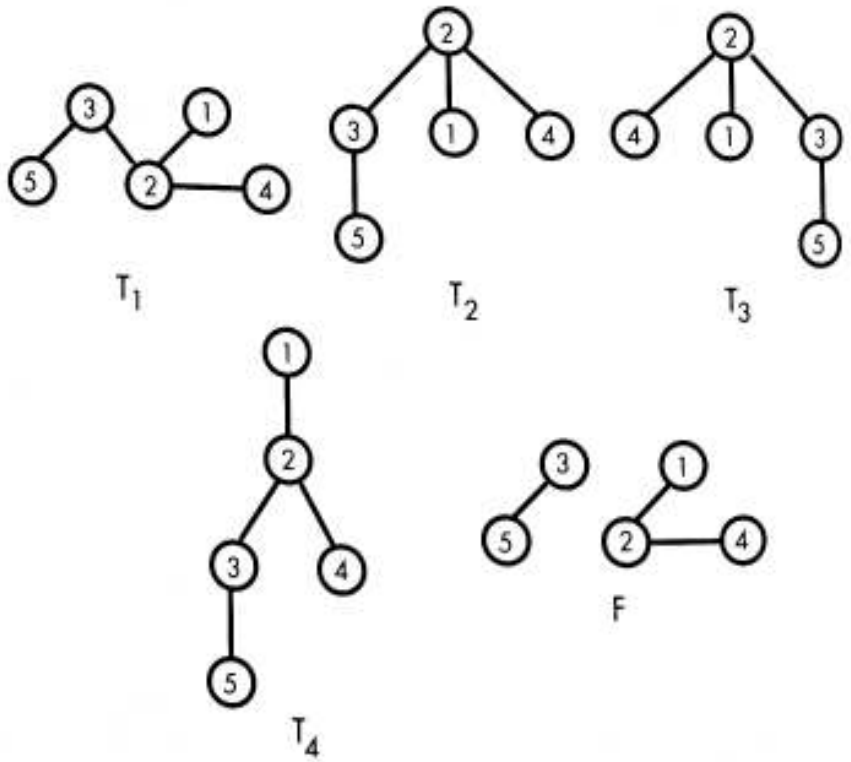


Fig. 9.4 Alberi liberi, radicati e foreste.

Un *albero di copertura* T di un grafo connesso $G = (V, E)$ è un albero libero $T = (V, E')$ composto da tutti i nodi di V e da un sottoinsieme degli archi ($E' \subseteq E$), tale per cui tutte le coppie di nodi del grafo sono connesse da una sola catena nell'albero. Il nome deriva dal fatto che tutti i nodi devono essere "coperti" dall'albero.

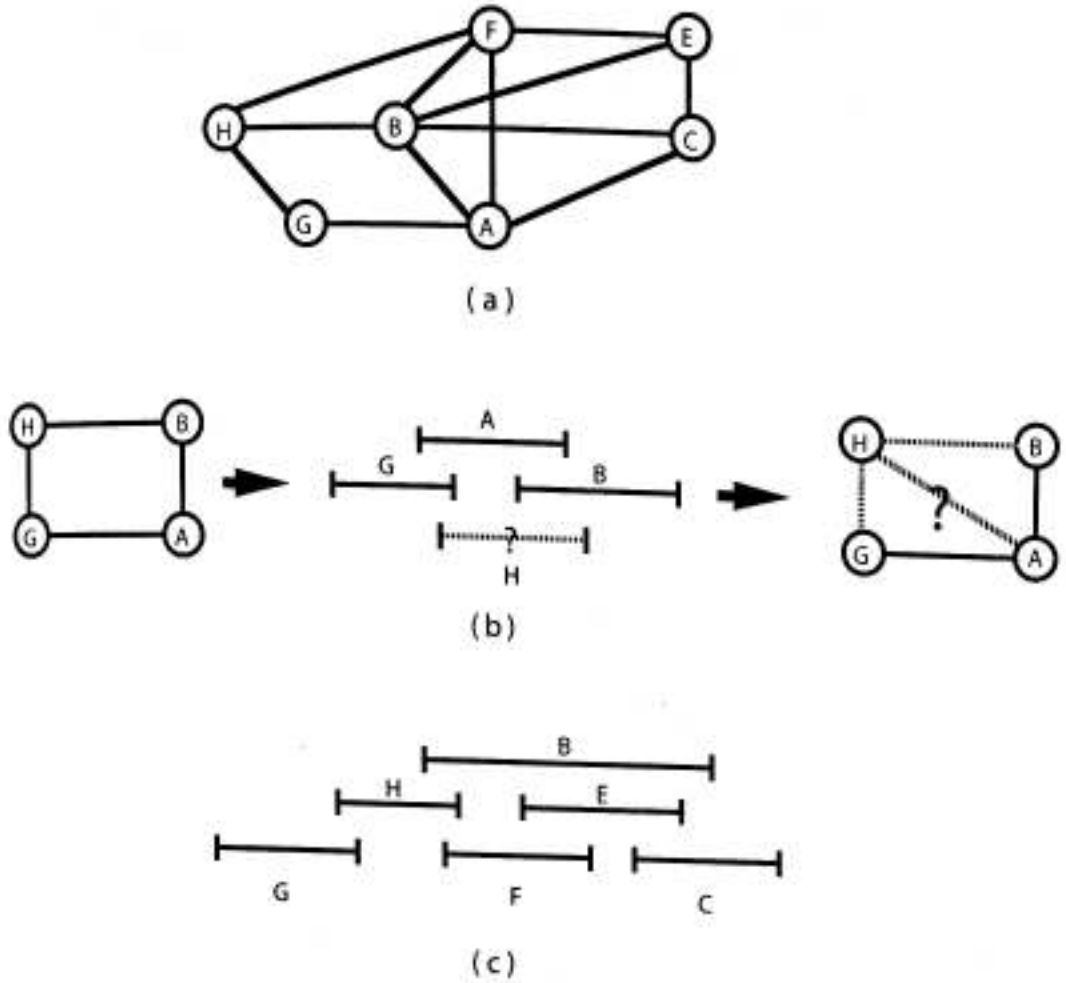


Fig. 9.5 a) Il grafo degli incontri. b) Rappresentazione di circuiti di lunghezza 4 come intersezione di intervalli della retta. c) Possibili periodi di soggiorno al castello cancellando il nodo A.

9.8 ESEMPIO – [Chi ha ucciso il duca McPollock?]

In molti problemi pratici, ci si imbatte in grafi che sono caratterizzati da proprietà particolari, che risultano utili per risolvere il problema stesso. Ad esempio, si consideri il problema seguente: Sherlock Holmes è stato incaricato di indagare sulla misteriosa morte del vecchio duca McPollock, ucciso da un'esplosione nel suo maniero scozzese. Il Dr. Watson si presenta a rapporto da Holmes in Baker Street:

Watson: "Ci sono novità, Holmes: pare che il testamento, andato distrutto nell'esplosione, fosse stato favorevole ad una delle sette 'amiche' del duca."

Holmes: "Ciò che è più strano, è che la bomba sia stata fabbricata appositamente per essere nascosta nell'armatura della camera da letto, il che fa supporre che *l'assassino abbia necessariamente fatto più di una visita al castello*."

Watson: "Ho interrogato personalmente le sette donne, ma ciascuna ha giurato di essere stata nel castello *una sola volta nella sua vita*. Dagli interrogatori risulta che:

1 Ann ha incontrato Betty, Charlotte, Felicia e Georgia;
2 Betty ha incontrato Ann, Charlotte, Edith, Felicia e Helen;
3 Charlotte ha incontrato Ann, Betty e Edith;
4 Edith ha incontrato Betty, Charlotte, Felicia;

5 Felicia ha incontrato Ann, Betty, Edith, Helen;
6 Georgia ha incontrato Ann e Helen;
7 Helen ha incontrato Betty, Felicia e Georgia.

Vedete, Holmes, che le testimonianze concordano. Ma chi sarà l'assassino?"

Holmes: "Elementare, mio caro Watson: ciò che mi avete detto individua inequivocabilmente l'assassino!"

Soluzione di Holmes:
Il grafo degli incontri tra le donne, indicate con le loro iniziali, è disegnato nella Fig. 9.5(a). Se ogni donna avesse detto la verità, questo grafo dovrebbe rappresentare l'intersezione di 7 intervalli della retta reale (un grafo siffatto si dice *grafo di intervalli*). In tal caso, però, ogni circuito di lunghezza maggiore o uguale a 4 dovrebbe possedere un arco che collegasse due nodi non consecutivi nel circuito, come esemplificato nella Fig. 9.5(b). Il grafo degli incontri ha ben 3 circuiti che violano questa proprietà: (1) *A*, G, *H*, *F*, *A*, (2) *A*, G, *A*, B, *A* e (3) *A*, *F*, E, C, *A*. L'unico nodo che compare in tutti e tre i circuiti è il nodo *A*: l'assassino è dunque Ann! Infatti, cancellando *A* dal grafo, i periodi di soggiorno nel castello delle altre 6 donne potrebbero essere quelli illustrati nella Fig. 9.5(c).

9.2 SPECIFICA

Poiché ogni grafo non orientato G può essere visto come un grafo orientato G', ottenuto da G sostituendo ogni arco $[u, v]$ con i due archi (u, v) e (v, u), sarà fornita una specifica solo per grafi orientati, che saranno chiamati semplicemente grafi. Le operazioni definite sui grafi prevedono la possibilità di inserire o cancellare nodi o archi, di ottenere l'insieme di tutti i nodi o dei nodi adiacenti ad un particolare nodo.

GRAPH	
Graph()	% Crea un grafo vuoto
insertNode(NODE u)	% Aggiunge il nodo u al grafo
insertEdge(NODE u, NODE v)	% Aggiunge l'arco (u, v) al grafo
deleteNode(NODE u)	% Rimuove il nodo u dal grafo
deleteEdge(NODE u, NODE v)	% Rimuove l'arco (u, v) nel grafo
SET adj(NODE u)	% Restituisce l'insieme dei nodi adiacenti ad u
SET V()	% Restituisce l'insieme di tutti i nodi

Di tutte queste procedure, discuteremo la realizzazione delle sole procedure adj() e V(); il motivo principale è che, in questo testo, i grafi sono usati per rappresentare relazioni tra

oggetti che non mutano nel tempo. In altri termini, non si effettuano mai né cancellazioni né inserzioni, né di nodi né di archi, ma si esplorano sistematicamente grafi forniti in ingresso, al fine di determinarne sottoinsiemi di nodi e/o di archi che soddisfino certe proprietà.

Qualora il grafo sia dinamico, la struttura di dati più adatta per rappresentarlo dipende dalla particolare utilizzazione del grafo stesso. Realizzazioni efficienti di insiemi e dizionari sono state trattate ampiamente nei capitoli precedenti, ed il lettore può cercare di combinarle per esercizio in modo da fornire realizzazioni efficienti per le operazioni di inserzione e cancellazione di nodi e/o archi.

Inoltre, nella vita di un programmatore capita di lavorare con i grafi senza menzionarli esplicitamente. Ad esempio, un file system può essere visto come un albero di directory e file, ma la presenza di *hard link* e *symbolic link* fanno sì che l'albero sia in realtà un grafo. Le operazioni per la creazione di nodi e archi sono quindi condizionate dalla particolare "semantica" associata al grafo, e parleremo di creazione di file al posto di creazione di nodi.

Infine, la situazione può essere ulteriormente complicata dal fatto che in molte applicazioni ulteriori informazioni (dette *etichette*, o *pesi*) sono associate ai nodi e/o agli archi. In tal caso, si parla di grafi *etichettati* (o *pesati*), i quali richiedono ulteriori operatori, per reperire e/o modificare le informazioni associate a nodi e/o archi. Non è quindi un caso che nelle librerie dei più diffusi linguaggi di programmazione siano disponibili numerose realizzazioni per le strutture di dati insieme, dizionario ecc., ma che non vi sia una realizzazione standard per i grafi. Non staremo quindi noi ad inventarne una fittizia.

Nelle prossime sezioni discuteremo dei principali approcci per realizzare la struttura di dati grafo. Partiremo dall'ipotesi semplificativa che l'insieme statico degli n nodi possa essere rappresentato dai numeri da 1 a n. Per eseguire un'operazione su tutti i nodi e su tutti gli archi, utilizzeremo il costrutto **foreach** come segue:

```
foreach u ∈ G.V()
    foreach v ∈ G.adj(u)
        fai un'operazione sull'arco (u, v)
```

Il primo **foreach** scorre tutti i nodi del grafo, mentre il secondo scorre tutti i nodi adiacenti al nodo u preso in considerazione. Per semplificare il testo inoltre, useremo il termine $G.n$ come abbreviazione di G.V().size().

9.3 REALIZZAZIONI CON MATRICI

Uno dei modi più semplici per rappresentare un grafo è basato su matrici. Dato un grafo $G = (V, E)$, la *matrice di adiacenza* $M = [m_{uv}]$ (detta anche *matrice di adiacenza nodi-nodi*) è una matrice di dimensione $n \times n$, tale che:

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

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

Fig. 9.6 Matrice di adiacenza per il grafo non orientato di Fig. 9.2.

Se il grafo non è orientato, la matrice è simmetrica, ovvero $m_{uv} = m_{vu}$ per ogni u,v. Un esempio di matrice di adiacenza è mostrata nella Fig. 9.6.

Con questa realizzazione è banale verificare in tempo $O(1)$ se un dato arco è presente nel grafo oppure no. Purtroppo, il tempo per ricavare l'insieme di adiacenza $G.\mathsf{adj}(u)$ di un qualsiasi nodo u è sempre $\Theta(n)$, indipendentemente dalla cardinalità di $G.\mathsf{adj}(u)$, perché occorre scandire un'intera riga della matrice alla ricerca degli elementi $m_{uv} = 1$. Inoltre, lo spazio di memoria occupato è $\Theta(n^2)$ e il tempo per esaminare tutti gli archi è $\Theta(n^2)$, anche se il grafo è sparso, ovvero contiene un numero m di archi che è $O(n)$. Se il grafo è pesato, allora nella matrice M si utilizzano i pesi degli archi al posto degli elementi binari. Se p_{uv} è il peso dell'arco (u, v), allora la matrice M diventa:

$$m_{uv} = \begin{cases} p_{uv} & \text{se } (u, v) \in E \\ +\infty \text{ (oppure } -\infty) & \text{se } (u, v) \notin E \end{cases}$$

La *matrice di incidenza nodi-archi* è una rappresentazione su matrice alternativa, in cui si utilizza una matrice rettangolare B di dimensione $n \times m$, dove ciascuna riga rappresenta un nodo e ciascuna colonna rappresenta un arco, tale che:

$$b_{uk} = \begin{cases} -1 & \text{se l'arco } k\text{-esimo esce dal nodo } u \\ +1 & \text{se l'arco } k\text{-esimo entra nel nodo } u \\ 0 & \text{altrimenti} \end{cases}$$

Se il grafo è non orientato, invece, si ha:

$$b_{uk} = \begin{cases} 1 & \text{se l'arco } k\text{-esimo è incidente nel nodo } u \\ 0 & \text{altrimenti} \end{cases}$$

9.10 ESEMPIO – [Matrice di incidenza nodi-archi]
La matrice di incidenza del grafo orientato di Fig. 9.1 è mostrata nella Fig. 9.7(a) (si noti che ogni colonna ha esattamente un +1 e un −1), mentre quella del grafo non orientato di Fig. 9.2 è mostrata nella Fig. 9.7(b) (si noti che ogni colonna ha esattamente due 1).

La matrice di incidenza è molto utile per rappresentare un grafo in certi problemi di PROGRAMMAZIONE LINEARE, nei quali si cerca un vettore x di m incognite (reali o intere) associate agli archi che soddisfi un sistema di equazioni del tipo $Bx = c$, dove c è un vettore di termini noti interi. Purtroppo, con questa rappresentazione, lo spazio di memoria occupa-

to è sempre $\Theta(nm)$. Inoltre, dato un nodo, non è agevole ricavare l'insieme di adiacenza. Infatti per calcolare un qualsiasi $G.\text{adj}(u)$ occorre scandire la riga u di B alla ricerca delle colonne k con elementi $b_{uk} = -1$ e per ciascuna colonna k scandirla per individuare l'indice di riga v tale che $b_{vk} = +1$, per un totale di $\Theta(nm)$ tempo.

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{array}{c}
\begin{array}{cccccc} (1,2) & (1,3) & (2,4) & (3,2) & (4,2) & (4,3) \end{array} \\
\begin{bmatrix}
-1 & -1 & 0 & 0 & 0 & 0 \\
+1 & 0 & -1 & +1 & +1 & 0 \\
0 & +1 & 0 & -1 & 0 & +1 \\
0 & 0 & +1 & 0 & -1 & -1
\end{bmatrix}
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{array}{c}
\begin{array}{ccccc} [1,2] & [1,3] & [2,4] & [3,2] & [4,3] \end{array} \\
\begin{bmatrix}
1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1
\end{bmatrix}
\end{array}
$$

(a) (b)

Fig. 9.7 Matrici di incidenza nodi-archi: a) per il grafo orientato di Fig. 9.1; b) per il grafo non orientato di Fig. 9.2.

9.4 REALIZZAZIONI CON LISTE DI ADIACENZA

È spesso conveniente mantenere esplicitamente gli insiemi di adiacenza per ogni nodo del grafo. Una possibilità consiste nell'utilizzare le cosiddette *liste di adiacenza*.

Questa realizzazione è analoga a quella con liste di trabocco viste per i dizionari. Si usa un vettore di dimensione n dove l'u-esima posizione del vettore contiene l'identificativo di una lista, di solito realizzata con puntatori, che memorizza gli elementi di $G.\text{adj}(u)$. Lo spazio di memoria è $\Theta(n)$ per il vettore e $\Theta(|G.\text{adj}(u)|)$ per ciascuna lista, per un totale di $\Theta(n + \sum_{u \in V} |G.\text{adj}(u)|) = \Theta(n + m)$, che è ottimo. Se il grafo è sparso, cioè m è $O(n)$, lo spazio risulta quindi essere $\Theta(n)$ anziché $\Theta(n^2)$. Inoltre, la scansione di ciascun insieme di adiacenza $G.\text{adj}(u)$ richiede tempo ottimo $\Theta(|G.\text{adj}(u)|)$ mentre la scansione di tutti gli archi del grafo richiede tempo ottimo $\Theta(n + m)$. Purtroppo, anche per verificare se $(u, v) \in E$ occorre una scansione di $G.\text{adj}(u)$ che richiede $O(|G.\text{adj}(u)|)$ tempo.

Nell'ipotesi semplificativa che non vengano effettuate né inserzioni né cancellazioni di nodi e archi, è possibile rappresentare gli insiemi di adiacenza senza utilizzare liste, ma utilizzando invece un *vettore di adiacenza* (di dimensione fissa) per ognuno dei nodi u del grafo, contenente esattamente i nodi adiacenti ad u.

9.11 ESEMPIO – [Vettori di adiacenza]
I vettori di adiacenza che rappresentano il grafo di Fig. 9.2 sono mostrati nella Fig. 9.8. Si noti che, essendo il grafo non orientato, ogni arco $[u, v]$ è rappresentato due volte, poiché $u \in G.\text{adj}(v)$ e $v \in G.\text{adj}(u)$.

Lo spazio di memoria occupato dai vettori di adiacenza è $\Theta(n + m)$, che è ottimo in ordine di grandezza come nel caso delle liste di adiacenza. Rispetto alle liste di adiacenza, però, c'è un ulteriore risparmio dovuto all'assenza dei campi per i puntatori nelle celle delle liste. Se il grafo è pesato, i pesi sui nodi e/o sugli archi possono essere memorizzati ad

esempio aggiungendo vettori simili (un vettore di dimensione n per i pesi sui nodi, un vettore di vettori per i pesi sugli archi).

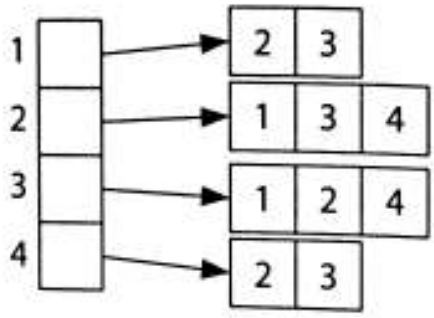


Fig. 9.8 Vettori di adiacenza per il grafo non orientato di Fig. 9.2.

9.5 ESPLORAZIONE DI UN GRAFO

Una *visita* di un grafo è una procedura sistematica per esplorare un grafo, visitando almeno una volta ogni suo nodo ed arco.

9.12 ESEMPIO

I motori di ricerca utilizzano i cosiddetti "web spider" per raccogliere i dati necessari a rispondere alle richieste dell'utente. Un web spider visita il grafo del Web, in cui le pagine web sono nodi e i link in esse contenuti sono archi.

Genericamente, la visita di un grafo può essere descritta dalla procedura visita().

```
visita(Graph G, NODE r)
  SET S ← Set()
  S.insert(r)
  {marca il nodo r come "scoperto"}
  while S.size() > 0 do
      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
              {marca il nodo v come "scoperto"}
              S.insert(v)
```

L'insieme S contiene i nodi che sono stati scoperti, ma non ancora visitati. I nodi vengono estratti da S uno dopo l'altro per essere visitati, e nel contempo si esaminano tutti gli archi uscenti dal nodo visitato per scoprire nuovi nodi. Grazie al controllo effettuato nel costrutto **if**, ogni nodo del grafo è inserito nell'insieme S una sola volta; poiché per ogni nodo v l'insieme G.adj(v) viene scandito una sola volta, questo algoritmo è ottimo e ha complessità $\Theta(n + m)$. Nel caso di grafi orientati e fortemente connessi, o non orientati e connes-

si, la visita parte da un generico nodo r e raggiunge tutti i nodi in una singola "passata". Nei grafi non connessi o non fortemente connessi, è necessario invece fare più passate, ovvero far ripartire l'algoritmo da un nodo non scoperto in precedenza.

L'ordine in cui vengono visitati i nodi dipende dalla politica di estrazione della procedura remove(). Esistono due approcci principali:

- *Breadth-First-Search* (BFS), ovvero *visita in ampiezza* (detta anche a ventaglio);
- *Depth-First-Search* (DFS), ovvero *visita in profondità* (detta anche a scandaglio).

9.5.1 VISITA BFS

Nella visita BFS, i nodi sono visitati in ordine di distanza crescente dal nodo di partenza r, dove la distanza da r ad un generico nodo u è il minimo numero di archi in un cammino da r a u. Tale metodo di visita è un'estensione di una visita per livelli di un albero radicato, in cui i figli di un nodo sono visitati dopo aver visitato tutti gli altri nodi che stanno allo stesso livello del padre.

Poiché la BFS visita i nodi più vicini ad r prima di quelli lontani, l'insieme S è una coda (con politica di rimozione FIFO). Il processo di marcatura è realizzato con un vettore di booleani; la posizione corrispondente al nodo u viene inizializzata a **false**, per diventare **true** quando il nodo viene "scoperto" durante la visita.

```
bfs(GRAPH G, NODE r)
QUEUE S ← Queue()
S.enqueue(r)
boolean[] visitato ← new boolean[1 ... G.n]
foreach u ∈ G.V() − {r} do visitato[u] ← false
visitato[r] ← true
while not S.isEmpty() do
    NODE u ← S.dequeue()
    {esamina il nodo u}
    foreach v ∈ G.adj(u) do
        {esamina l'arco (u, v)}
        if not visitato[v] then
            visitato[v] ← true
            S.enqueue(v)
```

9.12 ESEMPIO – [BFS]

L'ordine di visita BFS dei nodi e degli archi del grafo mostrato nella Fig. 9.9, assumendo di partire dal nodo $r = 1$ e che gli insiemi adj(v) siano memorizzati in ordine crescente, è il seguente, dove per chiarezza i nodi sono evidenziati in grassetto:

1, (1,2), (1,3), (1,4), (1,6), (1,8), **2**, (2,1), (2,4), (2,5), (2,7), (2,9), **3**, (3,1), (3,6), **4**, (4,1), (4,2), (4,5), **6**, (6,1), (6,3), **8**, (8,1), (8,7), **5**, (5,2), (5,4), **7**, (7,2), (7,8), (7,9), **9**, (9,2), (9,7).

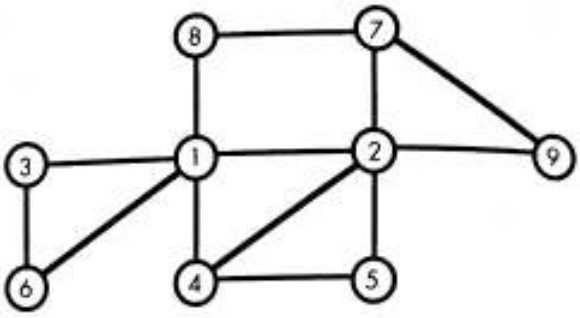


Fig. 9.9 Un grafo non orientato.

9.5.2
APPLICAZIONE BFS: IL NUMERO DI ERDŐS

La storia della matematica è piena di personaggi eccentrici; Paul Erdős è uno di questi. Fu un matematico estremamente prolifico, che arrivò a pubblicare circa 1500 articoli, con più di 500 collaboratori diversi. Gran parte della sua carriera fu spesa con una valigia in mano, passando da un dipartimento di matematica all'altro, collaborando con chi fosse disposto ad ospitarlo.

Come tributo scherzoso alla sua estrema prolificità, è nato il concetto di "numero di Erdős", un coefficiente assegnato ad ogni matematico (in generale, ad ogni autore scientifico) per misurare la sua "distanza" da Erdős. Paul Erdős ha *erdős* = 0. I co-autori dei suoi articoli hanno *erdős* = 1. Chi ha scritto articoli in collaborazione con autori con *erdős* =1 (ma non con Erdős) ha *erdős* = 2. Ricorsivamente, chi ha collaborato con autori con *erdős* = k (ma non con autori con *erdős* < k), ha *erdős* = $k + 1$. Chi non è raggiunto da questa definizione, ha *erdős* = ∞. Il massimo numero di Erdős finito è 15; è scherzosamente considerato un punto d'onore avere un numero di Erdős basso. Gli autori di questo libro hanno *erdős* 3 e 4, rispettivamente.

Dato il grafo delle collaborazioni scientifiche (dove ogni autore è un nodo e due autori sono uniti da un arco se e solo se hanno scritto un articolo scientifico insieme), è possibile calcolare il numero di Erdős di ogni autore utilizzando una visita in ampiezza a partire da Erdős.

In questa versione della BFS, invece di utilizzare un vettore di booleani per la marcatura, utilizziamo il vettore *erdős* di interi, in cui i nodi non ancora scoperti hanno numero di Erdős pari a ∞. Il nodo radice (Erdős) viene inizializzato a zero; ogni volta che viene analizzato un arco (u, v) e il nodo v non è ancora stato scoperto, *erdős*[v] viene posto uguale a *erdős*[u] + 1, come da definizione. Al termine, tutti i nodi non raggiunti dalla singola passata conservano (correttamente) il valore ∞.

```
erdos(GRAPH G, NODE r, integer[] erdős, NODE[] p)
QUEUE S ← Queue()
S.enqueue(r)
foreach u ∈ G.V() − r do erdős[u] ← ∞
erdős[r] ← 0
p[r] ← nil
while not S.isEmpty() do
    NODE u ← S.dequeue()
    foreach v ∈ G.adj(u) do                    % Esamina l'arco (u, v)
        if erdős[v] = ∞ then                   % Il nodo u non è già stato scoperto
            erdős[v] ← erdős[u] + 1
            p[v] ← u
            S.enqueue(v)
```

9.5.3
ALBERI DI COPERTURA BFS

Oltre al valore *erdős*[*s*] di un autore *s*, può essere interessante ottenere anche il cammino di lunghezza *erdős*[*s*] che porta da *r* a *s* (o meglio, uno dei cammini, visto che è possibile che ve ne sia più di uno). Per ottenere questa informazione, è possibile utilizzare l'*albero dei cammini* BFS, che è un albero di copertura del grafo *G* radicato in *r*. In questo albero, i figli di un nodo *u* sono tutti i nodi *v* scoperti durante la visita di archi (*u*, *v*). Il cammino nell'albero che unisce il nodo *r* ad un nodo *s* è ovviamente un cammino anche nel grafo, ed ha lunghezza *erdős*[*s*].

La procedura erdos() memorizza l'albero dei cammini BFS in un vettore di padri *p*, in cui *p*[*v*] contiene *u* se *v* è stato scoperto durante la visita del nodo *u* e in particolare dell'arco (*u*, *v*). Il vettore *p* è passato in input alla procedura erdos(), con ogni elemento del vettore inizializzato a **nil**.

Una volta ottenuto il vettore *p*, è possibile ottenere il cammino dal nodo radice *r* ad un generico nodo *s* tramite la procedura stampaCammino(). Si noti l'interessante uso della ricorsione, che permette di stampare gli autori nell'ordine atteso da *r* ad *s*.

```
stampaCammino(NODE r, NODE s, NODE[] p)
if r = s then print s
else if p[s] = nil then
    print "nessun cammino da r a s"
else
    stampaCammino(r, p[s], p)
    print s
```

9.5.4
VISITA DFS

Il metodo di visita DFS è una diretta estensione della visita in ordine anticipato di un albero ordinato. Quando viene visitato un nuovo nodo u, ci si allontana da esso il più possibile lungo un cammino (o catena), visitando nodi (e archi) nel cammino fino a giungere in un "vicolo cieco", ovvero in un nodo v i cui nodi adiacenti sono già stati tutti visitati. Quando si è raggiunto v si torna indietro lungo l'ultimo arco visitato e si riprende la visita allontanandosi lungo un altro cammino di nodi (e archi) non ancora visitati. Questa tecnica di visita è un'applicazione della tecnica backtrack [capitolo 16].

Nella visita DFS, l'insieme S potrebbe essere realizzato tramite una pila (politica di rimozione LIFO). È più naturale invece descrivere l'algoritmo in maniera ricorsiva, in quanto più coinciso ed elegante. L'algoritmo viene richiamato su un grafo G a partire da un nodo u, mentre *visitato* è il vettore di booleani che rappresenta l'insieme di nodi già scoperti (inizializzato a tutti **false**). Come negli alberi, è possibile distinguere fra una previsita (riga (1)) e una postvisita (riga (2)).

```
dfs(GRAPH G, NODE u, boolean[] visitato)
    visitato[u] ← true
(1) {esamina il nodo u (caso previsita)}
    foreach v ∈ G.adj(u) do
        {esamina l'arco (u, v)}
        if not visitato[v] then
        dfs(G, v, visitato)
(2) {esamina il nodo u (caso postvisita)}
```

ESEMPIO – [DFS]
L'ordine di visita DFS dei nodi e degli archi del grafo di Fig. 9.9, assumendo di partire dal nodo $r = 1$ e che gli insiemi adj(v) siano memorizzati in ordine crescente, è il seguente, dove per chiarezza i nodi sono evidenziati in grassetto:
1, (1,2), **2**, (2,1), (2,4), **4**, (4,1), (4,2), (4,5), **5**, (5,2), (5,4), (2,5), (2,7), **7**, (7,2), (7,8), **8**, (8,1), (8,7), (7,9), **9**, (9,2), (9,7), (2,9), (1,3), **3**, (3,1), (3,6), **6**, (6,1), (6,3), (1,4), (1,6), (1,8).

9.5.5
APPLICAZIONI DFS: COMPONENTI CONNESSE

Modificando opportunamente la DFS appena introdotta, è possibile risolvere in tempo $\Theta(n + m)$ molti problemi fondamentali relativi alla connessione dei grafi, quali ad esempio (i) verificare se un grafo non orientato G è connesso oppure no, oppure (ii) individuare le componenti connesse di cui G è composto.

La soluzione del primo problema è banale: un grafo è connesso se, al termine della visita DFS, tutti i nodi sono stati marcati. Se questo non accade, il grafo è composto da più componenti connesse e si passa al secondo problema. In questo caso, una singola passata non è sufficiente ed è quindi necessario far ripartire la visita da un nodo non marcato, sce-

prendo una nuova porzione del grafo. Vediamo quindi l'algoritmo per identificare le componenti connesse.

Il vettore *id* contiene, per ogni elemento *u*, un intero che identifica la componente connessa a cui *u* appartiene. Il contatore *conta* rappresenta il numero di componenti connesse scoperte finora; ogni volta che viene trovato un nodo *u* non ancora scoperto ($id[u] = 0$), si incrementa di uno tale contatore e si inizia una nuova visita DFS a partire da *u*, assegnando a tutti i nodi scoperti finora il valore di *conta*.

Questa versione di cc() prende in input anche una pila contenente una permutazione dei nodi di *G*; i nodi da cui far partire la visita vengono quindi scelti nell'ordine di estrazione dalla pila. Un simile comportamento non è necessario per il buon funzionamento dell'algoritmo, ma verrà utile in seguito per identificare le componenti fortemente connesse di un grafo orientato.

```
integer [] cc(GRAPH G, STACK S)
---------------------------------------------
integer[] id ← new integer[1 ... G.n]
foreach u ∈ G.V() do id[u] ← 0
integer conta ← 0
while not S.isEmpty() do
    u ← S.pop()
    if id[u] = 0 then
        conta ← conta + 1
        ccdfs(G, conta, u, id)
return id

ccdfs(GRAPH G, integer conta, NODE u, integer[] id)
    id[u] ← conta
    foreach v ∈ G.adj(u) do
        if id[v] = 0 then
            ccdfs(G, conta, v, id)
```

9.15 ESEMPIO – [Componenti connesse]

Il grafo di Fig. 9.3 è formato da tre componenti connesse. Eseguendo la procedura cc() l'ordine di visita dei nodi è 1, 2, 5, 6, 3, 7, 8, 4 ed alla fine il vettore *id* contiene: $id[1]=1$, $id[2]=1$, $id[3]=2$, $id[4]=3$, $id[5]=1$, $id[6]=1$, $id[7]=2$, $id[8]=2$.

9.5.6 ALBERI DI COPERTURA DFS

Così come nelle visite BFS, è possibile definire l'*albero dei cammini T* generato dalla visita DFS. Tutte le volte che viene incontrato un arco che connette un nodo marcato ad uno non marcato, esso viene inserito nell'albero *T*. Qualora non tutti i nodi siano raggiungibili dalla radice (ovvero dal nodo di partenza *r*), perché il grafo non è (fortemente) connesso, allora una visita completa del grafo si ottiene richiamando iterativamente la DFS, come mostrato nella procedura cc. In tal caso, la visita forma una *foresta* di copertura DFS. Se il grafo non è orientato, la

visita provoca un orientamento arbitrario (u, v) oppure (v, u) di ciascun arco $[u, v]$, dovuto alla direzione in cui è esaminato, mentre se il grafo è orientato viene rispettato l'orientamento.

Gli archi non inclusi in T possono essere divisi in tre categorie durante la visita:

- se l'arco è esaminato passando da un nodo di T ad un altro nodo che è suo antenato in T, è detto *arco all'indietro*;
- se l'arco è esaminato passando da un nodo di T ad un suo discendente (che non sia figlio) in T è detto *arco in avanti*;
- altrimenti, è detto *arco di attraversamento*.

Se il grafo è non orientato, ciascun arco sarà o un arco di T o un arco all'indietro. Se il grafo è orientato, tutte le quattro classi sono possibili.

9.16 ESEMPIO – [Albero di copertura DFS]

La Fig. 9.10 mostra un grafo orientato G e la corrispondente partizione degli archi ottenuta eseguendo una DFS a partire dal nodo 1. L'ordine di visita dei nodi e degli archi è: **1**, (1,2), **2**, (2,3), **3**, (3,1), (1,3), (1,4), **4**, (4,2), (4,3). Gli archi (1,2), (2,3) e (1,4) sono in T, l'arco (1,3) è in avanti, l'arco (3,1) è all'indietro, e gli archi (4,2) e (4,3) sono di attraversamento.

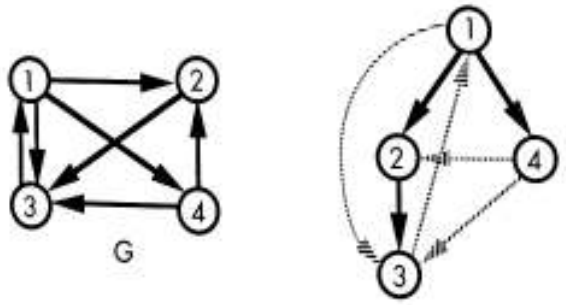


Fig. 9.10 Un grafo orientato e la partizione degli archi indotta da una DFS.

La distinzione tra le quattro classi di archi per i grafi orientati (due per quelli non orientati) è importante perché permette di definire uno schema generale di visita dfs-schema(), che può essere poi particolarizzato per risolvere problemi specifici.

Nello schema si fa uso di due vettori dt e ft (rispettivamente *discovery time* and *finish time*) di dimensione n e di un intero *time* con le funzioni di orologio discreto; per semplicità assumiamo che queste tre variabili siano globali. La variabile *time* è inizializzata a 0 ed è incrementata di uno sia all'inizio che alla fine della procedura DFS. Tutte le posizioni dei vettori dt e ft sono inizializzate a 0 (a significare che la visita non è ancora iniziata); durante la visita, nelle posizioni $dt[u]$ e $ft[u]$ sono registrati i valori dell'orologio, rispettivamente, all'inizio e alla fine della procedura di visita sul nodo u.

Confrontando i valori di $dt[u]$, $dt[v]$ e $ft[v]$ è facile stabilire il tipo dell'arco (u, v). Se $dt[v] = 0$, allora la visita del nodo v non è ancora iniziata, e quindi (u, v) è in T; altrimenti, se $dt[v] \neq 0$, allora (u, v) non è in T. In quest'ultimo caso, se $dt[u] > dt[v]$ e $ft[v] = 0$, la visita di u è iniziata dopo quella di v, la quale però non è ancora terminata, e pertanto (u, v) è all'indietro; quando invece $dt[u] < dt[v]$ e $ft[v] \neq 0$, la visita di v è sia ini-

ziata che finita dopo quella di u, che però non è ancora terminata, e quindi (u, v) è in avanti; nei rimanenti casi (u, v) è di attraversamento.

Per concludere, nelle prossime sezioni accenniamo alle risoluzioni di alcuni semplici problemi, ottenibili modificando opportunamente lo schema proposto.

```
dfs-schema(GRAPH G, NODE u)
esamina il nodo u prima (caso pre-visita)
time ← time + 1;   dt[u] ← time
foreach v ∈ G.adj(u) do
    esamina l'arco (u, v) di qualsiasi tipo
    if dt[v] = 0 then
        esamina l'arco (u, v) in T
        dfs-schema(G, v)
    else if dt[u] > dt[v] and ft[v] = 0 then
        esamina l'arco (u, v) all'indietro
    else if dt[u] < dt[v] and ft[v] ≠ 0 then
        esamina l'arco (u, v) in avanti
    else
        esamina l'arco (u, v) di attraversamento
esamina il nodo u dopo (caso post-visita)
time ← time + 1;   ft[u] ← time
```

9.5.7
APPLICAZIONE SCHEMA DFS: GRAFO ACICLICO

Si consideri il problema di verificare se un grafo $G = (V, E)$ è aciclico. Un grafo è aciclico se e solo se non esistono archi all'indietro. Infatti, 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. Se esiste un ciclo, sia v il primo nodo del ciclo che viene visitato e sia (u, v) un arco del ciclo. Il cammino che connette v ad u verrà prima o poi visitato, e da u verrà scoperto l'arco all'indietro (u, v). Adattando opportunamente lo schema di cui sopra, si ottiene l'algoritmo ciclico() che restituisce **true** se il grafo contiene un ciclo. Si noti che non appena viene trovato un arco all'indietro, la procedura termina restituendo **true** e chiudendo tutte le chiamate ricorsive.

```
boolean ciclico(GRAPH G, NODE u)
time ← time + 1;   dt[u] ← time
foreach v ∈ G.adj(u) do
    if dt[v] = 0 then
        if ciclico(G, v) then return true
    else if dt[u] > dt[v] and ft[v] = 0 then
        return true
time ← time + 1;   ft[u] ← time
return false
```

9.5.8
APPLICAZIONE SCHEMA DFS: ORDINAMENTO TOPOLOGICO

In alcuni casi, è possibile dimostrare la correttezza di un algoritmo in base ai tempi di inizio e di fine, senza doverli effettivamente calcolare! Come esempio, si consideri il problema dell'*ordine topologico* (o linearizzazione) di un grafo orientato (non necessariamente connesso), che consiste nell'identificare un ordinamento dei nodi tale che se (u, v) è un arco del grafo, allora u precede v nell'ordinamento.

9.47 ESEMPIO – [Ristrutturazione]
Ristrutturare casa è un compito improbo. Bisogna coordinare diverse figure professionali (muratori, idraulici, elettricisti, piastrellisti, imbianchini, etc.), ognuna delle quali è impegnata in varie attività (costruire un nuovo muro – creare le tracce per l'idraulico e l'elettricista – riempire le tracce con tubature e canalette – inserire i fili nelle canalette – ricoprire le tracce – imbiancare – applicare le piastrelle – applicare rubinetteria, ecc.). Il coordinamento di queste attività è un problema di ordine topologico: dato il grafo delle dipendenze tra attività, in quale ordine queste attività vanno eseguite?

Innanzitutto, dobbiamo chiederci se tutti i grafi hanno un ordine topologico. La risposta è no: solo i grafi aciclici possono essere ordinati topologicamente. Consideriamo infatti un qualsiasi ordinamento, e supponiamo per assurdo che esista un ciclo. Sia v_j il primo nodo del ciclo a comparire nell'ordinamento; per definizione, esiste un nodo v_i che precede v_j nel ciclo; esiste quindi un arco $(v_i, v_j) \in E$. Per definizione di ordine topologico, v_i deve comparire prima di v_j nell'ordine topologico, ma questo contraddice la definizione di v_j. L'implicazione inversa (dato un grafo acicliclo G, esiste sempre un ordinamento topologico di esso) verrà dimostrata in maniera costruttiva, proponendo un algoritmo che costruisce tale ordinamento.

```
topsort(GRAPH G, STACK S)
    boolean[] visitato ← boolean[1 ... G.n]
    foreach u ∈ G.V() do visitato[u] ← false
    foreach v ∈ G.V() do
    |   if not visitato[u] then
    |   |   ts-dfs(G, u, visitato, S)

ts-dfs(GRAPH G, NODE u, bolean[] visitato, STACK S)
    |   visitato[u] ← true
    |   foreach v ∈ G.adi(u) do
    |   |   if not visitato[v] then
    |   |   |   ts-dfs(G, u, visitato, S)
    |   S.push(u)
```

La procedura topsort(), coadiuvata da topsortdfs(), costruisce un ordine topologico eseguendo una visita DFS che calcola l'istante $ft[u]$ in cui finisce la visita di ciascun nodo u e ordinando i nodi per istanti di fine decrescenti. È interessante notare che questo algoritmo sfrutta il concetto di tempo di scoperta introdotto nel paragrafo 9.5.6, ma in realtà

non richiede che questo valore debba essere calcolato esplicitamente. È facile provare che in un grafo aciclico non esistono archi all'indietro, ed è pertanto agevole dimostrare che se (u, v) è un arco, allora $ft[v] < ft[u]$.

Anzichè calcolare il tempo di fine, quindi, è sufficiente memorizzare il nodo u in una pila S, presa in input, in modo che al momento dell'estrazione i nodi vengano analizzati in ordine inverso di tempo di fine.

9.5.9
APPLICAZIONE SCHEMA DFS: COMPONENTI FORTEMENTE CONNESSE

Vediamo ora un altro algoritmo che non calcola esplicitamente i tempi di inizio e di fine; tali concetti, tuttavia, vengono sfruttati pesantemente per dimostrare la correttezza dell'algoritmo. Il problema considerato è quello dell'identificazione delle componenti fortemente connesse di un grafo orientato. L'algoritmo cc() visto nel capitolo 9.5.5 non risolve questo specifico problema, perché il risultato dipende dall'ordine in cui i nodi vengono visitati. Ad esempio, se la visita del grafo in Fig. 9.1 inizia nel nodo 2, cc() restituirà correttamente due componenti connesse {2,3,4} e {1}; partendo dal nodo 1, restituirà invece un'unica componente {1,2,3,4}. Questo perché dalla componente {1} è possibile raggiungere {2,3,4} ma non viceversa.

Un algoritmo del 1978 attribuito a Kosaraju funziona nel modo seguente: (i) esegue una visita in profondità del grafo G utilizzando l'algoritmo topSort(); (ii) calcola il grafo trasposto G^T di G; (iii) esegue l'algoritmo cc() per ottenere le componenti del grafo G^T, ma nel ciclo principale seleziona i nodi in ordine inverso di tempo di fine rispetto alla prima visita. Ciascuno dei tre passi richiede tempo $O(m + n)$.

Dato un grafo $G = (V, T)$, il *grafo trasposto* $G^T=(V, E^T)$ è formato dagli stessi nodi, mentre gli archi hanno direzioni invertite: i.e, $E^T = \{(u, v) \mid (v, u) \in E\}$.

L'algoritmo è realizzato dalla procedura scc(). È interessante notare che i tempi di inizio e di fine non vengono calcolati esplicitamente; al loro posto, i nodi vengono inseriti in una pila al termine della prima chiamata. La pila verrà passata alla procedura cc(), che la utilizzerà come ordine di visita.

```
integer[] scc(GRAPH G)
    STACK ← topSort(G)                                  % Prima visita
    GRAPH G^T ← Graph()
    foreach u ∈ G.V() do G^T.insertNode(u)              % Calcolo grafo trasposto
    foreach u ∈ G.V() do
        foreach v ∈ G.adj(u) do
            G^T.insertEdge(v, u)
    return cc(G^T, S)                                   % Seconda visita
```

Questo algoritmo si basa sull'osservazione che un grafo G e il suo trasposto G^T hanno le stesse componenti fortemente connesse; questo deriva dalla simmetria della definizione, che prevede, per ogni coppia di nodi u, v, che esista almeno un cammino da u a v e almeno un cammino da v ad u.

Per capire il funzionamento, è utile considerare il *grafo delle componenti* $G^c = (V^c, E^c)$ del grafo G, definito come segue:

- $V^c = \{C_1, C_2, ..., C_k\}$, dove C_i è l'i-esima componente connessa di G;
- $E^c = \{(C_i, C_j): \exists\, (u_i, u_j) \in E: u_i \in C_i, u_j \in C_j\}$

In altre parole, le componenti sono "contratte" in un singolo vertice, e così gli archi che vanno da componente a componente. È facile notare che il grafo delle componenti è aciclico: se così non fosse, i nodi delle componenti appartenenti al ciclo potrebbero tutti raggiungersi l'uno con l'altro; farebbero quindi parte di un'unica componente, una contraddizione.

Estendiamo il concetto di tempo di inizio e di fine per le componenti. Sia C un sottoinsieme di V; definiamo $dt(C)$ come il primo tempo di scoperta e $ft(C)$ come l'ultimo tempo di completamento:

$$dt(C) = \min\{dt[u]: u \in C\}$$
$$ft(C) = \max\{ft[u]: u \in C\}$$

In base a queste definizioni, possiamo enunciare il seguente teorema.

TEOREMA 1

Siano C e C' due componenti distinte nel grafo orientato $G = (V, E)$. Supponiamo che esista un arco $(C, C') \in E^c$. Allora $ft(C) > ft(C')$.

DIMOSTRAZIONE

Sono dati due casi: viene scoperto prima un nodo di C oppure di C'.

- $dt(C) < dt(C')$: sia x il primo nodo scoperto in C, all'istante $dt(C)$. Poiché tutti i nodi in $C \cup C'$ sono raggiungibili da x e non ancora scoperti, la loro visita inizierà e terminerà prima della terminazione della visita di x. Quindi $ft(C) = ft(x) > ft(y)$, per ogni $y \in C \cup C' - \{x\}$. Ne segue che $ft(C) > ft(C')$.
- $dt(C') < dt(C)$: sia y il primo nodo scoperto in C'; tutti i nodi in C' sono raggiungibili da y e non ancora scoperti. Al termine della visita di tutti i nodi di C', tutti i nodi di C non sono ancora stati scoperti. La loro visita inizierà e terminerà dopo $ft(C')$; ne segue che $ft(C) > ft(C')$.

Questo teorema può essere letto nel modo seguente: l'ultimo nodo della componente C ad essere stato visitato verrà inserito nella pila prima dell'ultimo nodo della componente C'. Poiché la pila viene svuotata in ordine inverso, questo significa che almeno un nodo di C verrà visitato prima dei nodi di C' nella seconda visita. A questo punto entra in gioco il

grafo trasposto: se in G^c esiste un arco che connette C con C', in G^{T^c} esso è sostituito da un arco che connette C' con C: nella seconda visita quindi i nodi di C' non sono raggiungibili dai nodi di C. C verrà quindi individuata come una componente a sé stante.

9.6 REALITY CHECK

Le applicazioni dei grafi sono innumerevoli. All'ordinamento topologico su grafi orientati aciclici (in inglese *directed acyclic graph*, o DAG), ad esempio, è addirittura dedicato un programma di utilità specifico: tsort, presente in molti sistemi Unix e nelle GNU Coreutils. tsort prende in input la descrizione di un grafo come questo:

```
3 8
3 10
5 11
7 8
7 11
8 9
11 2
11 9
11 10
```

in cui ogni coppia di numeri identifica un arco, e va letta come "il primo numero precede il secondo", e restituisce il seguente ordinamento topologico: 3 5 7 11 8 10 2 9. Questo algoritmo è usato poi come punto di partenza per programmi di utilità quali GNU make e Apache ant, che permettono la compilazione automatica di progetti software di grandi dimensioni. make e ant costruiscono un ordine topologico di tutte le "regole" (rules) per la compilazione del progetto; se esiste un ciclo, essi si bloccano. L'ordinamento topologico viene anche utilizzato per risolvere le dipendenze fra librerie nei gestori di pacchetti software dei sistemi Linux, nei compilatori per determinare l'ordine ottimale delle istruzioni e negli spreadsheet per aggiornare i valori calcolati tramite formule.

ESERCIZI

9.1 [Occupazione di memoria] Sia $G = (V, E)$ un grafo orientato con n vertici ed m archi. Si supponga che puntatori e interi richiedano d bit. Indicare per quali valori di n ed m la rappresentazione a liste (monodirezionali) di adiacenza occupa meno memoria della matrice di adiacenza.

9.2 [DFS iterativa] Scrivere una versione iterativa della DFS basata su pila dove l'ordine di visita dei nodi e degli archi sia lo stesso di quello della versione ricorsiva descritta nel capitolo 9.5.4 ed eseguirla sul grafo dell'Esempio 9.14.

9.3 [Grafo bipartito] Un grafo non orientato G è *bipartito* se l'insieme dei nodi può essere partizionato in due sottoinsiemi tali che nessun arco connette due nodi appartenenti alla stessa parte. G è 2-colorabile se ogni nodo può essere colorato di bianco o di rosso in modo che nodi connessi da archi siano colorati con colori distinti. Si dimostri che G è bipartito: (1) se e solo se è 2-colorabile; (2) se e solo se non contiene circuiti di lunghezza dispari.

9.4 [Grafo bipartito] Scrivere una procedura basata su una visita BFS per verificare se un grafo non orientato è bipartito oppure no.

9.5 [Larghezza albero radicato] Si modifichi una visita BFS per calcolare la larghezza di un albero radicato [cfr. Esercizio 5.3].

9.6 [Albero libero, non orientato] Si scriva una procedura per verificare se un grafo non orientato è un albero libero.

9.7 [Scheduling] Siano dati n programmi $P_1, P_2, ..., P_n$ che richiedono rispettivamente tempo $T_1, T_2, ..., T_n$ per essere eseguiti. Tra i programmi è definita una relazione $<$ di precedenza aciclica tale che $P_i < P_j$ se l'esecuzione di P_i deve essere completata prima dell'inizio dell'esecuzione di P_j. Si fornisca una procedura per determinare in tempo polinomiale l'istante in cui iniziare l'esecuzione di ciascun programma in modo da rispettare i vincoli di precedenza e terminare l'esecuzione di tutti i programmi il prima possibile.

9.8 [Massima sovrapposizione di intervalli] Siano dati n intervalli distinti $I_1 = [a_1, b_1]$, ..., $I_n = [a_n, b_n]$ della retta reale. Si progetti un algoritmo di complessità $O(n \log n)$ per trovare il più grande sottoinsieme di intervalli che si intersecano tutti tra di loro (nel corrispondente grafo di intervalli, il problema consiste nel trovare un sottoinsieme di cardinalità massima di nodi tale che per ogni coppia di nodi esista un arco che li connetta).

9.9 [Distanza massima] Dato in input un grafo orientato $G = (V, E)$ rappresentato tramite liste di adiacenza e un nodo $r \in V$, restituire il numero dei nodi in G raggiungibili da r che si trovano alla massima distanza da r. Analizzare la complessità.

9.10 [Grafo quadrato] Il *quadrato* di un grafo orientato $G = (V, E)$ è il grafo $G^2 = (V, E^2)$ tale che $(u, w) \in E^2$ se e solo se $\exists\, u : (v, u) \in E \wedge (u, w) \in E$. In altre parole, se esiste un percorso di due archi fra i nodi u e w. Scrivere un algoritmo che, dato un grafo G rappresentato con matrice d'adiacenza, restituisce il grafo G^2.

9.11 [Diametro] Il *diametro* di un grafo è la lunghezza del "più lungo cammino breve", ovvero la più grande lunghezza del cammino minimo (in termini di numero di archi) fra tutte le coppie di nodi. Progettare un algoritmo che misuri il diametro di un grafo e valutare la sua complessità.

9.3 Per rappresentare la matrice, abbiamo bisogno di n^2 bit; per rappresentare le liste di adiacenza, abbiamo bisogno di n puntatori di inizio lista (d bit) più m strutture dati con $2d$ bit (d per il puntatore al successore della lista e d per il puntatore al nodo della lista). Le liste di adiacenza occupano meno memoria della matrice quando $n^2 > 2dm + dn$, ovvero quando $m < \frac{n^2 - dn}{2d}$.

9.4 Come visto nell'Esercizio 9.3, un grafo è bipartito se e solo se è 2-colorabile. Per verificarlo, la procedura bipartito() utilizza una visita BFS colorando la radice di rosso, i nodi raggiungibili dalla radice in un passo di bianco, i nodi raggiungibili da questi di rosso, e così via. Se nel processo si scopre un nodo che deve essere colorato sia di rosso che di bianco, il grafo non è 2-colorabile e quindi nemmeno bipartito. La colorazione viene mantenuta nel vettore di interi *color*. L'ultimo colore utilizzato viene mantenuto in *corrente*, che può assumere i valori rosso = 0, bianco = 1 e senzacolore = 2. Grazie alla scelta delle costanti di colore, l'espressione $1 - corrente$ trasforma il rosso in bianco e viceversa.

```
boolean bipartito(GRAPH G, NODE r)
    QUEUE S ← Queue()
    S.enqueue(r)
    corrente ← rosso
    integer[] color ← new integer[1 ... G.n]
    foreach u ∈ G.V() do color[u] ← senzacolore
    while not S.isEmpty() do
        NODE u ← S.dequeue()
        foreach v ∈ G.adj(u) do
            if color[v] = senzacolore then
                color[v] ← corrente
                S.enqueue(v)
            else
                if color[v] ≠ corrente then return false
        corrente ← 1 − corrente
    return true
```

9.7 Si possono rappresentare i programmi come nodi di un grafo orientato aciclico dove gli archi rappresentano le relazioni di precedenza, cioè se $P_i < P_k$ allora (i, j) è un arco del grafo. Gli istanti di partenza si calcolano scandendo la sequenza dei nodi ordinati topologicamente [capitolo 9.6], e assegnando all'istante di partenza di ogni nodo un valore che è determinato dalla somma dei tempi di esecuzione dei nodi precedenti. Oltre al vettore *ordine*, si utilizzano i vettori *durata* e *partenza*, che contengono, rispettivamente, i tempi di esecuzione di ogni programma e gli istanti di partenza di ciascun nodo. La complessità dell'algoritmo è $\Theta(n + m)$.

```
scheduling(GRAPH G, integer[] durata)
ordine ← new integer[1 ... G.n]
topSort(G, ordine)
partenza ← new integer[1 ... G.n]
partenza[ordine[1]] ← 0
for i ← 2 to G.n do
    integer j ← ordine[i]
    partenza[j] ← partenza[j] + durata[j]
```

6.8 Senza perdere in generalità assumiamo che tutti gli estremi degli intervalli siano distinti (se così non fosse, potremmo sempre perturbare due estremi coincidenti in modo da renderli distinti mantenendo la sovrapposizione tra i rispettivi intervalli). Osserviamo che se c intervalli hanno tutti intersezione non vuota, allora esiste un punto x che appartiene a tutti i c intervalli. Se il punto x non è un estremo di un intervallo, si può sempre spostarlo fino ad incontrare il più vicino estremo (destro o sinistro) di un intervallo. Pertanto possiamo cercare il punto x di massima sovrapposizione tra i $2n$ estremi degli intervalli.

Ordiniamo tali estremi $\{a_1, ..., a_n, b_1, ..., b_n\}$ in modo crescente, ponendoli in un vettore $S[1 ... 2n]$. Scandiamo S da sinistra verso destra e durante la scansione impostiamo opportuni valori in un altro vettore $Q[1 ... 2n]$. Per $k = 1, ..., 2n$, se l'elemento scandito $S[k]$ è un estremo iniziale, cioè un a_i, allora assegnamo $+1$ a $Q[k]$, mentre se $S[k]$ è un estremo finale, cioè un b_i, allora assegnamo -1 a $Q[k]$. Terminata la scansione, calcoliamo le somme prefisse di Q, ovvero per $k = 2, ..., 2n$ eseguiamo $Q[k] \leftarrow Q[k] + Q[k-1]$. Osserviamo che $Q[k]$ fornisce il numero di sovrapposizioni nel punto $S[k]$, se $S[k]$ è un estremo iniziale, ed il numero di sovrapposizioni meno uno, se $S[k]$ è un estremo finale. Adesso, basta trovare $c = \max_{1 \le k \le 2n}\{Q[k]\}$, che fornisce la massima sovrapposizione richiesta. Per ricavare anche il sottoinsieme di intervalli, si memorizza l'indice c che dà il massimo e poi si controlla, per ciascun intervallo I_i, se $a_i \le S[c] \le b_i$ oppure no. In caso affermativo, I_i è aggiunto alla soluzione. Il tempo complessivo è $O(n \log n)$, dovuto all'ordinamento. L'algoritmo si estende facilmente al caso più generale in cui ad ogni intervallo I_i sia associato un peso w_i e si voglia trovare un sottoinsieme di intervalli che si sovrappongono tra loro e per i quali la somma dei pesi sia massima. Basta infatti assegnare a $Q[k]$ il valore $+w_i$, se $S[k]$ è un estremo iniziale a_i, oppure $-w_i$, se $S[k]$ è un estremo finale b_i, e poi procedere come prima.

6.9 È possibile adattare una visita BFS che calcoli le distanze in modo simile a quanto fatto per i numeri di Erdős. Tutte le volte che viene trovato un nodo a distanza maggiore della maggiore distanza trovata finora, il contatore *count* viene riportato a 1 e la variabile *max* viene aggiornata; se il nodo trovato è a distanza uguale alla distanza massima, il contatore viene incrementato.

```
integer maxDistance(GRAPH G, NODE r)

integer d ← new integer[1 ... G.n]                    % Vettore distanze
integer max ← 0                                       % Distanza massima trovata finora
integer count ← 0                                     % Numero di nodi alla distanza massima
QUEUE S ← Queue()
S.enqueue(r)
foreach u ∈ G.V() − {r} do d[u] ← ∞
d[r] ← 0
while not S.isEmpty()
    NODE u ← S.dequeue()
    foreach v ∈ G.adj(u) do                           % Esamina l'arco (u, v)
        if d[v] = ∞ then                              % Il nodo u non è già stato scoperto
            d[v] ← d[u] + 1
            if d[v] > max then
                max ← d[v]
                count ← 1
            else if d[v] = max then
                count ← count +1
            S.enqueue(v)}
return count
```

CODE CON PRIORITÀ E INSIEMI DISGIUNTI

Capitolo 10

Le strutture di dati visti finora, con l'esclusione dei grafi, sono state utilizzate per realizzare i tipi di dato astratto sequenza, insieme e dizionario. Esistono dei casi, tuttavia, in cui questi tipi di dato non soddisfano le peculiari esigenze di un algoritmo, magari perché mancano di una particolare operazione, oppure perché la loro generalità li rende inefficienti per compiti specifici.

Può essere quindi necessario progettare nuovi tipi di dato, e nuove strutture per contenerli. L'elenco di strutture di dati speciali è lungo, e contiene per esempio gli alberi *trie*, gli *heap binomiali*, gli *heap di Fibonacci* ecc. Per motivi di spazio, ci limiteremo ad esaminare due esempi molto diffusi, ovvero code con priorità basate su *heap* binari e insiemi disgiunti.

10.1 CODE CON PRIORITÀ

Una *coda con priorità* (*priority queue*, in inglese) è una particolare struttura di dati, i cui elementi sono associati a *priorità*, valori numerici confrontabili tramite una relazione "$\leq$" di ordinamento totale. Il nome deriva dall'interpretazione in cui tale coda contiene elementi che devono essere serviti rispettandone la priorità.

10.1 ESEMPIO – [Pronto Soccorso]
Nella sala d'attesa del pronto soccorso di un ospedale, un paziente colpito da un incidente che può risultargli fatale sarà soccorso prima di altri pazienti, affetti da malanni meno gravi, che erano già in attesa prima di lui.

La mappatura fra il concetto di priorità e i valori numerici dipende da come viene utilizzata la coda. In alcuni casi, si dà la priorità ai valori numerici più bassi (*priorità decrescente*), come nel caso di un sistema operativo che esegue processi in base alla loro scadenza (*earliest deadline first*); in altri casi, si dà la priorità ai valori numerici più alti (*priorità crescente*).

Questo capitolo descrive le code con priorità decrescente, che verranno utilizzate per risolvere il problema dei CAMMINI MINIMI [capitolo 11]. In questo tipo di dati astratto, sono ammesse le operazioni insert(), che inserisce un nuovo elemento associato ad un valore di priorità; min(), che restituisce l'elemento con valore di priorità più basso; deleteMin(), che restituisce l'elemento con valore di priorità più basso dopo averlo cancellato; e infine decrease(), che diminuisce la priorità di un elemento esistente. In una coda con priorità crescente, insert() rimane inalterato e le operazioni max(), deleteMax() e increase() sostituiscono le altre operazioni.

PRIORITYQUEUE

% Crea una coda con priorità vuota
PriorityQueue()

% Restituisce **true** se la coda con priorità è vuota
boolean ISEMPTY()

% Restituisce l'elemento minimo di una coda con priorità non vuota
ITEM min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota
ITEM deleteMin()

% Inserisce l'elemento x con priorità p in una coda con priorità non piena
% Restituisce un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(ITEM x, **integer** p)

% Diminuisce la priorità dell'oggetto identificato da y portandola a p
decrease(PRIORITYITEM y, **integer** p)

Se più elementi hanno la stessa priorità, questa specifica non descrive in che ordine vengono estratti. La procedura insert() restituisce un oggetto che identifica l'elemento appena inserito nella coda con priorità. Questo identificatore è necessario unicamente per l'operazione decrease(), che dovendo modificare la priorità di un oggetto all'interno della coda ha bisogna di una maniera agile per ritrovare questo oggetto. Nel descrivere la realizzazione, quando questo non genera confusione, non faremo distinzione fra un elemento e la sua priorità.

10.1.1
REALIZZAZIONE CON LISTE

Si può realizzare una coda con priorità utilizzando liste, ordinate o non ordinate. Nel caso di liste ordinate, le operazioni min() e deleteMin() hanno complessità $O(1)$, mentre insert() e decrease() sono $O(n)$. Nel caso di liste non ordinate, la complessità di insert() e decrease() può scendere ad $O(1)$, mentre quella di min() e deleteMin() sale a $O(n)$.

In questa implementazione, l'oggetto di tipo PRIORITYITEM restituito dalla insert() è un puntatore alla cella che contiene l'elemento appena inserito.

10.1.2 REALIZZAZIONE CON ALBERI BILANCIATI

Un'ulteriore possibilità è quella di utilizzare alberi bilanciati; in questo caso, tutte le operazioni hanno costo $O(\log n)$, a scapito di una più complessa organizzazione della struttura di dati e di una maggiore occupazione di memoria.

10.1.3 REALIZZAZIONE CON HEAP

Se il numero di elementi presenti nella coda con priorità può essere limitato superiormente, allora è possibile rendere uniforme la complessità di deleteMin(), insert() e decrease() con una realizzazione sequenziale efficiente e particolarmente semplice, che usa un vettore chiamato *heap*.

Gli elementi della coda con priorità di dimensione n possono essere disposti in un vettore $H[1 \ldots n]$ (detto appunto *heap*) che può essere interpretato come un albero binario B. Ciascun nodo di B corrisponde ad una ben precisa posizione nel vettore H e memorizza un elemento della coda con priorità. L'albero B deve verificare le seguenti proprietà:

(1) se h è il livello massimo delle foglie, allora ci sono esattamente $2^h - 1$ nodi di livello minore di h;

(2) tutte le foglie di livello h sono addossate a sinistra;

(3) ogni nodo diverso dalla radice contiene un elemento della coda con priorità che è maggiore o uguale a quello contenuto nel padre.

10.2 ESEMPIO – [Albero B]
L'albero B di Fig. 10.1 contiene gli elementi della coda con priorità $C = \{5, 10, 8, 11, 13, 12, 9, 18, 3, 6\}$ e verifica le proprietà (1) – (3). Il livello massimo delle foglie è 3, ci sono $2^3 - 1 = 7$ nodi di livello al più 2, e le 3 foglie di livello 3 sono addossate a sinistra.

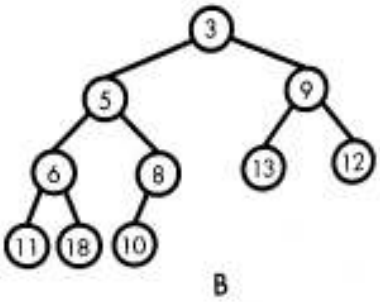


Fig. 10.1 Un albero binario B che rappresenta uno *heap*.

Grazie alle proprietà (1) e (2), ciascun livello k di B (tranne l'ultimo) contiene tutti i possibili 2^k nodi, $0 \leq k < h$, mentre i nodi di livello massimo h sono addossati sul lato "sinistro" dell'albero. In altri termini, B è "quasi pieno", cioè non ha nodi "mancanti" tranne le foglie a destra di livello massimo. Inoltre, tutti i nodi interni hanno grado 2, tranne al più un nodo interno con grado 1.

È possibile passare da una rappresentazione ad albero ad una rappresentazione a vettore in modo molto semplice: si scorre l'albero per livelli, dall'alto verso il basso, e da sinistra verso destra: prima il nodo radice al livello 0, poi i suoi figli al livello 1, poi i nodi del livello 2, e così via fino all'ultimo livello h. Poiché ad ogni livello il numero di nodi raddoppia, si ha che:

- la radice di B viene memorizzata in $H[1]$;
- i figli sinistro e destro del nodo i-esimo (memorizzato in $H[i]$) vengono memorizzati nelle posizioni $H[2i]$ e $H[2i + 1]$, rispettivamente;
- il padre del nodo $i > 1$ è memorizzato nella posizione $H[\lfloor i/2 \rfloor]$.

Per comodità, nel seguito utilizzeremo le seguenti abbreviazioni per indicare le posizioni del padre e dei figli sinistro e destro, se esistono:

- $p(i) = \lfloor i/2 \rfloor$ (padre)
- $l(i) = 2i$ (figlio sinistro)
- $r(i) = 2i + 1$ (figlio destro)

È facile verificare che il figlio sinistro (destro) di i esiste se e solo se $l(i) \leq n$ ($r(i) \leq n$). È possibile esprimere la proprietà (3) direttamente sul vettore *heap*: $H[i] \geq H[p(i)]$, $\forall i > 1$.

ESEMPIO – [Heap H]
Il vettore di Fig. 10.2 rappresenta l'albero di Fig. 10.1.

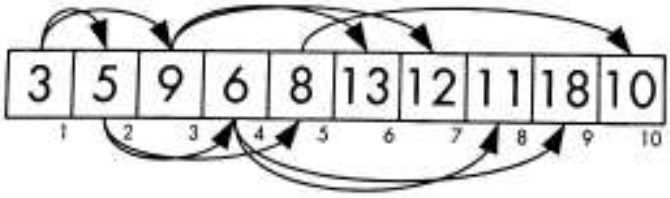


Fig. 10.2 L'*heap* corrispondente all'albero binario di Fig. 10.1.

Queste definizioni, e lo pseudocodice che segue, sono relativi ad una coda con priorità decrescente. In questo caso, la struttura di dati viene chiamata *min-heap*. La struttura simmetrica per code con priorità crescente viene chiamata *max-heap*.

La realizzazione delle operazioni della coda con priorità è molto semplice se si immagina di operare nell'albero B anziché sul vettore H. Infatti, per eseguire min() basta leggere l'elemento contenuto nella radice, mentre per eseguire deleteMin() si cancella la foglia di livello massimo più a destra, sì da mantenere verificate le proprietà (1) e (2), se

ne copia l'elemento nella radice, e si fa "scendere" tale elemento lungo un percorso radice-foglia, scambiandolo con il minimo degli elementi contenuti nei figli, fino a verificare la proprietà (3).

10.4 ESEMPIO – [deleteMin()]
Volendo cancellare 3 dall'albero *B* di Fig. 10.1, si cancella la foglia contenente 10, si ricopia 10 nella radice, si scambia 10 con 5 e successivamente 10 con 6 (Fig. 10.3).

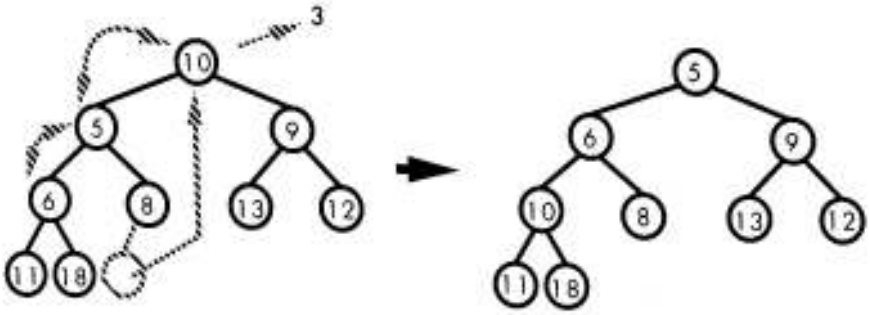


Fig. 10.3 Cancellazione del minimo.

Per eseguire insert(), si aggiunge una foglia col nuovo elemento al livello massimo più a sinistra, per verificare le proprietà (1) e (2), facendo poi "salire" tale elemento lungo un percorso foglia-radice scambiandolo col padre, fino a verificare la proprietà (3). Lo stesso meccanismo si applica alla decrease(): la nuova priorità potrebbe essere inferiore a quella del padre e dei suoi antenati.

10.5 ESEMPIO – [insert]
Volendo inserire l'elemento 4 nell'albero *B* di Fig. 10.1, si aggiunge un figlio, con 4, al nodo contenente 8, si scambia 4 con 8 e successivamente 4 con 5 (Fig. 10.4).

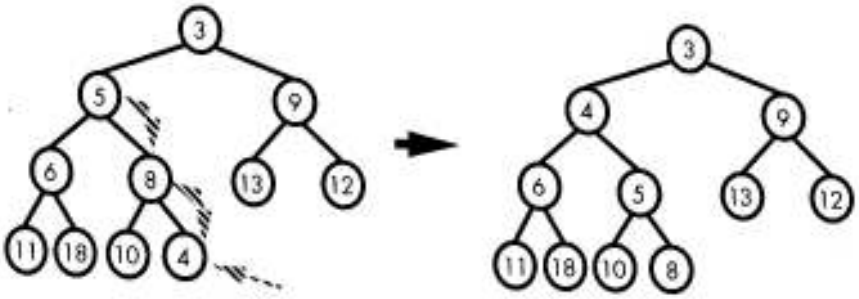


Fig. 10.4 Inserimento di 4.

La proprietà (1) di *B* permette di mantenere il livello massimo h delle foglie limitato da $\lfloor \log n \rfloor$. Infatti, uno *heap* con altezza h ha un numero di nodi n compreso fra 2^h e $2^{h+1} - 1$, da cui $h = \lfloor \log n \rfloor$. Poiché deleteMin() e insert() provocano scambi di elementi lungo un solo percorso radice-foglia, entrambi possono essere realizzati in modo da richiedere al più $O(\log n)$ passi.

Vediamo ora in dettaglio le procedure in pseudocodice che realizzano la coda con

priorità basata su *heap*. Per semplicità, le priorità saranno rappresentate da numeri interi, ma qualunque tipo dotato di una relazione di ordinamento $\leq$ potrebbe andare bene. La struttura di dati è mantenuta in un vettore *heap* di record di tipo PRIORITYITEM; ognuno di questi record contiene, oltre ad una coppia elemento-priorità, la posizione attuale del record all'interno del vettore. Questa informazione è necessaria per realizzare decrease().

```
PRIORITYITEM
    integer priorità                    % Priorità
    ITEM valore                         % Elemento
    integer pos                         % Posizione nel vettore heap
```

Oltre al vettore *heap* H, la struttura di dati contiene un intero *capacità* che indica la dimensione massima del vettore, inizializzata al momento di costruire la struttura di dati, e un intero *dim* che indica quanti elementi sono attualmente presenti nel vettore e corrisponde alla posizione (nel vettore *heap*) della foglia di B che sta più a destra fra quelle di livello massimo. Il valore 0 in *dim* indica una coda con priorità vuota. La funzione min() è molto semplice: se la coda con priorità non è vuota, è sufficiente restituire il primo elemento del vettore.

```
PRIORITYQUEUE
    integer capacità                    % Numero massimo di elementi nella coda
    integer dim                         % Numero attuale di elementi nella coda
    PRIORITYITEM[] H                    % Vettore heap

    PRIORITYQUEUE PriorityQueue(integer n)
        PRIORITYQUEUE t ← new PRIORITYQUEUE
        t.capacità ← n
        t.dim ← 0
        t.H ← new PRIORITYITEM[1 ... n]
        return t

    ITEM min()
        precondition: dim > 0

        return H[1].valore
```

Nella insert(), si incrementa il valore di *dim* per far spazio al nuovo elemento x. Questo corrisponde ad aggiungere una foglia nel livello massimo di B, nella posizione più a sinistra fra quelle disponibili. In questa posizione, l'elemento x potrebbe non rispettare la proprietà (3), ovvero potrebbe essere più piccolo di suo padre. Il cursore i viene inizializzato a *dim*; se i non coincide con la radice e l'elemento contenuto in $h[i]$ è più piccolo di quello di suo padre, questi vengono scambiati dall'operazione swap() e il cursore i risale alla posizione del padre $p[i]$. Il ciclo **while** si ripete fino a quando la proprietà (3) non vie-

ne rispettata. Nel caso pessimo, la risalita continua fino alla radice ($i = 1$). La procedura termina restituendo l'oggetto PRIORITYITEM in cui l'elemento è stato inserito.

```
PRIORITYITEM insert(ITEM x, integer p)
  precondition: dim < capacità

  dim ← dim + 1
  H[dim] ← new PRIORITYITEM()
  H[dim].valore ← x
  H[dim].priorità ← p
  H[dim].pos ← dim
  integer i ← dim
  while i > 1 and H[i].priorità < H[p(i)].priorità do
  |   swap(H, i, p(i))
  |   i ← p(i)
  return H[i]
```

La funzione swap(), oltre a scambiare le posizioni i e $p(i)$, deve anche tenere aggiornata la posizione dei record mano a mano che vengono spostati.

```
swap(PRIORITYITEM[] H, integer i, integer j)
  H[i] ↔ H[j]
  H[i].pos ← i
  H[j].pos ← j
```

Per realizzare deleteMin() useremo una procedura ausiliaria minHeapRestore(), che sarà utilizzata anche nella sezione successiva. deleteMin() si basa su un semplice trucco: sovrascriviamo la prima posizione dello *heap* (quella che contiene il minimo) con l'elemento che si trova nella foglia più a destra del livello massimo, e decrementiamo *dim* ad indicare che il numero di elementi presenti è diminuito di 1. L'albero risultante è "quasi" uno *heap*: i sottoalberi radicati nei figli sinistro e destro della radice rispettano la proprietà (3), mentre l'unica possibile eccezione è proprio la radice, che potrebbe essere maggiore dei propri figli.

La procedura minHeapRestore(A, i, *dim*) ha il compito di rettificare questa situazione; parte dall'assunto che gli alberi radicati in $l(i)$ e $r(i)$ (se esistenti) rispettino già le proprietà dello *heap*, e verifica se uno dei figli di i è minore di i stesso. Se questo accade ($i \neq min$), gli elementi in $A[i]$ e $A[min]$ vengono scambiati. A questo punto, il problema può essersi spostato nella posizione *min*: siamo certi che i sottoalberi radicati in $l(min)$ e $r(min)$ sono *heap* (lo erano prima di eseguire questa chiamata di minHeapRestore()), ma è possibile che l'elemento *min* violi la proprietà (3) dello *heap*, ovvero che $A[min]$ sia maggiore di $A[l(min)]$ o $A[r(min)]$. La procedura minHeapRestore() viene quindi chiamata ricorsivamente su *min*. La chiamata ricorsiva termina quando i e i suoi figli rispettano le proprietà di uno *heap*.

Il codice della deleteMin() e della minHeapRestore() è mostrato di seguito; si noti che deleteMin() chiama minHeapRestore() sulla radice, subito dopo aver eseguito lo scambio con l'ultimo elemento.

Concludiamo con la procedure decrease(), che prende in input un oggetto di tipo PRIORITYITEM e ne aggiorna la priorità, riposizionandolo nel vettore *heap* se necessario. Poiché la priorità è diminuita, viene confrontata con quella del padre e scambiata se necessario.

```
minHeapRestore(PRIORITYITEM[] A, integer i, integer dim)
  integer min ← i
  if l(i) ≤ dim and A[l(i)].priorità < A[min].priorità then min ← l(i)
  if r(i) ≤ dim and A[r(i)].priorità < A[min].priorità then min ← r(i)
  if i ≠ min then
      swap(A, i, min)
      minHeapRestore(A, min, dim)
```

```
ITEM deleteMin ()
  precondition dim > 0

  swap(H, 1, dim)
  dim ← dim − 1
  minHeapRestore(H, 1, dim)
  return H[dim + 1]
```

```
decrease(PRIORITYITEM x, integer p)
  precondition p < x.priorità
  x.priorità ← p
  integer i ← x.pos
  while i > 1 and H[i].priorità < H[p(i)].priorità do
      swap(H, i, p(i))
      i ← p(i)
```

10.1.4
HEAPSORT

L'impiego più famoso della struttura di dati *heap* permette di realizzare un algoritmo di ordinamento di complessità ottima.

10.6 ESEMPIO – [Ordinamento]
Si supponga che la sequenza di n elementi da ordinare sia memorizzata nel vettore A. È possibile ordinare gli elementi di A utilizzando una coda con priorità decrescente, nella quale sono dapprima inseriti in sequenza tutti gli elementi, e dalla quale sono poi estratti, sempre in sequenza, gli elementi minimi.

```
sort(ITEM[] A, integer n)
PRIORITYQUEUE Q ← PriorityQueue(n)
for i ← 1 to n do                          % Inserisce gli elementi nello heap
    Q.insert(A[i], A[i])
for i ← 1 to n do                          % Ricopia gli elementi in ordine
    A[i] ← Q.deleteMin()
```

Se la coda con priorità è realizzata con uno *heap*, la complessità della procedura sort() è $O(n \log n)$. Per quanto descritto nell'Esempio 5.7, una limitazione inferiore alla complessità del problema dell'ordinamento utilizzando confronti è $\Omega(n \log n)$. La procedura sort() ha quindi complessità ottima.

Benché la procedura sort() abbia complessità ottima se si usa uno *heap*, essa presenta lo svantaggio di richiedere un doppio trasferimento di elementi dal vettore A allo *heap* e dallo *heap* di nuovo al vettore A, con conseguente spreco di memoria. È possibile però effettuare l'ordinamento *in loco*, cioè direttamente nel vettore A senza un secondo vettore d'appoggio, dopo aver "ristrutturato" A stesso in modo che verifichi le proprietà di un *max-heap*. Il motivo per cui è necessario un *max-heap* sarà chiarito a breve.

Dato il vettore $A[1 \ldots n]$ in cui gli elementi sono disposti in un ordine qualunque, è possibile permutarli richiamando maxHeapRestore() più volte in modo che al termine il vettore A soddisfi le proprietà di *max-heap*. Poiché maxHeapRestore() lavora assumendo che gli alberi sottostanti siano già *max-heap*, dobbiamo partire dalle foglie e risalire verso la radice. Tutti gli elementi compresi fra $\lfloor n/2 \rfloor + 1$ e n sono foglie nell'albero B; sono quindi già dei *max-heap* per definizione, essendo composti da un solo elemento. Applicando maxHeapRestore() partendo da $\lfloor n/2 \rfloor$ e risalendo fino ad 1, si restaurano le proprietà di *max-heap* essendo certi che i sottoalberi siano *max-heap* (infatti, o sono foglie o sono nodi su cui abbiamo già chiamato maxHeapRestore()).

```
heapBuild(ITEM[] A, integer n)
for i ← ⌊n/2⌋ downto 1 do
    maxHeapRestore(A, i, n)
```

Invece di operare su una struttura di dati a parte e su associazioni elementi-priorità, la versione di maxHeapRestore() utilizzata qui lavora direttamente sul vettore di elementi preso in input.

Per valutare la complessità di heapBuild(), dobbiamo valutare la complessità di maxHeapRestore(), che dipende da i e dim. Sia m l'altezza del nodo i nel solito albero che rappresenta lo *heap* $A[i \ldots dim]$, definita come segue: l'altezza di una foglia è zero, mentre l'altezza di un nodo che non è una foglia è uguale al massimo delle altezze dei figli più uno. Poiché la procedura si basa su una discesa dal nodo i fino ad una foglia, essa richiede tempo $O(m)$.

```
maxHeapRestore(ITEM[] A, integer i, integer dim)
    integer max ← i
    if l(i) ≤ dim and A[l(i)] > A[max] then max ← l(i)
    if r(i) ≤ dim and A[r(i)] > A[max] then max ← r(i)
    if i ≠ max then
        A[i] ↔ A[max]
        maxHeapRestore(A, max, dim)
```

Si noti che in uno *heap* di n elementi ci sono al più $\lceil n/2^{m+1} \rceil$ nodi di altezza m. Poiché maxHeapRestore() richiede tempo $O(m)$ quando è richiamata su un nodo i di altezza m, la complessità di heapBuild() è

$$O\left(\sum_{m=0}^{\log n} \frac{n}{2^{m+1}} \cdot m\right)$$

che è dello stesso ordine di grandezza di

$$O\left(n \sum_{m=0}^{\log n} \frac{m}{2^m}\right)$$

Dato che

$$\sum_{m=0}^{\log n} \frac{m}{2^m} < \sum_{m=0}^{\infty} \frac{m}{2^m} = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$$

la complessità di heapBuild() è $O(n)$.

Terminata la ristrutturazione, l'elemento massimo di $A[1 \ldots n]$ si trova in $A[1]$. La posizione che gli compete è $A[n]$; facendo un'operazione deleteMax(), l'elemento in $A[n]$ viene sistemato nello *heap* $A[1 \ldots n-1]$ e $A[n]$ diviene libera. È quindi possibile scrivere l'elemento massimo in $A[n]$. L'operazione può quindi essere ripetuta con lo *heap* $A[1 \ldots n-1]$, poi con lo *heap* $A[1 \ldots n-2]$, e così via fino allo *heap* $A[1 \ldots 2]$; terminata quest'ultima operazione, il minimo si trova in $A[1]$.

La procedura di ordinamento risultante è detta heapsort() ed è dovuta a Williams (1964); si noti che invece di chiamare max() per leggere il massimo e una deleteMax() per rimuoverlo, ad ogni passo il primo e l'ultimo elemento vengono scambiati e maxHeapRestore() viene chiamata direttamente.

```
heapsort(ITEM[1 ... n] A)
    heapBuild(A, n)
    for i ← n downto 2 do
        A[i] ↔ A[1]
        maxHeapRestore(A, 1, i − 1)
```

La chiamata iniziale a heapBuild() costa $\Theta(n)$. Poiché lo *heap* $A[1 \ldots n]$ ha n elementi, l'altezza della radice (il nodo 1) è $O(\log n)$. Ciascuna chiamata a maxHeapRestore() sulla radice richiede pertanto $O(\log n)$ tempo. Essendoci un numero $O(n)$ di tali chiamate, la complessità di heapsort() è $O(n \log n)$.

10.2 UNIONE DI INSIEMI DISGIUNTI (MERGE-FIND)

Un'altra struttura che opera sugli insiemi è l'*unione di insiemi disgiunti* (in inglese *merge-find set* o *union-find set*). In tale struttura, un insieme generativo S viene partizionato in una collezione di sottoinsiemi (detti anche componenti o parti), su cui sono ammesse un'operazione di ricerca (che permette di stabilire a quale componente appartiene un generico elemento di S) e un'operazione di unione (che unisce due componenti distinte $X \subseteq S$ e $Y \subseteq S$ in una sola nuova componente $X \cup Y$, distruggendo X e Y e lasciando inalterate tutte le altre componenti).

La diversità rispetto all'usuale tipo di dato insieme consiste nel fatto che non sono previste operazioni di inserimento o di cancellazione; tutti gli elementi vengono inseriti all'inizio, suddivisi in una componente disgiunta per elemento, e le successive operazioni di unione fanno sì che le componenti rimangano comunque disgiunte.

Nel seguito, assumeremo che l'insieme generativo sia costituito dagli interi compresi fra 1 e n; in gran parte delle applicazioni pratiche, questa scelta non riduce la generalità della struttura, in quanto insiemi di dimensione fissa n possono essere mappati su vettori della stessa dimensione, e i valori $1 \ldots n$ rappresentano gli indici di tale vettore. Si ottiene quindi la seguente specifica:

MFSET

```
% Crea n componenti {1}, ..., {n}
MFSET Mfset(integer n)

% Restituisce il rappresentante della componente contenente x
integer find(integer x)

% Unisce le componenti che contengono x e y
merge(integer x, integer y)
```

Mfset(n) inizializza la struttura di dati, creando n componenti, ognuna delle quali contiene uno degli interi fra 1 e n. La funzione find(x) restituisce l'identificatore della componente contenente x; tale funzione, applicata a due elementi della stessa componente, deve restituire lo stesso identificatore; applicata a due elementi contenuti in componenti disgiunte, deve restituire due identificatori diversi. Nel seguito, l'identificatore di una componente sarà dato da uno dei suoi elementi, che prenderà il nome di rappresentante. La procedura merge(x, y) unisce le componenti contenenti gli elementi x e y, se sono disgiunte; non fa nulla, se x e y appartengono già alla stessa componente.

10.7 ESEMPIO – [MFSET]

Sia $A = \{1, 2, 3, 4\}$. Mfset(4) restituisce $S = \{\{1\}, \{2\}, \{3\}, \{4\}\}$. Poiché find(1) $\neq$ find(3), eseguendo merge(1, 3) si ottiene $S = \{\{1, 3\}, \{2\}, \{4\}\}$. Adesso, find(4) $\neq$ find(2) ed eseguendo merge(4, 2), si ha $S = \{\{1, 3\}, \{2, 4\}\}$. Essendo find(3) $\neq$ find(4), l'esecuzione di merge(3, 4) dà per risultato $S = \{\{1, 3, 2, 4\}\}$. D'ora in poi, non sono più possibili ulteriori fusioni.

10.2.1
REALIZZAZIONE BASATA SU LISTE

Una realizzazione semplice della struttura MFSET consiste nel creare una lista per ognuno degli insiemi. L'elemento in testa alla lista viene scelto come rappresentante. La procedura Mfset(n) crea n liste, una per ogni elemento. Utilizzando puntatori alla testa e alla coda, merge() può concatenare due liste in tempo $O(1)$. Sfortunatamente, la funzione find() richiede tempo $O(n)$ nel caso peggiore, quindi l'intera lista deve essere scandita per individuare il rappresentante. Questo può essere evitato includendo in ogni cella della lista un puntatore alla testa della lista. A questo punto, tuttavia, è l'operazione merge() a richiedere tempo $O(n)$, in quanto tutti i puntatori della lista che viene concatenata devono essere aggiornati. Sebbene siano possibili alcune ottimizzazioni, è più semplice cambiare approccio e utilizzare una realizzazione basata su foresta.

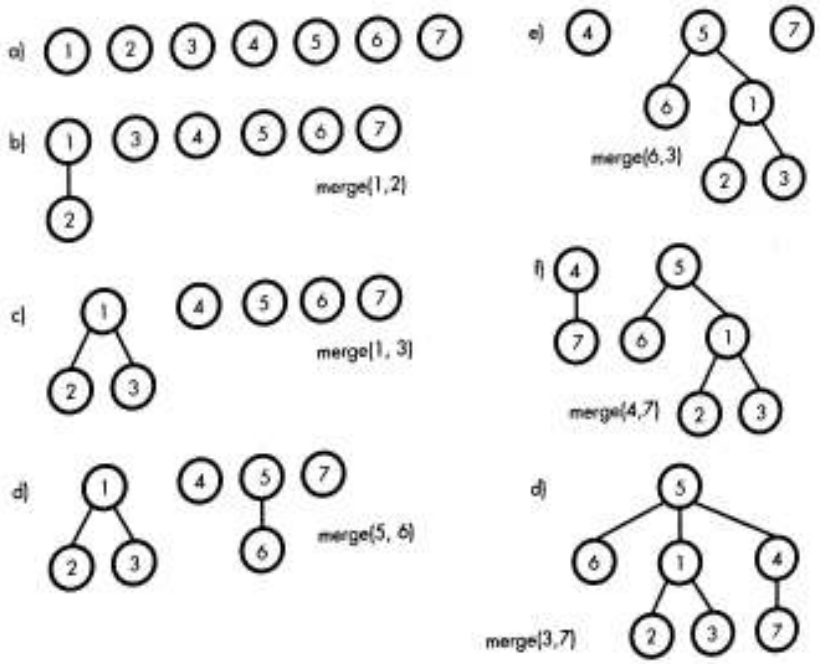


Fig. 10.5 Sequenza di fusioni in un mfset.

10.2.2
REALIZZAZIONE BASATA SU FORESTA

Una realizzazione efficiente per la struttura MFSET prevede che gli elementi di ciascuna componente siano ospitati nei nodi di un albero radicato (non necessariamente ordinato) così che la partizione in componenti sia rappresentata da una foresta di alberi radicati.

Ciascuna componente è identificata dalla radice dell'albero corrispondente. In questo modo, la funzione find() si riduce a restituire la radice dell'albero, "salendo" attraverso un percorso da un generico nodo fino alla radice; la procedura merge() lavora in modo analogo, risalendo gli alberi fino ad arrivare alla radice e imponendo che una delle due radici divenga un nuovo figlio dell'altra.

10.6 ESEMPIO – [Realizzazione basata su foresta]

Sia dato un MFSET S di 7 componenti, ciascuna contenente il solo intero i, $1 \leq i \leq 7$. S è rappresentato con la foresta di Fig. 10.5(a). L'effetto su S dell'esecuzione della sequenza di operazioni merge(1, 2), merge(1, 3), merge(5, 6), merge(6, 3), merge(4, 7) e merge(3, 7) è mostrato nelle Figg. 10.5(b)-(g).

La foresta sui nodi 1, ..., n è implementata come un vettore di padri p; quando $p[x] = x$, significa che x non ha padre e quindi è la radice del sottoalbero. La procedura Mfset inizializza ogni intero in 1, ..., n come radice del proprio sottoalbero. La funzione find(), dato un intero x, risale da esso fino alla radice che viene restituita come identificativo della componente. La merge() richiama due volte la find(), per identificare i rappresentanti r_x e r_y degli interi x e y; se i rappresentanti sono diversi, procede alla fusione degli insiemi, scrivendo nel campo $p[r_y]$ l'indice r_x e rendendo quindi la (ex) radice r_y figlia della radice r_x.

```
MFSET
integer[] p
integer[] rango

Mfset(integer n)
    MFSET t ← new MFSET
    t.p ← integer[1 ... n]
    t.rango ← integer[1 ... n]
    for i ← 1 to n do
        t.p[i] ← i
        t.rango[i] ← 0
    return t

integer find(integer x)
    if p[x] = x then
        return x
    else
        return find(p[x])

merge(integer x, integer y)
    r_x ← find(x)
    r_y ← find(y)
    if r_x ≠ r_y then
        p[r_y] ← r_x
```

10.2.3
EURISTICA DEL RANGO

Nella realizzazione basata su foresta, la creazione di un MFSET ha costo $O(n)$, che è ovviamente ottimale; mentre il costo di find() e merge() dipende dall'altezza degli alberi. Purtroppo, è possibile scegliere una sequenza particolarmente sfortunata di unioni, che renda l'albero una lista di lunghezza n; in questo caso, le operazioni find() e merge() hanno costo $O(n)$.

10.9 ESEMPIO – [Alberi di altezza massima]
Nella merge(x, y), l'albero contenente y diviene sempre figlio dell'albero contenente x; considerando il MFSET di Fig. 10.5(a), la sequenza di operazioni merge(6, 7), merge(5, 6), merge(4, 5), merge(3, 4), merge(2, 3), merge(1, 2) produce un albero di altezza massima.

La complessità può essere abbassata mantenendo il più possibile ridotto il livello massimo delle foglie di ciascun albero. A tal fine, durante la merge() è opportuno scegliere come radice della nuova componente quella relativa alla componente di altezza maggiore: questa tecnica prende il nome di *euristica sul rango*.

Questa informazione viene mantenuta in un vettore ausiliario *rango*; se r_x è la radice dell'albero contenente x, $rango[r_x]$ contiene l'altezza di tale albero. Nella creazione della struttura, tutte le altezze vengono inizializzate a zero. Nella funzione merge(), se $rango[r_x] > rango[r_y]$, si appende la componente radicata in r_y alla componente radicata in r_x e l'altezza non cambia; se $rango[r_y] > rango[r_x]$, si opera in maniera simmetrica; infine, se $rango[r_y] = rango[r_x]$, si sceglie arbitrariamente quale albero rendere figlio dell'altro e si aggiunge 1 alla relativa posizione in *rango*.

```
merge(integer x, integer y)
  r_x ← find(x)
  r_y ← find(y)
  if r_x ≠ r_y then
      if rango[r_x] > rango[r_y] then
          p[r_y] ← r_x
      else if rango[r_y] > rango[r_x] then
          p[r_x] ← r_y
      else
          p[r_x] ← r_y
          rango[r_y] ← rango[r_y] + 1
```

Per giustificare la bontà di questa scelta, cerchiamo di calcolare una limitazione superiore all'altezza dell'albero.

TEOREMA 1
Un albero MFSET con radice r ottenuto tramite euristica sul rango ha almeno $2^{rango(r)}$ nodi.

DIMOSTRAZIONE. Procediamo per induzione sul numero di operazioni merge() effettuate.

All'inizio (nessuna operazione merge()), tutti gli alberi hanno *rango* 0 ed esattamente 1 nodo; poiché $2^0 = 1$, il teorema è dimostrato nel caso base. Consideriamo l'$(n + 1)$-esima operazione merge(x, y), e supponiamo che la proprietà sia vera per tutte le operazioni precedenti. Siano X, Y gli insiemi contenenti x e y, con $X \neq Y$; siano $rango[r_x]$ e $rango[r_y]$ il rango delle radici r_x e r_y, rispettivamente; sia $rango[r]$ il rango della radice risultante dalla $(n + 1)$-esima operazione merge(). Sono dati tre casi:

- $rango[r_y] < rango[r_x]$: abbiamo che $rango[r] = rango[r_x]$; inoltre, per ipotesi induttiva:

$$|X \cup Y| = |X| + |Y| \geq 2^{rango[r_x]} + 2^{rango[r_y]} > 2^{rango[r_x]} = 2^{rango[r]}$$

(in quanto $2^{rango[r_y]} > 0$)

- $rango[r_x] < rango[r_y]$: simmetrico del precedente;
- $rango[r_x] = rango[r_y]$: la radice r_y prende r_x come nuovo figlio, e il suo rango viene incrementato di uno: $rango[r] = rango[r_y] + 1$. Dall'ipotesi induttiva, abbiamo che:

$$|X \cup Y| = |X| + |Y| \geq 2^{rango[r_y]} + 2^{rango[r_y]} > 2^{rango[r_y]+1} = 2^{rango[r]}$$

Sia quindi r la radice di un albero contenente k nodi; dal teorema, sappiamo che $k \geq 2^{rango[r]}$; applicando il logaritmo ad entrambi i lati della disequazione, otteniamo $rango[r] \leq \log k$, e quindi in generale possiamo dire che la complessità di find() e merge() è $O(\log n)$.

10.2.4
COMPRESSIONE DEI PERCORSI

Una limitazione superiore $O(\log n)$ potrebbe essere sufficiente, ma è possibile fare ancora meglio. Un'ulteriore euristica, chiamata compressione dei percorsi, consiste nel modificare la find() in modo che alteri l'albero durante la sua esecuzione, così da rendere più vantaggiose le esecuzioni successive di find() e merge(). In pratica, ogni nodo che viene incontrato da find() nel percorso di risalita dal generico nodo x alla radice, viene reso figlio della radice. Per far questo, la chiamata ricorsiva all'interno di find() è modificata introducendo un assegnamento al padre del nodo x, se questo è diverso da x. In tal modo, durante l'esecuzione, restano "sospesi" gli assegnamenti sui padri dei nodi nel percorso di risalita verso la radice. Quando con la chiamata ricorsiva più interna si giunge alla radice, la ricorsione di find() termina restituendo la radice stessa. La radice viene assegnata "a ritroso" ai padri in tutti gli assegnamenti "sospesi", via via che si richiudono tutte le chiamate ricorsive. È importante notare che il numero di operazioni della nuova versione di find() aumenta solo per un fattore costante rispetto alla vecchia versione e che pertanto la sua complessità nel caso pessimo non aumenta.

integer find(**integer** x)

if $p[x] \neq x$ **then**
  $p[x] \leftarrow$ find($p[x]$)
return $p[x]$

10.10 ESEMPIO – [Compressione dei percorsi]
L'effetto dell'esecuzione di find() con la compressione dei percorsi è mostrato nella Fig. 10.6.

Il vantaggio introdotto dalla compressione dei percorsi si ha quando viene effettuata una sequenza di operazioni formata da m find() e merge(), con $m \geq n$. Senza la compressione dei percorsi, la complessità dovuta all'esecuzione della sequenza sarebbe ovviamente $O(n + m \log n)$, mentre con la compressione dei percorsi è possibile dimostrare che tale costo scende ad $O(n + m\alpha(m, n))$, dove $\alpha(m, n)$ è la funzione inversa della funzione di Ackermann.

$\alpha(m, n)$ è una funzione che cresce lentissimamente, tanto che $\alpha(m, n) \leq 4$ per ogni m ed n rappresentabile con un calcolatore elettronico! Ne segue che ciascuna delle m operazioni find() e merge() della sequenza ha complessità "ammortizzata" $O((n + m\alpha(m, n))/m)$, che è pressoché $O(1)$ per ogni valore "realistico" di n ed m.

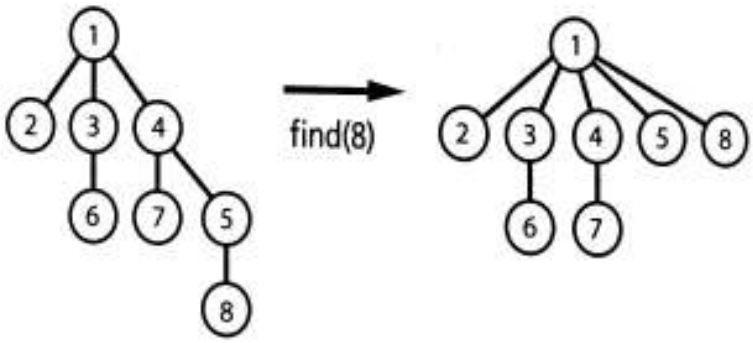


Fig. 10.6 Compressione dei percorsi.

10.2.5
COMPONENTI CONNESSE INCREMENTALI

Come abbiamo visto nel sottoparagrafo 9.5.5, una visita in profondità è sufficiente per identificare le componenti connesse di un grafo non orientato. Tale algoritmo è ottimale, avendo complessità $O(n + m)$. Supponiamo tuttavia di aggiungere un arco; come cambieranno le componenti? Utilizzando l'algoritmo proposto, l'unica possibilità è far ripartire l'algoritmo, pagando lo stesso costo $O(n + m)$. Supponendo di partire da un grafo senza archi e aggiungendo gli archi uno ad uno, il costo totale sarà $O(m(n + m))$. Questo problema prende il nome di *componenti connesse incrementali*.

Possiamo fare meglio? La struttura di dati MFSET può aiutarci. Si parte da n insiemi disgiunti, uno per nodo. Ogni volta che si aggiunge un arco $[u, v]$, si chiama merge(u, v): se le due componenti associate ad u e v sono disgiunte, vengono unite; altrimenti non succe-

de nulla. Poiché utilizzando la compressione dei percorsi l'operazione di unione ha costo ammortizzato costante e viene ripetuta m volte, e la creazione della struttura di dati ha costo $O(n)$, il tempo totale di questo algoritmo è $O(m + n)$, non diverso dalla visita in profondità. Tuttavia, la struttura MFSET ci permette di gestire facilmente l'aggiunta di ulteriori archi; per tale motivo, in questa particolare versione di cc() la struttura MFSET viene restituita (come risultato) al chiamante.

```
MFSET cc(GRAPH G)
MFSET M ← Mfset(|G.V|)
foreach u ∈ G.V do
    foreach v ∈ G.adj(u) do
        M.merge(u, v)
return M
```

10.3 REALITY CHECK

CODE CON PRIORITÀ

La *simulazione* è una branca dell'informatica che cerca di imitare, il più fedelmente possibile, il comportamento dinamico di un sistema o processo reale; esempi di sistemi che possono essere simulati con successo includono gli aerei (nei simulatori di volo), le reti peer-to-peer, ma anche liquidi in un sistema fluidodinamico. Lo scopo della simulazione è generare una storia artificiale del sistema, per permettere lo studio e la valutazione "in vitro" delle caratteristiche del sistema stesso, e rispondere a domande del tipo "cosa succede se"...

Una delle principali tecniche di simulazione, detta *ad eventi* (*event-driven*), richiede che tutte le azioni eseguite nel sistema vengano associate a particolari istanti temporali, detti eventi. Il simulatore non fa altro che eseguire un'azione dopo l'altra, generando nuovi eventi in risposta ad eventi precedenti. Ad esempio, la ricezione di un messaggio in una rete peer-to-peer all'istante t può causare la spedizione di una risposta che verrà ricevuta all'istante $t + \delta$, dove δ è il ritardo di comunicazione.

Una delle principali problematiche per realizzare una simulazione ad eventi è gestire l'insieme di eventi futuri; nuovi eventi possono essere generati (aggiunti) in ogni istante, per poi essere eseguiti nel corretto ordine temporale. È facile vedere che una coda con priorità si presta perfettamente allo scopo, utilizzando l'istante temporale come priorità. Quindi ad ogni passo il simulatore chiama min() e deleteMin() per leggere e rimuovere l'elemento con priorità minore (il prossimo evento da eseguire), e nuovi eventi vengo inseriti con insert().

INSIEMI DISGIUNTI

La struttura di dati MFSET è implementata nella Boost Graph Library per risolvere il problema delle componenti connesse incrementali. A sua volta, questa soluzione è utilizzata

per realizzare l'algoritmo di Kruskal [capitolo 14] per trovare il minimo albero di copertura di un grafo. L'implementazione proposta in questa libreria utilizza un approccio diverso per la creazione dell'insieme disgiunto e per la sua memorizzazione. Invece di utilizzare un vettore di dimensione fissa, permette la creazione di nuovi insiemi con una funzione makeset(a), che crea un nuovo insieme contenente l'oggetto a. Le informazioni relative al padre di un elemento e al suo rango sono memorizzate in dizionari invece che vettori, permettendo quindi la crescita dinamica del numero di elementi.

10.1 [Heap e ABR] Qual è la differenza fra la proprietà degli ABR e la proprietà *min-heap*? È possibile utilizzare la seconda per elencare in modo ordinato le chiavi di un albero di n nodi in tempo $O(n)$?

10.2 Si specifichi, passo dopo passo, il contenuto dello *heap* eseguendo la procedura heapsort() per ordinare alfabeticamente gli 11 elementi: O, R, D, I, N, A, M, E, N, T, O. Sorgono difficoltà dalla presenza di elementi con lo stesso valore?

10.3 [k-merging] Si supponga di avere k vettori ordinati, ognuno di m elementi. Si vuole ottenere un vettore ordinato contenente $n = km$ elementi. Per ordinarlo, un possibile algoritmo è il seguente: si fa il merge del primo vettore con il secondo, ottenendo un nuovo vettore di $2m$ elementi; poi si fa il merge del vettore così ottenuto con il terzo, ottenendo $3m$ elementi; e così via.

1. Calcolare la complessità di questo algoritmo
2. Mostrare un algoritmo più efficiente.

10.4 Si dimostri che nell'albero B che rappresenta uno *heap* di n elementi ci sono al più $\lceil n/2^{m+1} \rceil$ nodi di altezza m.

10.5 Per chiarezza di esposizione, maxHeapRestore() è una funzione ricorsiva. Nel caso pessimo, l'altezza dell'albero di ricorsione è potenzialmente pari a quello dell'albero *heap*; lo spazio aggiuntivo richiesto durante l'esecuzione dell'algoritmo Heapsort (oltre cioè a quello richiesto per memorizzare gli n elementi da ordinare) è quindi $O(\log n)$. Scrivere una versione iterativa di maxHeapRestore() che richieda uno spazio aggiuntivo costante.

10.6 Per chiarezza di esposizione, le implementazioni della coda con priorità e di Heapsort viste in questo capitolo sono basate su vettori in cui l'indice del primo elemento è pari ad 1. Ma molti linguaggi di programmazione (Java e C/C++, per esempio) sono basati su vettori il cui primo elemento è 0. Scrivere una coda con priorità e un algoritmo Heapsort per un linguaggio con vettori di questo tipo.

10.7 [Heap d-ario] Si descriva la realizzazione di uno *heap* d-ario, ovvero uno *heap* in cui ogni nodo ha al più d figli (uno *heap* binario ha $d = 2$).

ESERCIZI

10.8 [Dimensione] Data una radice r, si dimostri che *rango*[r] contiene l'altezza dell'albero radicato in r, se non si usa la compressione dei percorsi; e una sovrastima di essa, se si usa tale tecnica.

10.9 [Compressione senza ricorsione] Si scriva una versione iterativa della funzione find() con compressione dei percorsi senza usare una pila e senza aumentarne la complessità.

10.10 [Arco minimo della componente] Si progetti un algoritmo che dato un grafo non orientato pesato $G = (V, E)$, restituisca l'insieme delle componenti connesse e l'arco di peso minimo di ogni componente.

10.4 In un *min-heap*, la chiave di un nodo è più piccola di entrambe le chiavi dei suoi figli. In un albero binario di ricerca, la chiave di un nodo è più grande o uguale alla chiave del suo figlio sinistro, e più piccola o uguale alla chiave del suo figlio destro. Entrambe le proprietà sono ricorsive. È importante notare che la proprietà di *heap* non ci aiuta a cercare un valore, perché non ci indica che direzione scegliere, visto che i valori sinistro e destro non hanno ordine particolare.

Se la proprietà di *heap* ci permettesse di visitare lo *heap* in tempo $O(n)$, visto che la costruzione di uno *heap* richiede tempo $O(n)$ avremmo risolto il problema dell'ordinamento in tempo lineare, cosa contraria alla limitazione inferiore $\Omega(n \log n)$ per l'ordinamento basato su confronti.

10.6 La complessità dell'algoritmo proposto è la seguente:

$$T(n) = 2m + 3m + \ldots + km = \sum_{i=2}^{k} im = m\sum_{i=2}^{k} i = O(mk^2) = O(nk)$$

Un algoritmo alternativo, invece, utilizza un *min-heap* binario di dimensione k, inizializzato con i primi valori di tutti i vettori. Viene estratto il valore minimo dallo *heap*, che viene collocato nella prima posizione libera del vettore finale. Si inserisce quindi nello heap il prossimo valore preso dal vettore cui appartaneva il valore minimo appena rimosso (se ne esistono ancora). Il costo totale è quindi $O(n \log k)$. Si è utilizzata la notazione $<i, v>$ per operare su una struttura contenente una coppia di elementi e si sono memorizzati i k vettori in una matrice di dimensione $k \times m$.

```
integer [][] merge(integer[][] A, integer k, integer m)
  PriorityQueue Q ← new PriorityQueue(k)  % Coda con priorità per decidere il minimo
  integer[] V ← new integer[km]           % Vettore unito
  integer[ ] p ← new integer[k]           % Posizione attuale di ognuno dei vettori da unire
  for i ← 1 to k do
      p[i] ← 2
      Q.insert(⟨i, A[i, 1]⟩, A[i, 1])
  integer c ← 1                           % Posizione nel vettore unito
  while not Q.isEmpty() do
      ⟨i, v⟩ ← Q.deleteMin()
      V[c] ← v
      c ← c + 1
      if p[i] ≤ m then
          Q.insert(⟨i, A[i, p[i]]⟩, A[i, p[i]])
          p[i] ← p[i] + 1
  return V
```

10.7 La realizzazione è molto semplice: invece di funzioni $l()$ e $r()$, si utilizza una generica funzione $c(i, j) = (i - 1) \cdot d + j + 1$, dove i è l'indice del nodo di cui vogliamo conoscere i figli e j è l'indice del figlio, $1 \leq j \leq d$. La funzione $p()$ è ora definita come $\lceil (i - 1)/d \rceil$. L'albero *heap* è ora meno profondo, in quanto la sua altezza è $O(\log_d n)$; ma l'operazione di cancellazione del minimo, basata su minHeapRestore(), richiede una ricerca del minimo fra $d + 1$ valori ad ogni iterazione, e quindi costa $O(d \log_d n)$. Se d è una costante, entrambe le operazioni sono comunque $O(\log n)$.

10.8 Nel caso in cui non si attui la compressione dei percorsi, ci sono tre casi: quando un insieme viene creato, è costituito da un solo nodo x e il suo rango è inizializzato a 0, che è uguale all'altezza. Quando due alberi di altezze diverse vengono uniti, il più piccolo viene "agganciato" alla radice del più grande, la cui altezza non varia, così come il rango della radice; quando invece vengono uniti due alberi di altezza uguale, l'albero risultante ha l'altezza incrementata di uno, così come il rango della radice. Nel caso sia presente la compressione dei percorsi, il rango non viene toccato, ma ad ogni operazione find() l'altezza può diminuire; viene quindi rispettata la disequazione.

10.9 Una versione iterativa della find() con compressione può essere realizzata in due passate, cercando prima la radice r, e poi ripartendo da x e "comprimendo il cammino" di tutti i nodi che si incontrano fra x ed r.

```
integer find(integer x)
  integer r ← x                                   % Radice
  while p[r] ≠ r do r ← p[r]
  while p[x] ≠ r do
      integer temp ← p[x]
      p[x] ← r                                    % Comprimi il cammino
      x ← temp
  return r
```

10.10 Diamo uno schema di soluzione, lasciando al lettore i dettagli. Ovviamente la struttura di dati da utilizzare è basata su uno *heap*, ma questa struttura andrà opportunamente specializzata per ospitare l'informazione sull'arco di peso minimo. Tutte le volte che si analizza un arco $[u, v]$, ci sono due casi:

- Se $[u, v]$ unisce due componenti C_1 e C_2 prima sconnesse in una nuova componente C, il peso dell'arco minimo di C sarà dato dal valore minimo fra i pesi degli archi minimi di C_1 e C_2, e il peso di $[u, v]$.
- Se u e v appartengono alla stessa componente C, si verifica se il peso di $[u, v]$ sia minore dell'attuale peso minimo di C, nel qual caso si aggiorna quest'ultimo.

TECNICHE DI PROGETTAZIONE

Parte terza

STRUTTURE DI DATI E PROGETTAZIONE DI ALGORITMI

Capitolo 11

Dato un problema, non ci sono "ricette generali" per progettare algoritmi efficienti che lo risolvano. Al contrario, accade molto spesso di ripartire praticamente da zero ogniqualvolta si debba progettare un nuovo algoritmo!

Nel progettare algoritmi, comunque, possono essere evidenziate quattro "fasi" comuni attraverso le quali occorre usualmente passare. Esse sono le seguenti:

(1) *Classificazione del problema*. Innanzitutto, si cerca di stabilire se il problema da risolvere fa parte di una classe più ampia di problemi aventi caratteristiche comuni. Di solito, si usa distinguere tre classi di problemi:

- *problemi decisionali* la cui risposta è "sì" o "no" (o vero o falso) a seconda che il dato d'ingresso soddisfi o no una certa proprietà (per esempio: verificare se un grafo non orientato è connesso, o decidere se esiste una foglia di un albero radicato il cui livello sia maggiore di una certa costante k);
- *problemi di ricerca* dove, tra tutte le possibili soluzioni, si vuole trovarne una, detta *soluzione ammissibile*, che soddisfi una certa condizione (per esempio: ordinare una sequenza di numeri in senso crescente, o trovare una foglia di un albero radicato il cui livello sia maggiore di una certa costante k);
- *problemi di ottimizzazione* nei quali alle soluzioni ammissibili è associata una misura (o costo, o obiettivo) e si vuole trovare una *soluzione ottima*, cioè una soluzione ammissibile la cui misura sia minima o massima (per esempio: trovare una foglia di un albero radicato il cui livello sia massimo).

(2) *Caratterizzazione della soluzione*. In secondo luogo, si prova a caratterizzare matematicamente la soluzione del problema. Nella maggior parte dei casi, la matematica usata è "elementare", ma ciò non significa che i risultati siano banali da dimostrare. Tale caratterizzazione di solito suggerisce un algoritmo di risoluzione

semplice ma non molto efficiente (per esempio: in una sequenza ordinata di numeri, il primo è minore degli altri $n-1$, il secondo è minore dei rimanenti $n-2$, e così via fino all'ultimo che è il massimo; questa caratterizzazione suggerisce un algoritmo di ordinamento basato sulla ricerca iterata del minimo, come illustrato nel capitolo 2);

(3) *Tecnica di progettazione*. Successivamente, si cerca di applicare certe tecniche di progettazione di algoritmi per rendere gli algoritmi più veloci. Le tecniche principali sono: *divide-et-impera*, programmazione dinamica, *greedy*, ricerca locale, *backtrack*, algoritmi probabilistici. La prima di queste tecniche è già stata implicitamente introdotta, e consiste nel partizionare il problema in sottoproblemi più piccoli, risolubili indipendentemente, le cui soluzioni possono essere ricombinate per ottenere la soluzione del problema di partenza (per esempio: la procedura di ricerca binaria vista nel capitolo 1);

(4) *Strutture di dati*. Infine, si vede se c'è un modo astuto di strutturare i dati elaborati dall'algoritmo che permetta di eseguire velocemente le operazioni utilizzate nell'algoritmo (ad esempio: l'algoritmo di ordinamento basato sulla iterata ricerca del minimo è reso più veloce utilizzando uno *heap*).

Di solito, si passa per le fasi (2)-(4) nell'ordine citato, ma questo non sempre è vero. In alcuni casi, se ne possono considerare solo alcune e in un ordine qualsiasi, mentre in altri casi possono essere riconsiderate più di una volta per raffinare e rendere più efficienti diverse versioni successive dell'algoritmo.

Nel seguito di questo capitolo sono esemplificate le fasi di caratterizzazione matematica della soluzione e di utilizzo di strutture di dati nella progettazione di algoritmi efficienti. Per far questo, si considera un noto problema di ottimizzazione su grafi orientati pesati, quello dei cammini minimi, e si mostra come la caratterizzazione della soluzione permetta di definire un algoritmo "prototipo", a partire dal quale si possono derivare diversi algoritmi, più o meno veloci, a seconda della struttura di dati impiegata. Le tecniche di progettazione, invece, saranno l'argomento dei prossimi capitoli.

11.1 CAMMINI MINIMI

Dato un grafo orientato $G = (V, E)$, assumiamo che ogni arco $(u, v) \in E$ sia associato ad un *costo* $w(u, v)$ (o *peso*, o *lunghezza*). Dato un cammino $c = v_1, \ldots, v_k$ (con $k > 1$), il *costo del cammino* è definito da:

$$w(c) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$$

Si consideri il seguente problema di ottimizzazione su grafi:

CAMMINI MINIMI (SHORTEST PATH) Dato un grafo orientato $G = (V, E)$, una funzione di costo $w: E \longrightarrow \mathbf{R}$ e un nodo r, trovare un cammino da r ad u, per ogni nodo $u \in V$, il cui

costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da r a u.

11.1 ESEMPIO – [Cammini minimi]

Un proprietario di TIR è libero di trasportare container da una città ad un'altra. Un viaggio dalla città u alla città v porta un profitto di p_{uv} euro se c'è un container da trasportare da u a v, ma provoca una perdita di p_{uv} euro se il TIR viaggia scarico. Assumendo profitti $p_{uv} > 0$ e perdite $p_{uv} < 0$, il percorso più vantaggioso tra due città corrisponde ad un cammino minimo, in cui le lunghezze degli archi sono $w(u, v) = -p_{uv}$.

Una soluzione ammissibile di questo problema è data da un insieme di $n - 1$ cammini, ognuno dei quali parte dal medesimo nodo r, ma giunge ad uno e uno solo dei rimanenti nodi. In generale, non è detto che un cammino da r ad un qualsiasi altro nodo u esista, poiché u potrebbe non essere raggiungibile da r. Perché esista almeno una soluzione ammissibile, pertanto, occorre che ogni nodo sia raggiungibile da r con un cammino.

Il verificarsi di questa condizione, da sola, non garantisce l'esistenza di una soluzione ottima. Infatti, poiché sono ammesse lunghezze negative, potrebbe esistere nel grafo un ciclo per il quale la somma delle lunghezze degli archi nel ciclo dia un valore negativo. In tal caso, la lunghezza minima di un qualsiasi cammino che includa un nodo di tale ciclo non sarebbe inferiormente limitata. Occorre quindi che non esistano cicli di lunghezza negativa.

Si osservi che due cammini distinti possono avere un tratto iniziale comune, da r fino ad un certo nodo s, e poi divergere verso le rispettive destinazioni u e v, ma viceversa non possono convergere in un nodo comune s dopo aver percorso due tratti iniziali distinti, perché in tal caso ci sarebbero due cammini distinti da r ad s, mentre il problema ne richiede uno solo. Pertanto, una soluzione ammissibile altro non è che un albero di copertura, radicato in r, che include un cammino da r ad ogni altro nodo.

11.2 ESEMPIO – [Albero di cammini]

La Fig. 11.1 mostra un grafo G_1 che contiene un ciclo negativo ed un grafo G_2 privo di cicli negativi. Se $r = 1$, gli alberi T_1 e T_2 sono, rispettivamente, una soluzione ammissibile ed una ottima per G_2.

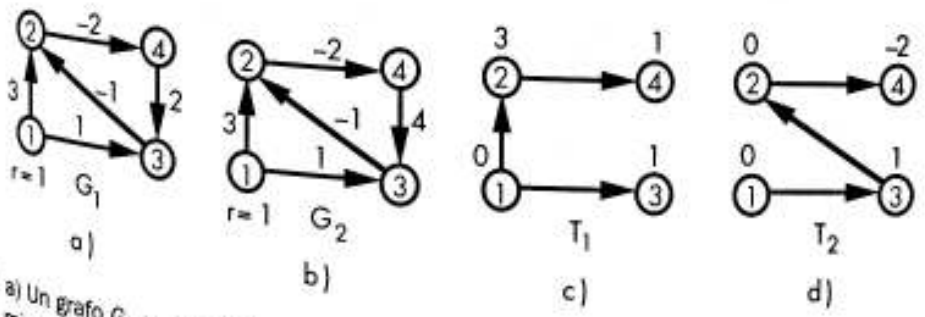


Fig. 11.1 a) Un grafo G_1 con un ciclo negativo; b) un grafo G_2 privo di cicli negativi; c) una soluzione ammissibile per G_2; d) una soluzione ottima per G_2.

Sia T una soluzione ammissibile. Ogni nodo u del grafo è caratterizzato da un valore reale d_u che indica la *distanza* di u da r in T, e che è uguale al costo del cammino fra r ed u in T. Quali caratteristiche devono avere le distanze affinché T sia una soluzione ottima?

(1) Sia T una soluzione ottima. Consideriamo un generico arco $(u, v) \in E$ e sia $w(u, v)$ la sua lunghezza. Ovviamente, se $(u, v) \in T$, allora $d_v = d_u + w(u, v)$. Se invece $(u, v) \notin T$, allora, poiché T è ottimo, deve risultare $d_u + w(u, v) \geq d_v$, poiché altrimenti esisterebbe nel grafo G un cammino tra r e v più corto di quello in T, e precisamente il cammino tra r ed u in T seguito dell'arco (u, v).

(2) Viceversa, sia $d_u + w(u, v) = d_v$ per ogni $(u, v) \in T$, e $d_u + w(u, v) \geq d_v$, per ogni arco $(u, v) \notin T$. Se, per assurdo, il cammino da r a u in T non fosse ottimo, allora esisterebbe un altro cammino in G da r a u tale che la distanza d'_u di u da r sarebbe minore di d_u. Sia d'_v la distanza da r ad un generico nodo v che appare in tale cammino. Poiché $d'_r = d_r = 0$, ma $d'_u < d_u$, esiste un arco (h, k) in questo cammino per cui $d'_h \geq d_h$ e $d'_k < d_k$. Per costruzione $d'_h + w(h, k) = d'_k$, mentre per ipotesi $d_k + w(h, k) \geq d_k$. Combinando queste due relazioni, si ottiene: $d_k = d'_h + w(h, k) \geq d_h + w(h, k) \geq d_k$, e quindi $d'_k \geq d_k$, che contraddice l'ipotesi.

Si è pertanto dimostrato il seguente risultato, dovuto a Bellman (1958).

TEOREMA 1 – [Teorema di Bellman]
Una soluzione ammissibile T è ottima se e solo se valgono le seguenti condizioni: $d_v = d_u + w(u, v)$ per ogni arco $(u, v) \in T$, e $d_u + w(u, v) \geq d_v$ per ogni arco $(u, v) \in E$.

11.2 ALGORITMO PROTOTIPO

Questa caratterizzazione della soluzione permette di formulare un algoritmo che si basa sulla verifica, arco per arco, delle condizioni di Bellman, con conseguente sostituzione degli archi che violano tali condizioni:

```
prototipoCamminiMinimi(GRAPH G, NODE r)
  % inizializza T ad una foresta di copertura composta da nodi isolati
  % inizializza d con una sovrastima della distanza (0 per r, + ∞ altrimenti)
  while ∃(u, v): d_u + w(u, v) < d_v do
      d_v ← d_u + w(u, v)
      % Sostituisci il padre di v in T con u
```

La foresta di copertura iniziale viene sostituita passo dopo passo dall'albero dei cammini minimi; se al termine dell'esecuzione qualche nodo mantiene ancora la distanza $+\infty$, esso non è raggiungibile dalla radice r.

Il passo cruciale dell'algoritmo consiste nel selezionare un arco (u, v) che violi le condizioni di Bellman. Ciò può essere effettuato mantenendo una struttura di dati S che permetta di aggiungere e togliere nodi tramite due operatori generici aggiungi() e estrai(). Inizialmente viene inserito solo il nodo r. Selezionando un nodo u da S è possibile verificare le condizioni di Bellman su tutti gli archi (u, v) tali che v appartiene all'insieme di adiacenza u.adj().

Se viene modificata la distanza d_v, il nodo v viene inserito nuovamente in S, perché è possibile che v possa ora essere un nodo intermedio più conveniente per raggiungere altri nodi.

Il codice di camminiMinimi() definisce meglio il prototipo di procedura appena abbozzato. Sono utilizzati tre vettori, contenenti una posizione per ognuno dei nodi del grafo. Il vettore d contiene le distanze di tutti i nodi dall'origine r; T è un vettore dei padri ricevuto in input e che viene modificato in modo che al termine dell'esecuzione T contenga l'albero dei cammini minimi; infine, b è un vettore booleano tale per cui $b[u]$ = **true** se e solo se u è contenuto nella struttura di dati S. I vettori delle distanze e dei padri sono inizializzati come descritto sopra. La procedura ha accesso ad una funzione di costo w, lasciando alla realizzazione effettiva l'onere di definire dove debbano essere memorizzati tali costi (e.g., nella matrice di adiacenza del grafo oppure nelle celle che formano la lista di adiacenza di un nodo).

Assumendo che la struttura prima o poi restituisca tutto quello che viene inserito, e in assenza di cicli negativi, tale procedura è corretta, ma non molto efficiente. La sua complessità dipende infatti dal numero massimo di inserzioni del medesimo nodo in S e dal numero massimo di miglioramenti delle distanze. In particolare, quest'ultima quantità può non essere polinomialmente correlata alla dimensione n del problema. Nel caso pessimo, le lunghezze $w(u, v)$ possono avere valori che crescono come $\Omega(2^n)$, mentre le distanze possono essere migliorate solo di una unità alla volta, provocando così un tempo complessivo di esecuzione superpolinomiale in n.

```
camminiMinimi(GRAPH G, NODE r, integer[] T)
    integer[] d ← new integer [1 ... G.n]
    boolean[] b ← new boolean [1 ... G.n]
    foreach u ∈ G.V() − {r} do
        T[u] ← nil
        d[u] ← +∞
        b[u] ← false
    T[r] ← nil
    d[r] ← 0
    b[r] ← true
(1) STRUTTURADATI S ← StrutturaDati(); S.aggiungi(r)
    while not S.isEmpty() do
(2)     NODE u ← S.estrai()
        b[u] ← false
        foreach v ∈ G.adj(u) do
            if d[u] + w(u, v) < d[v] then
(3)             if not b[v] then
                    S.aggiungi(v)
                    b[v] ← true
(4)             else
                    % Azione da intraprendere nel caso v sia già presente in S
                T[v] ← u
                d[v] ← d[u] + w(u, v)
```

Per ottenere una procedura efficiente, non basta utilizzare una generica "struttura di dati" con un'altrettanto generica operazione estrai(), ma occorre specificarla ulteriormente! Scelte ragionevoli per la struttura S possono sembrare la coda con priorità, la coda e la pila. Infatti, tutte e tre adottano criteri diversi ma sistematici di "lettura", al contrario della troppo vaga operazione estrai(). Nell'algoritmo, sono state evidenziate quattro righe (numerate da (1) a (4)) che devono essere specializzate a seconda della struttura di dati utilizzata. Nel seguito vedremo diverse implementazioni, e per ognuna di esse specificheremo come sostituire le quattro righe suddette.

11.3 ALGORITMO DI DIJKSTRA

Se la struttura S è una coda con priorità, realizzata con una lista o un vettore non ordinati, si ottiene un algoritmo noto fin dal 1959 e attribuito a Dijkstra. In tal caso, l'operazione estrai() si riduce a deleteMin() [capitolo 10.1]. Gli elementi della coda con priorità sono nodi del grafo e le loro priorità corrispondono alla distanza da r. Ad ogni iterazione è estratto da S il nodo u avente distanza $d[u]$ minima. Il codice da sostituire è il seguente:

(1) PRIORITYQUEUE $S \leftarrow$ PriorityQueue(); S.insert(r, 0)

(2) $u \leftarrow S$.deleteMin()

(3) S.insert(v, $d[u] + w(u, v)$)

(4) S.decrease(v, $d[u] + w(u, v)$)

Benché la complessità dell'algoritmo resti superpolinomiale qualora il grafo contenga archi di lunghezza negativa, tale complessità si riduce drasticamente se tutte le lunghezze sono non negative. Sotto questa ipotesi, infatti, ogni nodo del grafo viene estratto da S una e una sola volta e, al momento dell'estrazione, la sua distanza da r è minima. Per convincersi di ciò, si proceda per induzione sul numero di nodi estratti. Non essendoci lunghezze negative, l'asserto è chiaramente vero per il primo nodo estratto, cioè r, per il quale $d[r] = 0$. Quando viene estratto il k-esimo nodo, p.e. u, la sua distanza $d[u]$ dipende soltanto dai $k-1$ nodi già estratti, ognuno dei quali ha distanza minima per ipotesi d'induzione, ma non può dipendere dai nodi che si trovano ancora in S, che hanno distanza maggiore o uguale a $d[u]$. Non essendoci lunghezze negative, ed essendo $d[u]$ la distanza più piccola tra tutte quelle dei nodi in S, $d[u]$ è minima ed il nodo u non sarà più reinserito in S.

ESEMPIO – [Algoritmo di Dijkstra]
La Fig. 11.2 mostra un grafo G con lunghezze non negative degli archi e l'esecuzione dell'algoritmo di Dijkstra su tale grafo. L'ordine di estrazione da S è: 1, 2, 3, 5, 4, 6.

Poiché ogni nodo è estratto da S una e una sola volta, il ciclo **while** è eseguito esattamente n volte. All'interno del ciclo, l'operazione più costosa è deleteMin() che, utilizzando una struttura non ordinata, richiede tempo $O(n)$, mentre la insert() è $O(1)$ utilizzando una lista non ordinata e già sapendo (grazie al controllo sul vettore booleano) che l'elemento non è presente. Pertanto l'algoritmo di Dijkstra ha complessità $O(n^2)$.

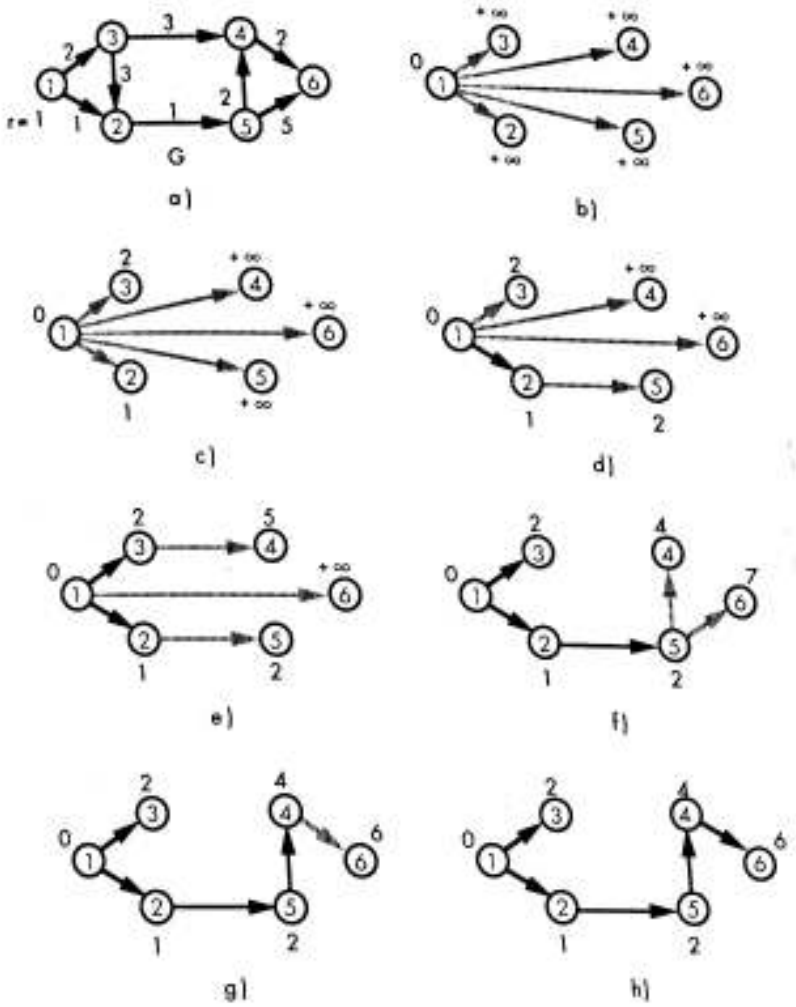


Fig. 11.2 Algoritmo di Dijkstra. a) Un grafo G con lunghezze non negative; b) inizializzazione ad un albero fittizio per $r = 1$; c) - h) aggiornamento dell'albero e delle distanze, estraendo i nodi nell'ordine: 1, 2, 3, 5, 4, 6.

11.4
ALGORITMO DI JOHNSON

Qualora la struttura S sia una coda con priorità realizzata con uno *heap* binario, si ottiene un algoritmo proposto da D.B. Johnson (1977). In molti testi questa versione è associata a Dijkstra, ma si tenga conto che gli *heap* binari sono stati introdotti solo nel 1964 a opera di Williams, quindi ben cinque anni dopo l'algoritmo di Dijkstra. Ad ogni buon conto, le linee da sostituire sono le stesse che nella sezione precedente.

Qualora le lunghezze degli archi siano non negative, ogni nodo è estratto dallo *heap* S una e una sola volta e il ciclo **while** è eseguito esattamente n volte, come nell'algoritmo di Dijkstra. All'interno del ciclo, però, le operazioni deleteMin() e decrease() richiedono tempo $O(\log n)$. Poiché, nel caso pessimo, decrease() è richiamata ad ogni iterazione su tutti i nodi di $G.\text{adj}(u)$, la complessità totale è

$$O\left(n \log n + \left(\sum_{u \in G.V()} |G.\text{adj}(u)|\right) \log n\right)$$

Pertanto l'algoritmo di Johnson ha complessità $O(m \log n)$.

Si noti che per grafi sparsi, nei quali il numero m di archi è $O(n)$, la complessità diviene $O(n \log n)$ e l'algoritmo di Johnson risulta più veloce di quello di Dijkstra. Per grafi densi, dove m è $\Omega(n^2)$, la complessità risulta $O(n^2 \log n)$ e l'algoritmo di Johnson è più lento di quello di Dijkstra.

11.5 ALGORITMO DI FREDMAN-TARJAN

La scarsa efficienza dell'algoritmo di Johnson nel caso di grafi densi deriva dal fatto che l'operazione decrease(), che viene eseguita fino a $O(m)$ volte, ha costo $O(\log n)$. Una possibile soluzione a questo problema prevede di adottare una struttura di dati esplicitamente studiata per migliorare le prestazioni del nostro algoritmo generico, gli *heap* di Fibonacci. Questa struttura di dati, non descritta in questo libro in quanto particolarmente complicata, conserva un costo di $O(\log n)$ per le operazioni insert() e deleteMin(), ma richiede un costo ammortizzato $O(1)$ per le operazioni decrease(). In questo modo, il costo del nostro algoritmo diventa $O(m + n \log n)$. Tuttavia, questo algoritmo ha più interesse teorico che pratico, in quanto gli heap di Fibonacci non sono implementati efficientemente.

11.6 ALGORITMO DI BELLMAN-FORD-MOORE

Se la struttura S è una coda si ottiene un algoritmo, noto sin dalla fine degli anni Cinquanta e scoperto indipendentemente da molti ricercatori, che ha complessità polinomiale anche se ci sono archi di lunghezza negativa! L'algoritmo di Bellman-Ford-Moore si ottiene dal prototipo sostituendo le seguenti righe di codice:

(1) QUEUE $S \leftarrow$ Queue(); S.enqueue(r)

(2) $u \leftarrow S$.dequeue()

(3) S.enqueue(v)

(4) Sezione non necessaria

La struttura dell'algoritmo è in pratica quella di una visita BFS in cui la marcatura di un nodo consiste nel diminuirne la distanza ed il medesimo nodo può essere visitato, cioè estratto dalla coda S, più di una volta, ma non più di $n - 1$ volte!

La dimostrazione di questa proprietà è un po' macchinosa e si rifà alla nozione di "passata", che è definita induttivamente come segue:

(1) per $k = 0$, la zeresima passata consiste nell'estrazione del nodo r dalla coda S;

(2) per $k > 0$, la k-esima passata consiste nell'estrazione di tutti i nodi presenti in S al termine della passata $(k - 1)$-esima.

Si indichino con $d_{v,k}$ il valore della distanza $d[v]$ all'inizio della passata k-esima, con v

l'ultima passata in cui il nodo v si trova in S, con S_k l'insieme dei nodi in S nella passata k-esima, con $I(v)$ l'insieme $\{u: (u, v) \in E\}$, e con T^* l'albero ottimo dei cammini minimi, con distanze d_{u*} per ogni nodo u.

La proprietà cruciale dell'algoritmo è che, per ogni arco (u, v) che sta nella soluzione ottima T^*, p_v è uguale a p_u oppure a $p_u + 1$. Innanzitutto, è facile verificare che, per ogni v e k, si ha:

$$d_{v,k} = \min\{d_{v,k-1} + w(u, v): u \in I(v) \cap S_{k-1}\}$$

poiché S è una coda. Da questa proprietà discende che, se $v \in S_{p_v}$, allora deve esistere un nodo $z \in S_{p_v - 1}$ che ha provocato l'inserimento di v in S per l'ultima volta. Ma allora la distanza ottima d_{v*} è stata determinata esplorando:

- l'arco (z, v), e quindi $p_v = p_z + 1$, oppure
- un arco (u, v) con $u \in S_{p_v}$, e allora $p_v = p_z$.

In entrambi i casi, un eventuale reinserimento di u o z causerebbe un'ulteriore diminuzione di p_v e un reinserimento di v in S, il che è impossibile. Poiché il cammino semplice più lungo contiene al più $n - 1$ archi, allora $p_v < n$ per ogni nodo v, ed il massimo numero di passate è n. Siccome ad ogni passata la coda S contiene al più tutti i nodi e ad ogni passata si esaminano al più tutti gli archi, l'algoritmo di Bellman-Ford-Moore ha complessità $O(n(m + n)) = O(nm)$, qualsiasi sia il segno delle lunghezze degli archi.

11.4 ESEMPIO – [Algoritmo di Bellman-Ford-Moore]

La Fig. 11.3 mostra l'esecuzione dell'algoritmo di Bellman-Ford-Moore su un grafo G per $r = 1$. L'ordine di estrazione dei nodi da S è: 1 (passata 0); 2, 3 (passata 1); 4, 2 (passata 2) e 4 (passata 3).

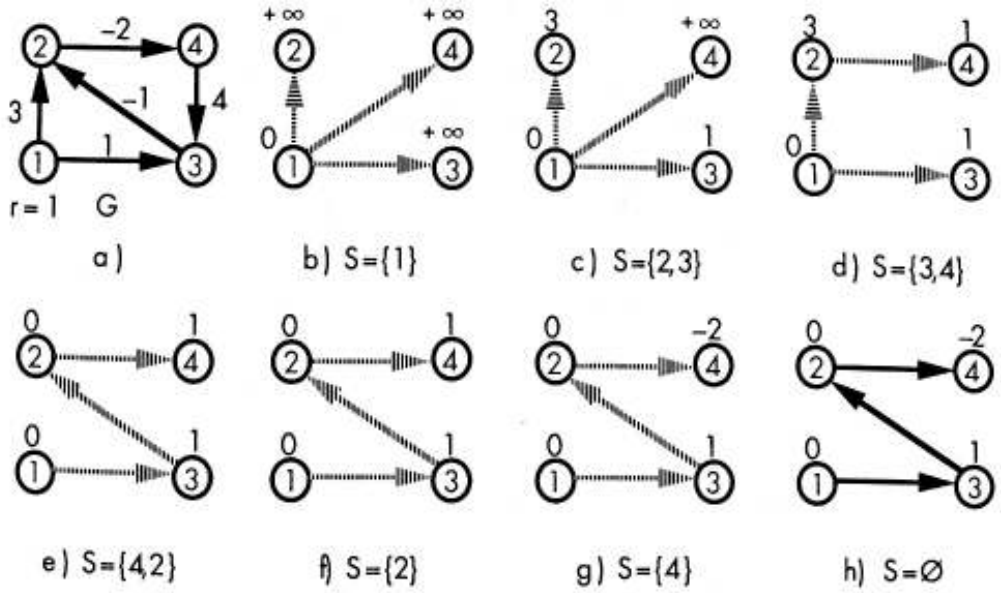
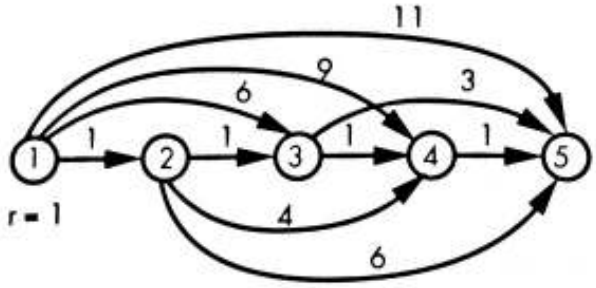


Fig. 11.3 Algoritmo di Bellman-Ford-Moore. a) Un grafo G ($r = 1$); b)-g) estrazione dalla coda dei nodi nell'ordine: 1, 2, 3, 4, 2, 4; h) la soluzione ottima.

11.7
UTILIZZO DI UNA PILA

Benché venga spontaneo pensare che un algoritmo che impieghi una pila anziché una coda debba avere la stessa complessità dell'algoritmo di Bellman-Ford-Moore, l'utilizzo di una pila per la struttura S rende la procedura camminiMinimi() di complessità superpolinomiale.

Si consideri il grafo $G_n = (V_n, E_n)$, con $n \geq 2$, così definito: $V_n = \{1, 2, ..., n\}$, $E_n = \{(u, v): u < v\}$, con lunghezze $w(u, v) = 2^{n-u-1} - 2^{n-v} + v - u$. Si noti che G_n è un grafo aciclico e che le lunghezze sono non negative. Nonostante questa apparente semplicità della struttura di G_n, è possibile dimostrare che, utilizzando una pila, l'estrazione del nodo u da S provoca ogni volta l'inserimento di tutti i nodi k con $k > u$. Pertanto ogni nodo può essere estratto da S per $O(2^n)$ volte. Il fatto più curioso è che l'algoritmo di Bellman-Ford-Moore richiede soltanto un tempo $O(m)$ qualora sia eseguito su G_n!



G_5

pila S 1 2 3 4 5	nodo estratto	distanze d_1 d_2 d_3 d_4 d_5
1 0 0 0 0	1	0 ∞ ∞ ∞ ∞
0 1 1 1 1	5	0 1 6 9 11
0 1 1 1 0	4	0 1 6 9 11
0 1 1 0 1	5	0 1 6 9 10
0 1 1 0 0	3	0 1 6 9 10
0 1 0 1 1	5	0 1 6 7 9
0 1 0 1 0	4	0 1 6 7 9
0 1 0 0 1	5	0 1 6 7 8
0 1 0 0 0	2	0 1 6 7 8
0 0 1 1 1	5	0 1 2 5 7
0 0 1 1 0	4	0 1 2 5 7
0 0 1 0 1	5	0 1 2 5 6
0 0 1 0 0	3	0 1 2 5 6
0 0 0 1 1	5	0 1 2 3 5
0 0 0 1 0	4	0 1 2 3 5
0 0 0 0 1	5	0 1 2 3 4
0 0 0 0 0		0 1 2 3 4

Fig. 11.4 Esecuzione della procedura camminiMinimi() sul grafo G_5 usando una pila.

11.6 ESEMPIO – [Utilizzo di una pila]

La Fig. 11.4 mostra il grafo G_n per $n = 5$, le configurazioni successive della pila S (sotto forma di vettore booleano), i nodi estratti da S e i valori delle distanze ad ogni iterazione dell'algoritmo camminiMinimi() eseguito su G_5 con $r = 1$. Si noti che le configurazioni della pila

corrispondono a tutti i numeri in base 2 compresi tra $16 = 2^{n-1}$ e 0, e che il nodo $n = 5$ viene estratto da S per $2^{n-2} = 8$ volte.

11.8 ALGORITMO DI PAPE-D'ESOPO

In pratica, l'algoritmo che risulta più veloce nei problemi reali impiega per S una struttura detta *dequeue* (per *double ended queue*), cioè una sequenza modificabile ad entrambi gli estremi che combina le proprietà di una pila e di una coda [cfr. Esercizio 4.9].

L'algoritmo che usa la *dequeue*, proposto da Pape (1974), permette di estrarre solo dalla testa di S, ma ammette l'inserzione sia in testa che in fondo. Per la precisione, quando un nodo è inserito in S per la prima volta, allora viene aggiunto in fondo ad S, mentre quando è inserito le volte successive, allora viene aggiunto in testa ad S. In altri termini, quando un nodo v va inserito in S, se $d[v] = +\infty$ allora S è trattata come una coda e v diviene l'ultimo elemento della sequenza, mentre se $d[v] < +\infty$ allora S è trattata come una pila e v diventa il primo elemento della sequenza. La giustificazione di questa modalità di inserimento è la seguente. Ogni volta che la distanza $d[v]$ di un nodo v è decrementata, allora conviene tentare di diminuire anche le distanze dei nodi che seguono v nei cammini che appaiono nell'albero T che rappresenta la soluzione parziale corrente. Questo è lo scopo dell'inserimento in testa quando $d[v] < +\infty$. D'altra parte, per evitare casi patologici con effetto "ping-pong" come si è visto nell'Esempio 11.5, non è conveniente inserire in testa quando la distanza corrisponde ad un cammino nell'albero "fittizio" T dovuto all'inizializzazione. Questo è lo scopo dell'inserimento in fondo quando $d[v] = +\infty$.

L'algoritmo di Pape-D'Esopo si ottiene dal prototipo sostituendo le seguenti righe di codice:

(1) DEQUEUE $S \leftarrow$ DeQueue(); S.insertHead(r)

(2) $u \leftarrow S$.removeHead()

(3) **if** $d[v] = +\infty$ **then** S.insertTail(v) **else** S.insertHead(v)

(4) Sezione non necessaria

11.6 ESEMPIO – [Algoritmo di Pape-D'Esopo]
Eseguendo l'algoritmo di Pape D'Esopo sul grafo G della Fig. 11.3, è facile verificare che l'ordine di estrazione dei nodi da S è: 1, 2, 3, 2, 4. Si noti che rispetto all'algoritmo di Bellman-Ford-Moore il nodo 4 è estratto una sola volta e che il numero complessivo di estrazioni è 5 anziché 6.

Benché nel caso pessimo l'algoritmo di Pape-D'Esopo richieda tempo superpolinomiale come quello che usa una pila, è stato verificato sperimentalmente che nella pratica esso risulta il più veloce di tutti, specie per grafi che rappresentano vere reti di comunicazione stradale. Questi grafi hanno la proprietà di essere sparsi, cioè con m che cresce come $O(n)$, e planari, cioè possono essere disegnati sul piano in modo che le linee corrispondenti a due archi distinti non si sovrappongano mai (dato un grafo G, verificarne la planarità richiede tempo $O(n + m)$ con un algoritmo complicato ideato da Hopcroft e Tarjan (1974); inoltre, se G è planare, è possibile fornirne un disegno planare in tempo polinomiale).

ESEMPIO – [Planarità]
I grafi illustrati nelle Figure 11.2(a) e 11.3(a) sono planari. Il grafo G_5 della Fig. 11.4 invece non lo è, in quanto non è possibile eliminare la sovrapposizione tra gli archi (1, 4) e (3, 5) senza introdurre ulteriori sovrapposizioni tra altre coppie di archi.

11.9
REALITY CHECK

I protocolli OSPF (*Open Shortest Path First*) e IS-IS (*Intermediate System to Intermediate System*) sono protocolli di routing utilizzati nella rete Internet. Sono protocolli *link-state*: ogni router costruisce una mappa completa della connettività della rete, un grafo in cui esiste un arco fra due nodi (router) solo se sono connessi direttamente. Ogni nodo calcola in maniera indipendente il cammino minimo per tutte le possibili destinazioni della rete, utilizzando l'algoritmo di Dijkstra basato su heap (proposto da Johnson).

Questi protocolli sono denominati anche *interior gateway protocol*; in altre parole vengono utilizzati all'interno di singoli *autonomous systems*, ovvero porzioni della rete sotto il controllo di un'unica entità (come, per esempio un Internet Service Provider). Protocolli diversi vengono utilizzati per instradare il traffico fra *autonomous systems* distinti.

11.1 [Algoritmo di Dijkstra] Cosa succede se l'algoritmo di Dijkstra è eseguito su un grafo in cui le lunghezze degli archi possono essere negative?

11.2 [Algoritmo di Bellman-Ford-Moore] Si utilizzi l'algoritmo di Bellman-Ford-Moore per verificare in tempo $O(mn)$ se un grafo possiede cicli negativi.

11.3 [Affidabilità] Sia dato un grafo orientato $G = (V, E)$ nel quale ogni arco $(u, v) \in E$ è associato ad un valore reale $r(u, v)$ preso dall'intervallo $[0, 1]$, che rappresenta l'affidabilità del canale di comunicazione dal vertice u al vertice v, ovvero la probabilità che tale canale trasmetta un messaggio. Si supponga che queste probabilità siano indipendenti. Progettare un algoritmo efficiente per trovare il cammino più affidabile fra due vertici dati, dove l'affidabilità di un cammino è data dal prodotto delle affidabilità degli archi esistenti.

11.4 [Insieme indipendente di intervalli] Dato un insieme di n intervalli della retta reale, si vuole trovarne un insieme indipendente massimo, cioè un sottoinsieme X di intervalli, avente massima cardinalità, tale che ogni coppia di intervalli in X abbia intersezione vuota. Si fornisca un algoritmo polinomiale, formulando il problema come problema di cammini minimi su un opportuno grafo.

11.5 [Insieme dominante di intervalli] Dato un insieme di n intervalli della retta reale, si vuole trovarne un *insieme dominante*, cioè un sottoinsieme X di intervalli, avente minima cardinalità, tale che ogni intervallo che non sta in X abbia intersezione non vuota con almeno un intervallo che sta in X. Si fornisca un algoritmo polinomiale, formulando il problema come problema di cammini minimi su un opportuno grafo.

11.6 [Insieme dominante totale pesato] Si riconsideri il problema dell'Esercizio 11.5 qualora ad ogni intervallo sia associato un peso intero non negativo e si richieda di trovare un insieme dominante totale di peso minimo, dove ogni intervallo (anche se sta in X) deve avere intersezione non vuota con almeno un intervallo che sta in X.

11.7 [Numero di cammini minimi] Dati un grafo orientato con lunghezze positive sugli archi e due nodi s ed t, possono esserci molti cammini minimi distinti fra essi. Si modifichi l'algoritmo di Dijkstra per calcolare il numero dei cammini minimi distinti da s a t.

11.8 [Cammino con arco più corto] Dati un grafo orientato con lunghezze positive sugli archi e due nodi r ed s, si vuole individuare il cammino da r ad s che contiene l'arco più

corto, cioè quello con lunghezza minima tra tutti gli archi che possono apparire nel cammino. Si fornisca un algoritmo per determinare il cammino da r ad s contenente l'arco più corto e se ne analizzi la complessità.

14.9 [Costi sui nodi] Dato un grafo orientato $G = (V, E)$ e una funzione di pesi $w: V \rightarrow \mathbb{R}$ che associa pesi ai nodi, invece che agli archi, il peso di un cammino è dato dalla somma dei pesi dei nodi inclusi nel cammino. Progettare un algoritmo per risolvere il problema dei cammini minimi da un nodo r a tutti gli altri nodi su un grafo pesato sui nodi.

14.10 [Pesi quadrati] Dato un grafo orientato $G = (V, E)$ e una funzione di pesi $w: E \rightarrow \mathbb{R}$ con pesi positivi, sia p un cammino di costo minimo fra una coppia di nodi s e t. Si consideri ora una funzione di pesi w' tale che $w'(u, v) = w(u, v)^2$ per ogni $(u, v) \in E$. Il cammino p è minimo anche nel grafo G pesato da w'? Se sì, fornire una dimostrazione; se no, fornire un controesempio.

11.4 Assumiamo senza perdere in generalità che gli n intervalli $[a_1, b_1], \ldots, [a_n, b_n]$ abbiano tutti i $2n$ estremi distinti e si aggiungano due intervalli fittizi $[a_0, b_0]$ e $[a_{n+1}, b_{n+1}]$, tali che $b_0 < a_u$ e $b_u < a_{n+1}$, per $1 \leq u \leq n$. Si costruisce un grafo orientato aciclico G con $n + 2$ nodi corrispondenti agli $n + 2$ intervalli tale che esiste un arco (u, v) se e solo se $b_u < a_v$. La lunghezza $w(u, v)$ dell'arco (u, v) è -1 se $u \neq 0$, mentre è 0 se $u = 0$. È facile verificare che un cammino minimo tra il nodo 0 ed il nodo $n + 1$ corrisponde ad un insieme indipendente massimo. La costruzione del grafo richiede tempo $O(n^2)$ e la risoluzione del problema di cammini minimi richiede lo stesso tempo poiché il grafo è aciclico (DAG).

11.5 Facciamo le stesse assunzioni dell'Esercizio 11.4, ma cambiamo la costruzione del DAG G. Per ciascun intervallo $[a_u, b_u]$, con $u = 0, 1, \ldots, n$, definiamo

$$P_u = \{k: a_u < a_k < b_u < b_k\}$$
$$Q_u = \{k: a_k > b_u \text{ e } \nexists h \text{ con } b_u < a_h < b_h < a_k\}$$

e introduciamo un arco (u, v) se e solo $v \in P_u \cup Q_u$. La lunghezza $w(u, v)$ dell'arco è 1 se $u > 0$, mentre è 0 se $u = 0$. Un cammino minimo tra il nodo 0 ed il nodo $n + 1$ corrisponde ad un insieme dominante minimo. Si noti che la definizione dell'insieme Q_u permette di evitare di "saltare" un eventuale intervallo $[a_h, b_h]$ che altrimenti resterebbe non dominato. Dopo aver preventivamente ordinato gli intervalli per estremi iniziali crescenti, ciascun P_u e ciascun Q_u si può costruire in tempo $O(n)$. Quindi la costruzione di G richiede tempo $O(n^2)$ così come trovare il cammino minimo perché G è aciclico.

11.6 Con le medesime ipotesi dell'Esercizio 11.5, definiamo gli insiemi P_u e Q_u nello stesso modo, ed inoltre definiamo

$$C_u = \{k: a_u < a_k < b_k < b_u\}.$$

Il grafo G ora contiene $2n + 2$ nodi, poiché ad ogni intervallo originario $[a_u, b_u]$ con $1 \leq i \leq n$ stavolta corrispondono due nodi, indicati con u_{IN} e u_{OUT} (per praticità denotiamo anche il nodo 0 sia con 0_{IN} sia con 0_{OUT}, e lo stesso facciamo col nodo $n + 1$). Indichiamo con w_u il peso di un intervallo, dove $w_0 = w_{n+1} = 0$. Gli archi sono definiti nel modo seguente:

(1) Esiste l'arco (u_{IN}, v_{IN}) con lunghezza w_u se e solo se $v \in P_u$;

(2) Esiste l'arco (u_{IN}, v_{OUT}) con lunghezza w_u se e solo se $v \in Q_u$;

(3) Esiste l'arco (u_{OUT}, v_{IN}) con lunghezza w_u se e solo se $v \in P_u$;

(4) Esiste l'arco (u_{OUT}, v_{OUT}) con lunghezza $w_u + w_k$ se e solo se $v \in Q_u$, $k \in C_u$, e $w_k = \min \{w_h: h \in C_u\}$.

Al solito, un cammino minimo tra il nodo 0 ed il nodo $n + 1$ corrisponde ad un insieme dominante totale di peso minimo ed il problema si risolve in tempo $O(n^2)$. Si noti che lo

dove c è una costante, mentre $D(n)$ e $C(n)$ sono il numero di operazioni per dividere il problema e per combinare i risultati, rispettivamente. Se i dati sono partizionati in maniera bilanciata, cioè tutti gli n_i sono all'incirca uguali, allora l'algoritmo può risultare molto efficiente.

Abbiamo già incontrato algoritmi basati sulla tecnica *divide-et-impera* durante l'introduzione all'analisi degli algoritmi [capitolo 2]. In particolare, abbiamo introdotto la ricerca binaria e l'algoritmo Merge Sort per l'ordinamento. In questo capitolo, raffineremo le nostre conoscenze su questa tecnica studiando le problematiche relative al rapporto fra la parte *divide* e la parte *impera* e a come dividere i problemi in maniera efficiente.

12.1
RICERCA BINARIA E RICERCA PER INTERPOLAZIONE

La procedura binarySearch() introdotta nel capitolo 1 è un esempio particolare di tecnica *divide-et-impera*. Il problema viene suddiviso in due sottoproblemi ($h = 2$) approssimativamente uguali ($n_2 \leq n_1 \leq n_2 + 1$), mentre i costi $C(n)$ e $D(n)$ sono costanti. La particolarità sta nel fatto che la chiamata ricorsiva avviene in solo uno dei sottoproblemi, invece che su entrambi.

Per questo motivo, alcuni autori considerano la ricerca binaria come un cattivo esempio per introdurre il *divide-et-impera*; noi riteniamo invece rilevante discuterne qui perché ci permette di enunciare una regola importante in questi casi: se non tutti i sottoproblemi devono essere analizzati, è buona norma affrontare ricorsivamente i problemi più piccoli possibili.

La complessità della ricerca può essere ulteriormente abbassata tenendo conto dei valori delle chiavi memorizzate nel vettore e della loro distribuzione di probabilità.

12.1 ESEMPIO – [Elenco telefonico]
Volendo ricercare il cognome Cherubini nell'elenco telefonico, non si apre l'elenco a metà, ma all'incirca ad un quarto del numero delle pagine.

Siano le n chiavi presenti nel vettore numeriche ed uniformemente distribuite nell'intervallo $[k_{min}, k_{max}]$. Dovendo ricercare la chiave k nel vettore $A[1 \ldots n]$, anziché provare nella posizione centrale è ragionevole tentare in quella più vicina a:

$$\frac{n(k - k_{min})}{(k_{max} - k_{min})}$$

Infatti, $k - k_{min}$ fornisce lo scarto tra i valori della chiave da ricercare e di quella più piccola, mentre $k_{max} - k_{min}$ dà lo scarto tra i valori della chiave più grande e quella più piccola. Il loro rapporto indica di quanto il valore di k si discosta dai valori estremi. Se $k = k_{min}$ allora il rapporto è 0 e conviene cercare nella prima posizione del vettore, mentre se $k = k_{max}$, il rapporto è 1 e conviene cercare nell'ultima posizione del vettore (la ricerca binaria

ignorando il valore di k, assume tale rapporto 1/2, e tenta in una posizione centrale). Il metodo di ricerca risultante è detto di *interpolazione*.

La procedura di ricerca binaria binarySearch() può essere trasformata in una di interpolazione, ricercaInterpolata(), semplicemente sostituendo l'istruzione

$$m \leftarrow \lfloor (i+j)/2 \rfloor$$

con

$$m \leftarrow i + \lfloor (k - A[i]) \cdot (j - i)/(A[j] - A[i]) \rfloor.$$

Infatti la ricerca viene effettuata in generale sulla porzione $A[i \ldots j]$ del vettore, che contiene la più piccola chiave in $A[i]$ e la più grande in $A[j]$. La formula per m si ricava da quella della ricerca binaria, che era $i + \lfloor (j - i)/2 \rfloor$, sostituendo $(k - A[i])/(A[j] - A[i])$ ad 1/2.

ESEMPIO – [Interpolazione]

Applicando la procedura ricercaInterpolata() al vettore dell'Esempio 1.3, sempre con $v = 13$, si ottiene la seguente lista di chiamate; si noti come, rispetto alla ricerca binaria, siano sufficienti 2 sole chiamate ricorsive anziché 4.

Chiamata	i	j	$A[i]$	$A[j]$	m	$A[m]$
1	1	12	1	42	4	8
2	5	12	13	42	5	13

È possibile dimostrare che, sotto l'ipotesi di uniforme distribuzione delle chiavi, la complessità di ricercaInterpolata() è $O(\log \log n)$. La dimostrazione è piuttosto complicata e non viene qui riportata. La funzione $\log \log n$ cresce molto lentamente (per esempio, vale meno di 5 per n uguale a un miliardo), ma è paragonabile a $\log n$ per n piccolo. Pertanto, se ci sono poche chiavi oppure se le chiavi non sono uniformemente distribuite è più conveniente usare la ricerca binaria.

12.2 QUICKSORT

L'algoritmo praticamente più efficiente per ordinare gli elementi di un vettore è stato proposto da Hoare (1962) e, non a caso, è stato chiamato Quicksort (*quick* significa "svelto"). Questo algoritmo è basato sulla tecnica *divide-et-impera*, ma differisce dal Merge Sort nel modo in cui divide il problema e combina i risultati. La strategia consiste nel selezionare un elemento del vettore, detto *perno*, attorno al quale riarrangiare gli altri elementi. Tutti gli elementi più piccoli del perno vengono spostati in posizioni del vettore che precedono quella del perno, mentre gli elementi più grandi del perno sono spostati in posizioni che la seguono. Lo stesso algoritmo è poi riapplicato alle due porzioni di elementi minori e maggiori del perno.

Sia $A[primo \ldots ultimo]$ la porzione di A che l'algoritmo, ad una generica chiamata, de-

ve ordinare. La seguente procedura QuickSort() richiama al suo interno una procedura perno() che sceglie un elemento perno x e permuta gli elementi trovando un indice j tale che $A[i] \leq x$ per $primo \leq i < j$, $A[j] = x$, e $A[i] \geq x$ per $j < i \leq ultimo$:

```
QuickSort(ITEM[] A, integer primo, integer ultimo)

  if primo < ultimo then
      integer j ← perno(A, primo, ultimo)
      QuickSort(A, primo, j − 1)
      QuickSort(A, j + 1, ultimo)
```

La procedura perno() può scegliere l'elemento x in vari modi. Supponiamo per semplicità che x sia uguale ad $A[primo]$. Gli altri elementi del vettore vengono via via suddivisi in tre sequenze di elementi: quelli minori di x, quelli maggiori o uguali ad x, e quelli non ancora esaminati. La procedura utilizza due cursori: il cursore i punta al primo elemento y non ancora esaminato e il cursore j dà la posizione finale in cui va spostato il perno. Ad ogni iterazione, se $y \geq x$ allora y non viene spostato, mentre se $y < x$ allora y è scambiato con il primo elemento tra quelli maggiori o uguali ad x. Al termine, x è scambiato con $A[j]$.

```
integer perno(ITEM[] A, integer primo, integer ultimo)

  ITEM x ← A[primo]
  integer j ← primo
  for i ← primo to ultimo do
      if A[i] < x then
          j ← j + 1
          A[i] ↔ A[j]
  A[primo] ← A[j]
  A[j] ← x
  return j
```

1	2	3	4	5	6	7	8	9	10
9	12	8	18	6	13	11	3	5	10
i		i							
9	8	12	18	6	13	11	3	5	10
	i			i					
9	8	6	18	12	13	11	3	5	10
		i					i		
9	8	6	3	12	13	11	18	5	10
			i					i	
9	8	6	3	5	13	11	18	12	10
				i					
5	8	6	3	9	13	11	18	12	10

Fig. 12.1 Esecuzione della procedura perno().

12.3 ESEMPIO – [perno()]

La Fig. 12.1 mostra l'esecuzione di perno(A, 1, 10) per i dati d'ingresso: $A[1] = 9$, $A[2] = 12$, $A[3] = 8$, $A[4] = 18$, $A[5] = 6$, $A[6] = 13$, $A[7] = 11$, $A[8] = 3$, $A[9] = 5$ e $A[10] = 10$. Al termine dell'esecuzione, $j = 5$.

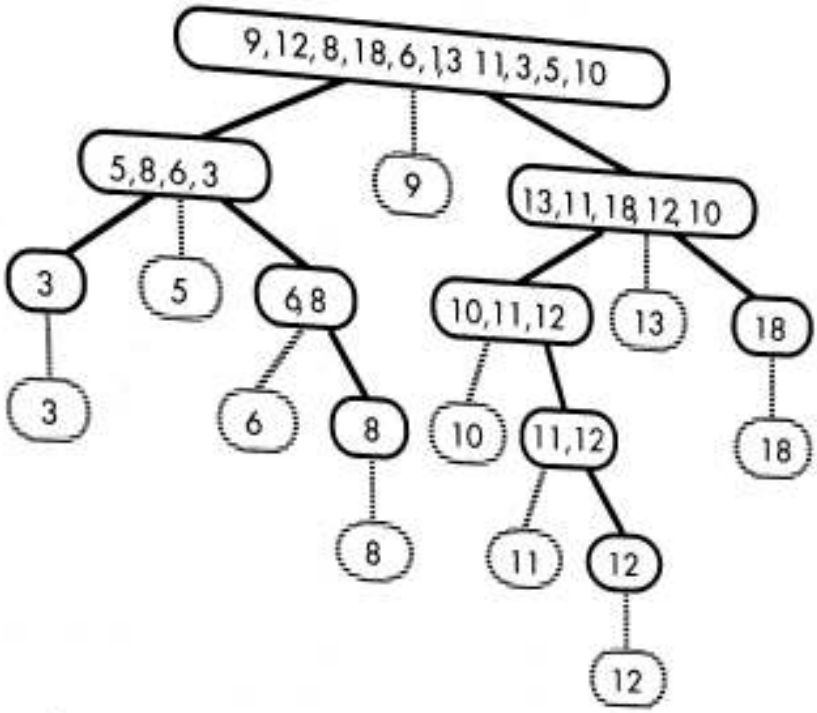


Fig. 12.2 Ordinamento con la procedura QuickSort().

12.4 ESEMPIO – [QuickSort()]

La Fig. 12.2 mostra la partizione dei dati indotta dall'esecuzione della procedura QuickSort() sul vettore A dell'Esempio 12.3.

Per valutare la complessità della procedura QuickSort(), impostiamone le relazioni ricorrenti. Siano $T(n)$ e $P(n)$ il numero di confronti tra elementi di A eseguiti, rispettivamente, dalle procedure QuickSort() e perno(). Poiché $P(n) = n - 1$, e le due porzioni di A sulle quali avviene la chiamata ricorsiva hanno $j - 1$ e $n - j$ elementi, si ottiene:

$$T(n) = \begin{cases} 0 & \text{per } n = 0, 1 \\ n - 1 + T(j-1) + T(n-j) & \text{per } n \geq 2 \end{cases}$$

Nel caso pessimo, $A[primo]$ è sempre l'elemento più piccolo, e quindi la porzione di A con elementi minori di x è vuota, mentre quella con elementi maggiori o uguali ne contiene $n - 1$. Pertanto, $T(n) = n - 1 + T(n - 1)$ e, sostituendo successivamente, si ha: $T(n) = \sum_{2 \leq k \leq n}(k - 1)$, che è $O(n^2)$. Nel caso pessimo, la procedura QuickSort() è lenta proprio come l'Insertion Sort. Paradossalmente, ciò avviene proprio quando il vettore A è già ordinato e non sarebbe necessaria alcuna elaborazione, tranne una verifica del corretto posizionamento degli elementi che richiede solo $O(n)$ tempo (esattamente quello che fa Insertion Sort in questo caso). Questo cattivo comportamento è dovuto ad un forte sbilanciamento dei due insiemi sui quali avvengono le chiamate ricorsive!

12.5 ESEMPIO – [Caso pessimo QuickSort()]

La Fig. 12.3 mostra la partizione dei dati fortemente sbilanciata indotta eseguendo QuickSort() su un vettore già ordinato.

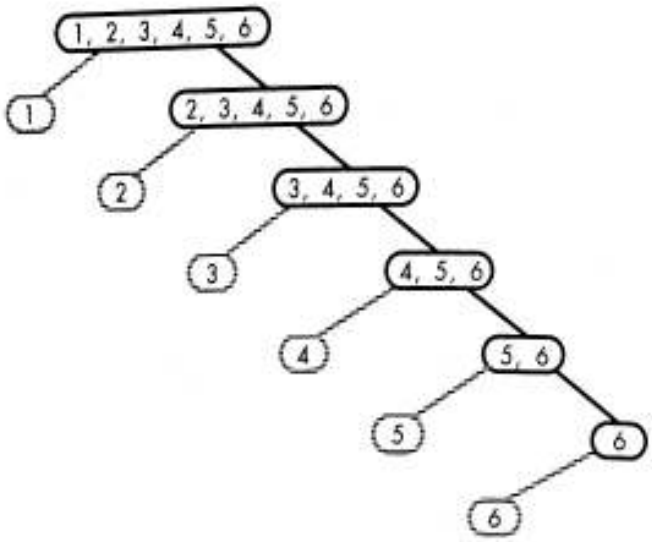


Fig. 12.3 Caso pessimo della procedura QuickSort().

Il termine "Quicksort", comunque, non è immeritato per questo algoritmo, poiché la sua velocità è elevata non nel caso pessimo, ma in quello medio. Per analizzarne la complessità nel caso medio, si supponga per semplicità che tutti gli elementi di A siano distinti, e che tutte le permutazioni degli elementi $A[1 \ldots n]$ siano equiprobabili. Allora, il valore j prodotto dalla procedura perno() può essere con la stessa probabilità $1/n$ in ciascuna posizione compresa tra 1 ed n, e quindi:

$$
\begin{aligned}
T(n) &= 0, && \text{per } n = 0,1,\\
T(n) &= n-1+\frac{1}{n}\sum_{j=1}^{n}(T(j-1)+T(n-j)) && \\
&= n-1+\frac{2}{n}\sum_{j=1}^{n-1}T(j) && \text{per } n \geq 2.
\end{aligned}
$$

Per risolvere la relazione, la si riscriva dapprima per $n-1$, e successivamente si calcoli $T(n)-(n-1)T(n-1)/n$, ottenendo:

$$
\begin{aligned}
T(n-1) &= n-2+\frac{2}{n-1}\sum_{j=1}^{n-2}T(j),\\
\frac{(n-1)T(n-1)}{n} &= \frac{(n-2)(n-1)}{n}+\frac{2}{n}\sum_{j=1}^{n-2}T(j),\\
T(n)-\frac{(n-1)T(n-1)}{n} &= n-1-\frac{(n-2)(n-1)}{n}+\frac{2T(n-1)}{n}.
\end{aligned}
$$

Semplificando, dall'ultima equazione si ricava:

$$
\begin{aligned}
T(n) &= \frac{2(n-1)}{n} + \frac{(n+1)T(n-1)}{n} \le 2 + \frac{(n+1)T(n-1)}{n} \\
&\le 2 + \left[2 + \frac{nT(n-2)}{n-1}\right]\frac{n+1}{n} \\
&= 2 + \frac{2(n+1)}{n} + \frac{(n+1)T(n-2)}{n-1} \\
&\le 2 + \frac{2(n+1)}{n} + \left[2 + \frac{(n-1)T(n-3)}{n-2}\right]\frac{n+1}{n-1} \\
&= 2 + \frac{2(n+1)}{n} + \frac{2(n+1)}{n-1} + \frac{(n+1)T(n-3)}{n-2} \\
&\vdots \\
&\le 2 + 2(n+1)\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right] + \frac{(n+1)T(1)}{2} \\
&= \frac{2(n+1)}{n+1} + 2(n+1)\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right] \\
&= 2(n+1)\sum_{k=3}^{n+1}\frac{1}{k}.
\end{aligned}
$$

Poiché, come illustrato nella Fig. 12.4, si ha che

$$\sum_{k=3}^{n+1}\frac{1}{k} < \int_2^{n+1}\frac{1}{x}dx = \ln(n+1) - \ln 2$$

$T(n)$ risulta essere $O(n \log n)$ e la procedura QuickSort() è quindi ottima nel caso medio!

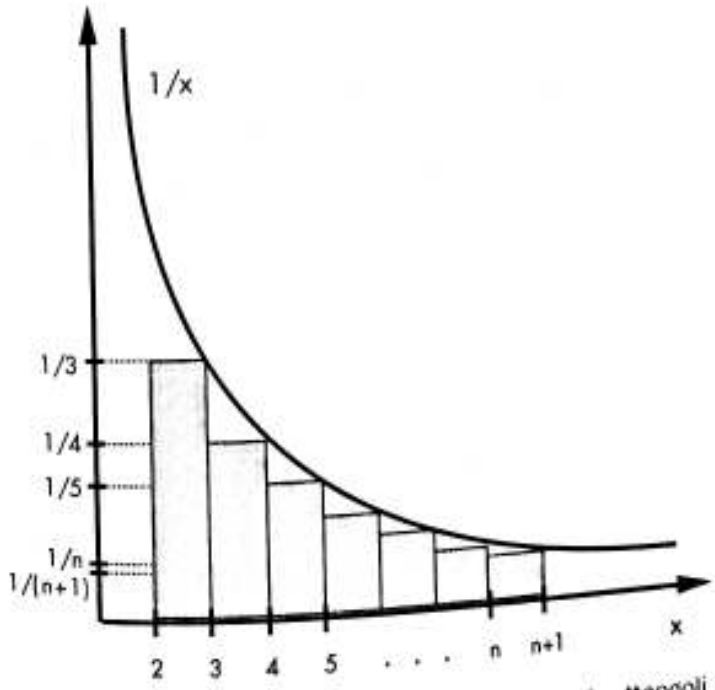


Fig. 12.4 Maggiorazione di $1/3 + 1/4 + \cdots + 1/n + 1/(n+1)$ (area dei rettangoli grigi) con $\int_2^{n+1}\frac{1}{x}dx$ (area sottesa dalla curva $1/x$ per $2 \le x \le n+1$).

Intuitivamente, il perno avrà in media un valore centrale tra $A[primo]$, $A[primo+1]$, ...,

A[*ultimo*] e la procedura sarà richiamata ricorsivamente su due sottoinsiemi all'incirca della stessa dimensione. In pratica, se si adottano alcuni "accorgimenti", la procedura QuickSort() non solo è veloce, ma è addirittura la più veloce tra tutti gli algoritmi di ordinamento noti finora per n abbastanza "grande" (per un fattore costante, ovviamente, non in ordine di grandezza, ché MergeSoft() e Heapsort() sono anch'esse asintoticamente ottime)! I principali accorgimenti, che possono essere combinati tutti insieme per esercizio ottenendo una nuova versione della procedura, sono i seguenti:

- *scelta del perno*: un buon criterio di scelta del perno x consiste nel considerare i tre elementi *A*[*primo*], *A*[(*primo* + *ultimo*)/2] e *A*[*ultimo*], e sceglierne il mediano, cioè quello che non è né il massimo né il minimo dei tre (dopo averlo scambiato con *A*[*primo*], naturalmente);
- *procedura* perno(): un metodo più veloce, ma meno intuitivo, di realizzare la procedura perno() consiste nell'utilizzare due cursori i e j per scorrere gli elementi di A, rispettivamente, da sinistra verso destra e da destra verso sinistra: i e j sono spostati finché non valgono entrambe $x > A[i]$ e $x < A[j]$, e a questo punto $A[i]$ e $A[j]$ sono scambiati tra loro; inoltre, il tempo perso per la chiamata della procedura può essere risparmiato scrivendo le istruzioni della perno() direttamente nella procedura QuickSort();
- *condizione di terminazione*: poiché il tempo perso nella gestione della ricorsione è elevato quando il numero di elementi da ordinare è piccolo, è opportuno scegliere una dimensione minima di A, per esempio 10, al di sotto della quale l'ordinamento è effettuato con una semplice procedura non ricorsiva (per esempio con shellSort(), che sarà illustrata nel capitolo 15);
- *gestione esplicita della ricorsione*: per abbassare il tempo necessario alla gestione della ricorsione, è opportuno gestire la ricorsione esplicitamente con una pila, come illustrato nel capitolo 4.4, avendo cura di salvare nella pila la minima informazione necessaria; per far questo, si possono salvare nella pila solo gli "estremi" di una porzione di A, ordinando direttamente l'altra porzione;
- *occupazione di memoria*: poiché nel caso pessimo la partizione di A può essere sbilanciata, la porzione di A i cui estremi sono salvati nella pila può essere sempre la più piccola e la pila richiedere così $O(n)$ spazio; per ridurre la dimensione della pila, basta salvare nella pila gli estremi della porzione più grande, e ordinare direttamente quella più piccola; in questo modo, l'occupazione di memoria della pila diviene $O(\log n)$.

12.3 MOLTIPLICAZIONE DI POLINOMI

Siano dati due polinomi $p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \ldots + p_1x + p_0$ e $q(x) = q_{n-1}x^{n-1} + q_{n-2}x^{n-2} + \ldots + q_1x + q_0$, ciascuno avente n coefficienti (e quindi grado $n - 1$). Tenendo conto che il polinomio prodotto $r(x)$ ha un numero doppio di coefficienti, è agevole scrivere un algoritmo banale di moltiplicazione che ha complessità $\Theta(n^2)$.

```
moltiplicaPolinomi(real[] p, real[] q, real[] r, integer n)
  for h ← 0 to 2 · (n − 1) do r[h] ← 0
  for h ← 0 to n − 1 do
  |   for k ← 0 to n − 1 do
  |   |   r[h + k] ← r[h + k] + p[h] · q[k]
```

Per abbassare la complessità, è opportuno impiegare la tecnica divide-et-impera, grazie alla quale si riescono ad utilizzare soltanto tre prodotti di polinomi di grado $n/2 - 1$ nella ricombinazione dei risultati dei sottoproblemi.

Si supponga per semplicità che il numero n di coefficienti sia una potenza di 2. Si può dividere il polinomio $p(x)$ di grado $n - 1$ in due polinomi, ciascuno avente $n/2$ coefficienti:

$$p^S(x) = p_{n-1}x^{n/2-1} + p_{n-2}x^{n/2-2} + \cdots + p_{n/2+1}x + p_{n/2}$$
$$p^D(x) = p_{n/2-1}x^{n/2-1} + p_{n/2-2}x^{n/2-2} + \cdots + p_1x + p_0$$

Dividendo anche $q(x)$ nello stesso modo, si ha:

$$p(x) = x^{n/2}p^S(x) + p^D(x)$$
$$q(x) = x^{n/2}q^S(x) + q^D(x)$$

e il prodotto $r(x) = p(x)q(x)$ può essere scritto come:

$$r(x) = x^n p^S(x)q^S(x) + x^{n/2}[p^D(x)q^S(x) + q^D(x)p^S(x)] + p^D(x)q^D(x)$$

Calcolando una volta per tutte, con solo tre prodotti, i polinomi:

$$r^S(x) = p^S(x)q^S(x)$$
$$r^D(x) = p^D(x)q^D(x)$$
$$r^M(x) = [p^S(x) + p^D(x)][q^S(x) + q^D(x)]$$

il prodotto $r(x)$ si può ottenere nel modo seguente:

$$r(x) = x^n r^S(x) + x^{n/2}[r^M(x) - r^D(x) - r^S(x)] + r^D(x)$$

Questa formula permette di calcolare il prodotto tra due polinomi di dimensione n risolvendo tre sottoproblemi di dimensione $n/2$, e addizionando tra loro polinomi per "preparare" i sottoproblemi e "combinare" le soluzioni.

```
moltiplicaPolinomiD&I(real[] p, real[] q, real[] r, integer n)
  if n = 1 then
  |   r[0] ← p[0] · q[0]
  else
  |   ripartisci p in p^S e p^D
  |   ripartisci q in q^S e q^D
  |   calcola r^S, r^M ed r^D richiamando moltiplicaPolinomiD&I()
  |   ricava r in funzione di r^S, r^M ed r^D
```

Osservando che la somma di polinomi ed il prodotto per $x^{n/2}$ e x^n richiedono $O(n)$ operazioni e che per $n = 1$ il problema si riduce ad una singola moltiplicazione tra due coefficienti costanti, il tempo $T(n)$ richiesto dall'algoritmo verifica le seguenti relazioni di ricorrenza:

$$T(n) = \begin{cases} c & \text{per } n = 1 \\ 3T(n/2) + dn & \text{per } n = 2^h \text{ con } h > 0 \end{cases}$$

dove c e d sono costanti. Essendo $\alpha = \log 3/\log 2 \approx 1.59 > 1 = \beta$, la complessità $T(n)$ dell'algoritmo è $O(n^{1.59})$.

È possibile adattare questo algoritmo per moltiplicare due numeri interi di n bit, con la stessa complessità $O(n^{1.59})$ invece che $O(n^2)$. L'algoritmo è dovuto a Karatsuba (1962). Tanto per curiosità, l'algoritmo descritto per la moltiplicazione di polinomi non è il migliore possibile. Si possono infatti moltiplicare i due polinomi in tempo ottimo $O(n \log n)$ usando trasformate ed antitrasformate di Fourier, che a loro volta possono essere calcolate con lo stesso tempo utilizzando algoritmi basati anch'essi sulla tecnica *divide-et-impera*.

12.4 MOLTIPLICAZIONE DI MATRICI

Si consideri il problema di moltiplicare tra loro due matrici quadrate $n \times n$ $A = [a_{ij}]$ e $B = [b_{ij}]$. Per definizione, la matrice prodotto $C = AB$ ha il generico elemento $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, per $1 \leq i \leq n$ ed $1 \leq j \leq n$. Il problema può essere risolto facilmente con la procedura moltiplicaMatrici().

```
moltiplicaMatrici(real[][] A, real[][] B, real[][] C, integer n)
for i ← 1 to n do
    for j ← 1 to n do
        C[i, j] ← 0
        for k ← 1 to n do
            C[i, j] ← C[i, j] + A[i, k] · B[k, j]
```

Questo banale algoritmo ha complessità $O(n^3)$. Vediamo come ridurre la complessità utilizzando la tecnica *divide-et-impera*. Assumendo per semplicità $n = 2^h$, la matrice A può essere ripartita in quattro sottomatrici A_{11}, A_{12}, A_{21} e A_{22}, ciascuna di dimensione $n/2 \times n/2$. Ripartendo in modo analogo anche B e C, poiché

$$c_{ij} = \sum_{k=1}^{n/2} a_{ik} b_{kj} + \sum_{k=n/2+1}^{n} a_{ik} b_{kj}$$

ogni sottomatrice C_{ij} della matrice prodotto può essere espressa come somma delle sottomatrici seguenti:

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} \quad \text{per } 1 \le i \le 2 \text{ ed } 1 \le j \le 2$$

Si definiscano adesso le sette matrici $X_1, ..., X_7$, ciascuna delle quali ha dimensione $n/2 \times n/2$ e può essere calcolata con un solo prodotto tra matrici:

$$\begin{aligned}
X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
X_2 &= (A_{21} + A_{22})B_{11} \\
X_3 &= A_{11}(B_{12} - B_{22}) \\
X_4 &= A_{22}(B_{21} - B_{11}) \\
X_5 &= (A_{11} + A_{12})B_{22} \\
X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
X_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

Le matrici C_{ij} possono essere ricavate in funzione delle matrici $X_1, ..., X_7$ come segue:

$$\begin{aligned}
C_{11} &= X_1 + X_4 - X_5 + X_7 \\
C_{12} &= X_3 + X_5 \\
C_{21} &= X_2 + X_4 \\
C_{22} &= X_1 + X_3 - X_2 + X_6
\end{aligned}$$

L'algoritmo risultante, proposto da Strassen (1969), può essere riassunto informalmente con la procedura strassen().

```
strassen(real[][] A, real[][] B, real[][] C, integer n)
  if n = 1 then
  |   C[1, 1] ← A[1, 1] · B[1, 1]
  else
  |   ripartisci A in A11, A12, A21 e A22
  |   ripartisci B in B11, B12, B21 e B22
  |   calcola X1, ..., X7 richiamando strassen()
  |   ricava C11, C12, C21 e C22 in funzione di X1, ..., X7
```

Poiché le somme di matrici di dimensione $n/2 \times n/2$ richiedono $O(n^2)$ operazioni, e la procedura richiama se stessa sette volte per calcolare $X_1, ..., X_7$, il tempo $T(n)$ richiesto dall'algoritmo verifica le seguenti relazioni di ricorrenza, dove c e d sono costanti positive:

$$T(n) = \begin{cases} d & \text{per } n = 1 \\ 7T(n/2) - cn^2 & \text{per } n = 2^h,\ h > 0 \end{cases}$$

Applicando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene $\alpha = \log 7/\log 2 \approx 2.81 > 2 = \beta$. Pertanto, $T(n)$ risulta essere $O(n^{2.81})$.

L'algoritmo di Strassen è valido per n qualsiasi, non necessariamente per $n = 2^k$. Basta infatti bordare le matrici A e B con righe e colonne di zeri fino ad ottenere $n = 2^k$ e l'ordine di grandezza di $T(n)$ non cambia. Inoltre, il metodo può essere adattato per calcolare, con la stessa complessità, il determinante e l'inversa di una matrice. L'algoritmo è efficiente per n grande, mentre per n piccolo l'algoritmo banale di complessità $O(n^3)$ può risultare più veloce. Ciò è dovuto alle costanti "nascoste" nell'ordine di grandezza della complessità, che sono elevate per l'algoritmo di Strassen. Sono noti persino algoritmi asintoticamente più efficienti di quello di Strassen (il più veloce ha complessità $O(n^{2.376})$, mentre una limitazione inferiore alla complessità del problema è ovviamente $\Omega(n^2)$), ma tali algoritmi non sono di utilità pratica a causa delle enormi costanti moltiplicative coinvolte negli ordini di grandezza delle funzioni di complessità.

12.5 REALITY CHECK

Poiché una limitazione inferiore al problema dell'ordinamento è $\Omega(n \log n)$, MergeSort() è ottimo sempre, mentre Quicksort() è ottimo nel caso medio, ma è $O(n^2)$ nel caso pessimo. Tuttavia, fra i due algoritmi si preferisce spesso Quicksort(), perché le sue costanti moltiplicative sono più basse.

Oltre all'efficienza, esistono ulteriori parametri per valutare gli algoritmi di ordinamento. Uno di questi è la *stabilità* dell'ordinamento: supponendo di ordinare coppie (chiave, valore), e che le chiavi possano apparire più di una volta, un algoritmo di ordinamento è stabile se, date due coppie con chiave uguale (k, v_1) e (k, v_2), tali che (k, v_1) appare prima di (k, v_2) nella sequenza da ordinare, allora quest'ordine viene rispettato nella sequenza ordinata. Fra gli algoritmi discussi in questo testo, Selection Sort, Heapsort, Quicksort e Shell Sort [capitolo 15] non sono stabili, mentre lo sono Insertion Sort, Merge Sort e Pigeonhole Sort [cfr. Esercizio 4.11].

Un altro parametro di valutazione è l'utilizzo di memoria aggiuntiva, oltre a quella necessaria per memorizzare l'input. Selection Sort, Insertion Sort, Heapsort e Shell Sort sono algoritmi iterativi e utilizzano $O(1)$ memoria aggiuntiva (solo lo spazio necessario per le variabili della procedura); Quicksort richiede in media $O(\log n)$ spazio aggiuntivo (lo spazio necessario per le chiamate ricorsive); mentre Merge Sort e Pigeonhole Sort richiedono spazio $O(n)$, anche se è possibile realizzare versioni iterative di Merge Sort. Gli algoritmi che non necessitano $O(n)$ spazio aggiuntivo sono comunemente detti *in-place*.

Comunque, algoritmi di ordinamento basati sulla tecnica Merge Sort risultano vantaggiosi qualora gli elementi da ordinare risiedano in memoria secondaria (per esempio su disco). In tal caso, il tempo per trasferire avanti e indietro dati tra la memoria principale, che è veloce ma poco capiente, e la memoria secondaria, che viceversa è capiente ma lenta, supera di gran lunga il tempo richiesto per effettuare l'ordinamento nella memoria principale. È quindi conveniente elaborare i dati in blocchi sequenziali, proprio come viene fatto dalla procedura Merge(). Pertanto, la maggior parte degli algoritmi di ordinamento su memoria secondaria si basa su una tecnica "Merge Sort polifase", distribuendo dapprima gli elementi da ordinare in molte sequenze ordinate (dette *corse*) memorizzate in memoria secondaria e poi eseguendo ripetutamente operazioni di fusioni tra corse finché non rimane che un'unica corsa.

Esistono competizioni per stabilire il più veloce algoritmo di ordinamento per grandi (enormi!) quantità di dati[1]. Per esempio, nel 2009 un programma sviluppato dall'Università di Padova è stato in grado di ordinare oltre 2 miliardi di chiavi da 100 byte, per un totale di 223 Gigabyte, in meno di un minuto, su un elaboratore "normale" (2.6 Ghz AMD Athlon, 4 GB RAM, 5 dischi da 160 GB - 7200 RPM Sata).

[1] http://sortbenchmark.org

ESERCIZI

12.1 [Ordinamento su lista] Si riscrivano le procedure MergeSort() e QuickSort() nel caso in cui la sequenza da ordinare sia contenuta in una lista realizzata con puntatori, mantenendo in entrambi i casi la stessa complessità $O(n \log n)$.

12.2 [Massima somma di sottovettori] Progettare un algoritmo che, preso un vettore A di n interi (positivi o negativi), trovi un sottovettore (una sequenza di elementi consecutivi del vettore) la somma dei cui elementi sia massima.

12.3 [Inversioni] Data una sequenza di n interi memorizzata in un vettore A, si ha un'inversione tra gli elementi $A[i]$ e $A[j]$ se $i < j$ e $A[i] > A[j]$. Si modifichi la procedura MergeSort() per contare il numero totale di inversioni presenti in A.

12.4 [Analisi Quicksort] Si analizzi la complessità di QuickSort() nel caso in cui tutti gli elementi del vettore da ordinare abbiano lo stesso valore.

12.5 [Merge Sort iterativo] Si fornisca una versione iterativa della procedura MergeSort() senza usare una pila e mantenendone la complessità di $O(n \log n)$.

12.6 [Ricerca in matrice ordinata] Sia data una matrice $n \times n$ di interi in cui gli elementi in ogni riga ed ogni colonna sono ordinati in modo crescente. Si progetti un algoritmo di tipo *divide-et-impera* per determinare se un dato intero k è contenuto nella matrice e se ne discuta la complessità.

12.7 [Torneo di tennis] Si progetti un algoritmo *divide-et-impera* per determinare il calendario di un torneo di tennis, cui partecipano $n = 2^k$ giocatori, in modo che ciascun giocatore incontri una ed una sola volta tutti gli altri, disputi un incontro al giorno, e la durata del torneo sia di $n - 1$ giorni.

12.8 [Prodotto di numeri complessi] Si individui un algoritmo per moltiplicare due numeri complessi usando solo tre operazioni aritmetiche di moltiplicazione anziché quattro.

12.9 [Conta gli zeri] Un vettore V di n numeri binari contiene tutti gli zeri nelle prime posizioni, seguiti da tutti gli uni. Si scrivano due funzioni, basate sulla ricerca binaria, per determinare il numero h di zeri in tempo $O(\log n)$ e $O(\log h)$, rispettivamente.

12.40 [Minimo in una matrice] Data una matrice $n \times n$ di interi con n potenza di 2, si vuole determinarne l'elemento minimo. Si scriva una funzione utilizzando la tecnica *divide-et-impera*.

12.41 [Chi manca?] Sia dato un vettore ordinato $V[1 \ldots n]$ contenente n elementi interi distinti appartenenti all'intervallo $1 \ldots n + 1$. Si scriva una procedura, basata sulla ricerca binaria, per individuare in tempo $O(\log n)$ l'unico intero dell'intervallo $1 \ldots n + 1$ che non compare in V.

12.42 [Massimo in sequenza unimodale] Una sequenza $a_1, a_2, \ldots, a_n$ di n elementi distinti è unimodale se esiste un indice i, $1 \leq i \leq n$, tale che $a_1 < a_2 < \ldots < a_i$ e $a_i > a_{i+1} > \ldots > a_n$ (per $i = 1$ oppure $i = n$, allora la sequenza unimodale è anche ordinata). Si scriva una funzione che trovi in tempo $O(\log n)$ il massimo di una sequenza unimodale memorizzata in un vettore.

12.43 [Occorrenze] Dato in input un vettore ordinato di n interi e un intero k, scrivere un algoritmo che restituisca in tempo $O(\log n)$ il numero di occorrenze di k in A.

12.2 Innanzitutto, è importante notare che il vettore di input deve contenere valori negativi, altrimenti la risposta è data dal vettore completo.
Un algoritmo banale e poco efficiente per risolvere questo problema consiste nel considerare tutti gli $n(n+1)/2$ sottovettori, calcolare la loro sommatoria e restituire il valore più alto. Un algoritmo del genere ha costo $O(n^3)$. Si noti che il massimo viene inizializzato a zero; infatti, se tutti i valori sono negativi un sottovettore vuoto ha somma zero ed è preferibile.

```
integer maxsum(integer[] A, integer n)
  integer max ← 0                                  % Massimo valore trovato
  for i ← 1 to n do
      for j ← i to n do
          integer sum ← 0                          % Somma sottovettore
          for k ← i to j do
              sum ← sum + A[k]
          if sum > max then max ← sum
  return max
```

Un algoritmo migliore si può ottenere notando che la somma di $A[i...j]$ è uguale alla somma di $A[i...j-1]$ più $A[j]$. L'algoritmo corrispondente ha costo $O(n^2)$. Questa semplice ottimizzazione ha il notevole effetto di ridurre la complessità di un ordine di grandezza; sebbene sia possibile interpretarla come un'applicazione della tecnica della programmazione dinamica, che vedremo solo nel capitolo successivo, in realtà questa ottimizzazione è una semplice applicazione del buon senso.

```
integer maxsum(integer[] A, integer n)
  integer max ← 0                                  % Massimo valore trovato
  for i ← 1 to n do
      integer sum ← 0                              % Somma sottovettore
      for k ← i to j do
          sum ← sum + A[k]
          if sum > max then max ← sum
  return max
```

Un approccio basato su divide-et-impera divide il vettore $A[i...j]$ in due parti $A[i...m]$, $A[m+1 ... j]$, con $m = \lfloor (i+j)/2 \rfloor$. Ricorsivamente, viene calcolato:

- max_s, il valore massimo fra tutti i sottovettori contenuti nella parte sinistra $A[i...m]$
- max_d, il valore massimo fra tutti i sottovettori contenuti nella parte destra $A[m+1 ... j]$

Inoltre, viene calcolato iterativamente il massimo sottovettore che sta a cavallo fra la parte sinistra e la parte destra, ottenuto calcolando:

- max'_s, il valore massimo fra tutti i sottovettori contenuti nella parte sinistra e confinanti con la parte destra (ovvero che terminano in $A[m]$);

– max'_d, il valore massimo fra tutti i sottovettori contenuti nella parte destra e confinanti con la parte sinistra (ovvero che iniziano in $A[m + 1]$);

A questo punto, il valore finale è dato dal massimo fra max_d, max_s e $max'_d + max'_s$. Come caso base, si e considerano vettori costituiti da 0 elementi (il cui valore è 0) o da un elemento (il cui valore è il massimo fra l'elemento stesso e 0, in modo da escludere valori negativi).

La chiamata iniziale è maxsumRic(A, 1, n); il costo computazionale è $O(n \log n)$.

```
integer maxsumRic(integer[ ] A, integer i, integer j)
  integer max_s, max_d, max'_s, max'_d, s, m, k
  if i > j then return 0
  if i = j then return max(0, A[i])
  m ← ⌊(i + j)=2⌋
  max_s ← maxsumRic(A, i, m)
  max_d ← maxsumRic(A, m + 1, j)
  sum ← 0
  max'_d ← 0
  for k ← m downto i do
      sum ← sum + A[k]
      if sum > max'_s then max'_s ← sum
  sum ← 0
  max'_s 0
  for k ← m + 1 to j do
      sum ← sum + A[k]
      if sum > max'_d then max'_d ← sum
      return max(max_s, max_d, max'_s, max'_d)
```

È possibile trovare una soluzione ancora più efficiente, che opera in tempo $O(n)$, ma rimandiamo la soluzione al capitolo successivo, quando introdurremo la programmazione dinamica.

10.3 Osservando la procedura Merge(), si nota che ogniqualvolta $A[i] > A[j]$ nel primo **if**, allora tutti gli elementi $A[i]$, $A[i + 1]$, ..., $A[mezzo]$ provocano un'inversione con $A[j]$, per un totale di $mezzo - i + 1$ inversioni. Pertanto, si può scrivere una funzione Contamerge(), ottenuta dalla Merge() aggiungendo un nuovo parametro intero *conta*, inizializzato a 0 e restituito come risultato della funzione, e modificando il primo **if** come segue:

```
if A[i] ≤ A[j] then
    B[k] ← A[i]
    i ← i + 1
else
    B[k] ← A[j]
```

```
    j ← j + 1
    conta ← conta + mezzo − i + 1
```

La risultante funzione Inversioni(), che conta il numero totale di inversioni di A, è ottenuta banalmente dalla procedura MergeSort() ed ha anch'essa complessità $O(n \log n)$.

```
integer Inversioni(ITEM[] A, integer primo, integer ultimo)

  if primo ≥ ultimo then
  |   return 0
  else
  |   mezzo ← ⌊(primo + ultimo)/2⌋
  |   return Inversioni(A, primo, mezzo) + Inversioni(A, mezzo + 1, ultimo) +
  |   Contamerge(A, primo, ultimo, mezzo)
```

12.6 Una limitazione inferiore di $\Omega(n)$ si può dimostrare applicando la tecnica dell'oracolo. Si supponga di avere una matrice $M[1 \ldots n, 1 \ldots n]$ con righe e colonne ordinate in modo crescente con elementi uguali a 0 oppure 1 sulla diagonale secondaria, con elementi negativi nel triangolo sopra la diagonale secondaria, ed elementi maggiori di 1 nel triangolo sotto la diagonale secondaria:

$$M[i,j]\begin{cases} < 0 & \text{per } i < n-j+1 \\ = 0 \text{ oppure } 1 & \text{per } i = n-j+1 \\ > 1 & \text{per } i > n-j+1 \end{cases}$$

Se l'elemento k da cercare è 1 e c'è un solo 1 nella matrice, tale elemento può occupare una posizione qualsiasi sulla diagonale secondaria, pertanto un qualsiasi algoritmo deve necessariamente scandirla tutta.

La funzione ricerca() è una generalizzazione sulle matrici della ricerca binaria per i vettori ordinati. Si comincia la ricerca di k a partire dalla posizione in alto a destra $M[1, n]$. Si esclude dalla ricerca successiva la colonna n, qualora $M[1, n] > k$, oppure la riga 1, se $M[1, n] < k$. La ricerca viene poi riapplicata ricorsivamente alla sottomatrice rettangolare $n \times (n-1)$ (oppure $(n-1) \times n$) privata dell'ultima colonna (oppure della prima riga). La chiamata iniziale è ricerca(M, 1, n, k, n). Ad ogni chiamata si esclude o una riga o una colonna di M. Poiché M ha n righe ed n colonne, ricerca() ha complessità ottima $O(n)$.

```
boolean ricerca(ITEM[][] M, integer i, integer j, integer k, integer n)

  if M[i, j] = k then
  |   return true
  else if M[i, j] < k then
  |   if i < n then return ricerca(M, i + 1, j, k, n)
  |   else return false
  else
  |   if j > 1 then return ricerca(M, i, j − 1, k, n)
  |   else return false
```

Si assuma che i giocatori siano indicati con numeri interi da 1 ad n. Si costruisce una matrice $n \times n$, dove la prima colonna contiene i giocatori in ordine crescente e nelle rimanenti $n - 1$ colonne si trovano tutti i giocatori che ognuno di essi dovrà incontrare in ciascuna giornata del calendario. In altri termini, la generica riga i della matrice contiene proprio i nella prima colonna e il giocatore che i deve incontrare nella giornata $j - 1$ nella colonna $j > 1$. Per esempio, per $n = 2$ si ottiene la matrice:

1	2
2	1

mentre per $n = 4$ si ha la seguente matrice, ottenuta combinando opportunamente quattro matrici per $n = 2$:

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

In generale, denotata con $M_{i,j}$ la matrice per i giocatori compresi tra i e j, la matrice $M_{1,n}$ si ottiene combinando $M_{1,\frac{n}{2}}$ e $M_{\frac{n}{2}+1,n}$ nel modo seguente:

$M_{1,\frac{n}{2}}$	$M_{\frac{n}{2}+1,n}$
$M_{\frac{n}{2}+1,n}$	$M_{1,\frac{n}{2}}$

A questo punto viene spontaneo progettare la seguente procedura torneo() che costruisce la matrice $M_{1,n}$ riapplicando ricorsivamente la tecnica alle quattro sottomatrici, fino a quando non viene raggiunto il caso di due soli giocatori. Oltre alla matrice M, la procedura necessita di altri sei parametri, gli indici minimi e massimi di riga e colonna, che individuano la porzione di M sulla quale avviene la generica chiamata ricorsiva, e gli indici minimi e massimi dei giocatori. La procedura va chiamata nel programma principale con torneo(M, 1, n, 1, n, 1, n).

```
torneo(ITEM[][] M, integer imin, imax, jmin, jmax, gmin, gmax)
  if gmin = gmax − 1 then
      M[imin, jmin] ← gmin
      M[imin, jmax] ← gmax
      M[imax, jmin] ← gmax
      M[imax, jmax] ← gmin
  else
      integer imed ← ⌊(imin + imax)/2⌋
      integer jmed ← ⌊(jmin + jmax)/2⌋
      integer gmed ← ⌊(gmin + gmax)/2⌋
      torneo(M, imin, imed, jmin, jmed, gmin, gmed)
      torneo(M, imed + 1, imax, jmed + 1, jmax, gmin, gmed)
      torneo(M, imed + 1, imax, jmin, jmed, gmed +1, gmax)
      torneo(M, imin, imed, jmed + 1, jmax, gmed + 1, gmax)
```

12.9 La funzione di complessità $O(\log n)$ esegue una ricerca binaria considerando il valore $V[m]$ dell'elemento centrale. Se $V[m] = 1$ viene fatta una chiamata ricorsiva sulla prima metà $V[i \ldots m-1]$, mentre se $V[m] = 0$ è effettuata una chiamata sulla seconda metà $V[m+1 \ldots j]$. Per evitare lo "scavalcamento" degli indici i e j, la ricorsione è arrestata quando la porzione ancora da esaminare comprende o un solo elemento (caso $i = j$) oppure due elementi (caso $i = j - 1$). In entrambi i casi, se l'elemento sinistro $V[i]$ è 1, allora l'ultimo 0 si trova in posizione $h = i - 1$ (se non c'è alcuno 0 allora deve essere $i = 1$ e correttamente $h = i - 1 = 0$); se invece l'elemento destro $V[j]$ è 0, allora $h = j$ (se ci sono tutti zeri, allora deve essere $j = n$ e quindi $h = j = n$). Infine, la terza possibilità occorre solo se ci sono esattamente due elementi (cioè quando $i = j - 1$) dei quali il sinistro non è 1 ed il destro non è 0; ma in tal caso deve essere $V[i] = 0$ e $V[j] = 1$, ed il numero h di zeri è i. La chiamata nel programma principale è conta0(V, 1, n).

Per contare il numero h di zeri in tempo $O(\log h)$, si scandisce con un ciclo **while** il vettore V da sinistra verso destra per potenze di due. Se l'elemento considerato $V[i]$ è uguale a 0, si procede nella scansione, raddoppiando i, ma evitando di superare la dimensione n del vettore. Se al termine della scansione l'ultimo elemento $V[i]$ esaminato vale 0, allora si chiama la conta0() tra tale elemento e l'ultimo elemento $V[n]$ del vettore. Altrimenti si chiama la conta0() tra l'ultimo elemento $V[i]$ esaminato (che vale 1) e l'elemento $V[i/2]$ esaminato in precedenza, se $i/2$ non è inferiore ad 1, oppure $V[1]$. La chiamata è trovaH(V, n).

Poiché la scansione procede raddoppiando ogni volta i finché non è individuato un intervallo tra due potenze di 2 che include h (diciamo $2^s \leq h \leq 2^{s+1}$), il ciclo **while** costa $O(\log 2^s) = O(\log h)$. La conta0() viene poi eseguita sugli elementi appartenenti a tale intervallo, il cui numero è $O(2^s) = O(h)$, e quindi la sua complessità è $O(\log 2^s) = O(\log h)$. Ovviamente, questo metodo è vantaggioso quando h è molto minore di n.

```
integer conta0(integer[] V, integer i, j)
  if i = j or i = j − 1 then
  |   while V[i] ≠ 1 and i ≤ j do i ← i + 1
  |   return i − 1
  else
  |   m ← ⌊(i + j)/2⌋
  |   if V[m] = 1 then
  |   |   return conta0(V, i, m − 1)
  |   else
  |   |   return conta0(V, m + 1, j)
```

```
integer trovaH(integer[] V, integer n)
  integer i ← 1
  while V[i] = 0 and 2 · i ≤ n do
  |   i ← 2 · i
  if V[i] = 0 then
  |   return conta0(V, i, n)
  else if i/2 > 1 then
  |   return conta0(V, i/2, i)
  else
  |   return conta0(V, 1, i)
```

12.10 La funzione min() divide la matrice in 4 sottomatrici di dimensioni dimezzate, richiama ricorsivamente se stessa sulle 4 sottomatrici, e poi trova il minimo tra i 4 minimi delle sottomatrici. Per semplicità, si assume di disporre di una funzione min4(a, b, c, d), che restituisce il minimo tra i quattro interi a, b, c, d.

```
integer min(ITEM[][] M, integer imin, integer imax, integer jmin, integer jmax)
  if imin > imax or jmin > jmax then
  |   return +∞
  else if imin = imax and jmin = jmax then
  |   return M[ imin, imax]
  else
  |   integer imed ← ⌊(imin + imax)/2⌋
  |   integer jmed ← ⌊(jmin + jmax)/2⌋
  |   integer a ← min(imin, imed, jmin, jmed)
  |   integer b ← min(imed + 1, imax, jmin, jmed)
  |   integer c ← min(imin, imed, jmed + 1, jmax)
  |   integer d ← min(imin + 1, imax, jmed + 1, jmax)
  |   return min4(a, b, c, d)
```

12.12 Se ci sono almeno due elementi, basta considerare l'elemento centrale del vettore, confrontarlo col successivo, e continuare la ricerca nella prima o nella seconda metà del vettore:

```
integer maxUnimodale(ITEM[] A, integer i, j)
    integer m ← ⌊(i + j)/2⌋
    if i = j then
    |   return A[m]
    else
    |   if A[m] < A[m + 1] then
    |   |   return maxUnimodale(A, m + 1, j)
    |   else
    |   |   return maxUnimodale(A, i, m)
```

12.13 È possibile modificare la ricerca binaria rimuovendo il caso in cui si verifica l'identità con k; in questo modo tutte le ricerche diventano ricerche con insuccesso e terminano sul primo valore maggiore o minore di k (a seconda se si verifica prima se un numero è maggiore di k o viceversa). Eseguendo quindi due ricerche binarie, con gli **if** scambiati, si ottengono in tempo $O(\log n)$ le posizioni dei valori immediatamente minore e maggiore. Successivamente si può calcolare il numero di occorrenze di k in tempo $O(1)$.

PROGRAMMAZIONE DINAMICA

Capitolo 13

Una filosofia simile al *divide-et-impera*, ma portata all'estremo, si basa sulla massima:

"Il pesce grosso mangia tutti i pesci più piccoli".

In altri termini, se non si sa con esattezza quali sottoproblemi risolvere, si può provare a risolverli tutti e conservare i risultati ottenuti per poterli usare successivamente nel risolvere il problema di dimensione più grande. Questa tecnica è detta *programmazione dinamica*, perché indica un processo di programmazione (o pianificazione) di un metodo risolutivo che sia applicabile dinamicamente per dimensioni via via crescenti del problema. Rispetto al *divide-et-impera*, la programmazione dinamica presenta tre grandi differenze: è iterativa e non ricorsiva, affronta i sottoproblemi "dal basso verso l'alto" anziché "dall'alto verso il basso", e memorizza i risultati dei sottoproblemi in una tabella. Comunque, mentre il *divide-et-impera* risulta vantaggioso quando i sottoproblemi più piccoli da risolvere sono indipendenti, la programmazione dinamica è conveniente quando i sottoproblemi non sono indipendenti, ma ne condividono altri. Procedendo dal basso verso l'alto, la programmazione dinamica risolve un sottoproblema comune una sola volta, mentre il *divide-et-impera* lo risolverebbe più volte.

13.1 ESEMPIO – [Coefficiente binomiale]

$C(n, k) = n!/(k!(n-k)!)$ rappresenta il numero di modi di scegliere k oggetti da un insieme di n oggetti, con $0 \leq k \leq n$, ed è definibile ricorsivamente come segue:

$$C(n, k) = \begin{cases} 1 & \text{se } k = 0 \vee n = k \\ C(n-1, k-1) + C(n-1, k) & \text{altrimenti} \end{cases}$$

Un algoritmo ricorsivo *divide-et-impera* si ottiene immediatamente dalla precedente definizione ricorsiva:

```
integer C(integer n, k)
  if n = k or k = 0 then return 1
                    else return C(n − 1, k − 1) + C(n − 1, k)
```

Purtroppo, la complessità dell'algoritmo *divide-et-impera* cresce come il numero di chiamate ricorsive, che è proprio uguale a $C(n, k)$. Questo è dovuto all'elevato numero di sottoproblemi identici che vengono risolti più volte. Per esempio già $C(n-2, k-1)$ è richiamata due volte, sia per calcolare $C(n-1, k)$ che $C(n-1, k-1)$, mentre al decrescere dei valori degli argomenti ciascun sottoproblema è risolto per un numero sempre maggiore di volte. Un algoritmo di programmazione dinamica, invece, risolve i sottoproblemi per valori crescenti degli argomenti secondo il ben noto schema del "triangolo di Tartaglia", memorizzando i risultati una volta per tutte in una matrice intera $C[0 \ldots n][0 \ldots n]$ in $\Theta(n^2)$ tempo.

```
tartaglia(integer n, integer[][] C)
  for i ← 0 to n do
      C[i, 0] ← 1
      C[i, i] ← 1
  for i ← 2 to n do
      for j ← 1 to i − 1 do
          C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
```

In questo modo, il valore di $C(m, k)$, per ogni coppia di interi m, k tali che $0 \leq k \leq m \leq n$, può essere successivamente letto direttamente dalla matrice C in tempo $O(1)$.

La programmazione dinamica è stata introdotta da Bellman (1957) per progettare algoritmi che risolvono problemi di ottimizzazione. Perché sia applicabile, occorre che:

(1) sia possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande;

(2) le decisioni prese per risolvere in modo ottimo un sottoproblema rimangano valide quando il sottoproblema diviene un "pezzo" di un problema più grande;

(3) una tecnica *divide-et-impera* richieda di risolvere più volte gli stessi sottoproblemi e sia pertanto inutilizzabile da un punto di vista computazionale.

Le proprietà (1) e (2) prendono il nome di *sottostruttura ottima*. Ma non sono sufficienti; in pratica, affinché un algoritmo basato sulla programmazione dinamica abbia complessità polinomiale, occorre inoltre che:

(4) ci sia un numero polinomiale di sottoproblemi da risolvere;

(5) per evitare di risolvere più di una volta lo stesso sottoproblema, si usi una tabella in cui si memorizzano le soluzioni di tutti i sottoproblemi, senza preoccuparsi se la soluzione di un particolare sottoproblema verrà poi utilizzata oppure no;

(6) il tempo per combinare le soluzioni dei sottoproblemi e trovare la soluzione del problema più grande sia polinomiale.

Questi principi generali sono di difficile comprensione se non vengono calati in esempi concreti. Questo capitolo ne contiene un discreto numero, ognuno dei quali insegna un particolare aspetto della programmazione dinamica.

13.1 STRING MATCHING APPROSSIMATO

Nei motori di ricerca e in generale nelle applicazioni che si occupano di ricerca di testo, può capitare di dover risolvere il seguente problema.

STRING MATCHING APPROSSIMATO (APPROXIMATE STRING MATCHING) Date una stringa $P = p_1 \cdots p_m$ (pattern) e una stringa $T = t_1 \cdots t_n$ (testo), con $m \leq n$, un'*occorrenza k-approssimata* di P in T, con $0 \leq k \leq m$, è una copia della stringa P nella stringa T in cui sono ammessi k "errori" (o differenze) tra caratteri di P e caratteri di T, del seguente tipo:

(1) i corrispondenti caratteri in P e in T sono diversi (sostituzione);

(2) un carattere in P non è incluso in T (inserimento);

(3) un carattere in T non è incluso in P (cancellazione).

Il problema consiste nel trovare un'occorrenza k-approssimata di P in T per cui k sia minimo.

13.2 ESEMPIO – [Occorrenza 2-approssimata]
Siano T = "questoèunoscempio" (dove $n = 17$) e P = "unesempio" (dove $m = 9$). Un'occorrenza 2-approssimata di P parte dalla posizione 8 di T: questoèunoscempio. Infatti, la prima e di P corrisponde ad una o di T (errore di tipo (1)), mentre la c di T è un carattere non incluso in P (errore di tipo (3)).

Per comodità, definiamo il *prefisso* $X(k) = x_1 \cdots x_k$ di una stringa $X = x_1 \cdots x_n$ come la sottostringa formata dai primi k caratteri.

Per applicare la programmazione dinamica, dobbiamo trovare una formulazione ricorsiva che abbia la proprietà di sottostruttura ottima. Definiamo $D[i, j]$, con $0 \leq i \leq m$ e $0 \leq j \leq n$, come il minimo valore k per cui esiste un'occorrenza k-approssimata di $P(i)$ in $T(j)$. È possibile calcolare $D[i, j]$, per $1 \leq i \leq m$ e $1 \leq j \leq n$, in funzione dei tre casi seguenti, riassunti in Fig. 13.1, che tengono conto del tipo di errore che può occorrere tra p_i e t_j:

(1) $D[i-1, j-1]$, se $p_i = t_j$, oppure $D[i-1, j-1] + 1$, se $p_i \neq t_j$, perché "avanzando" di un carattere sia nel pattern che nel testo il numero di errori resta identico se i due caratteri sono uguali mentre aumenta di uno se sono diversi (errore di tipo (1));

(2) $D[i-1, j] + 1$, perché "avanzando" di un carattere solo nel pattern il numero di errori aumenta di uno in quanto p_i non compare in T (errore di tipo (2));

(3) $D[i, j-1] + 1$, perché "avanzando" di un carattere solo nel testo il numero di errori aumenta di uno in quanto t_j non compare in P (errore di tipo (3)).

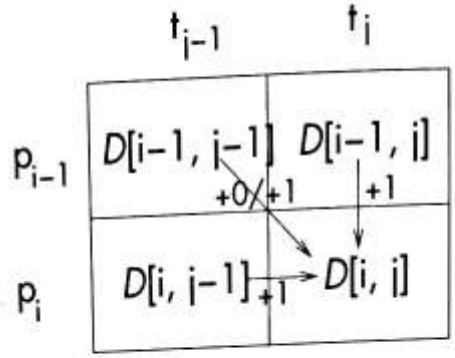


Fig. 13.1 Calcolo di $D[i, j] = \min\{D[i-1, j-1] + \delta, D[i-1, j] + 1, D[i, j-1] + 1\}$, dove $\delta = 0$, se $p_i = t_j$, oppure $\delta = 1$, se $p_i \neq t_j$.

La ricorsione termina quando una delle due stringhe diventa vuota. Se il pattern è vuoto, il numero di errori è 0, perché la stringa vuota $P(0)$ occorre in ogni posizione di T; se il testo è vuoto, invece, il numero di errori è pari alla lunghezza della stringa pattern, perché questi caratteri devono essere tutti cancellati. Riassumendo, si ottengono pertanto le seguenti relazioni di programmazione dinamica, per $0 \leq i \leq m$ e $0 \leq j \leq n$:

$$D[i,j] = \begin{cases} 0 & \text{se } i = 0 \\ i & \text{se } j = 0 \\ \min\{D[i-1,j-1]+\delta, D[i-1,j]+1, D[i,j-1]+1\} & \text{altrimenti} \end{cases}$$

dove $\delta = 0$, se $p_i = t_j$, oppure $\delta = 1$, se $p_i \neq t_j$. Non abbiamo fornito una dimostrazione formale, ma è chiaro che tale definizione illustra la proprietà di sottostruttura ottima del problema.

Si noti che $D[m, j] = k$ se e solo se c'è un'occorrenza k-approssimata di P in T che termina in t_j e che la soluzione del problema è data dal valore di $D[m, j]$ più piccolo, per $0 \leq j \leq n$. La funzione stringMatching() calcola la matrice D e restituisce la posizione di T in cui termina un'occorrenza approssimata di P col numero minimo di errori. La complessità della funzione è $\Theta(mn)$, poiché sono valutati esplicitamente tutti gli elementi di D.

13.3 ESEMPIO – [String matching approssimato]
La Fig. 13.2 mostra la matrice D ottenuta con le relazioni di programmazione dinamica per T = ababa e P = bab, dove $m = 3$ ed $n = 5$. Un'occorrenza 0-approssimata di P termina nella posizione 4 di T: ababa.

Le frecce in Fig. 13.2 indicano la provenienza del minimo; la funzione può essere facilmente modificata, senza aumentare la complessità, al fine di ricavare le occorrenze approssimate e i relativi errori; è sufficiente percorrere a ritroso tali frecce. Si noti tuttavia che non è effettivamente necessario memorizzare a parte le frecce; è sufficiente, per ogni posizione $[i, j]$, cercare quale fra le tre posizioni $[i-1, j]$, $[i, j-1]$ e $[i-1, j-1]$ ha dato origine al valore minimo. L'Esercizio 13.1 chiede di scrivere tale funzione.

```
integer stringMatching(ITEM[] P, ITEM[] T, integer m, integer n)
  integer[][] D ← new integer[0 ... m][0 ... n]
  for i ← 1 to m do D[i, 0] ← i
  for j ← 0 to n do D[0, j] ← 0
  for i ← 1 to m do
      for j ← 1 to n do
          integer t ← D[i − 1, j − 1] + iif(P[i] = T[j], 0, 1)
          t ← min(t, D[i − 1, j] + 1)
          t ← min(t, D[i, j − 1] + 1)
          D[i, j] ← t
  integer min ← D[m, 0]
  integer pos ← 0
  for j ← 1 to n do
      if D[m, j] < min then
          min ← D[m, j]
          pos ← j
  return pos
```

Una variante dello *string matching* approssimato prende il nome di *distanza di Levenshtein* o distanza di *editing* (in inglese, *edit distance*): date due stringhe, vogliamo conoscere il numero minimo di operazioni (sostituzione, inserimento, cancellazione) necessarie per trasformare una nell'altra (o viceversa, visto che inserimento e cancellazione sono simmetriche). Per esempio, la distanza di *editing* fra `google` e `yahoo` è pari a 6, perché bisogna cancellare i caratteri `gle`, inserire `ya` e sostituire `g` con `h`. Per adattarsi alla nuova semantica, la relazione di ricorrenza è la stessa ma le condizioni iniziali cambiano: come prima, $D[i, 0] = i$ per $1 \leq i \leq m$; ma ora anche $D[0, j] = j$, perché $T(j)$ ha j caratteri in più di $P(0)$ (da cancellare o inserire). Al termine, la distanza di *editing* si trova in $D[m, n]$.

Qualora sia richiesto, dato k, di trovare un'occorrenza approssimata di P in T con al più k errori, il problema può essere risolto in tempo $O(kn)$, ma il relativo algoritmo è complicato e non può essere qui riportato. Tale algoritmo, ideato da Landau e Vishkin (1988), riesce a valutare esplicitamente solo quegli elementi di D che provocano un incremento lungo una diagonale, cioè quelli per cui $D[i, j] > D[i - 1, j - 1]$, ed utilizza una struttura di dati ad albero detta *albero dei suffissi*.

P \ T		A	B	A	B	A
	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

Fig. 13.2 Risoluzione dello *string matching* approssimato con la programmazione dinamica.

13.2
INSIEME INDIPENDENTE DI INTERVALLI PESATI

Supponete di essere il manager di un grande albergo. La vostra sala riunioni è molto richiesta, e vi trovate a dover decidere come affittarla nel prossimo mese. Avete ricevuto un certo numero di richieste, ognuna caratterizzata da un orario di inizio e un orario di fine, e da un guadagno associato. Il vostro compito è massimizzare il guadagno dell'albergo. È facile vedere che il vostro problema è un caso particolare di questo problema classico:

INSIEME INDIPENDENTE DI INTERVALLI PESATI Siano dati n intervalli distinti $[a_1, b_1[, ..., [a_n, b_n[$ della retta reale, aperti a destra, dove all'intervallo i è associato un peso w_i, $1 \leq i \leq n$. Trovare un *insieme indipendente* di peso massimo, ovvero un sottoinsieme di intervalli tutti disgiunti tra di loro tale che la somma dei pesi degli intervalli nel sottoinsieme sia la più grande possibile.

Gli intervalli sono aperti a destra in quanto un evento che finisce a mezzogiorno può essere seguito da un evento che inizia alla stessa ora. Per risolvere questo problema, ordiniamo gli intervalli per estremi finali non decrescenti, in modo che $b_1 = b_2 \leq ... \leq b_n$. Per ogni intervallo i, sia $p_i = j$ il predecessore di i, dove $j < i$ è il massimo indice tale che $[a_j, b_j[$ non interseca $[a_i, b_i[$ (se non esiste j, allora $p_i = 0$).

Sia $P[i]$ il sottoproblema dato dai primi i intervalli e sia $S[i]$ una sua soluzione ottima di peso $D[i]$. Se l'intervallo i-esimo non fa parte di tale soluzione, allora deve valere $D[i] = D[i-1]$, dove si assume $D[0] = 0$; altrimenti, deve essere $D[i] = w_i + D[p_i]$.

Infatti, se $i \notin S[i]$, $S[i]$ è anche soluzione ottima per $P[i-1]$; se così non fosse, esisterebbe una soluzione ottima $S'[i-1]$ per $P[i-1]$ di peso $D'[i-1] > D[i]$. Ma tale soluzione sarebbe ottima anche per il problema $P[i]$, visto che i non è incluso, e questo è assurdo. Se invece $i \in S[i]$, $S[i] - \{i\}$ è una soluzione ottima per $P[p_i]$; se così non fosse, esisterebbe una soluzione ottima $S'[p_i]$ per $P[p_i]$ di peso $D'[p_i] > D[i] - w_i$. Ma allora basterebbe sostituire $S'[p_i]$ a $S[i] - \{i\}$ nella nostra soluzione ottima per il problema $P[i]$ e otterremmo una soluzione con costo $D'[p_i] + w_i > D[i] - w_i + w_i = D[i]$, assurdo.

Questo prova che il problema ha sottostruttura ottima; pertanto, la relazione di programmazione dinamica che si ottiene è:

$$D[i] = \max\{D[i-1], w_i + D[p_i]\} \qquad i = 1, ..., n$$

$D[n]$ è il costo della soluzione ottima del problema originario, partendo dalla quale è facile ricavare a ritroso gli intervalli di tale soluzione. Tutti i $p_1, ..., p_n$ si possono precalcolare in tempo $O(n)$ con una singola scansione degli intervalli. Basta ordinare i $2n$ estremi degli intervalli $\{a_1, ..., a_n, b_1, ..., b_n\}$ in modo non decrescente, utilizzare una variabile i inizializzata a 0 e scandire gli estremi da sinistra verso destra: se l'elemento scandito è un estremo iniziale, p.e. a_j, allora si assegna i a p_j, mentre se l'elemento è un estremo finale, p.e. b_j, allora si assegna j a i. Dopo la computazione dei predecessori,

ciascun $D[j]$ si ricava in tempo costante. La procedura maxinterval() ha quindi costo totale $O(n \log n)$, dovuto all'ordinamento.

```
SET maxinterval(integer[] a, integer[] b, integer[] w)
  {ordina gli intervalli per estremi di fine crescenti}
  {calcola p_i}
  integer[] D ← new integer [0 ... n]
  D[0] ← 0
  for i ← 1 to n do
  |   D[i] ← max(D[i − 1], w[i] + D[p_i])
  i ← n
  SET S ← Set()
  while i > 0 do
  |   if D[i − 1] > w[i] + D[p_i] then
  |   |   i ← i − 1
  |   else
  |   |   S.insert(i)
  |   |   i ← p_i
  return S
```

La procedura mostra che, con una pre-elaborazione non troppo costosa, è possibile risolvere un problema grazie alla programmazione dinamica.

13.3
MOLTIPLICAZIONE CATENA DI MATRICI

Date n matrici rettangolari $A_1, \ldots, A_n$, si consideri il problema di calcolare il prodotto $A_1 \cdots A_n$. Questo è possibile se per ogni i, $1 \leq i < n$, le matrici A_i e A_{i+1} sono *compatibili al prodotto*, ovvero il numero di colonne di A_i è uguale al numero di righe di A_{i+1}.

Moltiplicare due matrici A_i di dimensione $c_{i-1} \times c_i$ e A_{i+1} di dimensione $c_i \times c_{i+1}$ produce una matrice di dimensione $c_{i-1} \times c_{i+1}$ e richiede $c_{i-1} \cdot c_i \cdot c_{i+1}$ moltiplicazioni scalari, in quanto ogni elemento della matrice risultante viene calcolato dalla formula seguente:

$$\sum_{k=1}^{c_i} A_i[h, k] \cdot A_{i-1}[k, j] \qquad 1 \leq h \leq c_{i-1}, 1 \leq j \leq c_{i+1}$$

In generale, il numero totale di moltiplicazioni scalari necessarie per moltiplicare n matrici A_i di dimensione $c_{i-1} \times c_i$ dipende dall'ordine in cui vengono moltiplicate le matrici. Il prodotto fra matrici infatti è associativo.

13.4 ESEMPIO – [Moltiplicazione $A_1 \cdot A_2 \cdot A_3$]
Si consideri la moltiplicazione di A_1 di dimensione 30×5, A_2 di dimensione 5×25 e A_3 di dimensione 25×7. Il prodotto delle tre matrici può essere calcolato come $(A_1 \times A_2) \times A_3$ oppure $A_1 \times (A_2 \times A_3)$. Nel primo caso sono necessarie $(30 \cdot 5 \cdot 25 + 30 \cdot 25 \cdot 7) = 9000$ moltiplicazioni, mentre nel secondo ne sono necessarie solo $(30 \cdot 5 \cdot 7 + 5 \cdot 25 \cdot 7) = 1925$.

Una domanda importante è la seguente.

MOLTIPLICAZIONE CATENA DI MATRICI Date n matrici $A_1, \ldots, A_n$, trovare l'ordine di moltiplicazione di costo minimo, ovvero che richiede il numero minimo di moltiplicazioni scalari.

Avere queste informazione in anticipo permette di risparmiare tempo al momento della moltiplicazione effettiva.

Caratterizziamo la soluzione nel modo seguente: una *parentesizzazione* $P_{i,j}$ del prodotto $A_i \cdot A_{i+1} \ldots A_j$ consiste nella matrice A_i, se $i = j$; nel prodotto di due parentesizzazioni $(P_{i,k} \cdot P_{k+1,j})$, altrimenti. Una parentesizzazione $P_{i,j}$ si dice *ottima* se non esiste altra parentesizzazione $P'_{i,j}$ con costo inferiore.

Abbiamo caratterizzato la nostra soluzione in maniera ricorsiva; dobbiamo ora verificare che goda di sottostruttura ottima. Vogliamo dimostrare che nella parentesizzazione ottima $P_{i,j} = (P_{i,k} \cdot P_{k+1,j})$, le parentesizzazioni $P_{i,k}$ e $P_{k+1,j}$ sono anch'esse ottime. Per assurdo, supponiamo che esista una parentesizzazione $P'_{i,k}$ di $A_i \ldots A_k$ con costo inferiore a $P_{i,k}$ (il caso di $P'_{k+1,j}$ con costo inferiore a $P_{k+1,j}$ è simmetrico). Allora la parentesizzazione $(P'_{i,k} \cdot P_{k+1,j})$ avrebbe costo inferiore a $P_{i,j}$, ma questo è assurdo in quanto quest'ultima è ottima.

La definizione ricorsiva ci permetterebbe di scrivere una soluzione basata su *divide-et-impera*; ma molti problemi verrebbero ripetuti inutilmente. Il numero $F(n)$ di possibili parentesizzazioni è infatti definibile ricorsivamente nel modo seguente:

$$F(n) = \begin{cases} 1 & \text{se } n = 1 \\ \sum_{k=1}^{n-1} F(k)F(n-k) & \text{altrimenti} \end{cases}$$

dove una singola matrice ha un solo modo per essere espressa in parentesi, mentre nel caso di n matrici, per ognuno dei possibili indici k ci sono $F(k)$ modi di parentesizzare $A_1 \cdots A_k$ e $F(n-k)$ modi di parentesizzare $A_{k+1} \cdots A_n$. $F(n)$ è l'$(n-1)$-esimo "numero catalano" e sappiamo che $F(n) = \Omega(4^n/n^{3/2})$, quindi un algoritmo ricorsivo basato su *divide-et-impera* è superpolinomiale.

È possibile esprimere il costo della parentesizzazione ottima $P_{i,j} = (P_{i,k} \cdot P_{k+1,j})$ come la somma del costo delle parentesizzazioni ottime $P_{i,k}$ e $P_{k+1,j}$ più $c_{i-1}\, c_k\, c_j$ moltiplicazioni scalari per moltiplicare tra loro le matrici risultanti di dimensione $c_{i-1} \times c_k$ e $c_k \times c_j$. Sia $M[i, j]$ il minimo numero di moltiplicazioni aritmetiche necessarie per calcolare $P_{i,j}$; è possibile esprimere tale costo tramite le seguenti relazioni di programmazione dinamica:

$$M[i,j] = \begin{cases} \min_{i \le k \le j-1} \{M[i,k] + M[k+1,j] + c_{i-1}c_kc_j\} & \text{se } 1 \le i < j \le n \\ 0 & \text{se } 1 \le i = j \le n \end{cases}$$

La procedura parentesizzazione() calcola la matrice M, che viene riempita considerando il fatto che il costo per moltiplicare le sequenze di $h = j - i + 1$ matrici dipende soltanto dal

costo per moltiplicare sequenze con meno di h matrici, per $h = 2, 3, \ldots, n$. Per $h = 1$, invece, il costo è zero ed è rappresentato dalla diagonale principale. Fissato h, gli estremi i e j della sequenza $A_i \ldots A_j$ di h matrici sono tali che $1 \leq i \leq n - h + 1$, mentre $j = i + h - 1$.

La procedura parentesizzazione() lavora in tempo $O(n^3)$. Ovviamente, oltre a calcolare il costo di una parentesizzazione ottima, è necessario anche produrne una. Questo richiede una tabella aggiuntiva S che contiene, per ogni parentesizzazione $P_{i,j}$, l'indice $S[i, j] = k$ che rappresenta il punto di moltiplicazione fra le due sottoparentesizzazioni. Sfruttando questa informazione, possiamo chiamare la procedura stampaPar(S, 1, n) per stampare una rappresentazione della parentesizzazione in tempo $O(n)$.

Si noti che, a differenza dei problemi precedenti, in questo caso la tabella aggiuntiva riduce il costo di stampaPar(). In assenza della tabella, infatti, avremmo dovuto cercare nuovamente il valore k dell'ultima moltiplicazione per ogni posizione visitata da stampaPar, portando il suo costo ad $O(n^2)$.

```
parentesizzazione(integer[] c, integer[][] M, integer[][] S, integer n)
  integer i, j, h, k, t
  for i ← 1 to n do
  |   M[i, i] ← 0
  for h ← 2 to n do
  |   for i ← 1 to n − h + 1 do
  |   |   j ← i + h − 1
  |   |   M[i, j] ← +∞
  |   |   for k ← i to j − 1 do
  |   |   |   t ← M[i, k] + M[k + 1, j] + c[i − 1] · c[k] · c[j]
  |   |   |   if t < M[i, j] then
  |   |   |   |   M[i, j] ← t
  |   |   |   |   S[i, j] ← k
```

```
stampaPar(integer[][] S, integer i, integer j)
  if i = j then
  |   print "A["; print i; print "]"
  else
  |   print "("; stampaPar(S, i, S[i, j]); print "·"; stampaPar(S, S[i, j] + 1, j); print ")"
```

13.4 OCCUPAZIONE DI MEMORIA

Negli esempi visti finora, le tabelle di programmazione dinamica hanno una dimensione uguale al numero di possibili sottoproblemi. Una volta completate, le tabelle possono essere utilizzate da procedure come stampaPar() per ricostruire una soluzione ottima. Vi sono tuttavia alcuni casi in cui è possibile evitare di mantenere in memoria tutti i sottoproblemi risolti.

Si consideri come esempio la serie di Fibonacci. Dato un intero $n \geq 0$, l'n-esimo numero di Fibonacci F_n è definito ricorsivamente come segue: $F_0 = 1$, $F_1 = 1$, ed $F_n =$

$F_{n-1} + F_{n-2}$ per $n > 1$. Una soluzione *divide-et-impera* basata sulla definizione ricorsiva tende a risolvere un gran numero di problemi ripetuti. È infatti possibile dimostrare che il valore di F_n è circa $(1.6)^n$ e pertanto la soluzione *divide-et-impera* richiede un numero di chiamate che cresce anch'esso come $(1.6)^n$.

Una soluzione basata su programmazione dinamica memorizza tutti i numeri di Fibonacci fino ad F_n, evitando così problemi ripetuti. Calcolando i valori in maniera iterativa, si ottiene un algoritmo il cui costo è $O(n)$, sia in termini di tempo che di spazio.

```
integer fibonacci(integer n)
  integer[] F ← new integer[0 ... max(n, 1)]
  F[0] ← F[1] ← 1
  for i ← 2 to n do F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

Una domanda importante è la seguente: è veramente necessario un vettore di n posizioni? In realtà, la posizione $F[i]$ smette di essere utilizzata dopo aver calcolato $F[i + 2]$. In pratica, non serve un vettore di dimensione n, ma bastano tre variabili F_0, F_1, F_2. Queste variabili vedono scorrere i numeri di Fibonacci in modo che ad ogni iterazione il valore di F_0 viene scartato e sostituito con F_1, F_1 viene sostituito con F_2 e il nuovo termine delle serie di Fibonacci viene calcolato in F_2. La complessità è uguale, lo spazio questa volta è costante.

```
integer fibonacci(integer n)
  integer F0, F1, F2
  F0 ← F1 ← F2 ← 1
  for i ← 2 to n do
  |   F0 ← F1
  |   F1 ← F2
  |   F2 ← F0 + F1
  return F2
```

Il problema di Fibonacci può sembrare un caso particolare, non applicabile ad altri problemi. In realtà, sotto certe condizioni, una tecnica simile può essere applicata anche al problema dello *string matching* approssimato.

Supponiamo che non sia necessario produrre in output l'occorrenza con k errori, ma si voglia solamente ottenere il minimo valore k (analogamente, si voglia ottenere solamente la distanza di *editing*). Questo significa che al termine del problema saremmo interessati solo all'ultima riga della tabella, per individuare il valore minimo. Durante l'esecuzione, per calcolare una riga è sufficiente avere a disposizione i soli tre valori illustrati in Fig. 13.1. È quindi possibile memorizzare due sole righe alla volta, come illustrato nella procedura stringMatchingSingleRow(). L'algoritmo si avvale di due vettori, D che contiene la riga corrente (quella che viene attualmente calcolata) e PD che contiene la riga precedente. Ogniqualvolta viene calcolata una nuova riga, si sposta la riga corrente D nella riga precedente PD scambiando i puntatori dei due vettori, per poi utilizzare la formula ri-

corsiva illustrata in precedenza. Al termine del calcolo di m righe, è sufficiente identificare il valore minimo fra tutti quelli presenti in D. Si noti che è sufficiente utilizzare una sola riga, ma il codice risultante non sarebbe altrettanto chiaro.

```
integer stringMatchingSingleRow(ITEM[] P, ITEM[] T, integer m, integer n)
  integer[] PD ← new [0 ... n]
  integer[] D ← new [0 ... n]
  for j ← 0 to n do D[j] ← 0
  for i ← 1 to n do
      PD ↔ D
      integer D[0] ← i                                  % Valore a destra, colonna 0
      for j ← 1 to n do
      integer D[j] ← min(PD[j − 1] + iif(P[i] = T[j], 0, 1), PD[j] + 1, D[j − 1] + 1)
  return min(D)
```

13.5
CAMMINI MINIMI TRA TUTTE LE COPPIE

Nel capitolo 11 sono stati trattati diversi algoritmi per trovare i cammini minimi tra un dato nodo r e tutti i rimanenti nodi. Per trovare i cammini minimi tra tutte le coppie di nodi, è possibile applicare n volte uno degli algoritmi visti, considerando a turno come nodo "radice" r ciascuno degli n nodi del grafo. Se non ci sono archi con lunghezze negative, tale procedimento è $O(nm \log n)$, utilizzando l'algoritmo di Johnson, e $O(n^3)$ utilizzando quello di Dijkstra. Se ci sono lunghezze negative, invece, la complessità è $O(mn^2) = O(n^4)$, poiché si deve usare n volte l'algoritmo di Bellman, Ford e Moore.

Vediamo come la programmazione dinamica permetta di progettare un algoritmo di complessità inferiore ad $O(n^4)$ anche per grafi con lunghezze negative degli archi.

Dato G, un *cammino k-vincolato da u a v* è un cammino semplice da u a v che non attraversa i nodi $\{k, k+1, k+2, ..., n\} - \{u, v\}$.

Per i cammini minimi k-vincolati, vale la seguente proprietà di sottostruttura ottima: ogni sottocammino c_{ij} di un cammino minimo k-vincolato c_{uv} in G è esso stesso un cammino minimo k-vincolato in G. Per provarlo, scomponiamo il cammino c_{uv} in tre parti:

$$u \xrightarrow{c_{ui}} i \xrightarrow{c_{ij}} j \xrightarrow{c_{jv}} v$$

Il costo del cammino c_{uv} può quindi essere espresso come: $w(c_{uv}) = w(c_{ui}) + w(c_{ij}) + w(c_{jv})$. Ovviamente c_{ij} è k-vincolato. Supponiamo ora per assurdo che esista un cammino k-vincolato c'_{ij} con costo inferiore, ovvero $w(c'_{ij}) < w(c_{ij})$. Possiamo quindi costruire un cammino k-vincolato c'_{uv}, sostituendo c_{ij} con c'_{ij}, il cui costo è

$$w(c'_{uv}) = w(c_{ui}) + w(c'_{ij}) + w(c_{jv}) < w(c_{ui}) + w(c_{ij}) + w(c_{jv}) = w(c_{uv})$$

Ma questo è un assurdo, perché $w(c_{uv})$ è un cammino minimo k-vincolato.

Sia d_{uv}^k il costo di un cammino minimo k-vincolato fra u e v. Per la proprietà di sottostruttura ottima, è facile osservare che per un cammino minimo $(k+1)$-vincolato sono date due possibilità: o non passa nemmeno per il nodo k, e quindi il suo costo è anche quello di un cammino minimo k-vincolato, oppure include anche il nodo k, e allora è uguale al costo del cammino minimo k-vincolato fra u e k più il costo del cammino minimo k-vincolato fra k e v:

$$d_{uv}^{k+1} = \min\{d_{uv}^k, d_{uk}^k + d_{kv}^k\} \qquad 1 \le u \le n,\ 1 \le v \le n$$

con condizioni iniziali:

$$d_{uv}^1 = \begin{cases} w(u,v) & \text{se } (u,v) \in E \\ 0 & \text{se } u = v \\ +\infty & \text{altrimenti.} \end{cases}$$

Per ricavare, oltre alla lunghezza d_{uv}^k, anche il cammino minimo, si definisce in modo analogo p_{uv}^k come il predecessore del nodo v in un cammino minimo k-vincolato da u a v. Per $1 \le k \le n$, si ottiene:

$$p_{uv}^{k+1} = \begin{cases} p_{uv}^k & \text{se } d_{uv}^{k+1} = d_{uv}^k \\ p_{kv}^k & \text{se } d_{uv}^{k+1} = d_{uk}^k + d_{kv}^k \end{cases}$$

con condizioni iniziali:

$$p_{uv}^1 = \begin{cases} u & \text{se } (u, v) \in E \\ 0 & \text{altrimenti} \end{cases}$$

È facile convincersi che d_{uv}^{n+1} è la lunghezza di un cammino minimo dal nodo u al nodo v nel grafo G, mentre p_{uv}^{n+1} è il predecessore del nodo v in un cammino siffatto. Si noti che se il nodo v non è raggiungibile dal nodo u, allora $d_{uv}^{n+1} = +\infty$, mentre se esiste un ciclo negativo che passa per il nodo u, allora $d_{uu}^{n+1} < 0$.

La computazione di d_{uv}^{n+1} e p_{uv}^{n+1}, per $1 \le u \le n$, $1 \le v \le n$, può essere effettuata con la procedura floydWarshall(), proposta da Floyd (1962) basandosi su risultati precedenti di Warshall. La procedura usa iterativamente la riga k e la colonna k delle matrici d e p per aggiornare *in loco* i rimanenti elementi delle matrici stesse; in altre parole, come suggerito dalla sezione precedente, evita di memorizzare inutilmente k matrici. Il tempo richiesto della procedura è $\Theta(n^3)$. La procedura utilizza soltanto $O(n^2)$ spazio di memoria, in quanto i valori d_{uv}^k e p_{uv}^k, per ogni iterazione k, sono sempre riscritti nelle matrici d e p stesse. Si noti che l'inizializzazione della matrice d non è necessaria se il grafo è realizzato con matrice d'adiacenza rappresentata dalle lunghezze degli archi.

13.5 ESEMPIO – [Algoritmo di Floyd-Warshall]

La Fig. 13.3 mostra l'esecuzione dell'algoritmo di Floyd-Warshall su un grafo G avente 4 nodi e 6 archi. Per ogni iterazione k, sono evidenziate in grassetto la k-esima riga e la k-esima colonna delle matrici d e p. Gli elementi delle matrici all'iterazione $(k + 1)$-esima sono ottenuti applicando la cosiddetta regola del rettangolo rispetto alla la k-esima riga e alla k-esima colonna. Per esempio, per $k + 1 = 3$, l'elemento d_{14}^3 (Fig. 13.3c) è ottenuto da $\min\{d_{14}^2, d_{12}^2 + d_{24}^2\} = \min\{+\infty, 5 + 4\} = 9$ (Fig. 13.3b), mentre p_{14}^3 è ottenuto da $p_{24}^2 = 2$. L'u-esima riga della matrice p finale contiene l'albero dei cammini minimi di radice u, rappresentato come vettore dei padri, dove un nodo non raggiungibile da u ha padre uguale a 0 e distanza uguale ad $+\infty$.

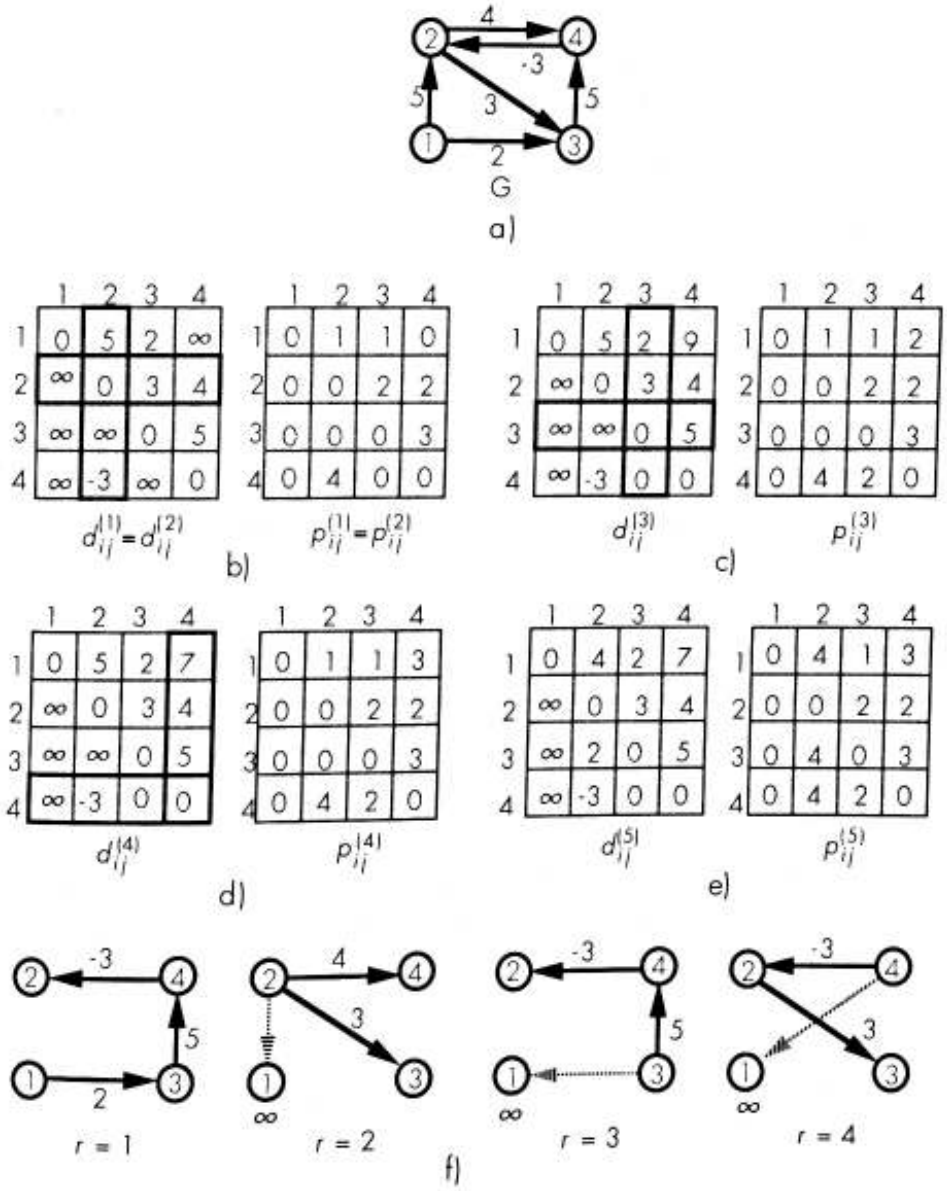


Fig. 13.3 Algoritmo di Floyd-Warshall. a) Un grafo G di 4 nodi; b)-e) matrici d e p per $k = 1, 2, 3, 4, 5$; f) alberi dei cammini minimi per radice $r = 1, 2, 3, 4$.

```
floydWarshall(GRAPH G, integer[][] d, integer[][] p)
foreach u, v ∈ G.V() do
    d[u, v] ← iif(u = v, 0, +∞)
    p[u, v] ← 0
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        d[u, v] ← w(u, v)
        p[u, v] ← u
for k ← 1 to G.n do
    foreach u ∈ G.V() do
        foreach v ∈ G.V() do
            if d[u, k] + d[k, v] < d[u, v] then
                d[u, v] ← d[u, k] + d[k, v]
                p[u, v] ← p[k, v]
```

13.6 MEMOIZATION

Nelle sezioni precedenti, abbiamo visto che non è sempre necessario memorizzare le soluzioni di tutti i sottoproblemi. Una domanda correlata è la seguente: è necessario calcolare le soluzioni di tutti i sottoproblemi? Anche in questo caso la risposta può essere negativa.

Si consideri il seguente problema: state svaligiando una casa piena di beni preziosi; volete sottrarre oggetti del maggior valore possibile, ma siete limitati dal fatto che lo zaino che avete portato con voi ha una certa capacità. Quali oggetti scegliete? Più formalmente, dovete risolvere il seguente problema:

ZAINO (KNAPSACK) Dati un intero positivo C (la capacità dello zaino) e un insieme di n "oggetti", tali che l'oggetto i-esimo è caratterizzato da un "profitto" p_i e da un "volume" v_i, entrambi interi positivi, trovare un sottoinsieme S di $\{1, ..., n\}$ tale che il volume totale $v(S) = \sum_{i \in S} v_i \leq C$ e il profitto totale $p(S) = \sum_{i \in S} p_i$ sia massimo.

È facile dimostrare che questo problema gode di sottostruttura ottima. Definiamo con $P(i, c)$ il sottoproblema dato dai primi i oggetti da inserire in uno zaino di capacità c. Il problema originale corrisponde a $P(n, C)$. Sia $S(i, c)$ una soluzione ottima per $P(i, c)$; possono darsi due casi:

- Se $i \in S(i, c)$, allora $S(i, c) - \{i\}$ è una soluzione ottima per il sottoproblema $P(i-1, c-v_i)$. Supponiamo per assurdo che ciò non sia vero. Allora è possibile trovare una soluzione S' per $P(i-1, c-v_i)$ tale per cui $p(S') > p(S(i, c) - \{i\})$ e $v(S') < c - v_i$. Se così fosse, potremmo ottenere una soluzione $S' \cup \{i\}$ al problema $P(i, c)$ tale che $p(S' \cup \{i\}) = p(S') + p_i > p(S(i, c) - \{i\}) + p_i = p(S(i, c))$ e $v(S' \cup \{i\}) \leq c - v_i + v_i = c$; ma questa è una contraddizione, visto che $S(i, c)$ è ottima.

- Se $i \notin S(i, c)$, allora $S(i, c)$ è una soluzione ottima per $P(i-1, c)$. Come sopra, assumiamo per assurdo che ciò non sia vero; allora esiste un insieme S' soluzione ottima di $P(i, c)$ tale che $p(S') > p(S(i, c))$ e $v(S') \leq c$, contraddicendo l'ipotesi di ottimalità di $S(i, c)$.

Detto $D[i, c]$ il profitto massimo ottenuto per il problema $P(i, c)$, la proprietà di sottostruttura ottima ci suggerisce la seguente definizione ricorsiva:

$$D[i,c] = \begin{cases} 0 & \text{se } i = 0 \vee c = 0 \\ -\infty & \text{se } c < 0 \\ \max\{D[i-1,c], D[i-1,c-v_i] + p[i]\} & \text{altrimenti} \end{cases}$$

Possiamo facilmente ottenere una procedura basata sulla programmazione dinamica con costo $O(nC)$. Questo problema, tuttavia, differisce da quelli precedenti in quanto non è detto che sia necessario risolvere tutti i sottoproblemi. Dipende in realtà dai particolari valori dei volumi e dalla capacità.

L'approccio iterativo, "dal basso verso l'alto" della programmazione dinamica risolve obbligatoriamente tutti i problemi. Esiste un approccio alternativo alla programmazione dinamica, ricorsivo e "dall'alto verso il basso", che prende il nome di *memoization* (o *annotazione*). L'idea è scrivere una procedura che utilizza la tabella come una cache per memorizzare le soluzioni dei problemi, controllando ogni volta se un problema è stato risolto; se sì, si riutilizza il risultato; se no, lo risolve (chiamando ricorsivamente la procedura sui sottoproblemi), ne memorizza il risultato nella tabella e lo restituisce al chiamante. La procedura zaino() risolve il problema dello zaino utilizzando *memoization*; prende in input due vettori p e v contenenti i profitti e i volumi, un indice i e la capacità residua c. Inoltre, prende in input anche un puntatore alla tabella D. La prima invocazione della procedura prende la forma zaino(p, v, n, C, D), con tutte le posizioni della matrice D inizializzate a $\perp$, un valore speciale che indica che il sottoproblema non è stato ancora risolto.

```
integer zaino(integer[] p, integer[] v, integer i, integer c, integer[][] D,)
  if i = 0 or c = 0 then
  |   return 0
  if c < 0 then
  |   return -∞
  if D[i, c] = ⊥ then
  |   D[i, c] ← max(zaino(p, v, i - 1, c, D), zaino(p, v, i - 1, c - v[i], D) + p[i])
  return D[i, c]
```

Il codice risultante è molto elegante e simile alla definizione ricorsiva; l'unica differenza è la presenza del controllo sul valore $\perp$, per evitare di risolvere sottoproblemi già risolti. Vi sono tuttavia alcune considerazioni da fare per valutare l'efficacia della tecnica di *memoization*. La complessità della procedura nel caso pessimo è comunque $O(nC)$; ma a se-

conda di volumi e capacità, il numero di sottoproblemi da risolvere potrebbe essere molto inferiore, anche di diversi ordini di grandezza.

Una possibile obiezione riguarda l'inizializzazione della tabella D, che non è inclusa nella procedura ma costa $O(nC)$ in ogni caso. Se il costo di ricombinare i risultati per riempire una posizione non è costante, il costo dell'inizializzazione è asintoticamente inferiore al costo per riempire la tabella e l'obiezione non si pone. Se questo non è il caso (come nel problema dello zaino), si ha una tipica situazione in cui può essere indicato usare una tabella hash: un universo delle possibili chiavi molto grande – $O(nC)$ – e un numero basso di chiavi attese.

Infine, una considerazione sulla complessità: $O(nC)$ non è polinomiale. Infatti, C non è la dimensione dell'input, è un parametro che fa parte dell'input. La dimensione effettiva di C (in numero di bit) è $d = \lceil \log C \rceil$, il che significa che il costo è esponenziale in d. Tuttavia, se $C = O(n^k)$ per qualche k, la soluzione proposta è praticabile. Algoritmi di questo tipo vengono detti *pseudo-polinomiali* [capitolo 19].

13.7 REALITY CHECK

Consideriamo un'ulteriore variante dello string matching.

SOTTOSEQUENZA COMUNE MASSIMALE (LONGEST COMMON SUBSEQUENCE) Data una sequenza X, è possibile ottenere una *sottosequenza* da X rimuovendone uno o più caratteri (non usiamo il termine sottostringa perché normalmente associato all'idea di una sottosequenza di caratteri *contigui*). Date due sequenze P e T, X è una *sottosequenza comune* se è sottosequenza sia di P che di T. Una sequenza X è una *sottosequenza comune massimale* (in inglese, *longest common subsequence* o LCS) di P e T se non esistono sottosequenze comuni a P e T più lunghe di X. Il problema consiste nel calcolare la lunghezza delle LCS ed eventualmente produrne una.

La lunghezza della LCS può essere utilizzata come una misura della similitudine fra due sequenze.

Il problema può essere risolto in modo molto simile al problema della distanza di editing. Sia $D[i, j]$ la lunghezza della LCS di $P(i)$ e $T(j)$. Ovviamente, se $i = 0$ oppure $j = 0$, tale valore è pari a 0 (non esiste infatti alcun carattere in comune con la sequenza vuota). Altrimenti, possono darsi due casi.

(1) Se $p_i = t_j$, la lunghezza della LCS di $P(i)$ e $T(j)$ è pari alla lunghezza della LCS di $P(i-1)$ e $T(j-1)$ più 1. Questo perché l'ultimo carattere fa sicuramente parte della LCS di $P(i)$ e $T(j)$.

(2) Se $p_i \neq t_j$, vi sono due sottocasi: o l'ultimo carattere di $P(i)$ non fa parte della LCS, oppure l'ultimo carattere di $T(j)$ non ne fa parte. Nel primo caso, possiamo ricondurci a calcolare la LCS di $P(i-1)$ e $T(j)$; nel secondo caso, possiamo ricondurci a calcolare la LCS di $P(i)$ e $T(j-1)$.

Queste considerazioni possono essere riassunte dalle seguenti relazioni di ricorrenza:

$$D[i,j]=\begin{cases} 0 & \text{se } i=0 \vee j=0 \\ D[i-1,j-1]+1 & \text{se } i>0 \wedge j>0 \wedge p_i=t_j \\ \max\{D[i-1,j],D[i,j-1]\} & \text{se } i>0 \wedge j>0 \wedge p_i \neq t_j \end{cases}$$

La prima invocazione della procedura prende la forma lcs(D, P, T, m, n), con tutte le posizioni della matrice D inizializzate a **nil**, un valore speciale che indica che il sottoproblema non è stato ancora risolto. Una versione basata su *memoization* è la seguente:

```
integer lcs(integer[][] D, ITEM[] P, ITEM[] T, integer i, integer j)
  if i = 0 or j = 0 then
  |  return 0
  if D[i, j] = nil then
  |  if P[i] = T[j] then
  |  |  D[i, j] ← lcs(D, P, T, i − 1, j − 1) + 1
  |  else
  |  |  D[i, j] ← max(lcs(D, P, T, i − 1, j), lcs(D, P, T, i, j − 1))
  return D[i, j]
```

In molti testi di algoritmi si menziona il fatto che questo è un tipico problema di bio-informatica, applicato a sequenze di basi del DNA tratte dall'alfabeto {A, T, G, C} – adenina, timina, guanina e citosina. Ma esiste un'applicazione informatica molto più comune e vicina all'uso quotidiano: il comando Unix `diff`, che esamina due file di testo, e ne evidenzia le differenze a livello di riga. Si noti che lavorare a livello di riga significa che i confronti fra caratteri sono in realtà fra righe, e che n ed m sono il numero di righe dei due file. L'output di `diff` può essere generato in tempo lineare a partire dalla tabella D, come richiesto nell'Esercizio 13.10.

Quando è applicato a due file di testo lunghi migliaia di righe, la complessità $O(mn)$ potrebbe essere troppo elevata, soprattutto per l'occupazione di memoria. È tuttavia possibile applicare numerose ottimizzazioni.

- `diff` è utilizzato per sorgenti di codice, in cui avvengono pochi cambiamenti per volta, raramente all'inizio del file, e molto spesso nemmeno alla fine del file. Conviene quindi scartare dal confronto tutte le prime s righe uguali all'inizio del file, e tutte le t righe uguali al termine; la complessità diventerebbe quindi $O(s+t+(n-s-t)\cdot(m-s-t))$.
- Confrontare righe è costoso; potrebbe essere utile calcolare, per ogni riga, un valore hash e confrontare questi ultimi. Se due valori hash sono diversi, siamo sicuri che le righe sono differenti; se sono uguali, dobbiamo spendere un po' di tempo in più per controllare che le righe siano effettivamente uguali e non abbiano dato origine ad una collisione.

- Utilizzando una tabella hash, possiamo anche scoprire che alcune righe compaiono in un solo file. È possibile rimuovere queste righe dal nostro problema, ricordando di aggiungerle all'output come aggiunte o rimozioni.

13.1 [k-occorrenza] Data la tabella D calcolata da stringMatching(), scrivere una procedura che stampa l'occorrenza k-approssimata di P che compare in T e termina in t_j.

13.2 [Massima sottostringa comune] Date due stringhe $P = p_1\, p_2 \ldots p_m$ e $T = t_1 t_2 \ldots t_n$, si progetti un algoritmo di programmazione dinamica per individuare la più lunga sottostringa (sottosequenza contigua) comune tra P e T.

13.3 [Algoritmo di Bellman, Ford e Moore] Si consideri il problema dei cammini minimi da un singolo nodo r a tutti i rimanenti $n - 1$ nodi in un grafo orientato pesato (con eventuali lunghezze negative degli archi). Si definisca d_v^k come la lunghezza di un cammino minimo dal nodo r al nodo v che contiene al più k archi. Si dimostri che l'algoritmo di Bellman-Ford-Moore [capitolo 11] equivale a risolvere le seguenti relazioni di programmazione dinamica:

$$d_v^{k+1} = \min\{d_v^k, \min_{h \neq v}\{d_h^k + w(h,v)\}\}$$

Qual è la complessità dell'algoritmo nella versione di programmazione dinamica?

13.4 [Cammini minimi su grafi aciclici] Si consideri il problema dei cammini minimi da un singolo nodo r a tutti i rimanenti nodi in un grafo (con eventuali lunghezze negative degli archi) nel quale esiste l'arco (u, v) solo se $u < v$. Si dimostri che le relazioni di programmazione dinamica dell'Esercizio 13.3 diventano semplicemente

$$d_v = \min_{h<v}\{d_h + w(h, v)\}$$

dove d_v è la lunghezza di un cammino minimo dal nodo r al nodo v. Qual è la complessità dell'algoritmo?

13.5 [Cammini minimi dinamici] Un algoritmo dinamico è un algoritmo che è in grado di aggiornare il risultato di una funzione a fronte di modifiche dei dati di ingresso senza doverlo ricalcolare tutto da capo. Sia $G = (V, E)$ un grafo orientato con funzione costo w: $V \times V \longrightarrow \mathbf{R}$. Progettare un algoritmo dinamico che sia in grado di aggiornare le distanze fra tutte le coppie di vertici del grafo a fronte dell'inserimento di un nuovo arco nel grafo. Se l'inserimento dell'arco introduce un ciclo negativo nel grafo, l'algoritmo deve essere in grado di lanciare un'eccezione. Quali difficoltà sorgerebbero se volessimo cancellare archi invece di inserirli?

13.6 [Massima sottosequenza crescente] Sia data una sequenza L di n numeri interi distinti. Si scriva una procedura efficiente basata sulla programmazione dinamica per

trovare la più lunga sottosequenza crescente di L (per esempio, se $L = 9, 15, 3, 6, 4, 2, 5, 10, 3$, allora la più lunga sottosequenza crescente è: 3, 4, 5, 10).

13.7 [Allineamento globale] Date due stringhe $P = p_1p_2 \ldots p_m$ e $T = t_1t_2 \ldots t_n$, all'incirca della stessa lunghezza, un *allineamento globale* consiste nell'inserzione di spazi in P o in T in modo da ottenere la stessa lunghezza, col vincolo che uno spazio in P non possa mai essere allineato a uno spazio in T. Ad un allineamento globale si associa un punteggio ottenuto valutando +1 se $p_i = t_j$, valutando −1 se $p_i \neq t_j$, e −2 quando p_i oppure t_j è uno spazio. Per esempio, se P = GACGGATTAG e T = GATCGGAATAG, l'allineamento globale tra GA_bCGGA_TTAG e GA_TCGGA_AATAG ha un punteggio di 9 − 1 − 2 = 6. Si progetti un algoritmo di programmazione dinamica per trovare l'allineamento globale di punteggio massimo.

13.8 [Allineamento semiglobale] Il problema dell'allineamento semiglobale è simile a quello dell'allineamento globale dell'Esercizio 13.7, con l'unica differenza che non si penalizzano gli eventuali spazi inseriti prima dell'inizio e/o dopo la fine di una stringa. Si indichi come modificare le relazioni di programmazione dinamica in modo da risolvere il problema.

13.9 [Allineamento locale] Il problema dell'allineamento locale è anch'esso una variante dell'allineamento globale dell'Esercizio 13.7, dove però si vuole trovare l'allineamento di punteggio massimo tra *sottostringhe* di P e T. Si mostri come modificare le relazioni di programmazione dinamica in modo da risolvere il problema.

13.10 [diff] Scrivere una procedura printdiff() che, data la tabella D calcolata dalla procedura lcs(), produca in output il file delle differenze come illustrato in Fig. 13.4.

```
Questo è il testo originale
alcune linee non dovrebbero
cambiare mai
altre invece vengono
rimosse
altre vengono aggiunte
```

```
Questo è il testo nuovo
alcune linee non dovrebbero
cambiare mai
altre invece vengono
cancellate
altre vengono aggiunte
come questa
```

```
- Questo è il testo originale
+ Questo è il testo nuovo
  alcune linee non dovrebbero
  cambiare mai
  altre invece vengono
- rimosse
+ cancellate
  altre vengono aggiunte
+ come questa
```

Fig. 13.4 Il file `original.txt` (a sinistra); il file `new.txt` (al centro); l'output di `diff original.txt new.txt` (a destra).

13.11 [Harry Potter] Al celebre maghetto Harry Potter è stata regalata una scopa volante modello Nimbus3000 e tutti i suoi compagni del Grifondoro gli chiedono di poterla provare. Il buon Harry ha promesso che nei giorni a venire soddisferà le richieste di tutti.

ma ogni ragazzo è impaziente e vuole provare la scopa il giorno stesso. Ognuno propone ad Harry un intervallo di tempo della giornata durante il quale, essendo libero da lezioni di magia, può fare un giro sulla scopa, e per convincerlo gli offre una certa quantità di caramelle Tuttigusti+1. Allora Harry decide di soddisfare, tra tutte le richieste dei ragazzi, quelle che gli procureranno la massima quantità di caramelle (che poi spartirà coi suoi amici Ron e Hermione). Aiutalo a trovare la migliore soluzione possibile.

12.12 [Massima somma di sottovettori] Riproponiamo l'Esercizio 12.2: Progettare un algoritmo che, preso un vettore A di n interi (positivi o negativi), trovi un sottovettore (una sequenza di elementi consecutivi del vettore) la somma dei cui elementi sia massima.

12.13 [Ma è italiano?] Vi viene data una stringa di caratteri $s[1...n]$; avete il dubbio che sia una stringa di testo italiano, in cui sono stati tolti tutti gli spazi e i segni di interpunzione. Ad esempio, "eraunanottebuiaetempestosa". Avete a disposizione una struttura di dati di tipo dizionario D che contiene... un dizionario di italiano! D.lookup(w) restituisce vero se la parola w è una parola italiana. Progettare un algoritmo che stabilisca se il testo che riceve in input è un test in italiano e calcolarne la complessità, misurata come numero di chiamate a D.lookup().

13.2 Si definisce una matrice $D[i, j]$, con $0 \leq i \leq m$ e $0 \leq j \leq n$, come segue: $D[i, j]$ è il numero dei caratteri della sottostringa comune al prefisso i-esimo di P e al prefisso j-esimo di T.

Chiaramente, $D[0, j] = 0$, per $0 \leq j \leq n$, poiché il prefisso 0-esimo di T è vuoto e non ci sono caratteri in comune; anche $D[i, 0] = 0$, per $0 \leq i \leq m$, perché il prefisso 0-esimo di P è vuoto. Valgono le seguenti relazioni di programmazione dinamica:

$$D[i,j] = \begin{cases} 0 & \text{se } p_i \neq t_j \\ D[i-1, j-1] + 1 & \text{se } p_i = t_j \end{cases}$$

Il massimo elemento $D[i, j]$ della matrice così ottenuta individua la sottostringa più lunga comune a P e T. L'algoritmo ha complessità $\Theta(mn)$.

13.7 Assumendo che P e T inizino con un carattere "vuoto" λ, che si trova in posizione 0, si definisce una matrice D tale che $D[i, j]$ è il punteggio del migliore allineamento globale tra $p_0 p_1 \ldots p_i$ e $t_0 t_1 \ldots t_j$. Valgono le seguenti relazioni di programmazione dinamica:

$$D[i,j] = \begin{cases} -2i & \text{se } j = 0 \\ -2j & \text{se } i = 0 \\ \max\{D[i, j-1] - 2, D[i-1, j] - 2, D[i-1, j-1] + \delta_{ij}\} & \text{altrimenti} \end{cases}$$

dove $\delta_{ij} = 1$ se $p_i = t_j$, e $\delta_{ij} = -1$ se $p_i \neq t_j$. Il valore $D[m, n]$ individua il punteggio dell'allineamento globale ottimo tra P e T e può essere calcolato in tempo $\Theta(mn)$ con una funzione la cui struttura è analoga a quella vista per lo *string matching*. Ovviamente, si può agevolmente modificare l'algoritmo in modo da ricavare, oltre al punteggio dell'allineamento ottimo, anche l'allineamento stesso. L'allineamento globale è un problema importante nella bio-informatica.

13.8 Le relazioni di programmazione dinamica sono analoghe a quelle viste nella soluzione dell'Esercizio 13.7, con le seguenti differenze. Per trascurare gli spazi inseriti prima dell'inizio di P basta inizializzare $D[0, j] = 0$ per $0 \leq j \leq n$, mentre per trascurare quelli prima di T si inizializza $D[i, 0] = 0$ per $0 \leq i \leq m$. Se invece non si vogliono penalizzare gli spazi inseriti dopo la fine di P occorre produrre come risultato il valore massimo della riga m di D, mentre per non penalizzare gli spazi dopo T si prende come risultato il massimo della colonna n di D.

13.9 Le relazioni di programmazione dinamica sono simili a quelle viste nella soluzione dell'Esercizio 13.7, ma l'interpretazione della matrice D è diversa. Nell'allineamento locale, $D[i, j]$ dà il punteggio del migliore allineamento tra un suffisso di $p_1 \ldots p_i$ ed un suffisso di $t_1 \ldots t_j$, dove tali suffissi possono essere anche vuoti. Per questo, l'inizializzazione prevede $D[i, 0] = 0$ per $0 \leq i \leq m$ e $D[0, j] = 0$ per $0 \leq j \leq n$. Siccome i punteggi devono tenere conto anche dell'allineamento tra suffissi vuoti, che ha punteggio nullo, la relazione di programmazione dinamica diventa:

$$D[i, j] = \max\{D[i, j-1] - 2, D[i-1, j] - 2, D[i-1, j-1] + \delta_{ij}, 0\}$$

dove δ_{ij} ha lo stesso significato dell'Esercizio 13.7. Si noti che adesso ogni $D[i, j]$ è non negativa. Al termine, il risultato del problema si trova nella casella $D[i, j]$ che contiene il massimo tra tutte le caselle di D.

13.10 La procedura printdiff() prende in input la tabella D, le due stringhe P e T e gli indici i, j della posizione sotto esame. La procedura viene chiamata con $i = m$ e $j = n$. Se i caratteri corrispondenti sono uguali, si chiama ricorsivamente printdiff() sulla posizione $i - 1, j - 1$, corrispondente all'unico sottoproblema risolto ricorsivamente da lcs(). Se sono diverse, la procedura lcs() ha scelto il massimo fra i due sottoproblemi $i, j - 1$ e $i - 1, j$; si verifica quindi quale sia il valore massimo per muoversi di conseguenza. Se sono uguali, viene arbitrariamente scelto il primo.

```
printdiff(integer[][] D, ITEM[] P, ITEM[] T, integer i, integer j)
  if i > 0 and j > 0 and P[i] = T[j] then
  |   printdiff(D, P, T, i − 1, j − 1)
  |   print "" P[i]
  else if j > 0 and (i = 0 or D[i, j − 1] ≥ D[i − 1, j]) then
  |   printdiff(D, P, T, i, j − 1)
  |   print "–" T[j]
  else if i > 0 and (j = 0 or D[i, j − 1] < D[i − 1, j] then
  |   printdiff(D, P, T, i − 1, j)
  |   print "+" P[i]
```

13.12 Come accennato nella soluzione dell'Esercizio 12.2, è possibile risolvere questo problema tramite programmazione dinamica. Sia t_i il valore del massimo sottovettore che termina in i. Se conosco t_{i-1}, posso calcolare t_i osservando due casi: se $A[i] + t_{i-1}$ ha un valore positivo, questo è il valore del massimo sottovettore che termina in i; se invece ha valore negativo, il massimo sottovettore che termina in i ha valore 0:

$$t_i = max(t_{i-1} + A[i], 0)$$

Nel codice seguente, questo viene calcolato passo passo dalla variabile *here*, che assume il valore max(*here* + $A[i]$, 0). Poiché il sottovettore di valore massimo deve terminare in qualche posizione, calcolando nella variabile m il valore massimo fra tutti i valori *here*, si ottiene il valore massimo per l'intero vettore. Questo algoritmo ha costo $O(n)$.

```
integer lmaxsum(integer[] A, integer n)
  integer max ← 0
  integer here ← 0
  for i ← 1 to n do
  |   here ← max(here + A[i], 0)
  |   max ← max(here, max)
  return max
```

Capitolo 14

GREEDY

Nella vita di tutti i giorni, si adottano strategie che danno immediatamente buoni risultati, ma che a lungo andare si rivelano scadenti. È questa la strategia dell'ingordo (o *greedy*):

"Meglio un uovo oggi che una gallina domani".

14.1 ESEMPIO – [Resto col numero minimo di monete]
Si supponga di avere monete da 50, 10, 5 e 1 centesimo, e di dover dare un resto di 72 centesimi. Senza neanche rendercene conto, utilizziamo la filosofia dell'ingordo, scegliendo una moneta da 50 centesimi, poi due da 10, e infine due da 1. La soluzione prodotta è formata da cinque monete ed è ottima, nel senso che non ce n'è una con un numero minore di monete. L'ottimalità della soluzione, però, non dipende dalla furbizia dell'ingordo, ma da una proprietà delle monete! Si supponga di avere monete da 11, 5 e 1 centesimo, e di dover dare un resto di 15 centesimi. L'ingordo giudicherà la moneta da 11 più "appetibile" delle altre, e darà un resto con una moneta da 11 e quattro da 1, utilizzando cinque monete, mentre la soluzione ottima è data da tre monete da 5 centesimi!

14.2 ESEMPIO – [Algoritmi di Dijkstra e Johnson]
Ad ogni passo, gli algoritmi di Dijkstra e Johnson effettuano la scelta ingorda di selezionare il nodo che può essere raggiunto con costo minore fra quelli non ancora visitati.

Il metodo *greedy* si può applicare a quei problemi di ottimizzazione in cui occorre selezionare un sottoinsieme S "ottimo" di oggetti, che verificano certe proprietà, da un insieme A di n oggetti. La procedura greedy() mostra uno "scheletro" di codice che adotta tale metodo.

Un algoritmo *greedy* ordina dapprima gli oggetti in base ad un criterio di "appetibilità". La soluzione del problema è poi costruita in modo incrementale considerando gli oggetti uno alla volta e aggiungendo ogni volta l'oggetto più appetibile, se possibile. In altri termini, l'algoritmo effettua una sequenza di scelte, preferendo ogni volta la scelta che fornisce immediatamente il miglior risultato. Fatta la scelta, è successivamente risolto un sottoproblema dello stesso tipo ma di dimensione più piccola. Il problema più piccolo dipende dalla sequenza di scelte passate, ma non dalle scelte future.

```
SET greedy(SET A)
  SET S ← Ø
  {ordina A per "appetibilità" decrescente}
  foreach a ∈ A do                    % Secondo l'ordinamento appena calcolato
  |   if a può essere aggiunto ad S then
  |   |  S ← S ∪ {a}
  return S
```

Ovviamente, non sempre esiste un algoritmo *greedy* che trovi la soluzione ottima di un qualsiasi problema di ottimizzazione, come abbiamo già potuto osservare per il problema del resto col minimo numero di monete. Affinché un algoritmo *greedy* fornisca la soluzione ottima di un problema, occorre che siano verificate due proprietà, tra loro correlate:

- *Sottostruttura ottima*. Per mostrare che una scelta *greedy* riduce il problema ad un sottoproblema più piccolo dello stesso tipo, occorre dimostrare che una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi.
- *Scelta greedy*. Dopo aver fornito una caratterizzazione matematica della soluzione ottima, occorre dimostrare che tale soluzione può essere modificata in modo da utilizzare una prima scelta *greedy*, che riduce il problema ad un sottoproblema più piccolo dello stesso tipo, e poi mostrare per induzione che il procedimento può essere esteso ad una sequenza di scelte *greedy*.

Iniziamo studiando due varianti di problemi che abbiamo già risolto con la programmazione dinamica, per poi vedere un esempio notevole di problema per il quale la tecnica *greedy* si comporta bene: trovare il minimo albero di copertura di un grafo pesato.

14.1 ZAINO REALE

ZAINO REALE Dati un insieme di n oggetti, con profitti $p_1, ..., p_n$ e volumi $v_1, ..., v_n$, tutti interi positivi, e un intero positivo C, la capacità dello zaino, il problema consiste nello scegliere oggetti o porzioni di essi in modo da massimizzare il profitto degli oggetti scelti senza superare la capacità; ovvero determinare un vettore $x = [x_1, ..., x_n]$ di reali compresi tra 0 e 1 tali che il profitto totale $p(x) = \sum_{1 \le i \le n} p_i x_i$ sia massimo e il volume $v(x) = \sum_{1 \le i \le n} v_i x_i \le C$.

Abbiamo visto che il problema originale (che viene anche chiamato ZAINO 0-1 o ZAINO INTERO, ovvero "prendere/1 o lasciare/0") gode di sottostruttura ottima. È facile adattare la dimostrazione nel caso di scelta frazionaria; una volta scelta una certa quantità di un oggetto, possiamo ridurci al sottoproblema in cui c'è un oggetto in meno e una capacità inferiore. Una domanda legittima è chiedersi se questo problema è anche dotato di scelta *greedy*. Facciamo qualche tentativo.

14.3 ESEMPIO – [Profitto decrescente]

Supponiamo di scegliere oggetti in ordine di profitto decrescente. È facile trovare un controesempio per cui questo approccio non funziona: $p_1 = 3$, $p_2 = 2$, $p_3 = 2$, $v_1 = 2$, $v_2 = 1$, $v_3 = 1$, $C = 2$. La scelta *greedy* richiede di prendere il primo oggetto, con profitto 3 e volume 2, riempiendo lo zaino. Ma scegliere l'insieme dato dal secondo e terzo oggetto ha profitto ottimo 4 e volume 2, migliore della scelta *greedy*.

14.4 ESEMPIO – [Volume crescente]

Neanche scegliere oggetti per volume crescente funziona. Il controesempio è dato da $p_1 = 1$, $p_2 = 1$, $p_3 = 3$, $v_1 = 1$, $v_2 = 1$, $v_3 = 2$ e capacità 2. La scelta *greedy* seleziona il primo e secondo oggetto, con profitto 2 e volume 2. Ma scegliere il terzo oggetto ha profitto ottimo 3 e volume 2, migliore della scelta *greedy*.

La scelta corretta è scegliere gli oggetti per "profitto specifico" decrescente, ovvero per rapporto profitto/volume. Si cerca poi di scegliere la porzione più grande possibile dell'oggetto che dà il maggior profitto per unità di volume, senza superare la capacità C; dopo di che, si risolve il sottoproblema dato dai rimanenti oggetti e dalla capacità residua.

Dimostriamo che la scelta *greedy* è sempre corretta. Supponiamo che esista una soluzione ottima x, che contenga una quantità $x[1] < 1$ tale che $x[1]v[1] < C$; ovvero, non contiene la porzione più grande possibile dell'oggetto 1 (quello con profitto specifico più alto dopo l'ordinamento). Costruiamo una nuova soluzione x' dove $x'[1] = \min\{v[1]/C, 1\}$, ovvero massimizziamo la porzione dell'oggetto 1 inserita nella soluzione, risolvendo poi il sottoproblema dato dai restanti $n - 1$ oggetti con capacità $C - x'[1]v[1]$. La soluzione x' ha profitto totale maggiore o uguale al profitto di x, in quanto l'oggetto 1 ha il più alto profitto specifico.

Alla fine, saranno scelti per intero gli oggetti più appetibili, può essere scelta una frazione dell'oggetto immediatamente successivo, ed i rimanenti oggetti non saranno scelti. Il tempo di calcolo è $O(n \log n)$, dominato dal costo dell'ordinamento.

```
zaino(integer[] p, integer[] v, integer C, integer n, integer[] x)
  {ordina p e v in modo che p[1]/v[1] ≥ p[2]/v[2] ≥ ··· ≥ p[n]/v[n]}
  integer i ← 1
  while i ≤ n and C > 0 do
      if v[i] ≥ C then
          x[i] ← C / v[i]
          C ← 0;
      else
          x[i] ← 1
          C ← C − v[i]
      i ← i + 1
  for j ← i to n do x[j] ← 0
```

14.2 INSIEME INDIPENDENTE DI INTERVALLI

Nel capitolo 13, abbiamo descritto un algoritmo basato sulla programmazione dinamica per trovare l'insieme indipendente massimale per intervalli pesati. Consideriamo ora un caso particolare, in cui tutti gli intervalli hanno peso 1.

INSIEME INDIPENDENTE DI INTERVALLI Dati n intervalli distinti $I_1 = [a_1, b_1[, \ldots, I_n = [a_n, b_n[$ della retta reale, trovare un insieme indipendente massimo (ovvero un sottoinsieme di massima cardinalità formato da intervalli tutti disgiunti tra loro).

La soluzione parte da premesse simili a quelle relative agli intervalli pesati. La dimostrazione di sottostruttura ottima resta valida. Ordiniamo gli intervalli per estremi finali non decrescenti, ovvero in modo che $b_1 \leq b_2 \leq \cdots \leq b_n$. Inseriamo il primo intervallo I_1 e ci riduciamo al sottoproblema dato da tutti gli intervalli che non intersecano I_1.

Per provare la correttezza di questo algoritmo, dobbiamo dimostrare che la scelta *greedy* è sempre corretta; ovvero che esiste una soluzione ottima che include il primo intervallo (dopo l'ordinamento). Sia S una soluzione ottima; se $I_1 \in S$, la dimostrazione è banalmente completata. Se $I_1 \notin S$, sia I_s l'intervallo con minor estremo finale in S. Si consideri la soluzione $S - \{I_s\} \cup \{I_1\}$; poiché $b_1 < b_s$, sostituire I_s con I_1 non modifica l'indipendenza dell'insieme (l'intervallo I_s era indipendente da tutti gli altri intervalli in S, e l'intervallo I_1 che termina prima resta indipendente); inoltre, la cardinalità resta la stessa. Abbiamo quindi dimostrato che $S - \{I_s\} \cup \{I_1\}$ è un insieme indipendente massimo che contiene I_1.

L'algoritmo independentSet() scandisce gli intervalli da sinistra verso destra e ad ogni passo include nella soluzione l'intervallo che finisce prima e che non interseca gli intervalli già inclusi. Il sottoinsieme di intervalli selezionati è un insieme indipendente massimo. La complessità è $O(n \log n)$, dovuta all'ordinamento.

```
SET independentSet(integer[] a, integer[] b)
  {ordina a e b in modo che b[1] ≤ b[2] ≤ ··· ≤ b[n]}
  SET S ← Set()
  S.insert(1)                                  % Ultimo intervallo inserito
  integer ultimo ← 1
  for i ← 2 to n do                            % Controllo indipendenza
      if a[i] ≥ b[ultimo] then
          S.insert(i)
          ultimo ← i
  return S
```

14.3
MINIMO ALBERO DI COPERTURA

Si consideri il seguente problema di ottimizzazione su grafi pesati.

MINIMO ALBERO DI COPERTURA (MINIMUM SPANNING TREE) Dato un grafo non orientato e connesso $G = (V, E)$, con pesi $w(u, v)$ non negativi sugli archi, trovare un albero di copertura $G' = (V, T)$ per G, cioè un albero avente tutti i nodi di V, ma solo $n - 1$ degli m archi in E, tale che la somma $\sum_{(u,v) \in T} w(u, v)$ sia la più piccola possibile.

14.5 ESEMPIO – [Minimo albero di copertura]
La Fig. 14.1 illustra un grafo non orientato pesato G e due alberi di copertura per G, dei quali il secondo è il minimo albero di copertura.

Questo problema può essere risolto con molti algoritmi, dei quali i più noti sono quelli di Kruskal (1956) e Prim (1957). Entrambi questi algoritmi sono basati sul principio *greedy*; illustriamo prima gli algoritmi, e poi discutiamo i principi alla base della loro correttezza.

14.3.1
ALGORITMO DI KRUSKAL

L'algoritmo di Kruskal analizza gli archi in ordine crescente di peso; un generico arco è aggiunto all'albero T se non forma circuiti con gli archi inseriti in T.

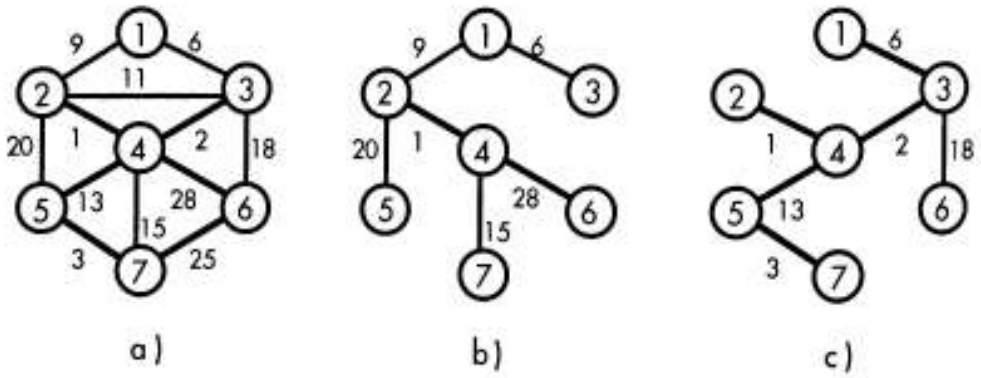


Fig. 14.1 Minimo albero di copertura. a) Un grafo pesato; b) una soluzione ammissibile; c) una soluzione ottima.

14.6 ESEMPIO – [Algoritmo di Kruskal]
La Fig. 14.2 mostra l'esecuzione dell'algoritmo di Kruskal sul grafo di Fig. 14.1. L'ordinamento degli archi è: [2,4], [3,4], [5,7], [1,3], [1,2], [2,3], [4,5], [4,7], [3,6], [2,5], [6,7], [4,6].

Il maggior sforzo computazionale dell'algoritmo di Kruskal è dato dall'ordinamento iniziale e dalla verifica che l'aggiunta di un nuovo arco non provochi la formazione di un circuito. L'ordinamento iniziale degli archi costa $O(m \log n)$ tempo, dato che $\log m \leq 2 \log n$. Inoltre, poiché la costruzione di T avviene per unione di insiemi disgiunti, cioè di due componenti connesse che si "fondono" in una sola con l'aggiunta del nuovo arco, è possibile uti-

lizzare una struttura merge-find MFSET [capitolo 10] per mantenere le componenti connesse di T, inizializzato con n componenti distinte, ciascuna formata da un singolo nodo del grafo. In questo modo, la verifica che un arco appartenga a due componenti distinte (cioè non provochi un circuito in T) ed anche il suo eventuale inserimento in T richiedono tempo ammortizzato costante $O(1)$ usando le operazioni find() e merge(). Poiché il ciclo **for** è ripetuto m volte, la complessità dell'algoritmo risulta essere $O(m \log n)$.

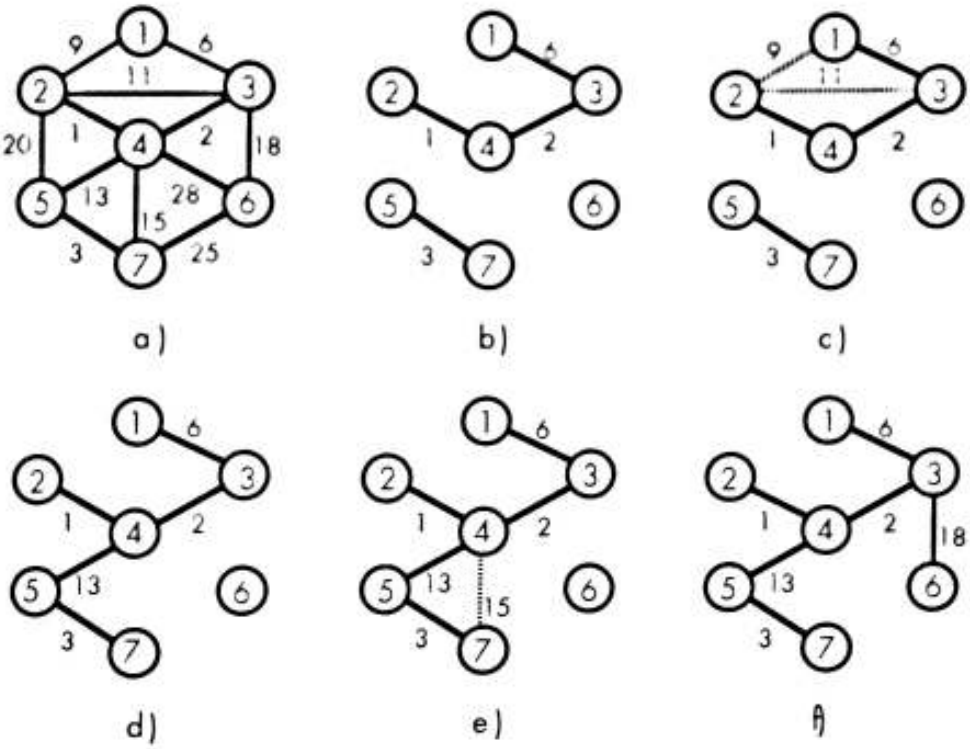


Fig. 14.2 Algoritmo di Kruskal; a) Il grafo di ingresso. b) la foresta di copertura dopo l'inclusione di [2,4], [3,4], [5,7] e [1,3]; c) [1,2] e [2,3] non sono inclusi; d) inclusione di [4,5]; e) [4,7] non è incluso; f) inclusione di [3,6] e soluzione ottima.

Per semplificare l'algoritmo di Kruskal, assumiamo che prenda in input un vettore di record ARCO, contenente i campi u, v e *peso*, corrispondenti agli estremi dell'arco e al suo peso.

Data l'usuale rappresentazione di un grafo con vettori di adiacenza, è facile vedere che il vettore degli archi può essere costruito in $O(n + m)$ tempo. L'albero T è restituito come un insieme di archi, che può essere realizzato con una lista non ordinata, in modo che l'inserimento di un arco in T richieda tempo $O(1)$.

Per risparmiare iterazioni inutili dopo che è stata raggiunta la soluzione ottima, la procedura kruskal() introduce una variabile intera c che conta il numero di inserimenti di archi nell'albero di copertura T, uscendo dal ciclo appena sono stati effettuati $n - 1$ inserimenti. L'ordine di grandezza della complessità resta comunque $O(m \log n)$.

```
SET kruskal(ARCO[] A, integer n, integer m)
    SET T ← Set()
    MFSET M ← Mfset(n)
    {ordina A[1, ..., m] in modo che A[1].peso ≤ ··· ≤ A[m].peso}
    integer c ← 0
    integer i ← 1
    while c < n − 1 and i ≤ m do                % Termina quando l'albero è costruito
        if M.find(A[i].u) ≠ M.find(A[i].v) then
            M.merge(A[i].u, A[i].v)
            T.insert(A[i])
            c ← c + 1
        i ← i + 1
    return T
```

14.3.2
ALGORITMO DI PRIM

L'algoritmo di Prim parte da un albero T contenente uno specifico nodo radice r, e accresce T aggiungendo sempre l'arco con peso minore fra quelli che "escono" da T, ovvero fra quelli che uniscono un nodo di T con un nodo non incluso in T.

L'algoritmo prende in input un grafo G e un nodo radice r; come nel caso dei cammini minimi [capitolo 11], la procedura ha accesso ad una funzione di costo w, lasciando alla realizzazione effettiva l'onere di definire dove debbano essere memorizzati tali costi (per es., nella matrice di adiacenza del grafo oppure nelle celle che formano la lista di adiacenza di un nodo). L'albero T viene rappresentato da un vettore di padri p.

La struttura di dati utilizzata in questo caso è una coda con priorità (min-heap). Tutti i nodi vengono inseriti nella coda con priorità $+\infty$, tranne il nodo r che ha priorità 0. La priorità del nodo u corrisponde al peso minimo di un arco che raggiunge u a partire da un nodo compreso in T, mentre $pos[u]$ è la posizione che u occupa nello heap. Quando un nodo u viene estratto dalla coda, $[p[u], u]$ rappresenta l'arco con peso minore fra quelli che escono dall'albero; sia l'arco $[p[u], u]$ che il nodo u entrano a far parte dell'albero T, indicato con $pos[u]$ = **nil**. Tutti gli archi incidenti ad u vengono analizzati, e se permettono di raggiungere un nodo v non ancora nell'albero ($pos[v] \neq$ **nil**) con costo inferiore, la priorità nella coda viene aggiornata, così come il vettore dei padri.

Utilizzando una coda con priorità realizzata tramite heap binario, il ciclo di inserimento dei nodi nella coda con priorità viene eseguito n volte, ognuna con costo $O(\log n)$; il ciclo principale viene eseguito n volte, ognuna con costo $O(\log n)$ per l'estrazione; il ciclo esterno viene eseguito m volte, ognuna con costo $O(\log n)$. La complessità è quindi $O(m \log n)$, asintoticamente uguale all'algoritmo di Kruskal.

```
prim(GRAPH G, NODE r, integer[] p)
  PRIORITYQUEUE Q ← MinPriorityQueue()
  PRIORITYITEM[] pos ← new PRIORITYITEM[1 ... G.n]
  foreach u ∈ G.V() − {r} do
  |   pos[u] ← Q.insert(u, +∞)
  pos[r] ← Q.insert(r, 0)
  p[r] ← 0
  while not Q.isEmpty() do
  |   NODE u ← Q.deleteMin()
  |   pos[u] ← nil
  |   foreach v ∈ G.adj(u) do
  |   |   if pos[v] ≠ nil and w(u, v) < pos[v].priorità then
  |   |   |   Q.decrease(pos[v], w(u, v))
  |   |   |   p[v] ← u
```

14.7 ESEMPIO – [Algoritmo di Prim]
La Fig. 14.3 mostra l'esecuzione dell'algoritmo di Prim sul grafo di Fig. 14.1 a partire dal nodo 1.

È possibile dare una versione dell'algoritmo di Prim che richiede tempo $O(n^2)$, sostituendo la struttura min-heap con una lista o un vettore non ordinati, analogamente a quanto visto per l'algoritmo di Dijkstra per i cammini minimi. Questa versione risulta vantaggiosa per grafi densi (dove $m = \Theta(n^2)$) mentre quella con min-heap è migliore per grafi sparsi (dove $m = O(n)$).

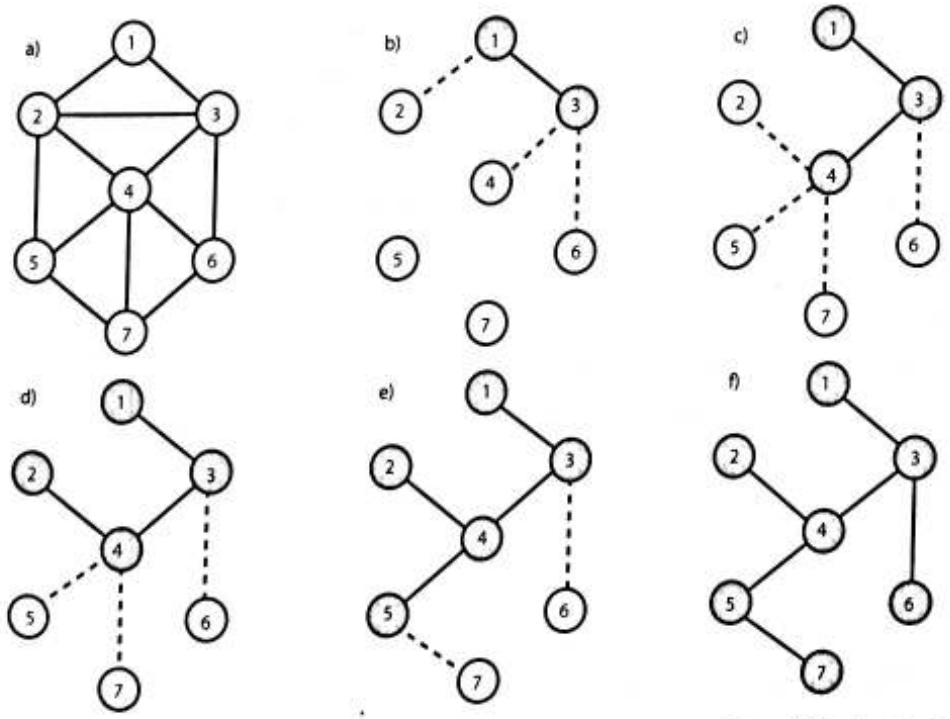


Fig. 14.3 Algoritmo di Prim. I nodi in grigio sono quelli estratti dalla coda. Gli archi tratteggiati sono quelli che "escono" da T; gli altri archi rappresentano T. a) Il grafo di ingresso; b) l'albero dopo aver estratto 1 e 3; c), d), e) l'albero dopo aver estratto 4, 2, 5, rispettivamente; f) soluzione ottima.

14.3.3
DIMOSTRAZIONE DI CORRETTEZZA

Per dimostrare la correttezza degli algoritmi di Kruskal e Prim abbiamo bisogno di alcune definizioni. Un *taglio* $(S, V-S)$ è una partizione di V in due sottoinsiemi non vuoti. Un arco $[u, v]$ *attraversa* il taglio se $u \in S$ e $v \in V-S$. Vale il seguente teorema.

TEOREMA 1

Sia $A \subseteq E$ un sottoinsieme di un minimo albero di copertura di G. Sia $(S, V-S)$ un taglio non attraversato da alcun arco di A. Sia $[u, v]$ l'arco di E con peso minimo fra quelli che attraversano il taglio. Allora l'insieme $A \cup \{[u, v]\}$ è un sottoinsieme di un minimo albero di copertura di G.

DIMOSTRAZIONE Sia T un minimo albero di copertura che contiene A. Se l'arco $[u, v]$ appartiene a T, la proprietà è banalmente dimostrata: $A \cup \{[u, v]\} \subseteq T$. Altrimenti, si noti che per definizione di albero u e v sono connessi da un cammino c in T. Poiché per ipotesi $u \in S$ e $v \in V-S$, esiste almeno un arco $[x, y]$ in c che attraversa il taglio. Si noti che $[x, y] \notin A$, visto che nessun arco di A attraversa il taglio.

Se sostituiamo $[u, v]$ al posto di $[x, y]$, otteniamo un insieme di archi $T' = T - \{[x, y]\} \cup \{[u, v]\}$ che è ancora un albero (perché una volta sconnesso $[x, y]$, l'albero si spezza in due componenti che si riuniscono con $[u, v]$); inoltre, il peso di T' è inferiore o uguale al peso di T (in quanto per definizione $w(u, v) \leq w(x, y)$). Questo dimostra la proprietà, in quanto T' è un minimo albero di copertura, e sappiamo che $A \cup \{[u, v]\} \subseteq T'$.

Per concludere la dimostrazione di correttezza di Kruskal e Prim, dobbiamo provare che quando un arco $[u, v]$ viene aggiunto all'insieme A di archi, l'ipotesi della proprietà sia rispettata, ovvero esista un taglio $(S, V-S)$ tale che nessun arco di A attraversi il taglio e $[u, v]$ sia l'arco di peso minimo fra quelli che attraversano il taglio.

Nel caso di Kruskal, l'arco $[u, v]$ che viene aggiunto connette due componenti connesse di $G' = (V, A)$; ponendo S uguale ad una delle due componenti, per definizione nessun arco di A attraversa il taglio (altrimenti sarebbero un'unica componente); essendo l'arco $[u, v]$ il primo nell'ordinamento ad attraversare il taglio ed unire tali componenti, esso ha il peso minore fra quelli che attraversano il taglio.

Nel caso di Prim, l'arco $[u, v]$ connette la componente connessa S formata dai nodi già visitati dell'albero e la componente $V-S$ che include tutti gli altri. Grazie all'utilizzo della coda con priorità, l'arco ha il peso minimo fra tutti quelli che escono dall'albero (ovvero attraversano il taglio).

14.4
REALITY CHECK

Un *algoritmo di compressione* prende in input una sequenza di bit e produce in output una sequenza di bit più corta, detta *sequenza compressa*. Un algoritmo di decompressione prende in input una sequenza compressa e cerca di ricostruire la sequenza originale. Se la ricostruzione avviene senza errori, la tecnica di compressione-decompressione

è di tipo *lossless* (senza perdite); altrimenti è di tipo *lossy* (con perdita). Il problema della compressione è di fondamentale importanza nell'informatica, sia per quanto riguarda lo spazio di memorizzazione, ma anche e soprattutto per risparmiare tempo di trasmissione.

Esistono innumerevoli tecniche per la codifica; una tecnica molto potente è la compressione di Huffman (1952) per la codifica di testi, utilizzata ancora oggi nei programmi quali PkZip. È una tecnica di codifica a caratteri: ogni carattere c viene codificato da una sequenza di bit b. Nella codifica ASCII, tutti i caratteri sono rappresentati con 8 bit. Ma in un qualunque testo scritto, i caratteri non appaiano tutti con la stessa frequenza. Ad esempio, in italiano la lettera `j` è molto meno frequente della `a`. In una *codifica con lunghezza variabile*, i caratteri più frequenti vengono rappresentati con una codifica più breve rispetto ai caratteri meno frequenti.

ESEMPIO – [Codifica di Huffman]

Si consideri un testo composto da n caratteri dall'alfabeto {a, b, c, d, e, f}, che appaiono con le frequenze 45%, 13%, 12%, 16%, 9%, 5%, nell'ordine. Se questo testo viene codificato in ASCII, richiederà $8n$ bit. Se viene codificato con 3 bit per carattere (sufficienti a rappresentare i 6 possibili caratteri), richiederà $3n$ bit. Se viene codificato nel modo seguente: `a = 0`, `b = 101`, `c = 100`, `d = 111`, `e = 1101`, `f = 1100`, date le frequenze di ognuno dei caratteri si ottiene una lunghezza pari a:

$$(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot n = 2.24n$$

Potremmo chiederci perché non si è utilizzata una codifica ancora più breve, quale: `a = 0`, `b =1`, `c = 00`, `d = 01`, `e = 10`, `f = 11`. Il motivo è semplice: perché sarebbe impossibile decomprimere il testo "`000`", che potrebbe corrispondere ad "`aaa`", "`ca`" o "`ac`". La codifica proposta nell'esempio non ha questo problema, essendo una *codifica a prefissi*: nessun codice è prefisso di un altro codice. Questa è una condizione necessaria per permettere la decodifica.

ESEMPIO – [Prefissi telefonici]

Il prefisso di Milano è 02; il prefisso di Bologna è 051; il prefisso di Trento è 0461. Non a caso le province hanno 3.2, 0.98, 0.52 milioni di abitanti: a provincia più grande (abitanti più "frequenti") corrisponde un prefisso più breve. Inoltre, non esiste il prefisso 02X, con X una qualunque cifra: sarebbe indistinguibile da Milano.

L'obiettivo è costruire la minima codifica a caratteri con lunghezza variabile: data una sequenza s, una codifica c è minima per s se non esiste un'altra codifica con cui s possa essere compressa impiegando un numero inferiore di bit. L'algoritmo di Huffman risolve questo problema di ottimizzazione tramite la tecnica *greedy*.

Si inseriscono i caratteri in un min-heap, con priorità data dalla frequenza. Vengono estratti i due caratteri c_1 e c_2 con frequenze più basse f_1 e f_2; si crea un nuovo carattere "virtuale" c che ha come frequenza la somma delle frequenze $f_1 + f_2$, che viene inserito nel min-heap. I caratteri c, c_1 e c_2 vengono organizzati ad albero binario, con c_1 e c_2 figli destro e sinistro di c. L'operazione di doppia estrazione ed inserimento viene ripetuta per

$n - 1$ volte, fino a quando non si rimane con la radice dell'albero. A questo punto, si assegna il codice 0 agli archi che collegano un nodo al suo figlio sinistro, e 1 agli archi che collegano un nodo al suo figlio destro. Tutti i caratteri originali (non virtuali) sono foglie; la loro codifica è data dai codici sugli archi che si trovano nel percorso radice-foglia.

La dimostrazione di correttezza non è complessa, ma è lunga e tediosa. Basti notare che i caratteri meno frequenti hanno codifica più lunga, in quanto si trovano a maggior profondità nell'albero.

14.1 [Zaino intero] Si dimostri con un controesempio che, al contrario dello ZAINO reale, il problema dello ZAINO INTERO non si può risolvere con la tecnica *greedy* suggerita in questo capitolo.

14.2 [Algoritmo di Kruskal] Si assuma che i pesi degli archi di un grafo non orientato G siano interi e compresi nell'intervallo $1, ..., n$, dove n è il numero di nodi di G. È possibile abbassare la complessità dell'algoritmo di Kruskal?

14.3 [Grafi blu] Un grafo G ha archi colorati di rosso e blu. Scrivere un algoritmo che restituisca un albero di copertura per G con il numero minimo di archi rossi.

14.4 [Algoritmo di Prim] Qual è la complessità dell'algoritmo di Prim se invece di utilizzare una realizzazione della coda con priorità basata su heap binario, si utilizza un heap di Fibonacci [capitolo 11.5]?

14.5 [Shortest Job First] Dati n programmi tali che l'i-esimo programma richiede t_i unità di tempo di esecuzione, trovare una sequenza di esecuzione (una permutazione) $A_1, ..., A_n$ di $\{1, ..., n\}$ che minimizzi il tempo medio di completamento di tutti i programmi, dove il tempo di completamento del programma A_i è dato dalla somma dei tempi di esecuzione dei programmi $A_1, A_2, ..., A_i$.

14.6 [Minima colorazione di intervalli] Siano dati n intervalli distinti $I_1 = [a_1, b_1[, ..., I_n = [a_n, b_n[$ della retta reale. Si progetti un algoritmo greedy di complessità $O(n \log n)$ per colorare ciascun intervallo in modo che intervalli che si intersecano siano colorati con colori diversi e che il numero di colori usati sia minimo.

14.7 [Torri cellulari lineari] Si consideri una strada diritta di lunghezza k punteggiata da n abitazioni collocate ai chilometri $k_1, k_2, ..., k_n$, con $k_i \in [0, k]$ (come esempio, considerate una di quelle strade americane che sembrano infinite). Una compagnia telefonica vuole collocare un insieme di torri cellulari, in modo tale che ogni abitazione sia ad una distanza massima d da una torre. Progettare un algoritmo per risolvere questo problema utilizzando il numero minimo di torri e discuterne la complessità.

14.8 [Sciatori e sci] Siano dati n sciatori di altezza $p_1, \ldots, p_n$, e n paia di sci di lunghezza $s_1, \ldots, s_n$. Il problema è assegnare ad ogni sciatore un paio di sci, in modo da minimizzare la differenza totale fra le altezze degli sciatori e la lunghezza degli sci; ovvero, se allo sciatore i-esimo è assegnato il paio di sci $h(i)$, minimizzare la seguente quantità:

$$\sum_{i=1}^{n} |p_i - s_{h(i)}|$$

Si consideri il seguente algoritmo *greedy*. Si individui la coppia (sciatore, sci) con la minima differenza. Si assegni allo sciatore questo paio di sci. Si ripeta con gli sciatori restanti fino a quando non si è terminato. Provare la correttezza di questo algoritmo o trovare un controesempio.

14.9 [Archi minimi] Sia dato un grafo non orientato $G = (V; E)$ e una funzione di pesi $w : E \to \mathbb{R}$, con pesi sugli archi tutti distinti. Siano e_1, e_2 ed e_3 l'arco con peso minimo, l'arco con il secondo peso minimo e l'arco con il terzo peso minimo, rispettivamente. Confutare o provare le seguenti affermazioni:

- Tutti gli alberi di copertura di peso minimo del grafo G contengono l'arco e_1.
- Tutti gli alberi di copertura di peso minimo del grafo G contengono l'arco e_2.
- Tutti gli alberi di copertura di peso minimo del grafo G contengono l'arco e_3.

14.10 [Massimizzazione e minimizzazione] Si dimostri se i problemi MASSIMO ALBERO DI COPERTURA e CAMMINI MASSIMI sono equivalenti o meno ai problemi MINIMO ALBERO DI COPERTURA e CAMMINI MINIMI.

14.1 Si consideri uno zaino di capacità 6 e tre oggetti di volume 3, 3, 4 e profitto 3, 3, 5, rispettivamente; l'algoritmo basato sul valore specifico sceglierebbe il terzo oggetto con profitto totale 5, in quanto il profitto specifico 5/4 è più alto del profitto 1 degli altri due. Ma data la capacità dello zaino, è possibile scegliere i primi due oggetti con profitto totale 6.

14.2 Se tutti i pesi sono compresi nell'intervallo 1, ..., n, è possibile utilizzare Counting Sort [capitolo 2] per ordinare gli archi in tempo $O(m + n)$. L'ordinamento in tempo $O(m \log n)$ viene a mancare e l'algoritmo di Kruskal può essere eseguito in tempo $O(m + n)$.

14.3 È necessario costruire una funzione di peso che associa il peso 1 agli archi rossi, il peso 0 agli archi blu. Si applica poi un algoritmo per ottenere il minimo albero di copertura, che conterrà il numero minimo di archi rossi.

14.5 Il titolo dell'esercizio suggerisce una soluzione. Un algoritmo greedy che minimizza il tempo medio di completamento ordina i programmi per tempo di esecuzione crescente. Dimostriamo che esiste sempre una soluzione ottima che contiene la scelta ingorda.

Supponiamo di avere una soluzione ottima A, rappresentata da una permutazione A_1, A_2, ..., A_n degli indici 1, 2, ..., n. Supponiamo che il programma di durata inferiore sia in posizione A_k, con $k > 1$ (se $k = 1$, abbiamo terminato). Costruiamo una soluzione A', scambiando gli elementi 1 e k della permutazione.

- Tutti i programmi eseguiti dopo il programma k hanno lo stesso tempo di completamento in A e A', perché lo scambio non influisce su di loro.
- Poiché la durata del k-esimo job è più breve di quella del primo, se ne deduce che, quando viene spostato al primo posto, ha un tempo di completamento inferiore; d'altra parte, il tempo di completamento del job in k-esima posizione in entrambe le sequenze è uguale, in quanto comunque devono essere completati tutti i primi k job.
- Tutti i programmi compresi fra 1 e k, estremi esclusi, hanno un tempo di completamento inferiore o uguale in A', perché lo scambio può avere ridotto il tempo di completamento per il primo programma.

Quindi il tempo di completamento totale di A' è inferiore o uguale al tempo di completamento di A; ma poiché A è ottima, questo significa che A e A' hanno lo stesso tempo di completamento e quindi anche A' è ottima.

14.6 È immediato verificare che il numero minimo di colori non può essere inferiore al numero massimo c di sovrapposizioni tra intervalli, calcolato nell'Esercizio 9.8. Supponiamo senza perdere in generalità che tutti gli estremi degli intervalli siano distinti. Ordiniamo gli intervalli per estremi iniziali crescenti, ovvero in modo che $a_1 < a_2 < \cdots < a_n$. Per $j = 1, ..., n$,

consideriamo l'intervallo I_j. Per ogni intervallo I_i con $a_i < a_j$ e che interseca I_j, escludiamo il colore di I_i ed assegnamo ad I_j un colore non escluso. Vediamo che in questo modo sono colorati tutti gli intervalli e che intervalli che si intersecano hanno colori distinti. Se per assurdo esistesse un intervallo I_j non colorato, preceduto nell'ordinamento da m intervalli che lo intersecano, allora esisterebbe una massima sovrapposizione di $m + 1 \leq c$ intervalli. Risulterebbe quindi $m \leq c - 1$ e si giungerebbe alla contraddizione che esiste un colore libero. Quindi I_j è stato colorato. Inoltre, non esistono intervalli colorati con lo stesso colore di I_j perché sono stati esclusi per costruzione. Infine, la colorazione è minima, dato che si sono usati c colori e c è una limitazione inferiore al numero di colori. In pratica, l'algoritmo dapprima calcola c come mostrato nell'Esercizio 9.8 e inserisce i colori in una pila. Indi scandisce i $2n$ estremi $\{a_1, ..., a_n, b_1, ..., b_n\}$ (ordinati in modo crescente) da sinistra verso destra. Se l'elemento scandito è un estremo iniziale, cioè un a_i, allora si estrae un colore dalla pila e lo si assegna ad I_i. Se invece è un estremo finale, cioè un b_i, allora si libera il colore di I_i reinserendolo nella pila. La complessità dell'algoritmo è $O(n \log n)$ a causa dell'ordinamento.

14.7 Un algoritmo *greedy* potrebbe essere il seguente. Innanzitutto, si assuma che i valori k_i siano ordinati, oppure si ordinino in $O(n \log n)$ tempo. Si costruisca una torre al chilometro $k_1 + d$; questa serve l'abitazione 1 e potenzialmente tutte quelle comprese in $[k_1, k_1 + 2d]$, che possono essere eliminate. Si ripete il procedimento con le torri restanti. La soluzione è corretta, in quanto qualunque soluzione ottima può essere trasformata in una soluzione che comprende la torre in $k_1 + d$.

```
torri(integer[] k, integer n, integer d)
  SET S ← Set()
  integer t ← k[1] + d                    % Ultima torre piazzata
  S.insert(t)
  for i ← 2 to n do
      if k[i] > t + d then
          t ← k[i] + d                    % Nuova torre, che copre la prima abitazione esclusa
          S.insert(t)
```

RICERCA LOCALE

Capitolo 15

Talvolta, anche una strategia estremamente "miope" può dare buoni risultati:

"Coltiviamo senza guardare più in là del proprio orticello".

In altri termini, se si conosce una soluzione ammissibile di un problema di ottimizzazione, si può cercare di trovarne una migliore che sia in qualche modo "vicina" a quella precedente. Data una soluzione ammissibile *Sol* del problema, si definisce un *intorno* $I(Sol)$ di questa soluzione come un insieme di soluzioni ammissibili che sono ottenibili da essa in base a certi criteri di trasformazione. Si può successivamente sostituire la soluzione con una che fa parte del suo intorno ed è "migliore" di essa, e ripetere il procedimento finché non si riesce più a migliorare. Tale strategia è detta ricerca locale e gli algoritmi basati su di essa hanno la seguente struttura generale:

```
ricercaLocale()
  Sol ← una soluzione ammissibile del problema
  while esiste S ∈ I(Sol) che è migliore di Sol do Sol ← S
  return Sol
```

15.1 ESEMPIO – [Intorno]

Si consideri il problema del minimo albero di copertura, introdotto nel capitolo 14. Una soluzione ammissibile *Sol* del problema è un albero di copertura, un albero avente tutti i nodi di V, ma solo $n - 1$ degli m archi in E. Un possibile intorno è il seguente:

$I(Sol) = \{T$: T è un albero di copertura ottenuto da *Sol* aggiungendo un arco non in *Sol* e cancellando un altro arco dall'unico circuito che si è formato$\}$

Un criterio di miglioramento consiste nello scegliere un albero $T \in I(Sol)$ in cui l'arco aggiunto ha un peso più piccolo dell'arco cancellato. Un esempio è fornito nella Fig. 15.1.

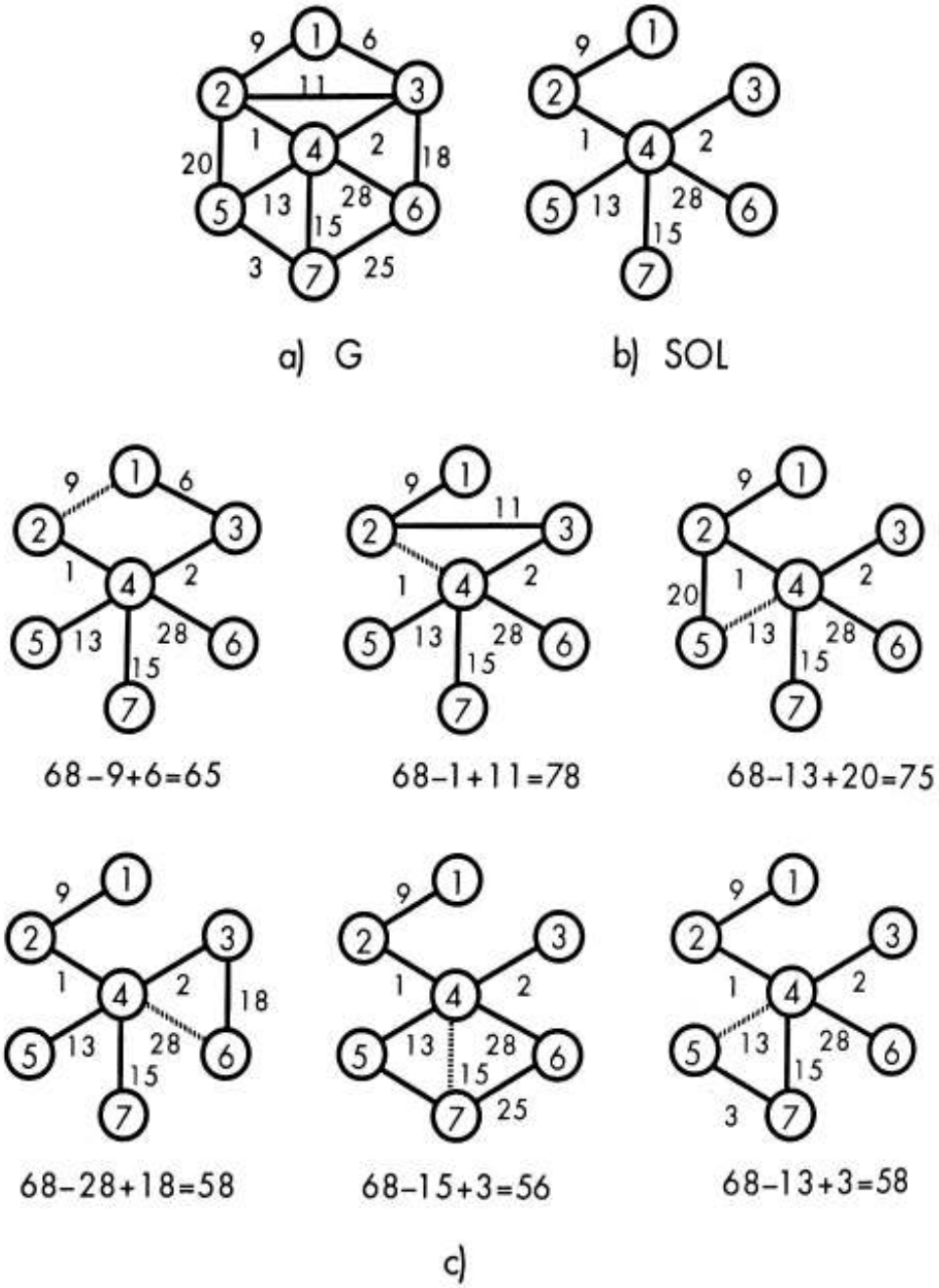


Fig. 15.1 Albero di copertura. a) il grafo G della Fig. 14.1; b) una soluzione ammissibile *Sol*; c) sei delle tredici soluzioni dell'intorno *I*(*Sol*) ottenute aggiungendo un arco e togliendone uno dal circuito formatosi: la migliore si ottiene sostituendo [4, 7] con [5, 7].

In generale, il procedimento di ricerca locale si arresta su una soluzione che è un ottimo locale, cioè che non può essere migliorata attraverso le trasformazioni possibili, ma che può non essere l'ottimo globale del problema. Inoltre, partendo da soluzioni iniziali diverse si possono raggiungere ottimi locali diversi, come illustrato nella Fig. 15.2. Ovviamente, se *I*(*Sol*) comprende tutte le soluzioni ammissibili del problema, allora ogni ottimo locale è anche ottimo globale, e la ricerca locale dà sempre la soluzione ottima del problema. Una tale ricerca, però, ha senso se la cardinalità di *I*(*Sol*) è molto più piccola della cardinalità dell'insieme di tutte le soluzioni ammissibili, e tale restrizione sulla cardinalità di *I*(*Sol*) può far arrestare la ricerca su un ottimo locale che non è quello globale!

Fig. 15.2 Ottimi locali e globali.

Perché l'intero procedimento abbia complessità polinomiale, occorre che la ricerca della soluzione S nell'intorno $I(Sol)$ abbia complessità polinomiale e che il numero complessivo di passi di "miglioramento", cioè di ripetizioni del ciclo **while**, sia anch'esso polinomiale. Per alcuni problemi, ma non per tutti, è possibile definire intorni di cardinalità "piccola" per i quali ogni ottimo locale è anche un ottimo globale che inoltre può essere trovato in tempo polinomiale con una ricerca locale.

15.2 ESEMPIO – [Minimo albero di copertura]

È possibile dimostrare che l'intorno definito nell'Esempio 15.1 per il minimo albero di copertura è tale che ogni minimo locale è anche globale. Pertanto una procedura di ricerca locale è in grado di trovare la soluzione ottima del problema. La complessità dell'algoritmo dipende dal numero di miglioramenti. Una sequenza di miglioramenti, fino all'ottimo globale, per l'esempio di Fig. 15.1, è mostrata nella Fig. 15.3. Purtroppo, già la sola verifica che nessuna trasformazione possa essere effettuata, cioè che una soluzione non sia migliorabile, richiede $O(nm)$ tempo, poiché devono essere provati al più $m - n + 1$ archi, e ciascuno può formare un circuito lungo al più n. Pertanto, l'algoritmo di Kruskal è nettamente più veloce di questo algoritmo.

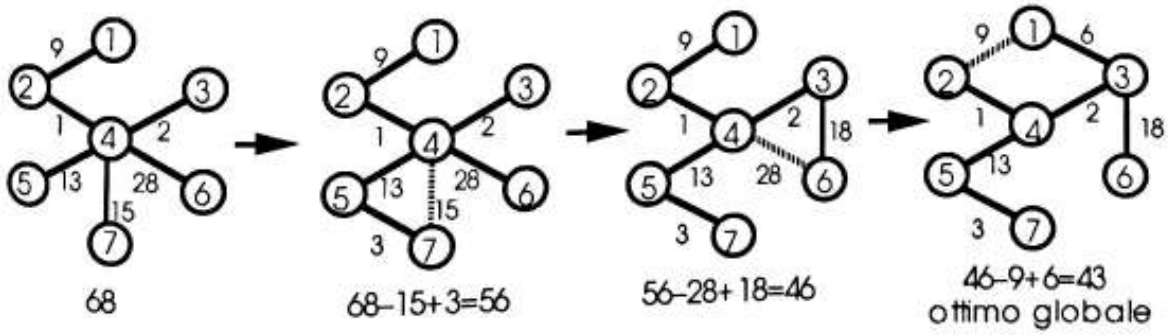


Fig. 15.3 Sequenza di miglioramenti ottenuta scegliendo ad ogni passo della ricerca locale la soluzione dell'intorno che dà il massimo miglioramento.

Il più famoso algoritmo di ricerca locale è senza dubbio il *metodo del simplesso*, ideato da Dantzig nel 1947 per risolvere il problema della PROGRAMMAZIONE LINEARE [capitolo 18].

Tale algoritmo non può però essere qui riportato perché una sua descrizione dettagliata richiede buone conoscenze matematiche di ricerca operativa. In questo capitolo, consideriamo invece la progettazione di altri due algoritmi di ricerca locale. Il primo algoritmo illustra un utilizzo piuttosto sorprendente della ricerca locale, per risolvere addirittura un problema che apparentemente non sembra neanche un problema di ottimizzazione: il solito problema dell'ordinamento degli elementi di un vettore. Il secondo invece ne mostra un utilizzo "classico", per risolvere un ben noto problema di ottimizzazione su grafi che è un caso particolare della programmazione lineare: il problema del flusso massimo.

15.1
SHELL SORT

In una sequenza di n elementi $a_1, ..., a_n$, un'*inversione* è data da una coppia di elementi a_i e a_j tali che $i < j$ e $a_i > a_j$ [cfr. Esercizio 12.3]. Il problema dell'ordinamento degli n elementi può essere formulato come problema di ottimizzazione nel modo seguente.

ORDINAMENTO Data la sequenza $a_1, \ldots, a_n$, trovare una permutazione degli n elementi che minimizzi il numero totale di inversioni.

15.3 ESEMPIO – [Inversioni]
La sequenza 1, 7, 3, 4 ha due inversioni, perché $a_2 = 7 > a_3 = 3$ e $a_2 = 7 > a_4 = 4$. La soluzione ottima 1, 3, 4, 7 ha il numero minimo di inversioni (zero), mentre la sequenza ordinata alla rovescia 7, 4, 3, 1 ne ha il numero massimo (sei). In generale, il numero minimo di inversioni per una sequenza di n elementi è sempre zero, mentre quello massimo è $\sum_{i=1}^{n-1}(n-i) = (n-1)n/2$.

Si supponga per semplicità che gli n elementi $a_1, ..., a_n$ siano tutti distinti, e si indichino con π e a_π, rispettivamente, una permutazione degli indici 1, ..., n e la corrispondente permutazione degli elementi. Indichiamo inoltre con $a_\pi - \{a_i\}$ la sequenza a_π dalla quale è stato tolto un generico elemento a_i. Un intorno $I(a_\pi)$ può essere definito come l'insieme di tutte le permutazioni di a_π che, a meno di un elemento a_i, hanno elementi uguali nella stessa posizione relativa:

$$I(a_\pi) = \{a'_\pi : a_\pi - \{a_i\} = a'_\pi - \{a_i\},\ 1 \le i \le n\}$$

15.4 ESEMPIO – [Intorno]
Si consideri la sequenza 1, 7, 3, 4; "scalando" l'1 a destra, si ottiene la sequenza 7, 1, 3, 4, indi 7, 3, 1, 4, e infine 7, 3, 4, 1. Queste quattro sequenze sono tra loro identiche, a meno dell'elemento 1: cancellando 1 in tutte, si ottiene infatti la medesima sequenza 7, 3, 4. Ragionando in modo analogo per i rimanenti elementi, si ricava: $I(1, 7, 3, 4) = \{(7, 1, 3, 4), (7, 3, 1, 4), (7, 3, 4, 1), (1, 3, 7, 4), (1, 3, 4, 7), (1, 7, 4, 3), (3, 1, 7, 4), (4, 1, 7, 3), (1, 4, 7, 3)\}$.

Un criterio di miglioramento è chiaramente quello di diminuire il numero di inversioni, cioè passare da a_π ad $a'_\pi \in I(a_\pi)$ in modo che il numero di inversioni di a'_π sia minore del numero di inversioni di a_π.

15.5 ESEMPIO – [Miglioramento]
Le sequenze di $l(1, 7, 3, 4)$ che diminuiscono il numero di inversioni sono soltanto 1, 3, 7, 4 e 1, 3, 4, 7.

Questo criterio di miglioramento può ricordare il semplice algoritmo di ordinamento insertionSort() [capitolo 2].

La "pecca" dell'Insertion Sort è quella di poter spostare solo elementi tra loro adiacenti, richiedendo pertanto $O(n)$ tempo per trasferire un elemento piccolo dall'estremità destra a quella sinistra della sequenza. Il suo "pregio", d'altronde, è quello di richiedere poco tempo di elaborazione se la sequenza è "quasi" ordinata. Per eliminare la "pecca" e mantenere il "pregio", è possibile spostare elementi che non siano adiacenti, ma "distanti" h, con $h > 1$. È questa l'idea venuta a Shell (1959), che ha proposto un'estensione dell'Insertion Sort che porta il suo nome: Shell Sort. L'idea fondamentale consiste nell'usare una sequenza di distanze decrescenti, in modo da ordinare dapprima elementi che sono tra loro molto lontani e solo successivamente considerare elementi tra loro adiacenti che, per l'effetto degli ordinamenti precedenti, dovrebbero essere ormai "quasi" ordinati.

La seguente procedura shellSort() è simile a insertionSort(); invece di considerare elementi a distanza 1, si considerano elementi a distanza h, con valori di h pari a ... 1093, 364, 121, 40, 13, 4, 1 (la scelta di questi particolari valori di h sarà chiarita in seguito):

```
shellSort(ITEM[] A, integer n)
    integer h ← 1
    while h ≤ n do h ← 3 · h + 1
    h ← ⌊h/3⌋
    while h ≥ 1 do
        for i ← h + 1 to n do
            ITEM temp ← A[i]
            integer j ← i
            while j > h and A[j − h] > temp do
                A[j] ← A[j − h]
                j ← j − h
            A[j] ← temp
        h ← ⌊h/3⌋
```

In pratica, la procedura riordina la sequenza di elementi con la proprietà che, scelto un elemento qualsiasi, la sottosequenza contenente un elemento ogni h è riordinata. La sequenza complessiva può essere vista come formata da h sottosequenze indipendenti ma "incastrate" tra loro, ognuna contenente all'incirca n/h elementi. Ciascuna sottosequenza è ordinata con una ricerca locale basata sull'Insertion Sort. Per h grande, sono spostati elementi della sequenza in ingresso molto "lontani" tra loro, mentre quando h diventa 1 la procedura è proprio una insertionSort() sull'intera sequenza in ingresso, in cui però ci si aspetta di spostare solo pochi elementi "vicini" tra loro.

15.6 ESEMPIO – [Shell Sort]

La Fig. 15.4 mostra l'ordinamento della sequenza di 15 elementi 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 con la procedura shellSort(). I valori di h usati sono: 13, 4 ed 1.

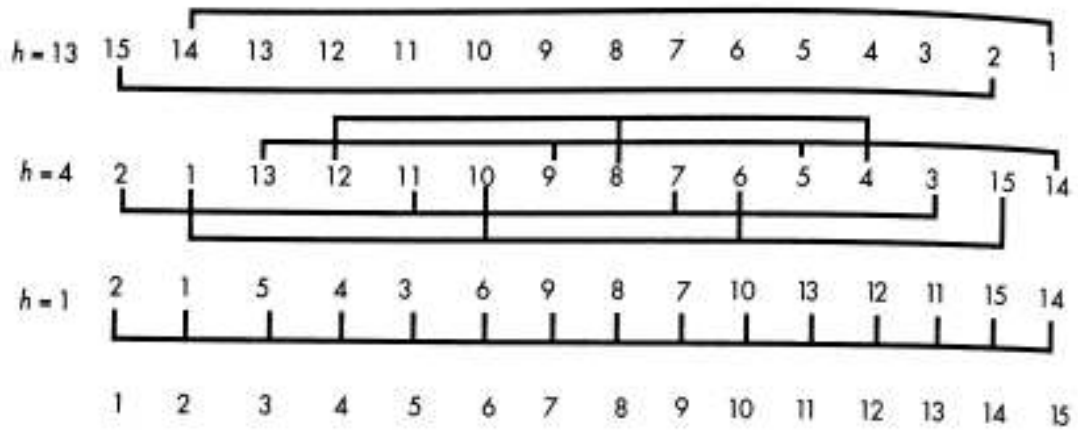


Fig. 15.4 Esecuzione della procedura shellSort(), con evidenziate le sottosequenze che sono ordinate per $h = 13, 4, 1$.

La procedura shellSort() è chiaramente corretta per un qualsiasi insieme di valori per h che comprenda il valore 1, perché in quest'ultimo caso essa si riduce ad insertionSort(). La sua complessità non solo dipende dalla sequenza in ingresso, ma anche, e soprattutto, dai particolari valori di h usati. Un'analisi completa della complessità non può essere qui riportata, perché troppo complicata. Ci limitiamo pertanto a citare i seguenti risultati:

(1) per valori di h uguali a $(3^i - 1)/2$, con $i = t, t-1, ..., 1$, dove t è il massimo indice s tale che $(3^{s+1} - 1)/2$ è maggiore di n (cioè proprio per i valori ... 1093, 364, 121, 40, 13, 4, 1 utilizzati precedentemente), è stato dimostrato che shellSort() ha complessità $O(n^{1.5})$ nel caso pessimo;

(2) per valori di h uguali soltanto ad $1.72n^{1/3}$ ed 1, è stato dimostrato che shellSort() ha complessità $O(n^{5/3})$ nel caso medio;

(3) per valori di h uguali a tutti gli interi della forma 3^i2^j minori o uguali ad n, la complessità si riduce ad $O(n \log^2 n)$ nel caso pessimo;

(4) esistono altri insiemi di valori di h per i quali la complessità è $O(n^{1.25})$ nel caso pessimo.

In pratica, i valori di h maggiormente utilizzati sono proprio quelli del caso (1), cioè ... 1093, 364, 121, 40, 13, 4, 1, perché generabili molto facilmente. Per questi valori, c'è chi ipotizza addirittura una complessità di $O(n \log^2 n)$ oppure $O(n^{1.25})$ nel caso medio, ma nessuno finora è stato capace di provare tale congettura. Tali considerazioni mostrano come shellSort() sia in pratica una procedura di ordinamento molto efficiente, da preferire addirittura ad altri algoritmi più sofisticati, specie per n non troppo elevato (p.e. $n \leq 5000$). I maggiori pregi di questa procedura, oltre alla facilità di programmazione e alla velocità di esecuzione, consistono nel fatto che l'ordinamento avviene iterativamente, cioè senza ricorsione, ed *in loco*, cioè senza bisogno di memoria ausiliaria né per copiare

porzioni di A, al contrario di MergeSort(), né per gestire la ricorsione, come avviene invece sia con MergeSort() che con QuickSort().

15.2 FLUSSO MASSIMO

Una *rete di flusso* $G = (V, E, s, p, c)$ è data da un grafo orientato $G = (V, E)$, da una coppia di vertici di V detti *sorgente* s e *pozzo* p, e da una funzione di *capacità* a valori interi positivi c: $V \times V \longrightarrow \mathbf{Z}^+ \cup \{0\}$, tale per cui $c(u, v) = 0$ se $(u, v) \notin E$.

Un *flusso* in G è una funzione a valori interi F: $V \times V \longrightarrow \mathbf{Z}$ che soddisfa le seguenti proprietà:

(1) *simmetria opposta*: $f(u, v) = -f(v, u)$ per ogni coppia $u, v \in V$;

(2) *vincolo di capacità*: $f(u, v) \leq c(u, v)$ per ogni coppia $u, v \in V$;

(3) *conservazione del flusso*: $\sum_v f(u, v) = 0$ per ogni nodo $u \in V - \{s, p\}$.

La quantità $f(u, v)$, che può essere positiva o negativa, è detta *flusso netto* dal nodo u al nodo v, mentre il valore $|f|$ del flusso f è il flusso netto totale che esce dalla sorgente (o che entra nel pozzo), ovvero $|f| = \sum_v f(s, v) = \sum_u f(u, p)$. Il problema del FLUSSO MASSIMO è definito nel modo seguente.

FLUSSO MASSIMO (MAXIMUM FLOW) Data una rete di flusso $G = (V, E, s, p, c)$, trovare un flusso f^* che ha valore $|f^*|$ più grande possibile.

Si osservi che la proprietà di simmetria opposta implica che $f(u, u) = 0$ per ogni $u \in V$. La stessa proprietà e i vincoli di capacità implicano che se sia $(u, v) \in E$ e $(v, u) \in E$, allora $f(u,v) = f(v, u) = 0$. Pertanto, un flusso netto $f(u, v) \neq 0$ implica che $(u, v) \in E$ oppure $(v, u) \in E$. Infine, la conservazione del flusso è la stessa legge della corrente elettrica di Kirchhoff che afferma che il flusso netto totale che entra in un nodo (esclusi la sorgente ed il pozzo) eguaglia il flusso netto totale che esce dallo stesso nodo.

Il problema di trovare un flusso massimo su un grafo è intimamente correlato a quello di trovare un taglio minimo sullo stesso grafo. Dato G, un *taglio* (S, P) è una partizione dell'insieme dei nodi tale che $s \in S$, $p \in P$, $S \cup P = V$, ed $S \cap P = \emptyset$. La *capacità* del taglio è $c(S, P) = \sum_{u \in S, v \in P} c(u, v)$. Un taglio di capacità minima è detto *taglio minimo*. Se f è un flusso ed (S, P) è un taglio, allora il flusso che attraversa il taglio è

$$f(S, P) = \sum_{u \in S, v \in P} f(u, v)$$

Una proprietà importante è che il valore di un flusso è uguale al flusso che attraversa un qualsiasi taglio, il quale a sua volta non supera la capacità del taglio stesso, cioè che $|f| = f(S, P) \leq c(S, P)$. Infatti, si ha che:

- $\sum_{u \in S, v \in P} f(u, v) = \sum_{v \in V} f(s, v) + \sum_{u \in S - \{s\}, v \in V} f(u, v) = |f|$, per la proprietà di conservazione del flusso,

$-\sum_{u \in S, v \in S} f(u, v) = 0$, per la proprietà di simmetria opposta,

da cui si ricava:

$$f(S, P) = \sum_{u \in S, v \in P} f(u, v) = \sum_{u \in S, v \in V} f(u, v) - \sum_{u \in S, v \in S} f(u, v) = |f| - 0 = |f|$$

mentre $f(S, P) \leq c(S, P)$ deriva banalmente dai vincoli di capacità.

Un importante teorema, detto del *flusso massimo/taglio minimo*, afferma che, se vale l'uguaglianza, cioè $f(S, P) = c(S, P)$, allora f è un flusso massimo ed (S, P) è un taglio minimo. Inoltre tale teorema dà lo spunto per progettare algoritmi di ricerca locale per trovare il flusso massimo.

Per dimostrare il teorema, si definisce *capacità residua* per un flusso f la quantità $R(u, v) = c(u, v) - f(u, v)$. La rete di flusso residua $R = (V, E_r, s, p, r)$ ha lo stesso insieme V di nodi di G, e l'insieme di archi E_r contiene un arco (u, v) con capacità $r(u, v)$ per ogni coppia di nodi $u, v \in V$ tale che $r(u, v) > 0$. Un *cammino aumentante* per f è un cammino da s a p in R. La capacità residua δ di un cammino aumentante è uguale alla capacità residua $r(u, v)$ più piccola tra quelle di tutti gli archi (u, v) che compaiono nel cammino.

È possibile definire un intorno $I(f)$ di un flusso f come l'insieme di tutti i flussi ottenibili da f variandone il valore, su ogni arco di un cammino aumentante, di una quantità non più grande della capacità residua del cammino aumentante stesso. In altri termini, dato un cammino aumentante per f, sia g il flusso di valore δ tale che $g(u, v) = \delta$ e $g(v, u) = -\delta$ se l'arco (u, v) appartiene al cammino aumentante, e $g(u, v) = 0$ altrimenti. Si può passare dalla soluzione f ad una soluzione migliore che appartiene all'intorno $I(f)$ sommando, componente per componente, il flusso g al flusso f, ottenendo così il flusso $f' = f + g$ di valore $|f'| = |f| + \delta$.

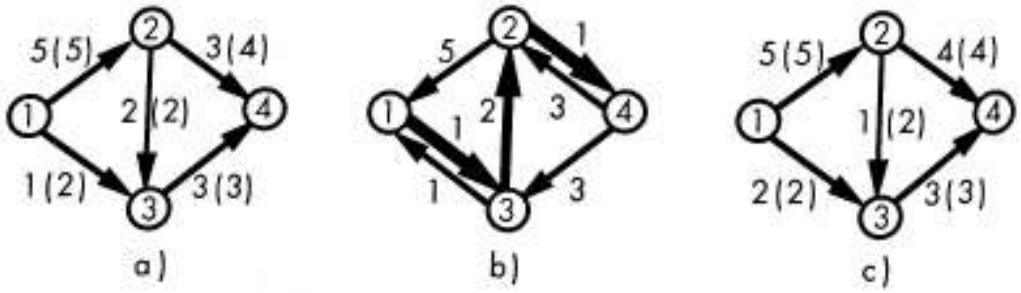


Fig. 15.5 Un grafo G con $s = 1$, $p = 4$, ed un flusso f di valore $|f| = 6$, dove per ciascun arco (u, v) è indicata la coppia di valori: $F(u, v)$ $(c(u, v))$. b) La rete di flusso residua R ed un cammino aumentante con capacità residua $\delta = 1$. c) Il grafo G con il flusso f aggiornato aggiungendovi δ lungo gli archi del cammino aumentante.

15.7 ESEMPIO – [Rete di flusso residua e cammini aumentanti]

La Fig. 15.5(a) mostra un grafo G di quattro nodi, ove $s = 1$ e $p = 4$, ed un flusso f per G di valore $|f| = 6$ (per chiarezza, sono indicate soltanto le componenti del flusso relative agli archi di G, seguite dalle capacità tra parentesi tonde). La Fig. 15.5(b) mostra invece la corrispondente rete residua R; ad esempio, l'arco $(2, 4)$ di R ha capacità residua $r(2, 4) = c(2, 4) - f(2, 4) = 4 - 3 = 1$, mentre l'arco $(4, 2)$ ha capacità residua $r(4, 2) = c(4, 2) - f(4, 2) = 0 - (-3) = 3$.

Si noti che sono comparsi alcuni archi che non esistevano in G, come ad esempio (4, 2). La Fig. 15.5(b) evidenzia in grassetto anche il cammino aumentante 1, 3, 2, 4 con capacità residua $\delta = r(1,3) = r(2,4) = 1$, sul quale si può definire un flusso g tale che $g(1,3) = g(3,2) = g(2,4) = \delta = 1$, $g(3,1) = g(2,3) = g(4,2) = -\delta = -1$, e $g(u, v) = 0$ altrove. Sommando g ad f si ottiene il flusso di Fig. 15.5(c), di valore $|f| = 7$.

È facile verificare che se f è un flusso qualsiasi per G ed f^* è un flusso massimo per G, allora $f^* - f$ è un flusso massimo per R di valore $|f^* - f|$. Si può quindi dimostrare il seguente teorema fondamentale:

TEOREMA 1 – [Flusso massimo/taglio minimo]
Le condizioni seguenti sono tra loro equivalenti:

(1) f è un flusso massimo;

(2) non esiste alcun cammino aumentante per f;

(3) esiste un taglio (S, P) tale che $|f| = c(S, P)$.

DIMOSTRAZIONE È immediato osservare che (1) implica (2), poiché se esistesse un cammino aumentante allora il flusso potrebbe essere aumentato e quindi non sarebbe massimo. Per verificare che (2) implica (3), siano S l'insieme dei nodi di R raggiungibili da s con un cammino e $P = V - S$. (S, P) è un taglio e poiché non esiste alcun cammino aumentante, per ogni $u \in S$ e $v \in P$ risulta che (u, v) non è un arco di R e quindi $c(u, v) = f(u, v)$. Ma ciò implica che $|f(S, P)| = c(S, P)$. Infine (3) implica (1), poiché da $|f| \leq c(S, P)$ per ogni flusso f ed ogni taglio (S, P) segue che se $|f| = c(S, P)$ allora f è un flusso massimo.

15.8 ESEMPIO – [continua]
Il flusso mostrato nella Fig. 15.5(c) è un flusso massimo. Infatti, considerando il taglio (S, P) dove $S = \{1\}$ e $P = \{2, 3, 4\}$, si vede immediatamente che $|f| = 7 = f(S, P) = c(S, P)$.

Basandosi sulla caratterizzazione matematica della soluzione fornita dal precedente teorema, sono stati proposti svariati algoritmi di ricerca locale che, partendo da un flusso f nullo (cioè con $|f| = 0$), lo aumentano ripetutamente lungo cammini aumentanti finché, non esistendo più alcun cammino aumentante nel grafo residuo, viene individuato il flusso massimo. La struttura generale è mostrata nella schema di procedura maxFlow().

L'aumento può essere fatto in base a molti criteri diversi, per esempio lungo un singolo cammino aumentante o su più cammini aumentanti contemporaneamente, preferendo quelli con massima capacità residua oppure col minimo numero di archi. Vediamo tre possibilità.

```
integer[][] maxFlow(GRAPH G, NODE s, NODE p, integer[][] c)
    NODE u, v                                          % Indici nodi
    integer[][] f ← new integer[][]                    % Flusso parziale
    integer[][] g ← new integer[][]                    % Flusso da cammino aumentante

    foreach u, v ∈ G.V() do f[u, v] = 0                % Inizializza un flusso nullo
    boolean stop ← false
    while not stop do
        R ← Rete di flusso residua del flusso f in G
        g ← flusso associato ad uno o più cammini aumentanti in R
        foreach u, v ∈ G.V() do f[u, v] = f[u, v] + g[u, v]      % f = f + g
        if ∀u, v ∈ G.V() : g[u, v] = 0 then stop ← true
    return f
```

15.2.1
ALGORITMO DI FORD-FULKERSON

L'algoritmo più semplice, proposto da Ford e Fulkerson (1956), richiede semplicemente di trovare un cammino aumentante qualsiasi, utilizzando un qualunque algoritmo di visita. Assumendo che le capacità siano rappresentate da valori interi, ogni cammino aumentante incrementa il flusso almeno di 1. Poiché una visita ha costo $O(n + m)$ e possono essere effettuati al massimo $|f^*|$ incrementi, il costo di questo algoritmo è $O((n + m)|f^*|)$. Se è possibile stimare superiormente il valore del flusso massimo e questa limitazione superiore è sufficientemente bassa, un algoritmo semplice come questo può essere considerato efficiente.

15.2.2
ALGORITMO DI EDMONDS-KARP

Se la ricerca del cammino aumentante è effettuata tramite una visita in ampiezza, si ottiene un algoritmo proposto da Edmonds e Karp (1972). È possibile dimostrare che la complessità dell'algoritmo è $O(nm^2)$; poiché questo algoritmo è un caso particolare di Ford e Fulkerson, è possibile calcolare entrambe le limitazioni superiori e scegliere quella più favorevole.

15.2.3
ALGORITMO DEI TRE INDIANI

Vediamo a grandi linee un algoritmo proposto da Kumar, Malhotra e Maheswari (1978) come raffinamento di altri algoritmi precedenti, che è noto come l'*algoritmo dei tre indiani* e richiede $O(n^3)$ tempo. Questo algoritmo si basa sulla strategia di aumentare ripetutamente lungo tutti i cammini aumentanti più corti, cioè aventi il minimo numero di archi. Il procedimento ha inizio con flusso f nullo e consta delle seguenti tre fasi, che vengono iterate:

(1) si costruisce un grafo residuo R, semplificato in modo che sia formato da tutti e soli i cammini aumentanti per f comprendenti il minimo numero k archi;

(2) si saturano in una sola volta tutti i cammini di R, in modo da non avere più cammini aumentanti per f con k archi, determinando un *flusso saturante* g di valore $|g|$ per R;

(3) si aggiorna f sommando ad esso il flusso saturante g.

Come sarà dimostrato nel seguito, le tre fasi dell'algoritmo sono iterate per $O(n)$ volte, in quanto il numero k di archi dei cammini minimi di R cresce ad ogni iterazione ed un cammino non può avere più di $n - 1$ archi. Poiché le fasi (2) e (3) richiedono $O(n^2)$ tempo, mentre la fase (1) ne richiede $O(m)$, la complessità dell'algoritmo è $O(n^3)$. La procedura kmn() descrive l'algoritmo dei 3 indiani.

```
integer [][] kmn(GRAPH G, NODE s, NODE p, integer[][] c)
NODE u, v                                          % Indici nodi
GRAPH R ← Graph()                                  % Grafo residuo
integer[][] r ← new integer[][]                    % Capacità residua
integer[][] f ← new integer[][]                    % Flusso parziale
integer[][] g ← new integer[][]                    % Flusso da cammino aumentante

foreach u, v ∈ G.V() do f[u, v] ← 0                % Inizializza un flusso nullo
boolean stop ← false
while not stop do
    residuo(G, s, p, c, f, R, r)
    satura(R, s, p, r, g)
    foreach u, v ∈ G.V() do f[u, v] ← f[u, v] + g[u, v]        % f = f + g
    if ∀u, v ∈ G.V() : g[u, v] = 0 then stop ← true
return f
```

Le procedure residuo() e satura() sono usate per realizzare le prime due fasi, mentre l'inizializzazione di f al flusso nullo di valore $|f| = 0$ e la sommatoria del flusso aumentante g al flusso f sono semplicemente realizzate con due cicli **foreach**, ripresi da maxFlow().

La procedura residuo() costruisce in $O(m)$ tempo il grafo residuo $R = (V, E_r)$ con una BFS modificata, dove un arco (u, v) viene aggiunto ad E_r con capacità $r(u, v) = c(u, v) - f(u, v)$ per ogni arco $(u, v) \in E$ o $(v, u) \in E$ tale che $r(u, v) > 0$. Si assume che G ed R siano rappresentati con gli insiemi di adiacenza sia degli archi uscenti che degli archi entranti di ciascun nodo. Durante la visita, è assegnata ad ogni nodo u la sua "distanza minima" d_u dalla sorgente s, intesa come il minimo numero di archi in un cammino tra s e u. Il grafo R è semplificato, con un'altra eventuale passata e sempre in $O(m)$ tempo, per eliminare tutti quei nodi e quegli archi che non risiedono sui cammini aumentanti più corti da s a p. Infatti, si possono eliminare tutti i nodi v diversi da p con $d_v \geq d_p$, insieme a tutti gli archi incidenti in essi. Inoltre, poiché un cammino più corto può contenere solo archi da un nodo u a distanza d_u ad un altro nodo v a distanza $d_v = d_u + 1$, si può eliminare ogni arco (u, v) tale che $d_v \leq d_u$. Il grafo R così semplificato è aciclico e "stratificato",

cioè ogni arco (u, v) unisce due nodi tali che $d_v = d_u + 1$, ed ogni nodo $u \in V - \{s, p\}$ ha distanza d_u tale che $0 = d_s < d_u < d_p$. Inoltre, R può essere ulteriormente semplificato con un'altra passata, che parte dal pozzo p ed esamina "a ritroso" gli archi, marcando così i nodi dai quali si può raggiungere p, e cancellando successivamente tutti i nodi non marcati, insieme agli archi incidenti. Dopo quest'ultima semplificazione, ogni nodo di R sta su un cammino minimo tra s e p, se esiste un tale cammino. In altri termini, R contiene tutti e soli i cammini aumentanti per f di lunghezza minima $k = d_p$. Una volta costruito il grafo residuo semplificato R in $O(m)$ tempo, la procedura satura() cerca un flusso saturante per esso, cioè un flusso g che saturi tutti i cammini presenti in R. Un arco è saturo se il flusso che lo attraversa è pari alla capacità dell'arco, ed un cammino di R è saturo se contiene almeno un arco saturo. Un flusso saturante per R non è detto che sia massimo per R (si potrebbe infatti aumentare percorrendo qualche arco nel verso opposto all'orientamento), ma garantisce comunque che in futuro non potranno più essere considerati cammini aumentanti di lunghezza $k = d_p$. Infatti, sia G'_r il grafo residuo all'iterazione successiva, con distanze d'. Ciascun arco (u, v) di R ha $d_v = d_u + 1$, mentre ciascun arco di G'_r o è un arco di R o è un arco di R con orientamento opposto. Ma allora ogni arco in G'_r ha $d_v \leq d_u + 1$, e quindi $d'_p \geq d_p$. Se per assurdo fosse $d'_p = d_p$, allora esisterebbe un cammino minimo da s a p in R che non è stato saturato tale che ogni arco (u, v) ha $d_v = d_u + 1$. Ma ciò contraddice il fatto che almeno un arco di ogni cammino in R è saturato e quindi non compare in G'_r. Pertanto, $d'_p > d_p$ e in futuro non potranno più essere considerati cammini aumentanti di lunghezza $k \leq d_p$.

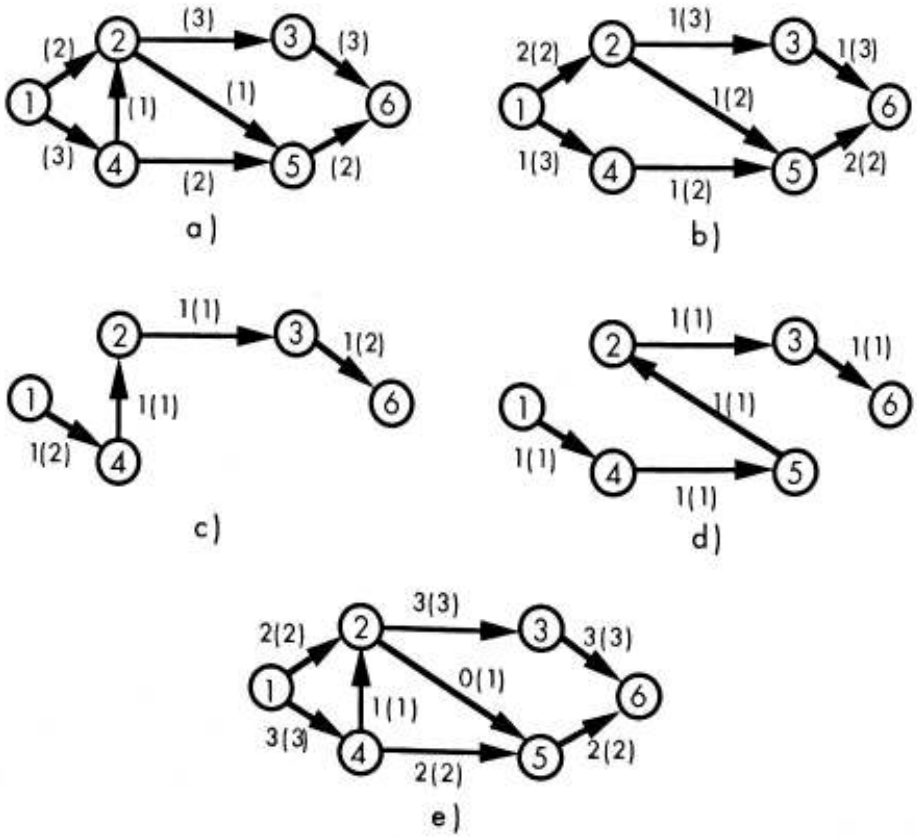


Fig. 15.6 Un grafo G, con capacità degli archi indicate tra parentesi. b) Il grafo residuo R semplificato alla prima iterazione, con un flusso saturante g e le capacità residue tra parentesi. c)-d) R alla seconda e terza iterazione, con un flusso saturante g. e) Il flusso massimo f per G.

15.9 ESEMPIO – [Flusso saturante]
La Fig. 15.6(a) mostra un grafo G con sorgente s = 1, pozzo p = 6, e capacità degli archi indicate tra parentesi. Alla prima iterazione il flusso f è nullo e il grafo residuo semplificato R, che contiene tutti i cammini aumentanti di lunghezza minima (uguale a 3), è mostrato nella Fig. 15.6(b), insieme ad un flusso saturante g per R. Le componenti del flusso g sono indicate senza parentesi, seguite dalle capacità residue di R tra parentesi. Alla seconda iterazione, il flusso f è uguale al flusso saturante g trovato alla prima iterazione, ed il grafo residuo semplificato R (che contiene cammini lunghi 4) è mostrato nella Fig. 15.6(c), insieme a un flusso saturante. Il grafo R alla terza iterazione (con cammini lunghi 5) ed un flusso saturante sono illustrati nella Fig. 15.6(d). Infine, la Fig. 15.6(e) mostra il grafo G con il flusso massimo f così trovato. Un taglio minimo si può individuare dal grafo residuo R per f ponendo in S la sorgente s e tutti i nodi raggiungibili da s con archi di R (nessuno nell'esempio di Fig. 15.6e), perché f satura entrambi gli archi uscenti da 1 in G), ottenendo così S = {1} e P = {2, 3, 4, 5, 6}.

Per trovare un flusso saturante g per R, si definisce per ogni nodo u di R la sua *portata*, ovvero l'ammontare massimo di flusso che può attraversare il nodo u, come il minimo tra la somma delle capacità residue degli archi entranti in u e quella degli archi uscenti da u:

$$portata[u] = \min\left\{\sum_v r(v,u), \sum_v r(u,v)\right\}$$

dove, per applicare la definizione anche ad s ed a p, si può assumere di avere un arco "fittizio di ritorno" (p, s) con capacità infinita.

Il vettore *portata* può essere calcolato in $O(m)$ tempo. Selezionando poi in $O(n)$ tempo il nodo u con portata minima, si è sicuri di poter trasmettere *portata*[u] unità di flusso da s a p in R con una opportuna procedura instrada(). Questa procedura, partendo dal nodo u, modifica il flusso g, aggiungendovi *portata*[u] unità di flusso che sono instradate da u verso p ed a ritroso da u verso s, ed aggiorna le portate dei nodi. Effettuato l'instradamento, con una semplice procedura aggiorna() sono eliminati i nodi diversi da s con portata nulla, tra i quali c'è almeno u, e gli archi incidenti in essi. Se il flusso g non è saturante, cioè la portata della sorgente è positiva, allora si instradano altre unità di flusso che sono sommate al flusso g. Il procedimento è iterato finché il flusso g non è saturante.

```
integer satura(GRAPH R, NODE s, NODE p, integer[][] r, integer[][] g)
                                                        % Indici nodi
  NODE u, v
  integer[] portata ← new integer[1 ... R.n]            % Vettore portata

  foreach u, v ∈ V do g[u, v] ← 0                       % Inizializza g al flusso nullo
  foreach u ∈ R.V() do
  |   portata[u] ← min{ Σ_v r[v, u], Σ_v r[u, v] }
  while portata[s] > 0 do
  |   seleziona il nodo u con portata[u] minima
  |   instrada(R, u, portata, g)
  |   aggiorna(R, portata, g)
  return g
```

La complessità della procedura satura() è determinata da quella del ciclo **while**, che a sua volta dipende da un'accorta realizzazione della procedura instrada(). Questa procedura aggiunge $x = portata[u]$ unità di flusso a g, instradandole da u verso p ed a ritroso da u verso s, secondo una visita BFS. Ad ogni chiamata, esamina ciascun nodo v di R al più una volta allontanandosi progressivamente dal nodo u nel grafo stratificato. In questo modo, quando è considerato il nodo v, il flusso sugli archi entranti in v (se v sta in un cammino tra u e p) oppure su quelli uscenti da v (se v sta tra s ed u) è già stato tutto instradato.

```
instrada(GRAPH R, NODE u, integer[] portata, integer[][] g)
  % Instrada portata[u] unità di flusso da u a p, modificando g
  % L'instradamento da s a u non è mostrato in quanto simmetrico
  QUEUE Q ← Queue()
  Q.enqueue(u)
  integer[] esci ← new integer[1 ... R.n]
  foreach v ∈ R.V() do esci[v] ← 0
  esci[u] ← portata[u]
  portata[u] ← 0
  while not Q.isEmpty() do
      v ← Q.dequeue()
      while esci[v] > 0 do
          considera un arco (v, w) uscente da R
          M ← min{r[v, w], esci[v]}
          r[v, w] ← r[v, w] – M
          if r[v, w] = 0 then cancella (v, w) da R
          esci[v] ← esci[v] – M
          esci[w] ← esci[w] + M
          g[v, w] = g[v, w] + M
          g[w, v] = – g[w, v]
          portata[w] = portata[w] – M
          Q.enqueue(w)
```

Per semplicità, nella procedura instrada() non è mostrato tutto l'instradamento, ma solo quello da u verso p poiché l'altro è perfettamente analogo. Durante la visita, instrada() calcola e aggiorna la quantità $esci[j]$, che denota quante unità di flusso devono uscire dal nodo v, ed aggiorna le portate dei nodi.

Il tempo richiesto della procedura satura() dipende dal numero di ripetizioni del ciclo **while**, ove è richiamata la instrada(). Poiché ad ogni iterazione è eliminato almeno un nodo saturo (con portata divenuta nulla), il numero di iterazioni è $O(n)$. Ogni arco di R può essere cancellato da aggiorna() oppure saturato e cancellato da instrada(), ma una sola volta, oppure può essere riempito con una certa quantità di flusso senza essere saturato, ma solo una volta per ciascuna iterazione e per non più di un arco per nodo. Inoltre, non sono mai esaminati tutti gli archi uscenti da v, ma soltanto i primi y_v, dei quali i primi $y_v - 1$ sono saturati e cancellati, mentre l'y_v-esimo è eventualmente riempito ma non saturato.

Pertanto, le saturazioni e cancellazioni di archi sono in totale $O(m)$, mentre i riempimenti senza saturazione sono in totale $O(n^2)$, ed il tempo richiesto da satura() è così $O(n^2)$.

Ne segue che ciascuna iterazione della procedura kmn() prende $O(n^2)$ tempo, dovuto alla satura(). Poiché kmn() effettua al più $n-1$ iterazioni in ciascuna delle quali cerca un flusso saturante, la complessità dell'algoritmo dei tre indiani per trovare il flusso massimo risulta essere $O(n^3)$.

15.3 REALITY CHECK

Un enorme numero di problemi può essere formulato come un problema di flusso massimo; ne potete vedere un paio di esempi negli esercizi (Esercizio 15.9 per gli abbinamenti bipartiti e Esercizio 15.10 per una misura dell'arco connettività). Alcuni autori (Skiena, 2008) arrivano a suggerire di progettare grafi, non algoritmi: utilizzando soluzioni esistenti di problemi ben noti[1]. Le reti di flusso possono essere utilizzate per modellare reti di trasporto, reti di telecomunicazioni, servizi logistici e molti altri ancora. La loro applicabilità è così vasta che risulta difficile identificare un unico esempio.

[1]Si suol dire scherzosamente che i matematici non risolvono i problemi, ma li trasformano in altri problemi dei quali conoscono già le soluzioni.

ESERCIZI

15.1 [Tagli minimi] Si elenchino tutti i tagli di capacità minima del grafo illustrato in Fig. 15.5(a) per la sorgente $s = 1$ e il pozzo $p = 4$.

15.2 [Scheduling] Dati n programmi con tempi di esecuzione $t_1, ..., t_n$, si vuole eseguirli su un processore in modo da minimizzarne il tempo medio di completamento $(1/n)\sum_i c_i$, dove il tempo di completamento c_i del programma i è la somma del suo tempo di esecuzione e dei tempi di esecuzione di tutti i programmi che lo precedono nell'ordinamento (*schedule*) finale. Si progetti un algoritmo di ricerca locale per risolvere il problema.

15.3 [Ciclo euleriano] Dato un grafo orientato $G = (V, E)$, si vuole trovare un *ciclo euleriano*, cioè un cammino chiuso (ma non semplice) che include ciascun arco una e una sola volta. Si determini una condizione necessaria e sufficiente affinché G ammetta un ciclo euleriano, e si progetti un algoritmo di ricerca locale per trovarlo.

15.4 [Riempimento di matrice] Dati $2n$ interi non negativi $c_1, c_2, ..., c_n, r_1, r_2, ..., r_n$, si vuole determinare una matrice $n \times n$ con elementi interi non negativi tali che la somma degli elementi della colonna i-esima sia uguale a c_i e la somma degli elementi della riga i-esima sia uguale ad r_i. Quali condizioni devono valere su $c_1, ..., c_n, r_1, ..., r_n$ affinché esista una soluzione? Si formuli il problema come un problema di flusso massimo. Si fornisca poi un algoritmo di complessità $O(n^2)$ senza usare il flusso massimo.

15.5 [Somma di matrici] Siano dati una matrice M di dimensione $m \times n$ con elementi non negativi e quattro interi non negativi l_1, l_2, t_1, t_2. Si vogliono determinare due matrici M_1 e M_2 di dimensione $m \times n$ con elementi interi non negativi tali che $M = M_1 + M_2$, la somma degli elementi di ogni linea (cioè riga o colonna) di M_i non superi l_i, e la somma di tutti gli elementi di M_i non superi t_i, per $i = 1, 2$. Si formuli il problema come un problema di flusso massimo.

15.6 [Formulazioni flusso massimo] Si dimostri che le seguenti varianti del problema del flusso massimo possono essere ricondotte alla formulazione originale del problema descritta nel presente capitolo:

(1) Il grafo ha più di una sorgente e più di un pozzo;

(2) I nodi, oltre agli archi, hanno capacità;

(3) Il grafo non è orientato.

15.7 [Cammini indipendenti] Dato un grafo orientato $G = (V, E)$ e due vertici u, v contenuti in V, trovare il numero totale di cammini "edge-independent", ovvero in cui un arco può comparire al massimo in un cammino.

15.8 [Intorno per abbinamento massimo] Dato un grafo non orientato, un *abbinamento* (*matching*, in inglese) M è un sottoinsieme di archi tale che non esistono due archi incidenti nel medesimo nodo. Si definisca un intorno per il problema di trovare un abbinamento di massima cardinalità.

15.9 [Abbinamento bipartito] Dato un grafo non orientato bipartito, il problema dell'ABBINAMENTO BIPARTITO consiste nel trovare un abbinamento di massima cardinalità [cfr. Esercizio 15.8]. Si dimostri che il problema è risolubile con l'algoritmo dei tre indiani con complessità $O(m\sqrt{n})$.

15.10 [Connettività] La *arco-connettività* di un grafo non orientato e connesso è la dimensione del più piccolo insieme di archi la cui rimozione causa la divisione del grafo in due componenti connesse separate. Si progetti un algoritmo per calcolare l'arco-connettività di un grafo.

15.1 I seguenti tagli hanno capacità minima:

- $S = \{1\}$, $P = \{2, 3, 4, 5, 6\}$
- $S = \{1, 4\}$, $P = \{2, 3, 5, 6\}$
- $S = \{1, 2, 3, 4, 5\}$, $P = \{6\}$
- $S = \{1, 2, 4, 5\}$, $P = \{3, 6\}$

15.2 Il famoso teorema di Eulero assicura che G contiene un ciclo euleriano se e solo se è connesso (trascurando l'orientamento degli archi), e per ciascun nodo il numero di archi entranti nel nodo è uguale al numero di archi uscenti dal nodo. Si assuma quindi che G verifichi tali condizioni.

Una *partizione di Eulero* P per G è una partizione di E in cammini chiusi (non necessariamente semplici) disgiunti sugli archi. Una partizione di Eulero può essere determinata specificando per ogni arco $a = (u, v)$ un unico successore $\text{succ}(a) = b = (v, w)$ in modo che non esistano mai due archi con lo stesso successore. Ovviamente, la cardinalità di una partizione di Eulero è compresa tra 1 ed $\lfloor n/2 \rfloor$ ed un ciclo euleriano corrisponde ad una partizione di Eulero di minima cardinalità. Una partizione iniziale P è facilissima da trovare, per esempio considerando un nodo alla volta di G e definendo l'h-esimo arco uscente dal nodo come il successore dell'h-esimo arco entrante. Dati due archi a e b, con $succ(a) = c$ e $\text{succ}(b) = d$, si può definire un'operazione $\text{scambio}(a, b)$ che scambia tra loro i successori in modo che $\text{succ}(a) = d$ e $\text{succ}(b) = c$. Si può allora definire l'intorno

$$I(P) = \{P' : P' \text{ è ottenuta da } P \text{ con una operazione scambio}\}$$

Ovviamente, uno scambio può far aumentare o diminuire di uno la cardinalità di una partizione, a seconda che divida un cammino chiuso in due cammini disgiunti sugli archi, oppure che fonda due cammini chiusi in uno solo. L'algoritmo di ricerca locale sceglierà ad ogni iterazione un'operazione di scambio che decrementi la cardinalità della partizione e si fermerà quando non saranno più possibili miglioramenti, cioè quando sarà stato individuato un ciclo euleriano.

15.7 Si costruisce una funzione di capacità c tale per cui $c(x, y) = 1$ se $(x, y) \in E$, mentre $c(x, y) = 0$ se $(x, y) \in E$. Si consideri il massimo flusso fra u e v; questo valore corrisponde al numero totale di cammini indipendenti, in quanto essendo la capacità di tutti gli archi pari ad 1, ogni cammino aumentante avrà valore di flusso 1 e tutti i suoi archi saranno distinti dagli altri cammini aumentanti.

15.8 Sia dato un abbinamento qualsiasi M. Si definisca un nodo "scoperto" se non ha alcun arco di M incidente in esso. Si definisca "catena alternante" una catena di archi costituita, alternatamente, da un arco non appartenente ad M e da uno appartenente ad M.

Si definisca "catena aumentante" una catena alternante tra due nodi scoperti. In una catena aumentante, il numero di archi che non fanno parte dell'accoppiamento è uno più degli archi che ne fanno parte. Pertanto, si può definire un intorno

$$I(M) = \{M' : M' \text{ è ottenibile da } M \text{ scambiando in una catena aumentante gli archi che fanno parte dell'accoppiamento con gli altri}\}$$

(Trovare una catena aumentante richiede tempo polinomiale, ma l'algoritmo non è affatto banale).

15.9 Sia $B = (V, E)$ un grafo non orientato bipartito. Per definizione, esiste una partizione dell'insieme V dei nodi in due insiemi disgiunti S e P tale che per ogni arco $[u, v]$ risulta $u \in S$ e $v \in P$ [cfr. Esercizi 9.3 e 9.4]. Si consideri un grafo orientato G con capacità unitarie sugli archi, ottenuto da B orientando gli archi da S verso P, aggiungendo un nodo sorgente s collegato con un arco ad ogni nodo di S ed un nodo pozzo p collegato con un arco da ogni nodo di P. Formalmente, si costruisce $G = (V', E')$ tale che:

$$V' = V \cup \{s, p\}$$
$$E' = \{(u, v): [u, v] \in E, u \in S, v \in P\} \cup \{(s, u): u \in S\} \cup \{(v, p): v \in P\}$$
$$c(u, v) = 1 \text{ per ogni } (u, v) \in E'$$

Come illustrato nella Fig. 15.7, è facile verificare che un flusso massimo f^* di valore $|f^*|$ in G corrisponde ad un abbinamento M di cardinalità $|f^*|$ in B, dove gli archi $[u, v] \in M$ sono dati dagli archi (u, v) con $u \in S$, $v \in P$ ed $f^*(u, v) = 1$.

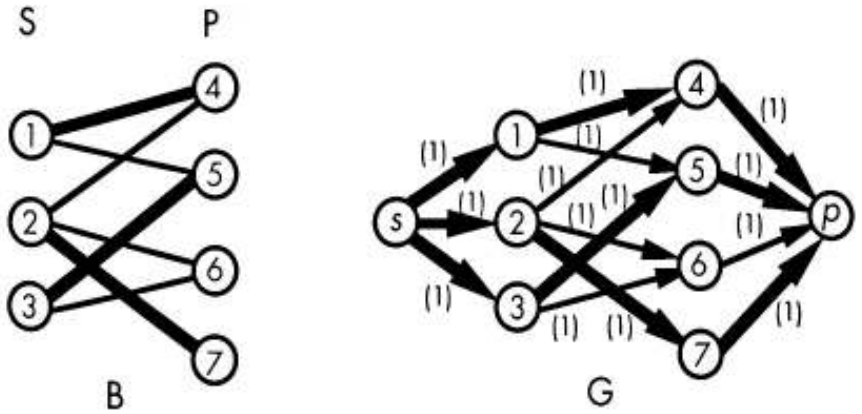


Fig. 15.7 Un grafo non orientato bipartito B ed il corrispondente grafo orientato G con capacità unitarie (tra parentesi). Gli archi in grassetto evidenziano un abbinamento massimo (di cardinalità 3) di B e le corrispondenti componenti unitarie del flusso massimo di G.

Per dimostrare che l'algoritmo dei 3 indiani ha complessità $O(m\sqrt{n})$ sul grafo G così definito, mostriamo che la procedura satura() richiede $O(m)$ tempo e che kmn() effettua $O(\sqrt{n})$ iterazioni in ciascuna delle quali cerca un flusso saturante.

Poiché ogni arco del grafo G ha capacità unitaria, anche il grafo residuo R ha capacità unitaria. Inoltre, dato che il flusso ha componenti intere, non ci sono mai riempimenti

senza saturazioni, ed ogni arco di R o è saturato e cancellato oppure non è neanche esaminato. Pertanto, il tempo richiesto da satura() è $O(m)$. Vediamo che il numero di volte che è cercato un flusso saturante è al più $\lceil 2\sqrt{n-2} \rceil$. Siano f il flusso ad una generica iterazione, f^* il flusso massimo, ed R il grafo residuo per f. Poiché $f^* - f$ è un flusso per R ed ogni componente del flusso vale 0 o 1 su ogni arco, è possibile partizionare gli archi di R per i quali il flusso è 1 in un insieme di $|f^* - f|$ cammini tra s e p (più qualche eventuale ciclo). Poiché le capacità residue sono unitarie ed ogni nodo di R ha al più un arco entrante oppure al più un arco uscente, ogni nodo (tranne s e p) appare in al più uno di tali cammini, e pertanto f ha un cammino aumentante di lunghezza $k \leq (n-2)/|f^* - f| + 1$. Ma dopo $\lceil 2\sqrt{n-2} \rceil$ iterazioni, $k \geq \sqrt{n-2} + 1$, perché ad ogni iterazione la lunghezza k dei cammini aumentanti più corti aumenta. Combinando quest'ultime due relazioni, si ricava che $|f^* - f| \leq \sqrt{n-2}$ e quindi dopo al più altre $\lfloor \sqrt{n-2} \rfloor$ iterazioni viene trovato il flusso massimo. Pertanto, il numero di iterazioni effettuate da kmn() è $O(\sqrt{n})$.

15.10 Si costruisca una versione orientata $G' = (V, E')$ del grafo, dove ad ogni arco $[u, v]$ in E corrispondono due archi (u, v) e (v, u) in E'. G' ha n vertici e $2m$ archi. Si associ una funzione di capacità c tale che per ogni arco $(u, v) \in E'$, $c(u, v) = 1$.

Sia $f^*(u, v)$ il flusso massimo fra u e v in G', per ogni u e v. Per il teorema Flusso massimo/Taglio minimo, $f^*(u, v)$ è la dimensione del minimo insieme di nodi che separa u da v. Scegliamo un nodo u ed eseguiamo l'algoritmo per calcolare $f^*(u, v)$ per tutti i nodi $v \neq u$. L'arco connettività è il minimo valore $|c^*| = \min_{v \neq u} |f^*(u, v)|$. La complessità è $O(n^2(n + m))$, in quanto si ripete n volte l'algoritmo di Ford-Fulkerson di costo $|f^*|(n + m)$, con $|f^*| = O(n)$.

BACKTRACK

Capitolo 16

Una tecnica non molto astuta, ma che talvolta può risultare utile, si basa sulla ovvia considerazione seguente:

"Prova a fare qualcosa e, se non funziona, disfala e riprova qualcos'altro".

Questa tecnica, detta *backtrack*, è alla base di molti algoritmi di visita di alberi e grafi, ed è usata per generare sistematicamente tutte le soluzioni ammissibili di un problema (*to backtrack* significa "tornare sui propri passi").

16.1 ESEMPIO – [Visita di alberi e grafi]
Le procedure previsita(), postvisita() e invisita() negli alberi [capitolo 5] sono basate sul *backtrack*, dove l'elaborazione su un nodo è sospesa temporaneamente per permettere la visita di alcuni suoi sottoalberi. Anche la dfs() nei grafi [capitolo 9], che è una diretta estensione della previsita(), è basata sul *backtrack*.

Per illustrare questa tecnica, vedremo innanzitutto un problema particolare, l'INVILUPPO CONVESSO; vedremo poi uno schema generale che può essere applicato ad una vasta classe di problemi.

16.1 INVILUPPO CONVESSO

Un poligono nel piano bidimensionale è *convesso* se ogni segmento di retta che congiunge due punti del poligono sta interamente nel poligono stesso, incluso il bordo.

INVILUPPO CONVESSO (CONVEX HULL) Dati n punti $p_1, \ldots, p_n$ nel piano, con $n \geq 3$, trovare il più piccolo poligono convesso che li contiene tutti.

Intuitivamente, i punti possono essere pensati come chiodi conficcati parzialmente in un'asse di legno ed il bordo del loro inviluppo convesso come la sagoma formata da un elastico che li stringe tutti.

16.2 ESEMPIO – [Inviluppo convesso]
La Fig. 16.1 mostra un poligono non convesso, un poligono convesso, un insieme di punti e il loro inviluppo convesso.

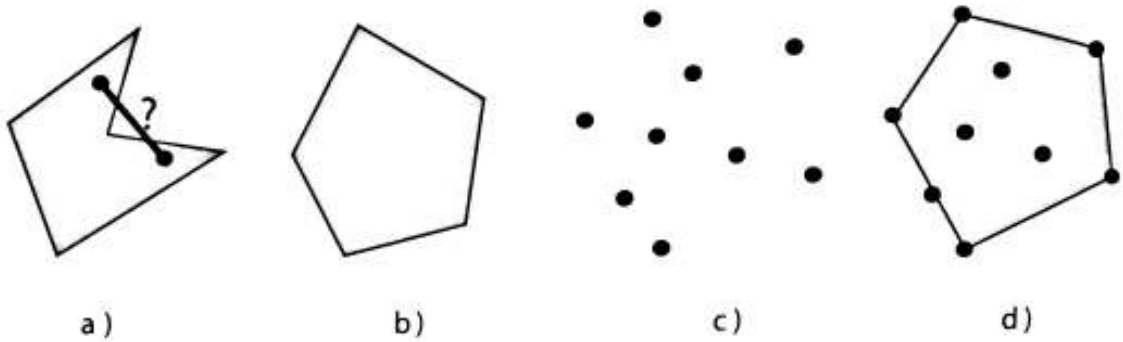


Fig. 16.1 Un poligono non convesso; b) un poligono convesso; c) un insieme di punti; d) il loro inviluppo convesso.

Un algoritmo banale per trovare l'inviluppo convesso si basa sulla seguente caratterizzazione della soluzione. Un poligono può essere rappresentato per mezzo dei suoi spigoli, cioè dei segmenti di retta che giacciono sul suo bordo. Si consideri allora un generico segmento S_{ij} che unisce due punti distinti p_i e p_j, $1 \leq i < j \leq n$, e si consideri la retta che include S_{ij}. Tale retta divide il piano in due parti (dette *semipiani chiusi*, ognuno dei quali comprende anche la retta). Se tutti i rimanenti $n - 2$ punti stanno dalla stessa parte rispetto alla retta che passa per S_{ij}, cioè stanno tutti nello stesso semipiano, inclusa la retta, allora S_{ij} è uno spigolo dell'inviluppo convesso e p_i e p_j ne sono due vertici (si veda la Fig. 16.2). Ripetendo per tutte le $n(n-1)/2$ coppie distinte di punti, si possono trovare tutti gli spigoli dell'inviluppo convesso. Individuati gli spigoli, il poligono può essere facilmente rappresentato tramite la sequenza dei suoi vertici, elencati secondo il verso antiorario (oppure orario).

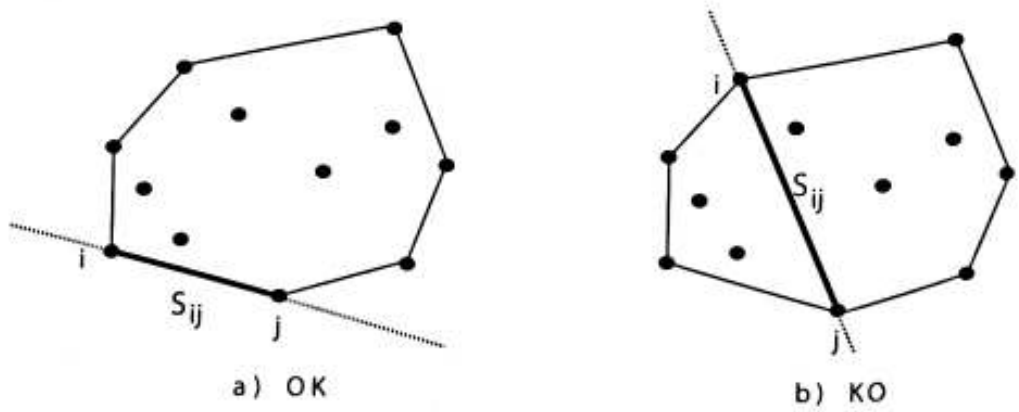


Fig. 16.2 a) Il segmento S_{ij} è uno spigolo dell'inviluppo convesso; b) il segmento S_{ij} non è uno spigolo dell'inviluppo convesso.

Per realizzare tale algoritmo, vediamo innanzitutto come sia possibile verificare in tempo $O(1)$ se due punti stanno dalla stessa parte rispetto ad una retta oppure no. Ricordiamo che l'equazione di una retta è $y = ax + b$, dove $a = \Delta_y/\Delta_x$ ne è il coefficiente angolare. La retta divide il piano nei due "semipiani" dati dalle disequazioni $y \leq ax + b$ e $y \geq ax + b$.

Poiché la retta potrebbe essere verticale (cioè Δ_x potrebbe valere 0), si moltiplica per Δ_x ottenendo $y\Delta_x - x\Delta_y - b\Delta_x = 0$, dalla quale è possibile ricavare $b\Delta_x$ se sono date le coordinate di un punto $p = (x_p, y_p)$ che appartiene alla retta: $b\Delta_x = y_p\Delta_x - x_p\Delta_y$. Consideriamo uno dei due semipiani, per esempio quello dato dalla disequazione

$$y\Delta_x - x\Delta_y - y_p\Delta_x + x_p\Delta_y \leq 0$$

Se un punto $q = (x_q, y_q)$ appartiene a tale semipiano, allora deve valere

$$y_q\Delta_x - x_q\Delta_y - y_p\Delta_x + x_p\Delta_y \leq 0$$

dalla quale segue che

$$(y_q - y_p)\Delta_x - (x_q - x_p)\Delta_y \leq 0$$

Pertanto, per ogni punto q che appartiene a tale semipiano, la precedente quantità $(y_q - y_p)\Delta_x - (x_q - x_p)\Delta_y$ deve essere negativa o nulla, mentre per ogni punto q appartenente all'altro semipiano la stessa quantità deve essere positiva o nulla. Ne consegue che due punti q e q' stanno dalla stessa parte se le rispettive quantità hanno lo stesso segno, ovvero se il loro prodotto è positivo o nullo.

Consideriamo adesso l'operazione stessaparte(), che prende in input una retta (rappresentata da due punti p_1 e p_2) e due punti p e q, e restituisce **true** se essi stanno dalla stessa parte rispetto alla retta (incluso il caso in cui almeno uno dei punti giaccia su di essa), e **false** se stanno da parti opposte rispetto ad essa. I punti sono realizzati come record contenenti due campi reali x e y. Dati una retta individuata dai due punti p_1 ed p_2, e due punti p e q, la funzione stessaparte(p_1, p_2, p, q) verifica in tempo $O(1)$ se le quantità discusse precedentemente hanno lo stesso segno, cioè se il loro prodotto è non negativo.

```
boolean stessaparte(POINT p1, POINT p2, POINT p, POINT q)
  real dx ← p2.x − p1.x
  real dy ← p2.y − p1.y
  real dx1 ← p.x − r.p1.x
  real dy1 ← p.y − r.p1.y
  real dx2 ← q.x − r.p2.x
  real dy2 ← q.y − r.p2.y
  return ((dx · dy1 − dy · dx1) · (dx · dy2 − dy · dx2) ≥ 0)
```

Utilizzando questa funzione, è un facile esercizio scrivere una procedura per trovare l'inviluppo convesso. Infatti, fissati due punti qualsiasi p_i e p_j, cioè un segmento S_{ij}, per ve-

dere se i rimanenti $n - 2$ punti stanno dalla stessa parte rispetto alla retta che passa per S_{ij} basta tenere fisso uno di tali punti, p.e. p, ed eseguire stessaparte(p_i, p_j, p, q) per $n - 3$ volte al variare di q per ciascuno dei rimanenti $n - 3$ punti. Pertanto verificare se un segmento S_{ij} è uno spigolo dell'inviluppo convesso richiede $O(n)$ tempo. Poiché ci sono in totale $n(n - 1)/2$ segmenti S_{ij} distinti, si possono individuare tutti gli spigoli dell'inviluppo convesso con complessità $O(n^3)$.

16.1.1
ALGORITMO DI GRAHAM

L'algoritmo appena illustrato non è molto efficiente, perché esamina gli n punti in modo troppo caotico. Vediamo come un esame sistematico dei punti, basato su un ordinamento ed un *backtrack*, permetta di progettare un algoritmo più efficiente, proposto da Graham (1972).

Innanzitutto, si osservi che in un insieme di punti, quello con ordinata minima è un vertice dell'inviluppo convesso. Inoltre, tracciando una retta orizzontale in tale punto e facendola ruotare intorno al punto stesso in senso antiorario si incontrano in sequenza tutti i rimanenti punti (uno per volta, se non ci sono due punti "allineati"). Come illustrato nella Fig. 16.3, i punti si possono rinumerare in base all'angolo α che la retta passante per ciascun punto p_i e quello di ordinata minima (il punto p_1) forma con l'asse orizzontale che passa per p_1. Se ci sono più punti con ordinata minima, allora quelli con ascissa minima e massima sono vertici dell'inviluppo convesso, mentre gli altri giacciono sullo spigolo che unisce i due punti con ascissa minima e massima e possono essere eliminati, e può essere scelto come punto p_1 quello di ascissa massima. Allo stesso modo, se ci sono più punti allineati, cioè con lo stesso angolo α, allora possono essere eliminati tutti tranne il più distante da p_1, cioè quello di ordinata massima, perché tali punti non possono essere vertici dell'inviluppo convesso. Una volta che i punti sono ordinati per angolo α crescente, l'eliminazione dei punti allineati può essere eseguita in tempo $O(n)$ con una scansione della sequenza ordinata dei punti. Pertanto, senza perdere in generalità, si può assumere che non ci siano punti allineati, tranne eventualmente l'ultimo punto, che può avere la stessa ordinata del primo, ma ascissa minore.

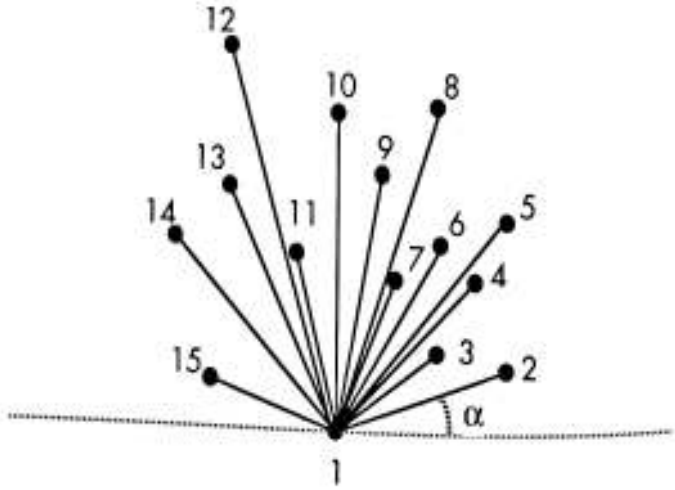


Fig. 16.3 Ordinamento dei punti in base all'angolo α formato con l'asse orizzontale quando sono connessi al punto p_1.

Si osservi che, una volta ordinati, il secondo e l'ultimo punto dell'ordinamento fanno parte dell'inviluppo convesso, oltre ovviamente a p_1. È possibile considerare i punti $p_1, \dots p_n$ per indici (cioè angoli α) crescenti e costruire in modo incrementale l'inviluppo convesso "corrente" dei primi i punti $p_1, \dots, p_i$, per $i = 3, 4, \dots, n$. Ovviamente, se $i = 3$, allora i punti p_1, p_2 e p_3 sono i vertici dell'inviluppo convesso di p_1, p_2 e p_3. Si assuma di aver costruito l'inviluppo convesso dei primi $i - 1$ punti, dato dalla sequenza dei suoi vertici in senso antiorario, e si provi ad aggiungere il nuovo punto p_i. Chiaramente, sia p_i che p_1 sono vertici dell'inviluppo convesso di $p_1, \dots, p_i$. Posto $j = i - 1$, si verifica se i punti p_i e p_1 stanno dalla stessa parte rispetto alla retta che passa per gli ultimi due vertici p_h e p_j, con $h < j$, dell'inviluppo di $p_1, \dots, p_{i-1}$; se questo non avviene, il punto p_j non può essere un vertice dell'inviluppo convesso di $p_1, \dots, p_i$ e va eliminato. Il procedimento è ripetuto aggiornando j a ritroso e considerando la retta che passa per gli ultimi due vertici p_h e p_j dell'inviluppo di $p_1, \dots, p_{i-1}$, continuando ad eliminare l'ultimo vertice p_j se i punti p_i e p_1 non stanno dalla stessa parte rispetto alla retta. Tale procedimento di eliminazione ha sicuramente termine, al più, quando si arriva a considerare la retta che passa per p_1 e p_2, poiché tali punti sono vertici dell'inviluppo convesso di $p_1, \dots, p_i$ per ogni i, $2 \le i \le n$. Queste osservazioni suggeriscono il seguente algoritmo informale:

```
STACK graham(POINT[] p, integer n)
integer min ← 1
for i ← 2 to n do
    if p[i].y < p[min].y then min ← i
p[1] ↔ p[min]
{ riordina p[2,...n] in base all'angolo formato rispetto all'asse orizzontale quando so-
  no connessi con p[1] }
{ elimina gli eventuali punti "allineati" tranne i più lontani da p1, aggiornando n }
STACK S ← Stack()
S.push(p1); S.push(p2)
for i ← 3 to n do
    while not stessaparte(S.top(), S.top2(), p1, pi) do
        S.pop()
    S.push(pi)
return S
```

La dimostrazione di correttezza dell'algoritmo segue facilmente per induzione su i in accordo alle considerazioni fatte precedentemente. La fase di *backtrack* avviene all'interno del ciclo **for**, dove si eliminano i punti che sono vertici dell'inviluppo convesso di $p_1, \dots, p_{i-1}$, ma che non lo sono più per $p_1, \dots, p_i$.

16.3 ESEMPIO – [Algoritmo di Graham]
L'esecuzione dell'algoritmo di Graham è illustrata nella Fig. 16.4, dove sono evidenziati gli spigoli dell'inviluppo "corrente" (in nero), le rette (tratteggiate) ed i punti che vengono eliminati ad ogni iterazione (con la croce), fino ad $i = 8$. Si noti che l'aggiunta del punto p_8 provoca l'eliminazione di due punti (p_7 e p_6), ma che ogni punto è inserito solo una volta ed è eliminato al più una volta.

L'inviluppo "corrente" è memorizzato utilizzando una pila S nella quale sono accatastati i vertici dell'inviluppo. In questo modo, il nodo p_j eventualmente da eliminare è sempre quello in testa alla pila, mentre il nuovo nodo p_i è anch'esso sempre inserito in testa alla pila. Per usare più semplicemente la pila, è introdotta la nuova operazione top2(), che restituisce il valore del secondo elemento a partire dalla testa della pila, senza modificare la pila.

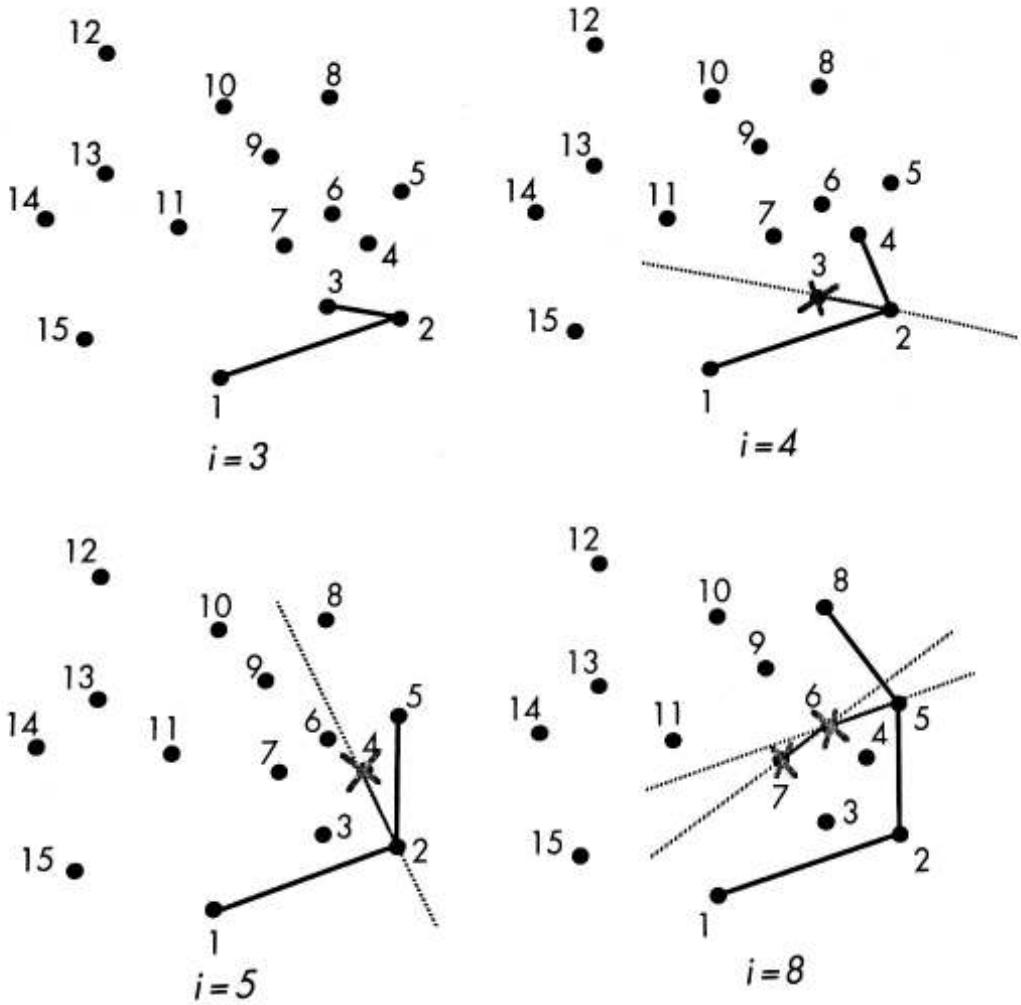


Fig. 16.4 Esecuzione dell'algoritmo di Graham fino ad $i = 8$.

Analizziamo la complessità del ciclo **for**. Tale ciclo è eseguito $n - 3$ volte ed al suo interno tutte le istruzioni, inclusa la condizione del ciclo **while**, richiedono tempo $O(1)$. Pertanto la complessità del **for** dipende esclusivamente dal numero di ripetizioni del **while**, che a sua volta dipende dal numero di punti che sono cancellati dalla pila. Ogni punto è inserito nell'inviluppo "corrente" una ed una sola volta ed è eliminato al più una volta. Quando un punto è eliminato, esso non viene più riconsiderato. Inoltre, ogni volta che è eseguito il corpo del **while** è eliminato un punto. Pertanto, la complessità del ciclo **for** è $\Theta(n)$. La complessità di tutto l'algoritmo è invece $O(n \log n)$, perché dominata dalla fase di ordinamento dei punti.

16.2 ENUMERAZIONE TRAMITE BACKTRACK

Introducendo le classi di problemi (decisionali, di ricerca, di ottimizzazione) [capitolo 11], abbiamo definito il concetto di *soluzione ammissibile*, che soddisfa un certo insieme di criteri. In alcuni casi, può essere necessario non solo cercare una soluzione ammissibile, ma anche essere in grado di elencarle tutte, dalla prima all'ultima; questo è il problema dell'*enumerazione*.

16.4 ESEMPIO – [Permutazioni]
Dato l'insieme $A = \{1, \ldots, n\}$, si consideri il problema di elencare tutte le permutazioni di A, ovvero tutte le sequenze ordinate di n elementi in cui gli elementi di A compaiono una e una sola volta.

16.5 ESEMPIO – [Sottoinsiemi]
Dato l'insieme $A = \{1, \ldots, n\}$, si consideri il problema di elencare tutti i sottoinsiemi di A contenenti esattamente k elementi.

In questi esempi, sottoinsiemi di k elementi e permutazioni rappresentano le soluzioni ammissibili.

Fra le motivazioni per elencare tutte le soluzioni ammissibili, possiamo ricordare le seguenti.

- Contare il numero di soluzioni ammissibili: tutte le volte che si ottiene una soluzione ammissibile, un contatore viene incrementato. Si tenga tuttavia conto che in molti casi il calcolo combinatorio può fornire risposte analitiche a tale problema. Ad esempio, il numero di permutazioni di un insieme di n elementi è $n!$ e il numero di sottoinsiemi di dimensione k di un insieme di n elementi è $n!/(k!(n-k)!)$
- Identificare la soluzione ottima, in base ad una certa funzione di costo. Anche in questo caso, è possibile sfruttare l'enumerazione elencando tutte le possibili soluzioni ammissibili, valutando la funzione di costo per ognuna di esse e memorizzando quella (o quelle) con costo ottimale. Sebbene sia preferibile adottare tecniche più efficienti quali programmazione dinamica e *greedy*, vi sono innumerevoli problemi [capitolo 18] in cui questo non è possibile e l'enumerazione è una delle possibili tecniche risolutive.
- Produrre (se esiste) almeno una soluzione ammissibile. Questo è il caso di molti giochi matematici, come il Sudoku. Questo problema può essere risolto semplicemente terminando l'enumerazione alla prima occorrenza di una soluzione.

Per enumerare tutte le soluzioni ammissibili, si utilizza un approccio a "forza bruta", basato su *backtrack*, che esplora lo spazio delle soluzioni in modo sistematico. Una soluzione viene rappresentata come un vettore $S[1 \ldots n]$. Il contenuto degli elementi $S[i]$ è preso da un insieme di scelte dipendente dal particolare problema. Ad ogni passo, si parte da una *soluzione parziale* $S[1 \ldots i-1]$; se è possibile, si estende tale soluzione con una delle

possibili scelte in una soluzione parziale $S[1 \ldots i]$ e si valuta la nuova soluzione parziale ricorsivamente; altrimenti, si opera un passo di *backtrack* (ritornando indietro nella ricorsione) e si effettua una nuova scelta fra quelle possibili a partire da $S[1 \ldots i-1]$.

La procedura enumerazione() realizza questo schema. Ad ogni chiamata della procedura, l'insieme C delle possibili opzioni per la scelta i-esima viene calcolato a partire dalle scelte precedenti $S[1 \ldots i-1]$. Tutte le possibili strade vengono quindi intraprese: uno dopo l'altro, tutti i valori di C vengono scelti e memorizzati in $S[i]$; se $S[1 \ldots i]$ è una soluzione ammissibile, tale soluzione viene elaborata in qualche modo (stampata, contata, valutata in base ad una funzione di costo ecc.) dalla procedura processSolution(). Infine, la procedura enumerazione() chiama ricorsivamente se stessa, dando per assodate le prime i scelte e spostandosi sull'$(i+1)$-esima. La ricorsione termina quando l'albero è stato visitato completamente.

Per supportare il caso in cui è sufficiente generare una e una sola soluzione ammissibile, le procedure enumerazione() e processSolution() restituiscono un valore booleano, vero se una soluzione ammissibile è stata trovata, falso altrimenti.

```
boolean enumerazione(ITEM[] S, integer n, integer i, ...)
  SET C ← choices(S, n, i, ...)          % Determina C in funzione di S[1 ... i − 1]
  foreach c ∈ C do
      S[i] ← c
      if S[1 ... i] è una soluzione ammissibile then
          if processSolution(S, n, i, ...) then return true
      if enumerazione(S, n, i + 1, ...) then return true
  return false
```

Applichiamo lo schema precedente all'Esempio 16.13. In questo caso, la scelta possibile è se inserire o meno un elemento; l'insieme delle scelte è quindi dato da {**true**, **false**} e il vettore S costituisce una rappresentazione booleana dell'insieme generato. L'insieme delle scelte è tuttavia vuoto quando tutti gli elementi sono già stati esaminati, ovvero quando $i > n$.

```
subsets(boolean[] S, integer n, integer k, integer i, integer count)
  SET C ← iff(i ≤ n, {true, false}, ∅)
  foreach c ∈ C do
      S[i] ← c
      count ← count + iff(S[i], 1, 0)
      if i = n and count = k then
          processSolution(S, n)
      subsets(S, n, k, i + 1, count)
      count ← count − iif(S[i], 1, 0)
```

Una prima versione dell'algoritmo, invocato da subsets(S, n, k, 1, 0), genera tutti i sottoinsiemi di un insieme di n elementi, e invoca processSolution() sui soli sottoinsiemi di k elementi. A tale scopo, la variabile *count* conta quanti elementi sono presenti nel sottoinsieme corrente. Il costo della procedura è ovviamente $O(2^n)$.

16.6 ESEMPIO – [Albero delle scelte per sottoinsieme]
L'enumerazione può essere visualizzata come la visita in profondità di un albero di scelte, dove ad ogni nodo corrisponde una soluzione (parziale oppure no). Nel caso di subsets(), l'albero è binario e completo, con tutte le sue foglie al livello n. La figura Fig. 16.5 mostra l'albero con $n = 4$, $k = 2$. I sottoinsiemi con esattamente $k = 2$ nodi sono evidenziati in grigio.

In realtà, non è necessario analizzare completamente l'albero. Ci sono tanti casi in cui una soluzione parziale non potrà mai generare una soluzione ammissibile: ad esempio, se un insieme ha già k elementi, oppure se le scelte ancora da prendere non sono sufficienti a ottenere un insieme di k elementi. In questo caso, si dice che l'albero di decisione viene "*potato*" (in inglese *pruned*): i rami che non possono generare (ulteriori) soluzioni possibili vengono ignorati.

Il nostro algoritmo può essere semplicemente modificato cambiando l'insieme delle scelte, e modificando la condizione di ammissibilità, ma il tempo resta comunque $O(2^n)$.

```
subsets(boolean[] S, integer n, integer k, integer i, integer count)
  SET C = iif(count < k and count + (n − i + 1) ≥ k, {true, false}, ∅)
  foreach c ∈ C do
      S[i] ← c
      count ← count + iif(S[i], 1, 0)
      if count = k then
          processSolution(S, n)
      subsets(S, n, k, i + 1, count)
      count ← count − iif(S[i], 1, 0)
```

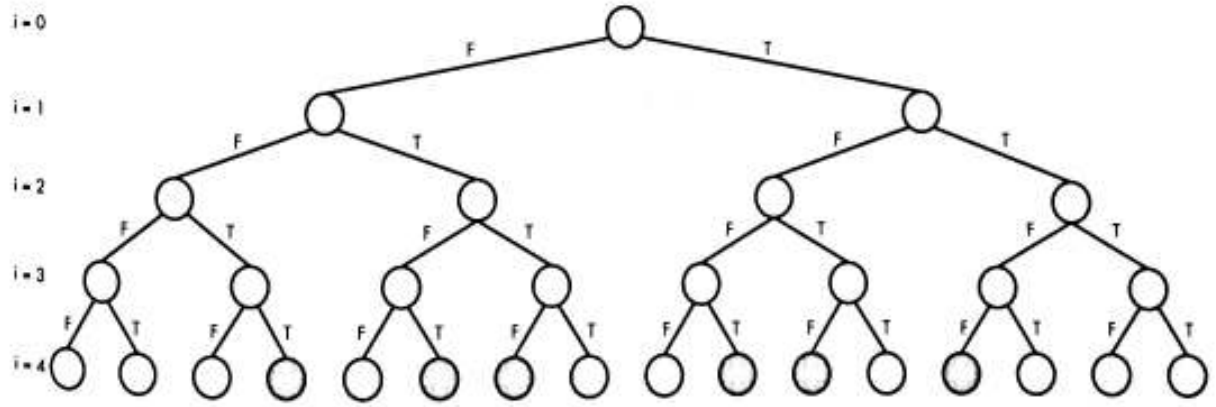


Fig. 16.5 L'albero delle scelte per la procedura subsets() con $n = 4$, $k = 2$.

16.3
REALITY CHECK

Per concludere questo capitolo, vediamo un gioco matematico che può essere risolto agilmente tramite *backtrack*. Nel Sudoku è data una tabella formata da 9 righe e 9 colonne suddivisa in 9 sottotabelle di 3 righe e 3 colonne ciascuna, dove alcune caselle sono riempite con numeri compresi tra 1 e 9. Si devono riempire le caselle vuote con numeri compresi tra 1 e 9 in modo che ciascuna riga, ciascuna colonna e ciascuna sottotabella contenga tutti i numeri tra 1 e 9. Per esempio, per la tabella in ingresso

2	5			9			7	6
			2		4			
		1	5		3	9		
	8	9	4		5	2	6	
1				2				4
	2	5	6			7	3	
		8	3		2	1		
			9		7			
3	7			8			9	2

la soluzione è:

2	5	3	8	9	1	4	7	6
8	9	7	2	6	4	3	1	5
6	4	1	5	7	3	9	2	8
7	8	9	4	3	5	2	6	1
1	3	6	7	2	9	8	5	4
4	2	5	6	1	8	7	3	9
9	6	8	3	5	2	1	4	7
5	1	2	9	4	7	6	8	3
3	7	4	1	8	6	5	9	2

L'algoritmo risolutivo è mostrato con la procedura sudoku(). La tabella del Sudoku (parzialmente inizializzata) viene memorizzata in una matrice $S[0 \ldots 8][0 \ldots 8]$ di interi, presa in input. Assumiamo che alcune caselle siano già riempite da numeri in $\{1, \ldots, 9\}$, mentre quelle vuote abbiano valore 0. Utilizziamo un indice i compreso fra 0 e 80 per indicare la casella che stiamo analizzando; questo indice viene tramutato in una coppia di coordinate (x, y) nella matrice. Se la casella è già piena, l'unica scelta possibile è data dal valore già presente. Altrimenti, un valore $c \in \{1, \ldots, 9\}$ è aggiunto all'insieme delle scelte C se e solo se scrivere c in $S[x, y]$ è una mossa ammissibile, condizione verificata dalla chiamata check(S, x, y, c). L'enumerazione termina quando viene trovata una soluzione completa (ovvero, è stata riempita l'ultima casella) e la procedura restituisce **true**. Se non esiste una soluzione ammissibile (l'input non è corretto), la procedura restituisce **false**. Il costo della procedura è... costante!, in quanto il Soduku ha dimensione fissa (però la costante è enorme). Ma è possibile generalizzare il problema a tabelle di dimensione $n^2 \times n^2$ da riempire con numeri nell'insieme $\{1, 2, \ldots, n^2\}$, dove il Sudoku classico corrisponde a $n = 3$. In questo caso il costo della procedura è superpolinomiale.

```
boolean sudoku(integer[][] S, integer i)
  SET C = Set()
  integer x ← i mod 9
  integer y ← ⌊i/9⌋
  if i ≤ 80 then
  |   if S[x, y] ≠ 0 then
  |   |   C.insert(S[x, y])
  |   else
  |   |   for c ← 1 to 9 do
  |   |   |   if check(S, x, y, c) then C.insert(c)
  integer old ← S[x, y]
  foreach c ∈ C do
  |   S[x, y] ← c
  |   if i = 80 then
  |   |   processSolution(S, n)
  |   |   return true
  |   if sudoku(S, i + 1) then return true
  S[x, y] ← old
  return false
```

ESERCIZI

16.1 [Angoli α] Se dx e dy sono le differenze tra le ascisse e le ordinate dei punti p_i e p_1, calcolare l'angolo $\alpha = \tan^{-1} dy/dx$ con la funzione di libreria per l'arcotangente richiede parecchio tempo. Dato che α serve solo nella fase di ordinamento dei punti dell'algoritmo di Graham, si può utilizzare una quantità diversa dall'angolo α, più veloce da calcolare e che produca lo stesso ordinamento. Si dimostri che $dy/(dy + dx)$ è adatta allo scopo e si scriva una opportuna funzione per calcolarla in tempo $O(1)$, tenendo conto dei casi in cui dx e dy possono essere positive, negative, o nulle.

16.2 [Tutte le permutazioni] Si scriva una procedura basata sulla tecnica *backtrack* per stampare tutte le permutazioni dei valori contenuti in un insieme A.

16.3 [Tutte le dismutazioni] Una *dismutazione* π dell'insieme $\{1, ..., n\}$ è una permutazione in cui nessun elemento è nella posizione "corretta", ovvero $\pi_i \neq i$, per $1 \leq i \leq n$. Si scriva una procedura basata sulla tecnica *backtrack* per stampare tutte le dismutazioni dei valori contenuti in un insieme A.

16.4 [Tutti i sottoinsiemi con somma k] Dato un vettore di n interi positivi $A[1 ... n]$, si scriva una procedura basata sulla tecnica *backtrack* per stampare tutti i sottoinsiemi la cui sommatoria sia pari a k.

16.5 [n regine] Il problema scacchistico delle n regine richiede di collocare n regine in una scacchiera $n \times n$, in modo che nessuna delle regine ne minacci un'altra – ovvero due regine non possono stare sulla stessa riga, colonna o diagonale. Scrivere un algoritmo che ritorni una disposizione delle regine seguendo queste regole, progettando anche un metodo per codificare la soluzione.

16.6 [Giro di cavallo] Il problema del "giro di cavallo" richiede di posizionare un cavallo su una casella di una scacchiera $n \times n$, e identificare un percorso che tocchi tutte le caselle una e una sola volta, seguendo le regole di movimento dei cavalli negli scacchi.

16.7 [Sudoku] Si completi la procedura sudoku() scrivendo la procedura check().

16.8 [Anagrammi] Si supponga di avere una funzione $d(S, i, j)$ che, dato un vettore S di caratteri, restituisce vero o falso a seconda che $S[i ... j]$ sia o non sia una parola della lingua italiana. Scrivere un programma per generare anagrammi, ovvero un programma che prende in input una stringa di caratteri (senza spazi) e genera tutte le possibili stringhe formate solo da parole italiane separate da spazi.

16.2 Seguendo lo schema discusso sopra, ad ogni passo è possibile scegliere uno degli elementi che non è già stato scelto in precedenza. Per fare questo, rimuoviamo da A l'elemento scelto, e lo re-inseriamo al momento di fare un passo di backtrack. La chiamata iniziale è permutations(A, n, S, 1).

```
permutations(SET A, integer n, ITEM[] S, integer i)
foreach c ∈ A do
    S[i] ← c
    A.remove(c)
    if A.isEmpty() then processSolution(S, n)
    permutations(A, n, S, i + 1)
    A.insert(c)
```

16.3 La soluzione è simile a quella vista per le permutazioni, con l'unico accorgimento che l'elemento c scelto nel ciclo **foreach** deve essere diverso da i.

16.5 Una regina ne può minacciare un'altra se si trovano sulla stessa riga, colonna o diagonale. Data una scacchiera $n \times n$, è possibile rappresentare la soluzione in vari modi, molti dei quali poco efficienti. Ad esempio:

- se l'insieme delle possibili scelte è metto/non metto una regina in una particolare casella, la soluzione è rappresentata da una matrice $n \times n$ di booleani e la dimensione dello spazio da analizzare (salvo potature) è pari a 2^{n^2};
- se l'insieme delle possibili scelte contiene le n^2 posizioni in cui una delle n regine può essere collocata, la soluzione è rappresentata da un vettore di n interi in $\{1, ..., n^2\}$ e la dimensione dello spazio da analizzare (salvo potature) è pari a $(n^2)^n$.

Queste dimensioni sono molto grandi; ma è possibile semplificare le cose notando che ogni riga deve contenere esattamente una regina, e lo stesso vale per le colonne. Quindi, la posizione di ognuna delle n regine può essere rappresentata da un numero in $\{1, ..., n\}$, e l'insieme delle posizioni deve essere una permutazione di tale insieme. Lo spazio da analizzare è quindi grande $n!$. Per completare la soluzione, lo spazio deve essere ulteriormente potato tenendo conto delle diagonali.

16.6 Rappresentiamo una soluzione con una matrice A di dimensione $n \times n$, dove $n = 8$ per la scacchiera classica. Poniamo $A[x, y] = k$ se il cavallo ha visitato la casella x, y al passo k-esimo. La funzione cavallo() prende in input la scacchiera e la sua dimensione n, nonché il contatore di passi k e la posizione attuale x, y. Al passo k inserisce k nella cella $A[x, y]$ e, se abbiamo riempito totalmente la scacchiera, restituisce **true**. Altrimenti, prova a fare un'ulteriore mossa delle 8 possibili mosse del cavallo, utilizzando due vettori $X = [-1, +1, +2, +2, +1, -1, -2, -2]$ e $Y = [-2, -2, -1, +1, +2, +2, +1, -1]$ per calcolare la

prossima posizione nx, ny; una mossa è ammissibile se il cavallo non esce dalla scacchiera e se la casella non è già occupata. La chiamata iniziale è cavallo(A, n, 1, x, y), dove A è inizializzato a tutti zero e x, y è una posizione qualsiasi. Al termine, se la risposta è **true**, la matrice A contiene il percorso del cavallo.

```
boolean cavallo(integer[][] A, integer n, integer k, integer x, integer y)
  A[x, y] ← k
  if k = n² then return true
  for i ← 1 to n do
      nx ← x + X[i]
      ny ← y + Y[i]
      if nx ≥ 1 and nx ≤ n and ny ≥ 1 and ny ≤ n and A[nx, ny] = 0 then
          if cavallo(A, n, k + 1, nx, ny) then return true
  A[x, y] = 0
  return false
```

16.7 La procedura check(S, x, y, c) controlla che l'aggiunta del numero c nella casella $S[x, y]$ non sia in contrasto con i valori già presenti.

```
boolean check(integer[][] S, integer x, integer y, integer c)
  for j ← 0 to 8 do
      if S[x, j] = c then return false                % Controllo sulla colonna
      if S[j, y] = c then return false                % Controllo sulla riga
  integer b_x ← ⌊x/3⌋
  integer b_y ← ⌊y/3⌋
  for i_x ← 0 to 2 do
      for i_y ← 0 to 2 do                             % Controllo sulla sottotabella
          if S[b_x · 3 + i_x, b_y · 3 + i_y] = c then return false
  return true
```

ALGORITMI PROBABILISTICI

Capitolo 17

Capita a volte di vedersi la strada bloccata da una persona, spostarsi a destra per passare, proprio mentre quella persona si sposta anch'essa nella stessa direzione; allora ci si sposta a sinistra, e di nuovo ci si trova bloccati. Se l'algoritmo seguito è "**if** destra **then** sinistra **else** destra", questo balletto non avrà fine. Ma se invece di fare una scelta *deterministica* ci si affida alla sorte, e si sceglie *casualmente* una direzione, la probabilità che la strada sia ancora bloccata dopo k mosse è $(1/2)^k$, un valore che diventa arbitrariamente piccolo con la crescita di k; prima o poi, sarà possibile passare.

Questo aneddoto suggerisce un approccio "audace" alla progettazione di algoritmi, che potrebbe essere riassunto da Virgilio in questo modo:

"*Audentes fortuna iuvat*" – "La sorte aiuta gli audaci".

Quando non si sa quale opzione prendere di fronte ad una scelta, oppure l'algoritmo per decidere in maniera corretta ha un costo computazionale troppo alto, è possibile affidarsi alla casualità: si sceglie un'opzione a caso.

Abbiamo già incontrato il concetto di probabilità parlando di analisi del caso medio, ove il tempo di calcolo è stato valutato mediando su tutti i possibili dati di ingresso, dopo aver individuato (o per lo più assunto arbitrariamente) una distribuzione di probabilità in accordo alla quale i dati di ingresso possono presentarsi. Per esempio, il caso medio del Quicksort [capitolo 12] parte dall'assunzione che tutte le permutazioni di un vettore di n elementi siano equiprobabili.

Nel caso di algoritmi *probabilistici* (detti anche algoritmi *randomizzati*) il calcolo delle probabilità è applicato non tanto ai dati di ingresso, quanto piuttosto ai dati di uscita. In un algoritmo probabilistico, la computazione dipende dai valori prodotti da un generatore di numeri casuali. Si assuma di avere a disposizione una funzione random(h, k), che quando è richiamata restituisce con la stessa probabilità un qualsiasi intero generato casualmente tra h e k, estremi inclusi[1].

[1] In pratica la funzione random() richiede tempo costante ed è realizzata con un generatore di numeri pseudo-casuali, cioè da un algoritmo deterministico che fornisce numeri che dal punto di vista statistico si comportano come se fossero veramente casuali.

17.1 ESEMPIO – [Testa o croce?]
random(0, 1) restituisce 0 oppure 1 con probabilità 1/2 ciascuno, come se fosse effettuato il lancio di una moneta e restituita testa oppure croce.

Il più semplice utilizzo della funzione random() consiste nel permutare casualmente i dati di ingresso di un problema. In questo modo una distribuzione casuale sui dati d'ingresso viene imposta preventivamente, anziché assunta successivamente nell'analisi di complessità. Vedremo questa soluzione risolvendo il problema della SELEZIONE, in cui si chiede di identificare il k-esimo elemento più piccolo all'interno di un insieme, senza dover ordinare tale insieme.

Un utilizzo più sofisticato, invece, permette di progettare algoritmi che nel risolvere un problema possono fornire un risultato scorretto (o addirittura non fornirne alcuno), ma che risultano tuttavia utilissimi qualora la probabilità che tali eventi si verifichino possa essere resa arbitrariamente piccola.

ESPRESSIONE POLINOMIALE NULLA Data un'espressione algebrica polinomiale $p(x_1, ..., x_n)$ in n variabili, determinare se p è identicamente nulla oppure no.

17.2 ESEMPIO – [Espressione Polinominale Nulla]
Si osservi che se p fosse dato già in forma semplificata, cioè come somma di monomi, allora il problema si risolverebbe banalmente verificando se esiste oppure no un coefficiente diverso da zero. Se invece p è dato come espressione algebrica arbitraria, allora il migliore algoritmo deterministico noto, basato sul metodo di semplificazione, richiede un tempo di calcolo molto elevato. È possibile definire un algoritmo probabilistico che dapprima genera casualmente una n-pla di valori $v_1, ..., v_n$ e poi valuta $p(v_1, ..., v_n)$. Se $p(v_1, ..., v_n) \neq 0$, allora l'espressione polinomiale è sicuramente non nulla. Altrimenti, se $p(v_1, ..., v_n) = 0$, c'è la possibilità che l'espressione sia nulla. Indicato con d il grado massimo delle variabili in p, è stato dimostrato che generando ciascun valore v_i tramite random(1, $2d$), $1 \leq i \leq n$, la probabilità di errore non supera 1/2. Iterando la valutazione di p più volte e assumendo l'espressione nulla se $p(v_1, ..., v_n) = 0$ per tutte le n-ple generate, la probabilità di errore può essere resa piccola a piacere.

Applicheremo tale tecnica ad un problema molto importante, quello della PRIMALITÀ: stabilire se un numero è primo oppure no.

Per concludere, si noti che questi non sono i primi esempi di algoritmi probabilistici del presente libro: la tabelle hash, infatti, si avvalgono di funzioni hash che distribuiscono le chiavi in maniera uniforme.

17.1 SELEZIONE

Dato un vettore A di n interi, la *mediana* è il numero che si troverebbe nella posizione centrale se questi fossero ordinati. A parte una piccola difficoltà nel definire la posizione centrale nel caso di n pari (si sceglie arbitrariamente $\lfloor n/2 \rfloor$ sia nel caso pari che nel caso dispari), è facile vedere che la mediana può essere calcolata in $O(n \log n)$ tempo ordinando il vettore. Ma è possibile domandarsi (i) se questo ordinamento sia necessario e, ancora più importante, (ii) se il tempo $\Omega(n \log n)$ sia necessario.

Ragionando su tali questioni, può essere interessante generalizzare e cercare di risolvere il problema della SELEZIONE.

SELEZIONE (SELECTION) Dati un vettore A di n interi ed un intero k tale che $1 \leq k \leq n$, trovare il k-esimo elemento più piccolo.

Si noti che per $k = 1$ ($k = n$) il problema si riduce a quello di trovare il minimo (massimo), mentre $k = n/2$ corrisponde al mediano.

Per trovare il k-esimo elemento più piccolo senza ordinare il vettore, si può utilizzare la procedura perno() [capitolo 12]. Ricordiamo che la procedura perno(A, *primo*, *ultimo*) scambia tra loro elementi di $A[primo \ldots ultimo]$ e restituisce l'indice j del "perno" tale che $A[i] \leq A[j]$, per $primo \leq i \leq j$, e $A[i] \geq A[j]$, per $j \leq i \leq ultimo$. La seguente procedura selezione(), proposta da Hoare (1961), richiama la perno() ed individua così il q-esimo elemento più piccolo di $A[primo \ldots ultimo]$, dove $q = j - primo + 1$ è il numero di elementi di $A[primo \ldots j]$. Se $k \leq q$, allora selezione() è riapplicata ad $A[primo \ldots j]$; se invece $k > q$, allora selezione() è riapplicata ad $A[j + 1 \ldots ultimo]$, ma per ricercare il $(k - q)$-esimo elemento più piccolo. La chiamata iniziale è selezione(A, 1, n, k).

```
ITEM selezione(ITEM[] A, integer primo, integer ultimo, integer k)
  if primo = ultimo then
  |   return A[primo]
  else
  |   integer j ← perno(A, primo, ultimo)
  |   integer q ← j − primo + 1
  |   if k = q then
  |   |   return A[j]
  |   else if k < q then
  |   |   return selezione(A, primo, j − 1, k)
  |   else
  |   |   return selezione(A, j + 1, ultimo, k − q)
```

Assumendo che perno() restituisca con la stessa probabilità una qualsiasi posizione j del vettore A, il numero medio di confronti $T(n)$ tra elementi di A effettuati da selezione() è dato dalla relazione di ricorrenza

$$
\begin{aligned}
T(n) &= 0 && \text{per } n = 0,1 \\
T(n) &= n - 1 + \frac{1}{n}\sum_{q=1}^{n-1} T(\max\{q, n-q\}) \\
&\leq n - 1 + \frac{2}{n}\sum_{q=\lceil n/2 \rceil}^{n-1} T(q) && \text{per } n \geq 2
\end{aligned}
$$

perché $\max\{q, n - q\}$ è uguale a q, se $q \geq \lceil n/2 \rceil$, mentre è uguale ad $n - q$, se $q < \lceil n/2 \rceil$.

Effettuiamo un ragionamento informale per cercare di ricavare una buona soluzione sulla quale applicare la tecnica di dimostrazione per tentativi. In media, selezione() do-

vrebbe essere richiamata ogni volta su una porzione dimezzata del vettore A. Poiché la perno() richiede tempo lineare nella lunghezza della porzione di A, il numero complessivo di confronti dovrebbe crescere come $n + n/2 + n/4 + \cdots + 1 \leq 2n$. Pertanto, una soluzione promettente di $T(n)$ con la quale tentare sembra essere $O(n)$.

Tentiamo quindi con $T(n) \leq cn$ per qualche costante c. La base dell'induzione è banalmente verificata perché $0 = T(0) = T(1) \leq c$. Si assuma che valga l'ipotesi induttiva $T(q) \leq cq$ per ogni $q \leq n - 1$. Sostituendo, si ricava:

$$
\begin{aligned}
T(n) &= n - 1 + \frac{2c}{n}\sum_{q=\lceil n/2 \rceil}^{n-1} q \leq n + \frac{2c}{n}\left(\sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil n/2 \rceil - 1} q\right) \\
&= n + \frac{2c}{n}(n(n-1)/2 - \lceil n/2 \rceil(\lceil n/2 \rceil - 1)/2) \\
&\leq n + \frac{2c}{n}(n(n-1)/2 - (n/2)(n/2-1)/2) \\
&= n + c(n-1) - (c/2)(n/2-1) = n + cn - c - cn/4 + c/2 \\
&= cn\left(\frac{1}{c} + \frac{3}{4} - \frac{1}{2n}\right)
\end{aligned}
$$

Poiché l'ultima quantità tra parentesi è minore di 1 per n sufficientemente grande e per una opportuna scelta di c, la relazione $T(n) \leq cn$ vale per ogni $n \geq m$ per le stesse costanti c ed m. Pertanto, la funzione selezione() trova il k-esimo elemento più piccolo di A in tempo lineare nel caso medio.

Tutta questa analisi è partita dall'assunzione che perno() restituisca con la stessa probabilità una qualsiasi posizione j del vettore A; questo non è vero per particolari casi dell'input, quali ad esempio un vettore già ordinato. Per ovviare a questo problema, è possibile forzare una scelta casuale del valore su cui impostare il perno, scambiando subito prima della chiamata perno() un elemento casuale fra *primo* e *ultimo* con l'elemento A[primo], in questo modo:

$$A[\text{random}(primo, ultimo)] \leftrightarrow A[primo]$$

Gli effetti sfavorevoli dovuti al caso pessimo sono pertanto ridotti e l'analisi della complessità nel caso medio risulta così più affidabile.

Tanto per curiosità, esistono altri algoritmi più sofisticati per la selezione() che richiedono tempo $O(n)$ nel caso pessimo, il primo dei quali è stato quello di Blum-Floyd-Pratt-Rivest-Tarjan (1973).

17.2 PRIMALITÀ

Un problema interessante che può essere risolto in maniera elegante tramite un algoritmo probabilistico è quello della PRIMALITÀ:

PRIMALITÀ (PRIMALITY) Dato un numero n, è tale numero primo, cioè non esiste alcun numero diverso da 1 e da n stesso che lo divida esattamente?

Il problema può essere risolto con la seguente procedura deterministica:

```
boolean primo(integer n)
integer j ← 2
while j ≤ ⌊√n⌋ do
    if n mod j = 0 then return false
    j ← j + 1
return true
```

La funzione primo() prova a dividere n per ciascun numero j compreso tra 2 e $\lfloor\sqrt{n}\rfloor$, e restituisce come risultato **true** se non è stato trovato alcun j tale che $n \bmod j = 0$, cioè tale che il resto della divisione di n per j sia zero. La sua complessità è $O(n^{1/2})$, che non è polinomiale nella dimensione del problema. Infatti, poiché un numero intero n codificato in base 2 necessita di $d = \log n + 1$ bit, la dimensione del problema è d. La complessità della funzione primo() è pertanto $O(2^d)$, cioè esponenziale nella dimensione del problema (secondo il criterio di costo logaritmico).

L'algoritmo più famoso per risolvere PRIMALITÀ è quello probabilistico di Rabin (1980). Tale algoritmo si basa sul concetto di *testimone di compostezza*.

Sia $n > 2$ il numero intero (dispari) in ingresso del quale si vuole verificare la primalità, e sia b un intero compreso tra 1 ed $n - 1$. Se n è primo, devono valere le tre proprietà seguenti. In primo luogo, $\text{mcd}(n, b) = 1$ per definizione di numero primo, dove mcd() è il massimo comun divisore. Poi deve essere $b^{n-1} \bmod n = 1$ per il Piccolo Teorema di Fermat[2]. Infine, si ha che $b^2 \bmod n = 1$ se e solo se $b = 1$ oppure $b = n - 1$. Combinando queste tre proprietà dopo aver riscritto $n - 1 = 2^v m$, dove v è il massimo esponente assegnabile a 2 ed m è dispari, si può dimostrare che se n è primo devono valere le condizioni:

(1) $\text{mcd}(n, b) = 1$;

(2) $b^m \bmod n = 1$ oppure $\exists i, 0 \le i \le v - 1$: $b^{2^i m} \bmod n = -1$.

L'intero b è un testimone che n è composto (cioè non primo) se almeno una delle condizioni (1) e (2) è falsa. Se b è un testimone che n è composto, allora n è effettivamente un numero composto. Se invece b non testimonia che n è composto, allora n può essere primo oppure composto. La verifica che b sia un testimone di compostezza per n può essere effettuata usando $O(\log n)$ operazioni aritmetiche su numeri interi di $O(\log n)$ bit. Infatti, v ed m si possono calcolare dimezzando $n - 1$ per $O(\log n)$ volte. La condizione (1) è verificabile in tempo $O(\log n)$ con il ben noto algoritmo di Euclide. Il calcolo di $b^m \bmod n$ si può svolgere ancora in tempo $O(\log n)$ per esponenziazioni successive di b [cfr. Esercizio 17.1], mentre le potenze $b^{2^i m} \bmod n$ si calcolano sempre con lo stesso tempo per quadrature successive di $b^m \bmod n$.

[2] Il Piccolo Teorema di Fermat è così chiamato per distinguerlo dal più famoso Ultimo Teorema di Fermat.

Rabin ha dimostrato che se n è un numero composto, allora ci sono almeno $3(n-1)/4$ numeri compresi tra 1 ed $n-1$ che ne testimoniano la compostezza. Ciò implica che scegliendo a caso un numero b compreso tra 1 ed $n-1$ e verificando se b testimonia la compostezza di n oppure no, c'è una probabilità inferiore ad 1/4 di rispondere erroneamente che n è primo quando invece è composto. Questo porta alla progettazione del seguente algoritmo probabilistico.

```
boolean primo(integer n)
  for j ← 1 to K do
      b ← random(1, n − 1)
      if b testimonia che n è composto then
          return false
  return true
```

Il precedente algoritmo ripete la verifica di compostezza su K valori indipendenti e uniformemente distribuiti di b e assume che n sia primo se e solo se nessuno di tali K valori è un testimone della compostezza. Mentre la risposta **false** (cioè che n è composto) è sempre corretta, la risposta **true** (cioè che n è primo) è corretta con probabilità maggiore o uguale a $1-(1/4)^K$. In altri termini, la probabilità di errore, cioè di assumere che n sia primo quando invece è composto, non supera $(1/4)^K$. Per $K = 50$, una costante moltiplicativa piccola dal punto di vista computazionale, questa probabilità è talmente piccola da essere di gran lunga inferiore alla probabilità che la risposta sbagliata sia dovuta a guasti occorsi nei circuiti hardware del calcolatore! Inoltre, poiché il valore K è una costante, il numero di operazioni aritmetiche della funzione primo() resta $O(\log n)$.

L'algoritmo di Rabin mostra come la probabilità di errore possa essere resa arbitrariamente piccola iterando il procedimento probabilistico per un opportuno numero di volte (50, nella funzione primo()), permettendo così la risoluzione probabilistica in tempo polinomiale di molti problemi decisionali. Si noti che questa tecnica di riduzione dell'errore non richiede che la probabilità iniziale di errore sia piccola, ma solo che sia minore di $1-\epsilon$ per qualche $\epsilon > 0$. Se invece di 1/4 la probabilità di errore fosse 0.99, basterebbe aumentare opportunamente il numero di iterazioni per ottenere la stessa probabilità $(1/4)^{50}$ di errore pur mantenendo la stessa complessità di $O(\log n)$ operazioni aritmetiche!

Per concludere, è interessante notare che fino a qualche anno fa l'approccio probabilistico di Rabin era l'unico conosciuto e non erano noti algoritmi deterministici di costo polinomiale. La situazione è cambiata nel 2002, quando i tre matematici indiani Agrawal, Kayal e Saxena introdussero un algoritmo di complessità polinomiale $O(\log^{12+\epsilon} n)$. Da allora, enormi sforzi di ricerca hanno portato alla scoperta di innumerevoli varianti, alcune delle quali basate su congetture matematiche non ancora dimostrate. Al momento, la versione migliore sembra essere quella di Pomerance e Lenstra, che ha complessità $O(\log^{6+\epsilon} n)$. Nonostante la presenza di algoritmi deterministici polinomiali, tuttavia, l'algoritmo di Rabin è ancora il più veloce e il più usato.

17.3 REALITY CHECK

Grazie all'algoritmo probabilistico di Rabin, decidere se un intero n è primo oppure composto non sembra essere un problema difficile. Al contrario, fattorizzare n esprimendolo come prodotto di numeri primi pare essere molto difficile. La difficoltà di questo problema, però, torna utile nella crittografia digitale a chiave pubblica, ove si sfrutta proprio tale difficoltà per proteggere la segretezza di messaggi scambiati attraverso canali di trasmissione insicuri, ad esempio in Internet.

La crittografia si usa per criptare un messaggio in modo che, anche se intercettato da una spia, non possa essere decifrato, se non da chi possieda la chiave di decodifica. La crittografia a chiave pubblica si applica quando il mittente vuole spedire un messaggio criptato ad un destinatario senza che i due condividano una chiave di codifica/decodifica prestabilita. Attraverso procedimenti correlati, basati sulle stesse proprietà, è anche possibile l'apposizione delle cosiddette firme digitali.

Se Bruno vuole spedire messaggi sicuri deve disporre di una coppia di chiavi: una chiave pubblica P, che può diffondere liberamente, e una corrispondente chiave privata S, che solo lui dovrà conoscere. Quando Alice vuole inviare un messaggio sicuro a Bruno, basta che codifichi il messaggio M attraverso la chiave pubblica di Bruno, ottenendo $C = P(M)$. Bruno, per riottenere il messaggio originale, non deve fare altro che codificare nuovamente il messaggio C ricevuto usando S, in questo modo riotterrà $M = S(C)$. Se qualcuno intercettasse il messaggio spedito da Alice non potrà conoscerne il contenuto, in quanto per decifrare il messaggio si deve conoscere S e solo Bruno conosce S. Affinché il sistema sia corretto deve valere $M = S(P(M))$ per ogni M. Affinché sia anche sicuro deve essere molto difficile risalire ad S nota P.

Il procedimento per apporre una firma digitale ad un documento D è analogo, con la differenza che stavolta lo si firma codificandolo per mezzo della chiave privata S, ottenendo così $S(D)$, e lo si riconosce per autentico decodificandolo tramite la chiave pubblica P, riottenendo $D = P(S(D))$.

Consideriamo il sistema crittografico a chiave pubblica noto come RSA, dalle iniziali degli inventori Rivest, Shamir e Adleman (1978), che utilizza l'algoritmo probabilistico di Rabin per generare in tempo polinomiale una coppia di chiavi P ed S (pubblica e privata). Il sistema si basa proprio sull'asimmetria tra la semplicità nel poter generare grandi numeri primi e la difficoltà nel trovare una fattorizzazione di grandi numeri interi.

Nel sistema RSA, ogni chiave è una coppia di interi: $P = (N, p)$ ed $S = (N, s)$. N ha circa 200 cifre decimali, mentre p e s circa 100 (questi sono valori indicativi, poiché al crescere del numero delle cifre cresce il livello di sicurezza offerto dal sistema, ma ad oggi sono consigliate chiavi di circa 1024-2048 bit). Assumiamo che il messaggio da trasmettere consista di una stringa binaria b e che M sia il corrispondente numero intero rappresentato in binario da b, e che considerazioni analoghe valgano per il messaggio criptato C. La codifica e la decodifica sono operazioni semplicissime. Infatti, $C = P(M)$ si calcola come $C = M^p \bmod N$ mentre $M = S(C)$ si calcola come $M = C^s \bmod N$. Queste operazioni si possono eseguire in tempo polinomiale (si veda l'Esercizio 17.1).

Per ottenere p e s, si procede nel modo seguente. Si generano casualmente tre numeri

primi s, x e y di 100 cifre (ciò è fattibile in tempo polinomiale usando la funzione random() per generare interi e poi l'algoritmo di Rabin per verificare se tali interi sono primi o composti). Indi si ricava $N = xy$, mentre p deve essere tale che $ps \bmod (x-1)(y-1) = 1$. Anche questa operazione è eseguibile in tempo polinomiale (si veda l'Esercizio 17.3). Si può dimostrare che, per ogni M, vale $M^{ps} \bmod N = M$, cioè che $S(P(M)) = M$. Per decifrare una codifica conoscendo $P = (N, p)$ bisogna ricavare s, ovvero risalire ad x e y. Ma anche sapendo che $xy = N$, questo resta un problema complicato perché la fattorizzazione di un numero in fattori primi pare essere un problema intrattabile. Per garantire la sicurezza, però, occorrono numeri di circa 100-200 cifre, poiché la fattorizzazione di interi piccoli non è un problema intrattabile. Comunque, finora non è stato provato formalmente alcun risultato generale: né che la fattorizzazione sia un problema risolubile in tempo polinomiale né che sia un problema inerentemente difficile [capitolo 18].

17.1 [Potenze] Si dimostri che l'elevamento a potenza x^v mod m si può effettuare in tempo polinomiale, dove x, v ed m sono interi.

17.2 [Algoritmo esteso di Euclide] Si dimostri che, dati s ed m interi, l'equazione $sp + mq = \text{mcd}(s, m)$, si risolve in tempo polinomiale con una modifica dell'algoritmo di Euclide.

17.3 [Inverso moltiplicativo] Si dimostri che, dati tre numeri primi s, x ed y, l'equazione $ps \bmod (x - 1)(y - 1) = 1$ nell'incognita p è risolvibile in tempo polinomiale usando l'algoritmo esteso di Euclide (vedi Esercizio 17.2).

17.4 [Prodotto di matrici] Date tre matrici A, B e C, quadrate $n \times n$, si vuole decidere se $AB = C$. Si fornisca un algoritmo probabilistico di complessità $O(n^2)$ per verificare se $AB = C$ con probabilità di errore al più 1/2 (Suggerimento: si generi un vettore x di n elementi uguali a -1 oppure $+1$ e si verifichi se $A(Bx) \neq Cx$).

17.5 [1/2-Soddisfattibilità] Data una formula booleana, si vuole verificare se la formula è soddisfatta da più della metà dei possibili assegnamenti dei valori di verità alle variabili booleane. Si fornisca un algoritmo probabilistico.

17.1 Assumiamo che x, v ed m siano rappresentati in binario usando d bit. Scomponiamo v come somma di potenze di 2, con una scansione della rappresentazione binaria di v, ottenendo $v = 2^{i_1} + \ldots + 2^{i_k}$, dove $i_1 < \cdots < i_k$. Calcoliamo le potenze $x^{2^j} \bmod m$ per $j = 1, 2, \ldots, i_k$, dove ciascuna è ricavata facendo il quadrato della precedente. Infine si ottiene $x^v \bmod m$ effettuando il prodotto modulo m tra le potenze: $x^{2^{i_1}} \bmod m \times \cdots \times x^{2^{i_k}} \bmod m$. Il numero di moltiplicazioni è $O(\log v) = O(d)$, quindi lineare nella dimensione del problema. Se inoltre vogliamo anche contare il numero totale di operazioni tra bit, dato che ciascuna moltiplicazione ne richiede $O(d^2)$ (in pratica, come moltiplicare due polinomi di d coefficienti), l'intero algoritmo effettua $O(d^3)$ operazioni tra bit.

17.2 Ricordiamo che, assunti $s > 0$, $m \geq 0$, ed $s \geq m$, l'algoritmo di Euclide calcola $\mathrm{mcd}(s, m)$ in tempo $O(\log m)$ tramite la ricorrenza:

$$\mathrm{mcd}(s, m) = \begin{cases} s & \text{se } m = 0 \\ \mathrm{mcd}(m, s \bmod m) & \text{altrimenti} \end{cases}$$

Il seguente algoritmo ricorsivo euclideEsteso riceve in input s ed m e restituisce in tempo $O(\log m)$ la terna $(\mathrm{mcd}(s, m), p, q)$ che verifica l'equazione $sp + mq = \mathrm{mcd}(s, m)$:

euclideEsteso$(s, m) = (s, 1, 0)$, se $m = 0$;
euclideEsteso$(s, m) = (d, p, r - \left\lfloor \frac{s}{m} \right\rfloor r)$, altrimenti,
dove $(d, p, r) =$ euclideEsteso$(m, s \bmod m)$.

17.3 Innanzitutto, che la soluzione p (detta *inverso moltiplicativo*) esista e sia unica è garantito dal fatto che per costruzione s è un numero primo minore di $(x - 1)(y - 1)$. Ponendo $m = (x - 1)(y - 1)$, l'equazione $ps \bmod m = 1$ è equivalente all'equazione $sp = mz + 1$ per un valore opportuno di z. Inoltre, ponendo $q = -z$, si ottiene l'equazione $sp + mq = \mathrm{mcd}(s, m)$ poiché, essendo s ed m primi tra loro, il loro massimo comun divisore è proprio uguale ad 1. Quindi l'equazione di partenza è un caso particolare dell'equazione generale $sp + mq = \mathrm{mcd}(s, m)$, la cui risoluzione è stata discussa nell'Esercizio 17.2.

PROBLEMI INTRATTABILI

Parte quarta

TEORIA DELL'NP-COMPLETEZZA

Capitolo 18

Per trascorrere il tempo durante un viaggio in aereo, ci si può essere cimentati con il seguente rompicapo, o con qualche sua variante semplificata.

DOMINO LIMITATO (SQUARE TILING) Dati un intero positivo n ed un insieme finito D di m tipi di "tessere orientate", cioè di quadrati di lato unitario, divisi in quattro parti dalle diagonali con ciascun quarto colorato con un colore, è possibile ricoprire un'area quadrata di lato n con copie delle tessere in D in modo che:

- nessuna tessera è "ruotata";
- una particolare tessera $d \in D$ occupa la posizione più in basso a sinistra;
- due tessere che si toccano hanno i lati adiacenti dello stesso colore?

18.1 ESEMPIO – [Domino limitato]
La Fig. 18.1 illustra due insiemi D e D', con $m = 3$ tipi di tessere ciascuno. Posto $n = 3$, è facile vedere che esiste una copertura legale con tessere di D (Fig. 18.1(a)), mentre non ne esiste alcuna con tessere di D' (Fig. 18.1(b), dove tutte le scelte effettuate sono obbligate e non c'è alcun modo di coprire la casella vuota). Si noti che se fossero ammesse rotazioni, allora la sola tessera d, qualunque essa fosse, sarebbe sufficiente a coprire ogni area di dimensione arbitraria!

Per dati di ingresso banali, come quelli dell'Esempio 18.1, il DOMINO LIMITATO può essere risolto a colpo d'occhio, ma per n ed m "grandi" e tessere "perverse", cercare una soluzione può essere alquanto frustrante. Ciò accade specialmente quando esistono molte soluzioni "parziali", che coprono cioè solo alcune porzioni del quadrato di lato n, ma ci sono pochissime soluzioni "complete" che coprono l'intero quadrato.

Un algoritmo banale che risolve il problema "provando tutte le possibilità" non è però difficile da trovare. Si osservi infatti che nel quadrato di lato n ci sono n^2 caselle da coprire con copie delle tessere. Per la casella in basso a sinistra la scelta è obbligata, mentre

ognuna delle rimanenti caselle può essere coperta con una copia di una tessera scelta tra m possibili tipi diversi. Il numero di coperture distinte è pertanto m^k, con $k = n^2 - 1$, e l'algoritmo deve semplicemente generarle tutte e verificare, per ognuna di esse, che la copertura sia "legale", ovvero che tessere adiacenti abbiano i lati che si toccano dello stesso colore. Purtroppo, questo banale algoritmo non è utilizzabile in pratica, poiché richiede un inaccettabile tempo esponenziale!

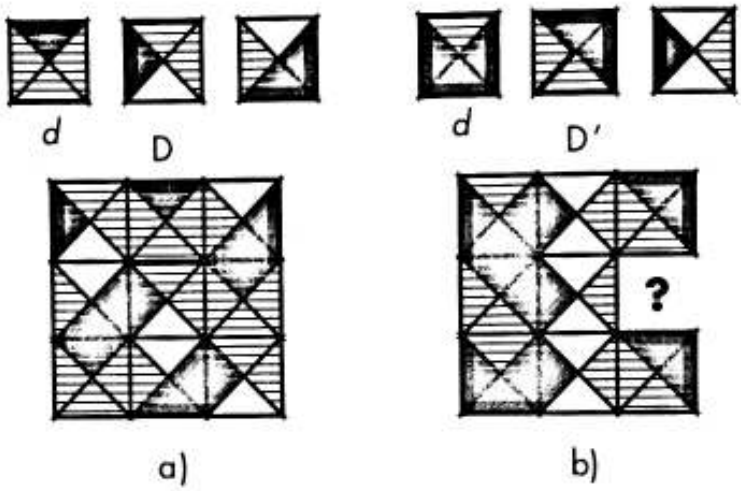


Fig. 18.1 DOMINO LIMITATO ($m = 3$, $n = 3$). a) Insieme D di tessere con copertura (risposta SÌ); b) Insieme D' di tessere senza copertura (risposta NO).

Il DOMINO LIMITATO è il problema "più difficile" tra tutti quelli visti finora. Infatti, per quasi tutti i problemi trattati nei capitoli precedenti (p.e. CAMMINI MINIMI, ORDINAMENTO, INVILUPPO CONVESSO, STRING MATCHING), oltre ad algoritmi banali del tipo "prova tutte le possibilità", erano stati introdotti algoritmi molto efficienti se non addirittura ottimi. Per il DOMINO LIMITATO, al contrario, non solo non è conosciuto un algoritmo migliore di quello che "prova tutte le possibilità", ma si suppone addirittura che un tale algoritmo non esista! Nella stessa situazione stanno molti altri problemi, come, per esempio, i seguenti:

COLORAZIONE (COLORING) Dati un grafo non orientato ed un intero k, è possibile colorarne i nodi usando al più k colori in modo che ogni nodo sia colorato con un colore diverso da tutti i nodi adiacenti?

CRICCA (CLIQUE) Dati un grafo non orientato ed un intero k, esiste un sottoinsieme di almeno k nodi tutti mutuamente adiacenti?

COMMESSO VIAGGIATORE (TRAVELING SALESPERSON, TSP) Date n città, le distanze tra esse, ed un intero k, è possibile partire da una città, attraversare ogni città esattamente una volta tornando alla città di partenza, percorrendo una distanza totale non superiore a k?

PROGRAMMAZIONE LINEARE 0/1 (0/1 LINEAR PROGRAMMING) Data una matrice A di elementi interi e di dimensione $m \times n$, ed un vettore b di m elementi interi, esiste un vettore x di n elementi 0/1 tale che $Ax \leq b$?

SODDISFATTIBILITÀ (SATISFIABILITY) Data un'espressione booleana in forma normale congiuntiva, esiste un'assegnazione di valori di verità vero/falso alle variabili booleane che rende l'espressione vera?

18.2 ESEMPIO – [COLORAZIONE]
La Fig. 18.2 mostra una colorazione con 3 colori per un grafo che rappresenta la relazione di adiacenza tra le 10 province della Toscana. Per ogni k maggiore o uguale a 3 il problema ha risposta Sì, mentre per k minore di 3 la risposta è NO.

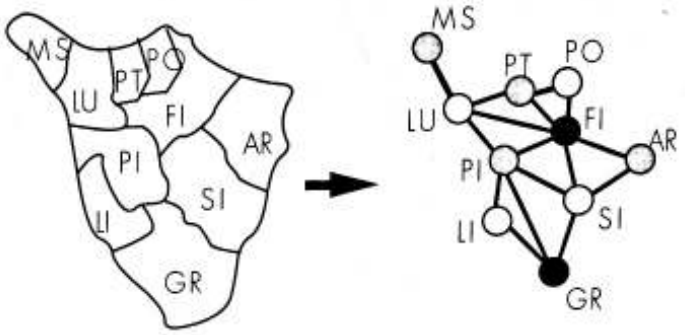


Fig. 18.2 Una colorazione con 3 colori del grafo che rappresenta la relazione di adiacenza tra province della Toscana.

18.3 ESEMPIO – [CRICCA]
Nel grafo illustrato nella Fig. 18.2, esiste una cricca con 3 nodi (p.e. {FI, PT, PO} o {PI, GR, SI}), ma non esiste alcuna cricca con più di 3 nodi.

18.4 ESEMPIO – [COMMESSO VIAGGIATORE]
La Fig. 18.3 mostra la tabella delle distanze (in linea d'aria) tra sette località della Toscana ed un percorso ottimo per il commesso viaggiatore. Per ogni k inferiore alla distanza ottima il problema ha risposta No, mentre per ogni k maggiore o uguale a tale distanza la risposta è ovviamente Sì.

18.5 ESEMPIO – [PROGRAMMAZIONE LINEARE 0/1]
Il sistema:

$$x_1 + x_2 + x_3 + x_4 \geq 2$$
$$x_1 - x_2 - x_3 + x_4 \geq 0$$
$$x_1 + x_3 + x_4 \leq 1$$

è verificato per $x_1 = x_2 = 1$ ed $x_3 = x_4 = 0$.

18.6 ESEMPIO – [SODDISFATTIBILITÀ]
La formula su u_1, u_2 e u_3:

(**not** u_1 **or not** u_2 **or not** u_3) **and** (u_2 **or not** u_3) **and** (u_1 **or not** u_2 **or** u_3)

è soddisfatta per u_1 = **false** e $u_2 = u_3$ = **true**.

Oltre a questi, centinaia di altri problemi di rilevante interesse pratico si trovano nella stessa "barca". Tutti sono tra loro computazionalmente correlati e destinati ad "affogare" o a "salvarsi" insieme. Ognuno può essere risolto in tempo superpolinomiale, ma per nes-

suno è noto un algoritmo polinomiale. Inoltre, vedremo in questo capitolo che se uno solo fosse risolubile in tempo polinomiale, allora lo sarebbero tutti, ma si suppone che non esista alcun algoritmo polinomiale per nessuno di essi!

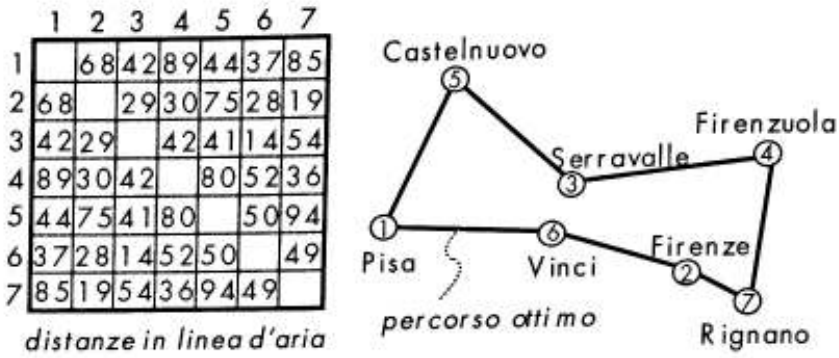


Fig. 18.3 Il percorso ottimo per il COMMESSO VIAGGIATORE tra sette località della Toscana.

18.1 CERTIFICATI POLINOMIALI

Oltre ad essere tutti di tipo decisionale, i problemi elencati precedentemente hanno in comune un'altra semplice proprietà. Benché sia estremamente difficile rispondere Sì o No, è abbastanza facile convincersi se un'ipotetica soluzione dia risposta Sì. Formalmente, tutti questi problemi possiedono un certificato polinomiale, cioè un algoritmo che, data una presunta soluzione del problema, verifica in tempo polinomiale che tale soluzione sia effettivamente una soluzione che dà risposta Sì.

18.7 ESEMPIO – [Certificato per DOMINO LIMITATO]
Dati D, n, ed una copertura, è possibile scrivere un certificato che verifica in tempo $O(n^2)$ che tale copertura sia "legale", cioè che una tessera di tipo d copra l'angolo in basso a sinistra, che ogni tessera usata sia una copia di una in D, e che le 4 tessere ad essa adiacenti abbiano i lati colorati con gli opportuni colori.

18.8 ESEMPIO – [Certificato per CRICCA]
Dati $G = (V, E)$, k ed un insieme S sottoinsieme di V, il certificato certificatoCricca() verifica che per ogni coppia $u, v \in S$ risulti $[u, v] \in E$ e che la cardinalità di S sia maggiore o uguale a k. L'algoritmo opera in tempo $O(k^2)$ se l'insieme è realizzato con una lista, e in tempo $O(n^2)$ se è realizzato con un vettore di booleani.

```
boolean certificatoCricca(GRAPH G, SET S, integer k)
  if S.size() < k then return false
  foreach u ∈ S do
      foreach v ∈ S − {u} do
          if v ∉ G.adj(u) then return false
  return true
```

18.9 ESEMPIO – [Certificato per COMMESSO VIAGGIATORE]
Dati k, la matrice D di dimensione $n \times n$ delle distanze tra le n città ed una permutazione π di $1, 2, \ldots, n$, certificatoTsp() verifica in tempo $O(n)$ che la somma delle distanze non superi k.

```
boolean certificatoTsp(integer[][] D, integer n, integer k, integer[] P)
integer t ← D[P[n], P[1]]
for i ← 2 to n do t ← t + D[P[i − 1], P[i]]
return t ≤ k
```

Un modo equivalente di descrivere questo fenomeno consiste nella possibilità di risolvere tutti questi problemi con algoritmi non deterministici di complessità polinomiale.

18.2 NON DETERMINISMO

Un *algoritmo non deterministico* è un algoritmo che, posto di fronte alla necessità di dover effettuare una "decisione", ha la "virtù magica" di scegliere sempre la strada giusta! In termini equivalenti, è come se l'algoritmo, di fronte a più alternative, le seguisse tutte contemporaneamente, generando più "copie" di se stesso. Ciascuna copia procede la computazione, indipendentemente dalle altre, seguendo una e una sola delle alternative possibili.

18.10 ESEMPIO – [Non determinismo]
In un racconto dello scrittore francese Marcel Aymé, Sabina, la protagonista, ha il dono sovrannaturale di generare una nuova copia di se stessa ogni volta che vuole sedurre un nuovo amante. Le varie Sabine hanno avventure indipendenti le une dalle altre e, al termine del racconto, il numero di Sabine è salito a qualche migliaio!

Gli algoritmi non deterministici, pur di interesse teorico, non sono ovviamente utilizzabili in pratica. Per definirli in modo formale, si supponga di avere a disposizione le tre seguenti istruzioni elementari, ciascuna delle quali richieda $O(1)$ tempo di computazione:

- **choice**(C), che sceglie arbitrariamente un elemento dell'insieme finito C;
- **failure**, che blocca la computazione in uno stato di "fallimento";
- **success**, che blocca la computazione in uno stato di "successo".

Un algoritmo non deterministico è un algoritmo che, per risolvere un problema decisionale, utilizza queste tre istruzioni. Quando viene eseguita un'istruzione **choice**(C) l'algoritmo produce $|C|$ copie di se stesso ed ogni copia procede indipendentemente dalle altre dopo aver "scelto" un elemento distinto di C. L'algoritmo fornisce risposta Sì se e solo se esiste una sequenza di scelte che portano allo stato di "successo", mentre fornisce risposta No altrimenti. Si noti che, per definizione, è data risposta Sì se una qualsiasi copia esegue un'istruzione **success**, mentre è data risposta No se tutte le copie eseguono un'istruzione **failure**!

18.41 ESEMPIO – [Algoritmo non deterministico per CRICCA]
Sia S un vettore di n elementi booleani, con n uguale al numero di nodi del grafo $G = (V, E)$. Un algoritmo non deterministico per CRICCA è illustrato nella procedura ndCricca(). Nel ciclo **for** vengono effettuate le scelte ed il nodo u va a far parte della soluzione parziale se **choice**({**true**, **false**}) = **true**. A quel punto non ci resta che verificare, tramite il certificato polinomiale, che l'insieme S così generato sia una cricca di dimensione k o maggiore. Il costo computazionale è $O(n^2)$, se per esempio l'insieme è realizzato con un vettore booleano.

```
ndCricca(GRAPH G, integer k)
  SET S ← Set()
  for i ← 1 to G.n do
      if choice({true, false}) then
          S.insert(i)
  if certificatoCricca(G, S, k) then success else failure
```

18.42 ESEMPIO – [Algoritmo non deterministico per COMMESSO VIAGGIATORE]
Presi in input la matrice delle distanze D, la dimensione del problema n e il valore k, sia S un vettore di n elementi che rappresenta la permutazione e sia C un insieme contenente le città non ancora scelte, inizializzato con $C = \{1, 2, ..., n\}$. Un algoritmo non deterministico è illustrato nella procedura ndTsp(). Nel ciclo **for** vengono effettuate le scelte, e la città scelta $S[i]$ è eliminata dall'insieme C in modo che non possa essere più scelta. In base alla chiamata a certificatoTsp() si decide il successo o il fallimento della computazione. La complessità della procedura dipende da come è realizzato l'insieme C. Per esempio, se la cancellazione di $S[i]$ da C richiede $O(n)$ tempo, allora la complessità di ndTsp() risulta essere $O(n^2)$.

```
ndTsp(integer[][] D, integer n, integer k)
  integer[] S ← new integer[1 ... n]
  SET C ← {1, 2, .... n}
  for i ← 1 to n do
      S[i] ← choice(C)
      C.remove(S[i])
  if certificatoTsp(D, n, k, S) then success else failure
```

Come nel caso dell'enumerazione, un algoritmo non deterministico è caratterizzato da un albero delle scelte. I nodi interni dell'albero rappresentano "soluzioni parziali", gli archi rappresentano "scelte", e le foglie rappresentano un "successo" o un "fallimento". Ogni percorso radice-foglia dell'albero corrisponde ad una sequenze di scelte, ovvero alla computazione di una "copia" dell'algoritmo non deterministico.

18.43 ESEMPIO – [Albero delle scelte per ndCricca()]
La Fig. 18.4 mostra un grafo G e il corrispondente albero delle scelte ottenuto eseguendo ndCricca() su G per $k = 3$.

Gli algoritmi non deterministici operano una sequenza di scelte $S[1]$, $S[2]$, ... per ottenere la soluzione. Tale sequenza di scelte è talmente correlata alla soluzione stessa da venire

spesso identificata con essa. Ogni scelta $S[i]$ avviene su un sottoinsieme C che è funzione delle scelte precedenti $S[1, ..., i-1]$ effettuate dall'algoritmo. Supponendo per semplicità che il numero delle scelte sia limitato da n, lo "scheletro" generale di una procedura non deterministica è illustrato nella procedura nonDeterministica().

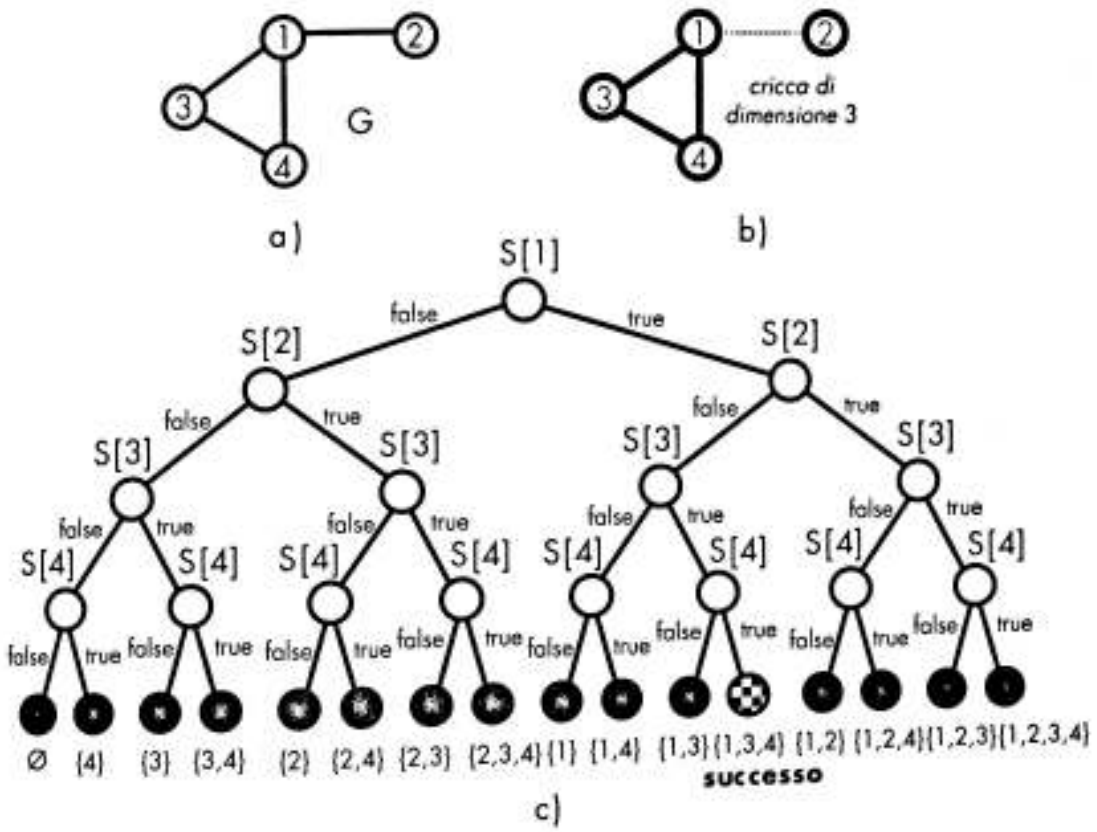


Fig. 18.4 Procedura ndCricca(). a) Il grafo G; b) una cricca di dimensione 3; c) albero delle scelte: i nodi bianchi corrispondono ad esecuzioni di istruzioni **choice**, quelli neri ad esecuzioni di **failure**, e quello a scacchi a una **success**.

```
nonDeterministica({opportuni parametri})
integer[] S ← new integer[1 ... n]
for i ← 1 to n do
    SET C ← choices(S, n, i, ...)        % Determina C in funzione di S[1 ... i − 1]
    if C = Ø then
        failure
    else
        S[i] ← choice(C)
        if S[1 ... i] è una soluzione then
            success
```

Si noti che la procedura è caratterizzata da due fasi distinte: nella prima fase viene costruita un'ipotetica soluzione, mentre nella seconda viene certificata. Se entrambe queste fasi richiedono tempo polinomiale, allora la complessità della procedura è polinomiale. Ciò accade per tutti i problemi precedentemente elencati!

18.3
ENUMERAZIONE

Poiché il non determinismo non è realistico, occorre "simulare" il comportamento di una procedura non deterministica con una deterministica. L'approccio da seguire è già noto: enumerazione basata su *backtrack*, per esplorare sistematicamente l'albero delle scelte corrispondente alla computazione deterministica. Vediamo quindi come utilizzare gli esempi visti nel capitolo 16 per risolvere i problemi della CRICCA e del COMMESSO VIAGGIATORE.

18.14 ESEMPIO – [Enumerazione per CRICCA]
Per CRICCA, possiamo partire dall'algoritmo per generare tutti gli insiemi di esattamente k membri (infatti, se esiste un insieme di nodi tutti mutuamente adiacenti e di dimensione maggiore di k, allora ne esiste uno di dimensione k) e utilizzare il certificato polinomiale per verificare se abbiamo trovato una cricca o meno; se questo avviene, l'enumerazione termina restituendo **true**, altrimenti l'enumerazione proverà tutti i possibili casi e restituirà **false**. Per semplificare lo pseudocodice e utilizzare la procedura certificatoCricca() definita in precedenza, utilizziamo un generico insieme senza dettagliare la sua realizzazione. La chiamata nel programma principale è enCricca(G, k, S, 1), dove S è un insieme vuoto. Sia che l'insieme sia realizzato con liste ordinate, liste non ordinate o vettori booleani, la complessità è $O(n^2\, 2^n)$.

```
boolean enCricca(GRAPH G, integer k, SET S, integer i)
  SET C ← iif(S.size() < k and S.size() + (G.n − i + 1) ≥ k, {true, false}, Ø)
  foreach c ∈ C do
      if c then S.insert(i)
      if certificatoCricca(G, S, k) then return true
      if enCricca(G, k, S, i + 1) then return true
      if c then S.remove(i)
  return false
```

18.15 ESEMPIO – [Enumerazione per COMMESSO VIAGGIATORE]
La procedura enTsp() descrive un algoritmo di enumerazione per il problema del COMMESSO VIAGGIATORE. Per semplificare il codice, assumiamo che la matrice delle distanze D, la dimensione del problema n e il parametro k siano variabili globali; non dovranno quindi essere passate nelle chiamate ricorsive. Il vettore S contiene una permutazione (parziale) della città da visitare; l'insieme C contiene le città ancora da visitare, ovvero le possibili scelte; l'indice i identifica la posizione in cui scrivere in S. L'algoritmo prova tutte le scelte rimanenti, rimuovendo la città da C e inserendola nel vettore S. Se la permutazione è completa e il certificato restituisce **true**, la computazione può terminare con successo. Altrimenti, si chiama ricorsivamente enTsp(); nel caso quest'ultimo restituisca **false**, si fa un passo di *backtrack* e si continua, fino all'esaurimento di tutte le permutazioni. La chiamata iniziale è enTsp(S, C, 1), dove $C = \{1, \ldots, n\}$. La complessità della procedura enTsp() è $O(nn!)$. Lasciamo per esercizio le possibili "potature" che possono essere applicate quando il costo di un cammino parziale eccede il valore k.

```
boolean enTsp(integer[] S, SET C, integer i)
foreach c ∈ C do
    C.remove(c)
    S[i] ← c
    if i = n and certificatoTsp(D, S, n, k) then return true
    if enTsp(S, C, i + 1) then return true
    C.insert(c)
return false
```

18.4 LE CLASSI $\mathbb{P}$ ED $\mathbb{NP}$

Per poter trattare in modo omogeneo problemi estremamente eterogenei tra loro (come problemi su grafi, algebrici, geometrici, numerici) facciamo alcune ipotesi semplificative:

(1) i problemi siano di tipo decisionale;

(2) il linguaggio di programmazione sia dotato delle operazioni **choice**, **success** e **failure**;

(3) la codifica dei numeri utilizzati nel calcolo sia *concisa*, ovvero si adotti una rappresentazione in base $b > 1$.

Vediamo di giustificare queste tre ipotesi e di mostrare come esse non siano affatto restrittive. Innanzitutto, un problema di tipo decisionale non può essere più difficile di uno di ricerca o di ottimizzazione. Pertanto, se si dimostra l'intrattabilità computazionale del problema in forma decisionale, allora si è dimostrata l'intrattabilità del problema formulato in uno degli altri due modi.

18.16 ESEMPIO – [Formulazioni per CRICCA]
Il problema della CRICCA è stato formulato sotto forma decisionale. Nella formulazione di ricerca si richiede di trovare una cricca di dimensione maggiore o uguale a k e in quella di ottimizzazione di trovare una cricca di dimensione massima. Ovviamente, se si risolve il problema di ricerca, allora si risolve anche quello decisionale, perché l'esibizione di una cricca è una prova della sua esistenza. Inoltre, se si risolve il problema di ottimizzazione, allora si risolvono anche gli altri due. Infatti, una cricca di dimensione massima h permette sia di rispondere al problema decisionale per ogni k, in quanto la risposta è Sì per ogni $k \leq h$ e No per ogni $k > h$, che di esibire una soluzione del problema di ricerca per ogni $k \leq h$ e di affermare che non esiste una soluzione per $k > h$.

In secondo luogo, sebbene non possiamo fornire qui una dimostrazione matematica, lo pseudocodice utilizzato in questo testo è un linguaggio di programmazione "universale", cioè non ci sono problemi risolubili con algoritmi scritti in linguaggi diversi dal nostro pseudocodice. Non solo, ma lo pseudocodice è "polinomialmente correlato" agli altri linguaggi universali, cioè non ci sono problemi di complessità polinomiale che non siano risolubili con procedure in pseudocodice anch'esse di complessità polinomiale (al più, può

cambiare il grado del polinomio). Pertanto, non è restrittivo considerare lo pseudocodice, integrato dalle operazioni **choice**, **success** e **failure**.

Infine, è ben noto che, utilizzando i calcolatori elettronici, tutti i dati sono rappresentati alla fin fine con sequenze di bit. Comunque, rappresentare i dati di un problema in base 2 o in base 5 non influenza la risolubilità polinomiale del problema. Infatti, è facilmente verificabile come si possa passare in tempo polinomiale dalla rappresentazione in base 2 di un dato alla sua rappresentazione in base 5, e viceversa. In altri termini, finché si codificano numeri con basi $b > 1$, la dimensione dell'input è polinomialmente correlata alla dimensione dello stesso input rappresentata con un'altra base qualsiasi $b' > 1$. Questa proprietà invece non è verificata rappresentando l'input con una base unaria, in cui il numero k è codificato con una sequenza di k "uno". Evitando di usare la base $b = 1$, si impedisce che il problema diventi "difficile" solo a causa di un'inopportuna codifica non concisa.

18.47 ESEMPIO – [Codifica in base $b > 1$]

Nell'ORDINAMENTO, abbiamo assunto la dimensione del problema pari al numero n di elementi da ordinare. A voler essere rigorosi, la dimensione dovrebbe essere $n \log_b a$, con $a = \max\{a_i\}$, poiché per codificare il numero a in base $b > 1$ occorrono $\log_b a$ caratteri[1]. Anche per a molto grande, comunque, la funzione $\log_b a$ cresce polinomialmente e le due dimensioni $d_1 = n$ e $d_2 = n \log_b a$ sono tra loro polinomialmente correlate. Per esempio, anche se il valore a del più grande elemento in ingresso cresce esponenzialmente col numero n di elementi da ordinare, cioè se $a = c^n$ con $c > 1$, allora $\log_b a$ è $O(n)$ e le due dimensioni d_1 e d_2 sono tali che d_2 è $O(d_1^2)$ mentre d_1 è $O(\sqrt{d_2})$. Usando la base $b = 1$, questo non è più vero. La codifica di a richiede a caratteri e la dimensione $d_3 = na$ non è più polinomialmente correlata né a $d_1 = n$ né a $d_2 = n \log_b a$. Infatti, per $a = c^n$, si ha $d_3 = na = nc^n$, che è esponenziale in n, e quindi d_3 è $O(d_1 n^{d_1})$.

Possiamo così definire le classi $\mathbb{P}$ ed $\mathbb{NP}$ come segue:

- $\mathbb{P}$ è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi deterministici;
- $\mathbb{NP}$ è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi non deterministici.

Ovviamente, la classe $\mathbb{P}$ è contenuta nella classe $\mathbb{NP}$, poiché ogni algoritmo deterministico è anche un algoritmo non deterministico (che semplicemente non usa mai **choice**), ma non si sa se $\mathbb{P}$ sia propriamente contenuta in $\mathbb{NP}$ oppure se le due classi coincidano. Tale questione resta a tutt'oggi la più importante e famosa questione irrisolta dell'Informatica!

[1] Nel provare la limitazione inferiore $\Omega(n \log n)$ per l'ORDINAMENTO, si è implicitamente assunto che i numeri a_i fossero codificati con una quantità costante di caratteri. Questa ipotesi, benché teoricamente scorretta, è realistica. Infatti, una parola di memoria di un moderno computer ha 64 bit, che permettono di codificare numeri grandi fino a $2^{63} - 1$ (un bit è usato per il segno). Numeri più grandi di $2^{63} - 1$ assai raramente si incontrano in pratica.

18.5 RIDUCIBILITÀ POLINOMIALE

Tra problemi in $\mathbb{NP}$ è utile definire una relazione $\propto$ di riducibilità polinomiale. Siano A e B due problemi decisionali. Si dice che A si riduce in tempo polinomiale a B, e si scrive $A \propto B$, se esiste una funzione f di trasformazione

$$f\text{: (dati d'ingresso per } A) \longrightarrow \text{(dati d'ingresso per } B)$$

tale che:

(1) f è computabile in tempo polinomiale con un algoritmo deterministico;

(2) x è un dato d'ingresso per cui A dà risposta Sì se e solo se $f(x)$ è un dato d'ingresso per cui B dà risposta Sì (si veda la Fig. 18.5).

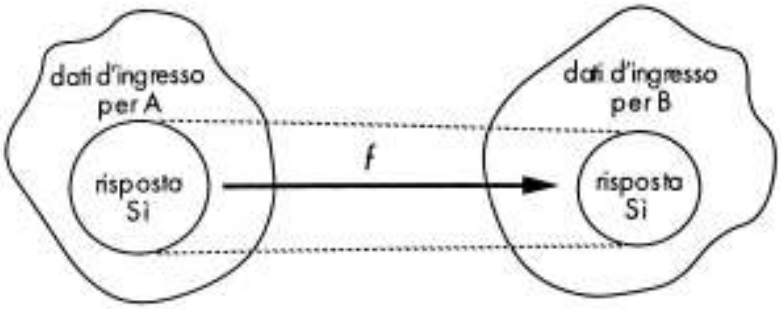


Fig. 18.5 Riduzione da un problema A ad un problema B.

In pratica, se si dimostra che $A \propto B$, si trasforma un qualsiasi dato d'ingresso per A in un particolare dato d'ingresso per B e quindi A risulta essere, in un certo senso, un sottoproblema di B! Si hanno le seguenti conseguenze.

(1) Se $B \in \mathbb{NP}$, allora anche $A \in \mathbb{NP}$. Infatti, per risolvere il problema A con un algoritmo polinomiale non deterministico su un dato x, basta trasformare x in $f(x)$ e risolvere il problema B sul dato $f(x)$: l'algoritmo risultante è ovviamente polinomiale non deterministico.

(2) Se $B \in \mathbb{P}$, allora anche $A \in \mathbb{P}$. Infatti, si opera come nel caso precedente e poiché la trasformazione f è computabile con un algoritmo polinomiale deterministico, l'algoritmo risultante è ovviamente polinomiale deterministico.

(3) Se una limitazione inferiore alla complessità di A è $\Omega(p(n))$ e la trasformazione f richiede al più tempo $O(p(n))$, allora $\Omega(p(n))$ è anche una limitazione inferiore alla complessità del problema B.

Vediamo un paio di esempi che, sebbene non siano problemi decisionali, ma problemi di ricerca e di ottimizzazione, si prestano bene ad illustrare la tecnica di riduzione polinomiale e le sue conseguenze[2].

[2] Il concetto di riduzione è già stato implicitamente utilizzato nei capitoli 11 e 15, dove si richiedeva di formulare alcuni problemi "incogniti" (l'INSIEME DOMINANTE per grafi di intervalli e l'ABBINAMENTO per grafi bipartiti [cfr. Esercizi 11.4, 11.5, 11.6 e 15.9]), come problemi di CAMMINI MINIMI o di FLUSSO MASSIMO.

In effetti, la riduzione potrebbe essere considerata come un'ulteriore tecnica di progettazione di algoritmi, che però in generale non porta ad algoritmi molto efficienti (con l'eccezione notevole dell'ABBINAMENTO per grafi bipartiti). Infatti, riducendo un problema particolare ad uno più generale, in genere si perdono alcune proprietà specifiche del problema di partenza che possono essere sfruttate per progettare algoritmi specializzati molto più efficienti.

18.18 ESEMPIO – [DOMINO LINEARE $\in \mathbb{P}$]

Si consideri una semplificazione del DOMINO LIMITATO, nota come DOMINO LINEARE, in cui si chiede di coprire un rettangolo di altezza unitaria e larghezza uguale ad n. Questo problema può essere risolto in tempo polinomiale con un algoritmo deterministico riducendolo ad un problema di "visita" di un grafo orientato che, come abbiamo visto nel capitolo 9, è in $\mathbb{P}$. Infatti, dati l'insieme D di m tipi di tessere, inclusa la tessera $d \in D$ da sistemare a sinistra, e l'intero n, si costruisca un grafo orientato $G = (V, E)$ il cui insieme di nodi V corrisponde all'insieme delle tessere D, ed esiste l'arco $(d_u, d_v) \in E$ se e solo se il lato destro della tessera d_u ha lo stesso colore del lato sinistro della tessera d_v. Chiaramente, il grafo G può essere costruito in $O(m^2)$ tempo, con $m = |D| = |V|$. È facile verificare che esiste una copertura "legale" per il DOMINO LINEARE se e solo se esiste un cammino in G a partire dal nodo corrispondente a d che include $n - 1$ archi e/o esiste un ciclo in G i cui nodi sono raggiungibili con un cammino che parte dal nodo corrispondente a d. Poiché queste due condizioni sono verificabili in tempo polinomiale applicando una BFS, il DOMINO LINEARE è risolubile con un algoritmo polinomiale deterministico. Un esempio di riduzione è fornito nella Fig. 18.6 per $m = 3$ tipi di tessere ed $n = 6$.

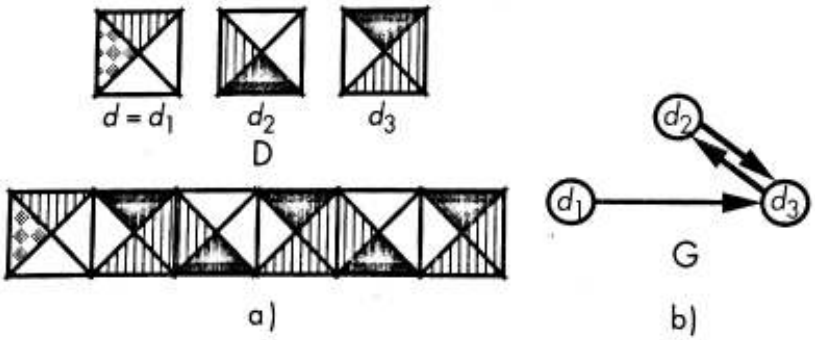


Fig. 18.6 DOMINO LINEARE ($n = 6$, $m = 3$). a) Insieme D dei tipi di tessere con copertura (risposta Sì). b) Il grafo G corrispondente con un ciclo raggiungibile da $d = d_1$.

18.19 ESEMPIO – [INVILUPPO CONVESSO è $\Omega(n \log n)$]

Dimostriamo che INVILUPPO CONVESSO è $\Omega(n \log n)$ fornendo una riduzione ORDINAMENTO $\propto$ INVILUPPO CONVESSO in cui la trasformazione f ha complessità $O(n)$. Siano dati n numeri $a_1, \ldots, a_n$ per l'ORDINAMENTO, che per semplicità assumiamo interi positivi e tra loro distinti. Si costruisca un insieme di n punti $p_1, \ldots, p_n$ del piano per l'INVILUPPO CONVESSO come segue. Si consideri la parabola $y = x^2$, e si ponga $p_i = (a_i, a_i^2)$, $1 \leq i \leq n$. La soluzione dell'INVILUPPO CONVESSO è data dai vertici del medesimo che, per i dati costruiti, sono tutti gli n punti ordinati per ascisse crescenti. Pertanto, ogni algoritmo che risolva l'INVILUPPO CONVESSO deve poter implicitamente ordinare le n ascisse $a_1, \ldots, a_n$, cioè risolvere l'ORDINAMENTO. Un esempio è illustrato nella Fig. 18.7 per $n = 4$. Poiché quest'ultimo problema è $\Omega(n \log n)$, come si è visto nel capitolo 5, e la trasformazione f è $O(n)$, anche l'INVILUPPO CONVESSO è $\Omega(n \log n)$, e quindi l'algoritmo di Graham descritto nel capitolo 16 è ottimo!

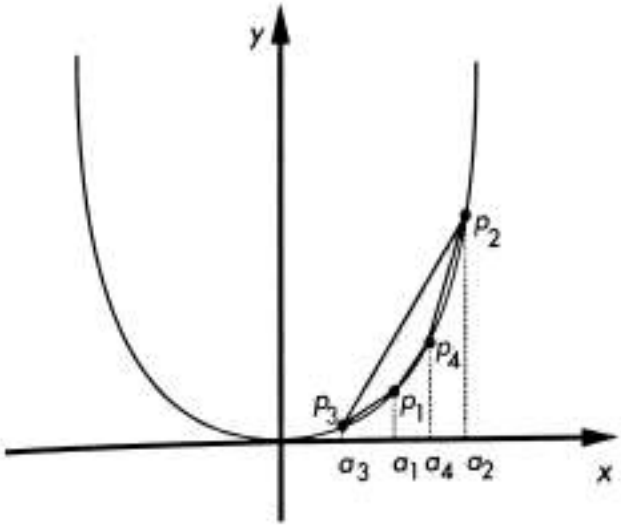


Fig. 18.7 Esempio di riduzione ORDINAMENTO $\propto$ INVILUPPO CONVESSO ($n = 4$).

18.6 TEOREMA DI COOK-LEVIN

Agli inizi degli anni settanta, S. Cook e L. Levin si sono chiesti se esistesse un particolare problema decisionale in $\mathbb{NP}$ tale che, se si dimostrasse la sua appartenenza a $\mathbb{P}$, allora dovrebbe risultare sicuramente $\mathbb{P} = \mathbb{NP}$. La risposta a questo quesito è affermativa: il problema esiste ed è il DOMINO LIMITATO! Il risultato fondamentale, dimostrato indipendentemente dall'uno e dall'altro, recita:

TEOREMA 1 - di Cook-Levin
Ogni problema in $\mathbb{NP}$ si riduce in tempo polinomiale al DOMINO LIMITATO.

La dimostrazione del teorema, benché concettualmente non molto difficile, è un po' macchinosa e viene qui presentata soltanto per sommi capi. Si assume di avere un calcolatore elettronico nella cui memoria sono caricati un programma non deterministico per risolvere il problema A ed un dato d'ingresso x per A di dimensione n. Poiché A è in $\mathbb{NP}$, il calcolatore risolverà A con dato x in tempo $p(n)$, con p funzione polinomiale in n. Poiché ogni istruzione del calcolatore manipola una quantità costante d'informazione, il calcolatore utilizzerà complessivamente un numero $cp(n)$ di bit di memoria, con c costante. Si costruisce allora un insieme opportuno D di $(cp(n))^k$ tessere, con k costante, tali che la regione da "coprire" corrisponda grossomodo a (numero di bit di memoria) $\times$ (numero di istruzioni eseguite). Le tessere sono definite "astutamente" in modo che la prima riga sia ricopribile se e solo se l'input x è corretto per A, la riga i sia ricopribile dopo la riga $i - 1$ se e solo se il calcolatore effettua un'operazione corretta per risolvere A, e l'ultima riga sia ricopribile se e solo se il calcolatore si arresta su **success**.

Il teorema di Cook-Levin ha come conseguenza il seguente risultato:

COROLLARIO 1
$\mathbb{P} = \mathbb{NP}$ se e solo se DOMINO LIMITATO $\in \mathbb{P}$.

DIMOSTRAZIONE Infatti, se DOMINO LIMITATO $\in \mathbb{P}$ ed un problema $A \in \mathbb{NP}$, allora anche $A \in \mathbb{P}$, poiché $A \propto$ DOMINO LIMITATO. Viceversa, se DOMINO LIMITATO $\notin \mathbb{P}$, allora $\mathbb{P} \neq \mathbb{NP}$, poiché DOMINO LIMITATO $\in \mathbb{NP}$.

In pratica, questo corollario del teorema di Cook-Levin asserisce che il DOMINO LIMITATO è il più difficile problema in $\mathbb{NP}$. Formalmente, un problema siffatto viene detto *NP-completo*.

Un problema A è detto *NP-arduo* se $B \propto A$ per ogni problema $B \in \mathbb{NP}$. A è detto NP-completo se, oltre ad essere NP-arduo, appartiene ad $\mathbb{NP}$. Il DOMINO LIMITATO è dunque NP-completo, ma non è il solo problema NP-completo. Da quando è stato dimostrato il teorema di Cook-Levin, sono stati scoperti centinaia di altri problemi NP-completi, inclusi CRICCA, COLORAZIONE, COMMESSO VIAGGIATORE, PROGRAMMAZIONE LINEARE 0/1 e SODDISFATTIBILITÀ. Poiché la relazione $\propto$ è transitiva (cioè se $P \propto Q$ e $Q \propto R$, allora anche $P \propto R$), tutti i problemi NP-completi sono computazionalmente equivalenti: o sono tutti risolubili in tempo polinomiale con algoritmi deterministici (e allora $\mathbb{P} = \mathbb{NP}$) oppure sono tutti intrinsecamente superpolinomiali (e quindi $\mathbb{P} \neq \mathbb{NP}$). Fino al giorno d'oggi, non è noto alcun algoritmo polinomiale deterministico per un problema NP-completo, e si pensa che non lo sarà mai! L'esistenza di un tale algoritmo è legata alla possibilità di "simulare" un algoritmo non deterministico che opera in tempo polinomiale con un algoritmo deterministico che operi anch'esso in tempo polinomiale, ma allo stato attuale non si conoscono che simulazioni superpolinomiali. Abbiamo visto infatti che la complessità di un algoritmo non deterministico è proporzionale all'altezza dell'albero delle scelte, cioè al massimo livello delle foglie, e che una simulazione deterministica richiede tempo esponenziale in tale altezza. La tesi più accreditata, quindi, è che tale simulazione sia intrinsecamente superpolinomiale e che quindi la classe $\mathbb{P}$ sia propriamente contenuta nella classe $\mathbb{NP}$.

18.7 PROVE DI NP-COMPLETEZZA

Per dimostrare che un problema A è NP-completo, non è necessario mostrare ogni volta che $B \propto A$ per ogni problema $B \in \mathbb{NP}$. Infatti, per la transitività della relazione $\propto$, "basta" mostrare che:

(1) $A \in \mathbb{NP}$;

(2) esiste un problema B, già notoriamente NP-completo, tale che $B \propto A$.

In pratica, poiché $B \propto A$ implica da sola che A è almeno tanto difficile quanto B, si fa a meno di mostrare la parte (1), e si dimostra direttamente, e in modo costruttivo, la parte (2). In linea di principio, un qualsiasi problema B che sia NP-arduo può essere un candidato per provare la riduzione, ma in pratica occorre "fiuto" per scegliere un problema B la cui "somiglianza" con A agevoli la dimostrazione. Un problema che si presta abbastanza bene, per generalità e flessibilità, a provare riduzioni polinomiali è comunque il DOMINO LIMITATO. Vediamo come questo problema possa essere ridotto a CRICCA, SODDISFATTIBILITÀ e PROGRAMMAZIONE LINEARE 0/1.

TEOREMA 2
CRICCA è NP-completa.

DIMOSTRAZIONE Mostriamo che DOMINO LIMITATO $\propto$ CRICCA. Siano dati $n > 0$, $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$, e indichiamo il quadrato di lato n da ricoprire con $[0, n-1] \times [0, n-1]$. Definiamo un grafo non orientato $G = (V, E)$ ed un intero k come segue:

(1) Per ogni casella $(i, j) \neq (0, 0)$ in $[0, n-1] \times [0, n-1]$ ed ogni $d_h \in D$, con $0 \leq h \leq m-1$, V contiene un nodo u_{hij} (l'appartenenza di u_{hij} ad una cricca "codifica" il fatto che la tessera d_h copre la casella (i, j));

(2) Per $(i, j) = (0, 0)$, V contiene il nodo u_{000} (che "codifica" la copertura di $(0, 0)$ con la tessera $d = d_0$);

(3) Ciascun nodo u_{hij} è connesso con un arco $[u_{hij}, u_{h'i'j'}] \in E$ ad ogni nodo $u_{h'i'j'}$ tale che:

(3.a) $i' = i, j' \neq j$ e $j' \neq j \pm 1$, oppure

(3.b) $j' = j, i' \neq i$ e $i' \neq i \pm 1$, oppure

(3.c) $j' \neq j$ e $i' \neq i$;

(cioè a nodi corrispondenti a caselle (i', j') distinte da (i, j) e che non sono adiacenti ad (i, j));

(4) Ciascun nodo u_{hij} è connesso con un arco $[u_{hij}, u_{h'i'j'}] \in E$ ad ogni nodo $u_{h'i'j'}$ tale che:

(4.a) d_h e $d_{h'}$ hanno i lati adiacenti dello stesso colore, ed inoltre

(4.b) $i' = i$ e $j' = j \pm 1$, oppure

(4.c) $j' = j$ e $i' = i \pm 1$,

(cioè a nodi corrispondenti a caselle (i', j') che sono adiacenti alla casella (i, j) e che possono essere coperti in modo "legale" con due tessere);

(5) $k = n^2$.

È facile verificare che la riduzione richiede tempo polinomiale in n ed m. Per costruzione, il problema DOMINO LIMITATO ha una copertura "legale" se e solo se il grafo G ha una cricca di dimensione k. Poiché CRICCA $\in \mathbb{NP}$, se n'è dimostrata l'NP-completezza.

18.20 ESEMPIO – [DOMINO LIMITATO $\propto$ CRICCA]
La Fig. 18.8 mostra un esempio di riduzione da un insieme D contenente $m = 2$ tessere per coprire un quadrato di lato $n = 2$. Per semplicità, sono disegnati separatamente gli archi corrispondenti a caselle non adiacenti (condizione (3): Fig. 18.8(b)) e quelli corrispondenti a caselle adiacenti (condizione (4): Fig. 18.8(c)). La Fig. 18.8(a) mostra una copertura legale e la Fig. 18.8(d) la corrispondente cricca.

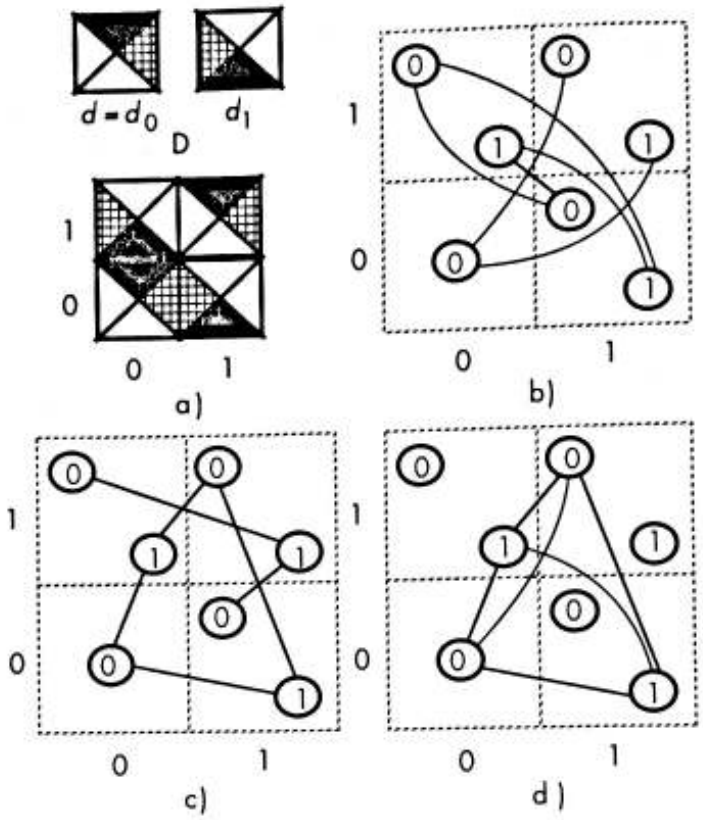


Fig. 18.8 DOMINO LIMITATO ∝ CRICCA. (a) Insieme D di $m = 2$ tipi di tessere e copertura di un quadrato di lato $n = 2$. (b) Archi di G corrispondenti a caselle non adiacenti (condizione 3). (c) Archi di G corrispondenti a caselle adiacenti (condizione 4). (d) Una cricca di dimensione $k = n^2 = 4$ corrispondente alla copertura mostrata in (a).

TEOREMA 3
SODDISFATTIBILITÀ è NP-completa.

DIMOSTRAZIONE Forniamo una riduzione dal DOMINO LIMITATO. Siano dati $n > 0$ e $D = \{d_0, d_1, ..., d_{m-1}\}$, con $d = d_0$.

Reinterpretiamo le condizioni di ricoprimento del DOMINO LIMITATO in termini di una formula booleana composta da "clausole" tra loro in "**and**". Si introducono le variabili logiche u_{hij}, $0 \le h \le m-1$, $0 \le i, j \le n-1$, con il seguente significato:

$$u_{hij} = \textbf{true} \text{ se e solo se la tessera } d_h \text{ copre la casella } (i, j)$$

Si definiscono quindi le seguenti clausole:

(1) u_{000}: questa clausola "codifica" la copertura di (0, 0) con la tessera d;

(2) $\textbf{and}_{0 \le i,j \le n-1}$ ($\textbf{or}_{0 \le h \le m-1}\ u_{hij}$): queste clausole asseriscono che almeno una tessera copre ciascuna casella del quadrato $[0, n-1] \times [0, n-1]$;

(3) $\textbf{and}_{0 \le i,j \le n-1}$($\textbf{and}_{0 \le h < h' \le m-1}$(**not** u_{hij} **or not** $u_{h'ij}$)): queste clausole asseriscono che al più una tessera copre ciascuna casella del quadrato $[0, n-1] \times [0, n-1]$;

(4) $\mathbf{and}_{0 \le i \le n-2, 0 \le j \le n-1}$ ($\mathbf{and}_{h,h'}$ (**not** u_{hij} **or not** $u_{h'i+1j}$)): per ogni coppia di tessere d_h e $d_{h'}$ che non possono essere adiacenti orizzontalmente, con d_h alla sinistra di $d_{h'}$;

(5) $\mathbf{and}_{0 \le i \le n-1, 0 \le j \le n-2}$ ($\mathbf{and}_{h,h'}$ (**not** u_{hij} **or not** $u_{h'ij+1}$)): per ogni coppia di tessere d_h e $d_{h'}$ che non possono essere adiacenti verticalmente, con d_h sotto $d_{h'}$.

La formula booleana F data dall'**and** di tutte le clausole definite in (1)-(5) è in forma normale congiuntiva e contiene un numero polinomiale, in n ed m, di variabili logiche, di clausole, e di variabili (affermate o negate) per clausola. Inoltre, il problema DOMINO LIMITATO ha una copertura "legale" per i dati n e D se e solo se la formula F è soddisfattibile. Poiché la riduzione richiede tempo polinomiale in n ed m e SODDISFATTIBILITÀ $\in \mathbb{NP}$, si è dimostrato che tale problema è NP-completo.

TEOREMA 4
PROGRAMMAZIONE LINEARE 0/1 è NP-completa.

DIMOSTRAZIONE La dimostrazione è analoga a quella del Teorema 18.3. Siano dati $n > 0$ e $D = \{d_0, d_1, ..., d_{m-1}\}$, con $d = d_0$, per il DOMINO LIMITATO. Reinterpretiamo le condizioni di ricoprimento del DOMINO LIMITATO in termini di un sistema di disequazioni, introducendo le variabili intere 0/1 x_{hij}, $0 \le h \le m-1$, $0 \le i, j \le n-1$, con il seguente significato:

$$x_{hij} = 1 \text{ se e solo se la tessera } d_h \text{ copre la casella } (i, j)$$

Si definiscono le seguenti disequazioni:

(1) $x_{000} = 1$: "codifica" la copertura di (0, 0) con la tessera d;

(2) $\sum_{0 \le h \le m-1} x_{hij} = 1$, $0 \le i, j \le n-1$: queste equazioni asseriscono che esattamente una tessera copre ciascuna casella di $[0, n-1] \times [0, n-1]$;

(3) $x_{hij} + \sum_{h'} x_{h'i+1j} \le 1$, $0 \le h \le m-1$, $0 \le i \le n-2$, $0 \le j \le n-1$, per tutte le tessere $d_{h'}$ che non possono essere adiacenti orizzontalmente a d_h, con d_h alla sinistra di $d_{h'}$;

(4) $x_{hij} + \sum_{h'} x_{h'ij} + 1 \le 1$, $0 \le h \le m-1$, $0 \le i \le n-1$, $0 \le j \le n-2$, per tutte le tessere $d_{h'}$ che non possono essere adiacenti verticalmente a d_h, con d_h sotto $d_{h'}$.

Poiché un'equazione $a = b$ può essere espressa come coppia di disequazioni $a \le b$ e $a \ge b$, il sistema definito in (1)-(4) è un sistema di disequazioni. Tale sistema contiene un numero polinomiale, in n ed m, di variabili 0/1, di disequazioni, e di variabili per disequazione. È facile verificare che il DOMINO LIMITATO ha una copertura "legale" per n e D se e solo se il sistema di disequazioni ha soluzione. La riduzione richiede tempo polinomiale in n ed m. Essendo PROGRAMMAZIONE LINEARE 0/1 $\in \mathbb{NP}$, si è provato che questo problema è NP-completo.

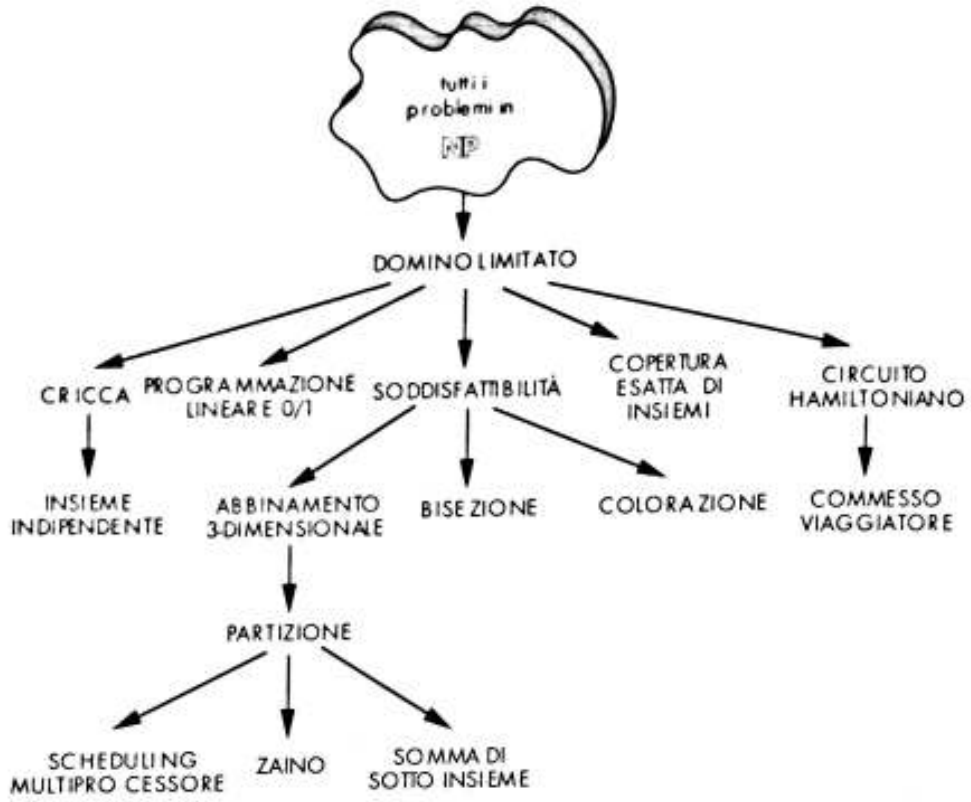


Fig. 18.9 Riduzioni tra problemi in NP.

Vediamo un breve elenco di nuovi problemi che, al pari dei precedenti, sono tutti NP-completi.

COPERTURA ESATTA DI INSIEMI (EXACT COVER) Dato un insieme X e una famiglia di suoi sottoinsiemi $Y = \{Y_1, ..., Y_n\}$, esiste una sottofamiglia F di Y che partizioni X?

ABBINAMENTO 3-DIMENSIONALE (3-DIMENSIONAL MATCHING) Dato un insieme M, sottoinsieme di $W \times X \times V$, con W, X, V insiemi disgiunti di eguale cardinalità q, esiste un sottoinsieme M' di M, con $|M'| = q$, tale che se (w, x, v) e (w', x', v') sono elementi distinti di M', allora $w \neq w'$, $x \neq x'$ e $v \neq v'$?

PARTIZIONE (PARTITION) Dato un insieme $A = \{a_1, ..., a_n\}$ di interi positivi, esiste un sottoinsieme S di $\{1, ..., n\}$ tale che $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?

SOMMA DI SOTTOINSIEME (SUBSET SUM) Dati un insieme $A = \{a_1, ..., a_n\}$ di interi positivi ed un intero positivo k, esiste un sottoinsieme S di indici in $\{1, ..., n\}$ tale che $\sum_{i \in S} a_i = k$?

ZAINO (KNAPSACK) Dati un intero positivo C (la capacità dello zaino) e un insieme di n "oggetti", tali che l'oggetto i-esimo è caratterizzato da un "profitto" p_i e da un "volume" v_i, entrambi interi positivi, esiste un sottoinsieme S di $\{1, ..., n\}$ tale che il volume totale $v(S) = \sum_{i \in S} v_i \leq C$ e il profitto totale $p(S) = \sum_{i \in S} p_i$ è maggiore o uguale a k?

SCHEDULING MULTIPROCESSORE (MULTIPROCESSOR SCHEDULING) Dati un insieme $P = \{p_1, ..., p_n\}$ di "programmi", tali che p_i richiede un "tempo (intero) di esecuzione" t_i, m "processori" identici, ed un intero d, è possibile eseguire tutti i programmi sui processori in al più d unità di tempo?

CIRCUITO HAMILTONIANO (HAMILTONIAN CIRCUIT) Dato un grafo non orientato G, esiste un circuito che attraversi ogni nodo una e una sola volta?

INSIEME INDIPENDENTE (INDEPENDENT SET) Dati un grafo non orientato G ed un intero k, esiste un sottoinsieme di almeno k nodi mutuamente non connessi da archi?

BISEZIONE (BISECTION) Dati un grafo non orientato $G = (V, E)$ ed un intero k, esiste una partizione di V in due sottoinsiemi di $|V|/2$ nodi tale che il numero di archi con un estremo in un sottoinsieme ed un estremo nell'altro non supera k?

Tra questi problemi, si possono dimostrare tutte le riduzioni illustrate nella Fig. 18.9, dove una riduzione $A \propto B$ è mostrata con una freccia da A verso B. Le riduzioni

- DOMINO LIMITATO $\propto$ CRICCA
- DOMINO LIMITATO $\propto$ SODDISFATTIBILITÀ
- DOMINO LIMITATO $\propto$ PROGRAMMAZIONE LINEARE 0/1

sono state appena dimostrate. Vediamo di provarne altre molto più semplici.

TEOREMA 5
CRICCA $\propto$ INSIEME INDIPENDENTE.

DIMOSTRAZIONE Dato un grafo non orientato $G = (V, E)$ per il problema CRICCA, si definisca il grafo "complemento" $G^c = (V, E^c)$ di G avente gli stessi nodi, ma in cui sono scambiati archi e "non archi": $[u, v] \in E^c$ se e solo se $[u, v] \notin E$. È immediato verificare che un sottoinsieme S di nodi è una soluzione di CRICCA per il grafo G se e solo è anche una soluzione di INSIEME INDIPENDENTE per G^c. Un esempio è fornito nella Fig. 18.10.

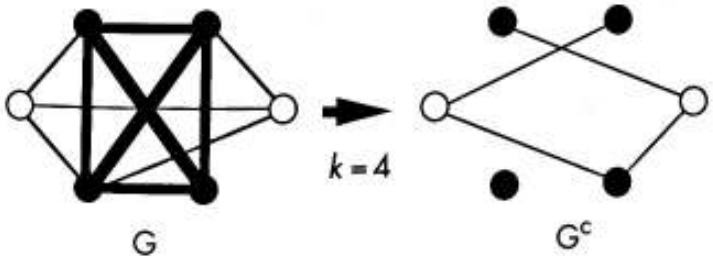


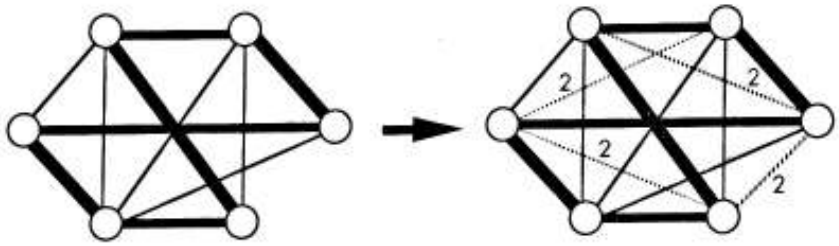
Fig. 18.10 Riduzione CRICCA $\propto$ INSIEME INDIPENDENTE. I nodi neri formano una cricca di G ed un insieme indipendente di G^c di cardinalità maggiore o uguale a $k = 4$.

TEOREMA 6
CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE.

DIMOSTRAZIONE Dato un grafo non orientato $G = (V, E)$, si definisca un insieme di dati per il COMMESSO VIAGGIATORE come segue. Si introducano n città, con $n = |V|$, si ponga $k = n$, e si definisca così la matrice $D = [d_{ij}]$ delle distanze tra le città i e j, per $1 \leq i, j \leq n$:

$$d_{ij} = \begin{cases} 1, & \text{se } [i, j] \in E, \\ 2, & \text{se } [i, j] \notin E. \end{cases}$$

È immediato verificare che esiste un circuito Hamiltoniano in G se e solo se esiste un percorso del COMMESSO VIAGGIATORE con distanza complessiva k. Un esempio è fornito nella Fig. 18.11.



G

le distanze non segnate sono tutte 1

Fig. 18.11 Riduzione CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE. Le distanze tra le città sono tutte 1, tranne quelle tratteggiate, che valgono 2 perché non corrispondono ad archi di G. Un circuito Hamiltoniano di G ed il percorso corrispondente per il commesso viaggiatore di lunghezza $k = n = 6$ sono mostrati in grassetto.

TEOREMA 7
PARTIZIONE $\propto$ SCHEDULING MULTIPROCESSORE

DIMOSTRAZIONE Dato un insieme $A = \{a_1, ..., a_n\}$ per PARTIZIONE, si definisca un insieme $P = \{p_1, ..., p_n\}$ di "programmi", tali che p_i richiede tempo $t_i = a_i$, per $1 \leq i \leq n$. Si considerino $m = 2$ "processori" e si ponga $d = \frac{1}{2}\sum_{1 \leq i \leq n} a_i$. È facile vedere che è possibile eseguire tutti i programmi sui due processori in al più d unità di tempo se e solo se esiste una soluzione per PARTIZIONE.

TEOREMA 8
PARTIZIONE $\propto$ SOMMA DI SOTTOINSIEME.

DIMOSTRAZIONE Dato l'insieme $A = \{a_1, ..., a_n\}$ per PARTIZIONE, si ponga $k = \frac{1}{2}\sum_{1 \leq i \leq n} a_i$.

TEOREMA 9
PARTIZIONE $\propto$ ZAINO.

DIMOSTRAZIONE Dato un insieme $A = \{a_1, ..., a_n\}$ per PARTIZIONE, si pongano $p_i = v_i = a_i$, $1 \leq i \leq n$, e $k = c = \frac{1}{2}\sum_{1 \leq i \leq n} a_i$.

Lo scopo delle precedenti riduzioni è mostrare come molti problemi, che nascono da contesti diversi, siano in realtà riformulazioni dello stesso problema o sue semplici varianti. In questo caso, è facile dimostrare l'NP-completezza di un problema fornendo una riduzione polinomiale da uno "che gli somigli il più possibile". La semplicità di queste riduzioni non deve però trarre in inganno, poiché molte riduzioni sono, al contrario, piuttosto complicate come, per esempio, SODDISFATTIBILITÀ $\propto$ ABBINAMENTO 3-DIMENSIONALE e DOMINO LIMITATO $\propto$ CIRCUITO HAMILTONIANO, che qui non stiamo a dimostrare.

I problemi NP-completi sono in un certo senso i più difficili problemi in $\mathbb{NP}$. Benché non sia stata dimostrata alcuna limitazione inferiore superpolinomiale alla complessità dei problemi NP-completi, il fatto che per decenni nessuno sia riuscito a trovare neanche un algoritmo polinomiale deterministico per un solo problema NP-completo fa supporre che questi problemi siano inerentemente intrattabili. Non si deve però pensare che in questi decenni non siano state provate ulteriori relazioni tra le classi $\mathbb{P}$ ed $\mathbb{NP}$ né che i problemi NP-completi siano i problemi più difficili. Ad esempio, un teorema dovuto a Ladner ci dice che se $\mathbb{P} \neq \mathbb{NP}$, allora esistono problemi in $\mathbb{NP}$ (detti *NP-intermedi*) che né stanno in $\mathbb{P}$ né sono NP-completi. Inoltre, esistono classi di problemi che non ammettono certificati polinomiali, bensì esponenziali. Un esempio è il rompicapo delle Torri di Hanoi [Capitolo 4], in cui non solo è difficile trovare una sequenza di trasferimenti che risolva il problema, ma è altrettanto difficile convincersi che una sequenza data di trasferimenti lo risolva effettivamente. Ulteriori informazioni su questi argomenti possono essere trovati sul sito del libro.

18.8 REALITY CHECK

Il Clay Mathematics Institute (CMI) è una fondazione non-profit dedicata alla diffusione della conoscenza matematica. Nel 2000, il CMI ha istituito un premio da un milione di dollari per chiunque risolva uno qualsiasi dei sette "Millenium problems", una collezione di problemi matematici "classici" ancora aperti. Fra i sette problemi c'è la questione $\mathbb{P} = \mathbb{NP}$, ovvero la domanda se le due classi di complessità coincidono oppure no. Un motivo in più per fare ricerca in questo campo! Dei sette problemi originali, uno è stato risolto: la Congettura di Poincaré è stata dimostrata da Grigori Perelman nel 2003. Nel marzo 2010 il Clay Institute ha assegnato a Perelman il premio da un milione di dollari, ma lui lo ha rifiutato, in precedenza, aveva rifiutato la medaglia Fields, il più importante premio nel campo della matematica.

Il premio Turing è invece il più prestigioso riconoscimento nel campo dell'informatica ed annovera tra i vincitori gli "algoritmisti" Edsger Dijkstra, Donald Knuth, Michael

Rabin, Robert Floyd, Antony Hoare, Richard Karp, John Hopcroft, Robert Tarjan, Ronald Rivest, Adi Shamir, Leonard Adleman, nonché i "teorici della complessità" Stephen Cook, Juris Hartmanis, Richard Stearns, Andrew Yao e Leslie Valiant.

Nel 2012, il premio Turing è stato assegnato a Silvio Micali e Shafi Goldwasser, per i loro lavori nel campo della crittografia. Micali ha nazionalità italiana e si è laureato all'Università La Sapienza di Roma nel 1978.

18.1 [Tangentopoli] Dati un insieme M di n "mazzette" ed un insieme P di n "politici corrotti" appartenenti al Partito dei Soldi (PS) oppure al Partito della Felicità (PDF), si vogliono assegnare tutte le mazzette a tutti i politici in modo che il PS riceva complessivamente il doppio del PDF. Si scriva una procedura non deterministica di complessità polinomiale.

18.2 [Zaino] Si scriva una procedura non deterministica di complessità $O(n)$ per la versione decisionale del problema dello ZAINO.

18.3 [Colorazione] Si scriva una procedura non deterministica di complessità polinomiale che risolva il problema della COLORAZIONE.

18.4 [Tangentopoli] Si scriva una procedura deterministica di enumerazione per il problema dell'Esercizio 18.1.

18.5 [Zaino] Si scriva una procedura deterministica di enumerazione per il problema dello ZAINO.

18.6 [Colorazione] Si scriva una procedura deterministica di enumerazione per il problema della COLORAZIONE.

18.7 [3-Copertura di insiemi] Si scriva una procedura deterministica di enumerazione per il problema della 3-COPERTURA DI INSIEMI: "Dati un insieme $X = \{1, 2, ..., m\}$ ed una famiglia $Y = \{Y_1, ..., Y_n\}$ di n sottoinsiemi di X, ciascuno avente cardinalità 3, esiste una sottofamiglia S di insiemi di Y che partiziona X?"

18.8 [Dimensione] Si stabilisca se le complessità delle funzioni pippo() e follia() dell'Esercizio 2.12 siano polinomiali nella dimensione dell'input oppure no.

18.9 [Counting Sort] Si stabilisca se la complessità della procedura countingSort() sia polinomiale nella dimensione dell'input oppure no.

18.10 [Coefficiente binomiale] Si stabilisca se la complessità della procedura tartaglia() definita nel capitolo 13 sia polinomiale nella dimensione dell'input oppure no.

18.11 [Copertura con nodi] Dati un grafo non orientato G ed un intero k, il problema della COPERTURA CON NODI chiede se esiste un sottoinsieme S di al più k nodi tale che ciascun arco abbia almeno un estremo in S. Si dimostri che INSIEME INDIPENDENTE $\propto$ COPERTURA CON NODI.

18.12 [Ciclo Hamiltoniano] Dato un grafo orientato G, il problema del CICLO HAMILTONIANO chiede se esiste un ciclo che include ogni nodo di G esattamente una volta. Si dimostri che CIRCUITO HAMILTONIANO $\propto$ CICLO HAMILTONIANO.

18.13 [Cammino Hamiltoniano] Dato un grafo orientato G, il problema del CAMMINO HAMILTONIANO chiede se esiste un cammino che include ogni nodo di G esattamente una volta. Si dimostri che CICLO HAMILTONIANO $\propto$ CAMMINO HAMILTONIANO. Si mostri che la riduzione vale anche se sono dati il nodo di partenza r ed il nodo di arrivo t per il CAMMINO HAMILTONIANO.

18.14 [Catena Hamiltoniana] Dato un grafo non orientato G, il problema della CATENA HAMILTONIANA chiede se esiste una catena che include ogni nodo di G esattamente una volta. Si dimostri che CICLO HAMILTONIANO $\propto$ CATENA HAMILTONIANA (Suggerimento: si modifichi la riduzione dell'Esercizio 18.13, sostituendo ogni nodo del grafo orientato D di partenza con una catena di tre nodi del grafo non orientato G di arrivo, e sistemando opportunamente in G le connessioni relative agli archi di D).

18.15 [Cammino massimo] Dati un grafo orientato G, due nodi r e t, ed un intero k, il problema del CAMMINO MASSIMO chiede se esiste un cammino semplice da r a t che include almeno k archi. Si dimostri che il problema è NP-completo. Si dimostri inoltre che il problema diventa polinomiale qualora G sia aciclico.

18.16 [Albero di copertura limitato] Dati un grafo non orientato G ed un intero k, il problema dell'ALBERO DI COPERTURA LIMITATO chiede se esiste un albero di copertura T per G tale che in ogni nodo di T siano incidenti al più k archi. Si dimostri che il problema è NP-completo.

18.1 Si rappresentino le mazzette con un vettore M di n interi ed i politici con un vettore P di n booleani tale che $P[i]$ = **true** se il politico i è del PS e $P[i]$ = **false** se è del PDF. Si ottiene la seguente procedura di complessità $O(n)$. Si noti il modo con cui nel primo **for** si generano in tempo $O(n)$ tutte le permutazioni del vettore M nel vettore S.

```
mazzette(integer[] M, integer[] S, integer[] P)
for i ← 1 to n do
    j ← choice ({i, i + 1, ..., n})
    S[i] ← M[j]
    M[j] ← M[i]
integer M_PS ← 0
integer M_PDF ← 0
for i ← 1 to n do
    if P[i] then M_PS ← M_PS + S[i]
            else M_PDF ← M_PDF + S[i]
if M_PS = 2 · M_PDF then success else failure
```

18.2 La procedura ndZaino() risolve il problema, prendendo in input le variabili V (vettore dei volumi), P (vettore dei profitti), n (dimensione del problema), C capacità dello zaino e k (profitto che si vuole raggiungere).

```
ndZaino(integer[] V, integer[] P, integer n, integer C, integer k)
SET S ← Set()
for i ← 1 to n do
    if choice({true, false}) then S.insert(i)
integer T_p ← 0
integer T_v ← 0
foreach i ∈ S do
    T_p ← T_p + P[i]
    T_v ← T_v + V[i]
if T_p ≥ k and T_v ≤ C then success else failure
```

18.5 La procedura enZaino() risolve il problema. Per semplicità, si assume che le variabili V, P, n, C e k (le stesse dell'Esercizio 18.2) siano globali. Le variabili T_v e T_p rappresentano il volume e il profitto raggiunti finora, mentre S è l'insieme di oggetti scelti. La chiamata è enZaino(S, 0, 0, 1), con S vuoto.

18.7 La procedura enCopertura() risolve il problema. Si assuma che la famiglia Y sia rappresentata con una matrice $n \times 3$, e che la sottofamiglia S sia rappresentata con un vettore booleano. Per verificare che S partizioni X, si utilizza un vettore booleano $B[1 \ldots m]$. Se Y_j

è stato selezionato ($S[j]$ = **true**), allora $B[Y[j, k]]$ è posto a **true** per $k = 1, 2, 3$. Se però $B[Y[j, k]]$ era già **true**, allora non si ha una partizione. Infine, si verifica che non sia rimasto alcun elemento di B a **false**, poiché anche in tal caso non si ha una partizione. La chiamata è enCopertura(i) e la complessità è $O((n + m)2^n)$. Si può migliorare l'algoritmo "potando" tutti quei casi in cui due sottoinsiemi sono intersecanti, prima di arrivare a $i = n$.

```
boolean enZaino(SET S, integer T_p, integer T_v, integer i)
  if i ≤ n then
      if enZaino(S, T_p, T_v, i + 1) then return true          % Elemento i non inserito
      S.insert(i)
      if T_v + V[i] ≤ C and T_p + P[i] ≥ k then return true
      if enZaino(S, T_p + P[i], T_v + V[i], i + 1) then return true   % Elemento i inserito
      S.remove(i)
  return false
```

```
boolean enCopertura(integer[][] Y, integer n, integer m, boolean[] S, integer i)
  if i ≤ n then
      S[i] ← false
      if enCopertura(Y, n, m, S, i + 1) then return true
      S[i] ← true
      if enCopertura(Y, n, m, S, i + 1) then return true
  else
      for j ← 1 to m do B[j] ← false
      for j ← 1 to n do if S[j] then
          for k ← 1 to 3 do
              if B[Y[j, k]] then return false
                          else B[Y[j, k]] ← true
      for j ← 1 to m do if not B[j] then return false
      return true
```

18.8 Le funzioni pippo() e follia() ricevono in ingresso un intero n. Poiché per codificare n in modo conciso occorrono $O(\log n)$ caratteri, la dimensione d dell'input è $O(\log n)$. La funzione pippo() richiede $O(\log n) = O(d)$ tempo, e quindi ha complessità polinomiale nella dimensione dell'input. La funzione follia() invece richiede $O(n) = O(2^d)$ tempo, e quindi ha complessità esponenziale nella dimensione dell'input.

18.9 La complessità di Counting Sort è $O(n + k)$, dove n è il numero di elementi e k è il valore dell'elemento maggiore. Poiché k è espresso con $d = \log k$ bit, la complessità è $O(n + 2^d)$, superpolinomiale in d.

TECNICHE RISOLUTIVE PER PROBLEMI INTRATTABILI

Capitolo 19

Se si deve risolvere un particolare problema, e si riesce a dimostrare la sua NP-completezza, che cosa si può fare? In generale, bisogna adottare un approccio diverso, che può essere riassunto dalla massima seguente:

"Chi si accontenta gode".

Non si può avere tutto dalla vita; bisogna rinunciare a qualcosa. Ad esempio, alla

- *generalità*: si può vedere se esiste qualche valore particolare dell'input che renda il problema risolubile in tempo polinomiale, utilizzando un algoritmo *pseudo-polinomiale*;
- *ottimalità*: si può cercare di individuare un algoritmo di *approssimazione* che non restituisca necessariamente la soluzione ottima, ma che garantisca di ottenerne una non troppo "distante" da essa;
- *efficienza*: si possono progettare algoritmi esponenziali di tipo branch-&-bound, che cercano di evitare di enumerare tutte le soluzioni con un'accurata potatura;
- *formalità*: si possono ricercare algoritmi euristici, di solito basati su tecniche greedy o di ricerca locale, che siano particolarmente facili da programmare, abbiano se possibile complessità polinomiale, e forniscano sperimentalmente risultati buoni – anche se non vi è alcuna dimostrazione formale che questo avvenga sempre.

19.1 ALGORITMI PSEUDO-POLINOMIALI

Si consideri il problema SOMMA DI SOTTOINSIEME, del quale è stata menzionata l'NP-completezza nel capitolo 18:

SOMMA DI SOTTOINSIEME Dati un insieme $A = \{a_1, ..., a_n\}$ di interi positivi ed un intero positivo k, esiste un sottoinsieme S di indici in $\{1, ..., n\}$ tale che $\sum_{i \in S} a_i = k$?

Si supponga di memorizzare l'insieme A in un vettore $A[1 ... n]$ di interi e si introduca un vettore booleano $M[0 ... k]$ di $k + 1$ elementi. Il problema può essere facilmente risolto dalla funzione subsetSum(), basata sulla programmazione dinamica.

```
boolean subsetSum(integer[] A, integer n, integer k)
  boolean[] M ← new boolean[0 ... k]
  for j ← 1 to k do M[j] ← false
  M[0] ← true
  for i ← 1 to n do
  |   for j ← k downto A[i] do
  |   |  M[j] ← M[j – A[i]] or M[j]
  return M[k]
```

Il problema viene risolto via via per $i = 1, 2, ..., n$. All'iterazione i-esima, il valore booleano di $M[j]$ indica se esiste un sottoinsieme di $\{a_1, ..., a_i\}$ la cui somma dà esattamente j. Inizialmente $M[0]$ è posto a **true** perché l'insieme vuoto dà somma 0. All'iterazione i-esima, viene considerata la possibilità di aggiungere a_i. L'assegnamento "$M[j] \leftarrow M[j - A[i]]$ **or** $M[j]$" prevede i due casi possibili in cui a_i sia aggiunto ad un sottoinsieme di $\{a_1, ..., a_{i-1}\}$ che dà somma $j - a_i$, oppure non sia aggiunto ad un sottoinsieme di $\{a_1, ..., a_{i-1}\}$ che dà somma j. Nel primo caso, $M[j]$ è uguale ad $M[j - A[i]]$, mentre nel secondo caso $M[j]$ mantiene il suo vecchio valore. Effettuando l'operazione **or**, $M[j]$ indica correttamente se c'è un sottoinsieme di $\{a_1, ..., a_i\}$ con somma j. Si noti che il ciclo **for** più interno procede per valori decrescenti di j, onde evitare di considerare più occorrenze di a_i nel medesimo sottoinsieme, come invece potrebbe accadere procedendo per valori crescenti di j. La funzione restituisce il valore **true** se e solo se $M[k]$ = **true**, ovvero se esiste un sottoinsieme di $\{a_1, ..., a_n\}$ la cui somma dà k.

10.1 ESEMPIO – [Somma di sottoinsieme]
Siano $n = 3$, $k = 24$, $a_1 = 7$, $a_2 = 11$, $a_3 = 13$. La seguente tabella illustra il contenuto del vettore M per $i = 0, 1, 2, 3$:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	1																								
$i = 1$	1							1																	
$i = 2$	1							1				1							1						
$i = 3$	1							1				1		1					1		1				1

La complessità della funzione subsetSum() è ovviamente $O(nk)$, che non è polinomiale nella dimensione del problema. Infatti, assumendo $a_i \leq k$ per $1 \leq i \leq n$, poiché altrimenti a_i può essere eliminato, la dimensione d dei dati di ingresso è $O(n \log k)$, ma non $O(nk)$. Se k è limitata da un polinomio in n, cioè k è $O(n^c)$ con c costante, allora la funzione subsetSum() ha complessità polinomiale, poiché nk risulta essere $O(n^{c+1})$. Se invece k

è superpolinomiale in n, allora la complessità della funzione risulta essere anch'essa superpolinomiale. Infatti, se k è $O(2^n)$, allora nk è $O(n2^n)$.

In altri termini, la complessità di subsetSum() non dipende soltanto dalla cardinalità n del sottoinsieme in ingresso, ma anche dai valori degli elementi del sottoinsieme. Tale complessità risulta polinomiale oppure no a seconda che i valori degli elementi in ingresso siano "piccoli" oppure "grandi" rispetto al loro numero. Una situazione analoga si è incontrata per la procedura countingSort() [capitolo 2], che permette di ordinare n elementi in tempo $O(n)$ qualora i valori di tali elementi siano "piccoli", cioè limitati anch'essi da $O(n)$, e per il problema dello ZAINO [capitolo 13].

19.1.1
NP-COMPLETEZZA "FORTE"

Ci si potrebbe chiedere sotto quali condizioni un problema NP-completo può essere risolto con un algoritmo polinomiale per valori "piccoli" dei dati in ingresso.

Dato un problema decisionale P, con insieme di dati di ingresso I, distinguiamo tra la "dimensione" d di I, cioè la lunghezza della stringa che codifica I, ed il "più grande numero" intero # che appare in I.

19.2 ESEMPIO – [dimensione d e numero #]

- SOMMA DI SOTTOINSIEME:
  $I = \{n, k, a_1, \ldots, a_n\}$, $\# = \max\{n, k, \max_i\{a_i\}\}$, $d = O(n \log \#)$;
- COMMESSO VIAGGIATORE:
  $I = \{n, k, [d_{ij}]\}$, $\# = \max\{n, k, \max_{ij}\{d_{ij}\}\}$, $d = O(n^2 \log \#)$;
- CRICCA:
  $I = \{n, m, k, G = (V, E)\}$, $\# = \max\{n, m, k\}$, $d = O(n + m + \log \#)$.

Sia P_p il problema P ristretto a quei dati d'ingresso per i quali # è limitato superiormente da $p(d)$, con p funzione polinomiale in d. P è *fortemente NP-completo* se P_p è NP-completo.

19.3 ESEMPIO – [NP-completezza forte]

- La SOMMA DI SOTTOINSIEME non è fortemente NP-completa; infatti, se $\# = n$ il problema è risolubile in tempo polinomiale, mentre se $\# = k$, allora # non può in generale essere limitato superiormente da alcun polinomio nella dimensione $d = O(n \log k)$ (ovviamente se $\# = a_i > k$ allora a_i non può far parte di alcuna soluzione e può essere eliminato);
- Il COMMESSO VIAGGIATORE è fortemente NP-completo, perché resta NP-completo anche quando $d_{ij} \in \{1, 2\}$, $1 \le i, j \le n$, e $k = n$ [si veda la dimostrazione del Teorema 18.6]; in tal caso, $\# = n$, che è ovviamente limitato superiormente da un polinomio nella dimensione $d = O(n^2 \log n)$;
- Anche la CRICCA è fortemente NP-completa, poiché k può essere assunto minore o uguale ad n (se $k > n$, la risposta del problema è sempre NO) e quindi # è limitato superiormente da un polinomio in $d = O(n + m + \log(m + n))$.

In pratica, sono "fortemente" NP-completi tutti quei problemi che possono essere dimostrati NP-completi con riduzioni che non usano numeri interi esponenzialmente grandi, mentre sono "debolmente" NP-completi quei problemi la cui dimostrazione di intrattabilità dipende pesantemente da riduzioni che usano numeri interi esponenzialmente grandi. Per esempio, le riduzioni illustrate per il DOMINO LIMITATO, la CRICCA, la SODDISFATTIBILI-

TÀ, la PROGRAMMAZIONE LINEARE 0/1, l'INSIEME INDIPENDENTE, il COMMESSO VIAGGIATORE richiedono numeri interi che sono polinomialmente limitati dalla dimensione d dell'input (in alcuni casi sono addirittura limitati da costanti). Pertanto, tutti questi problemi sono fortemente NP-completi. Viceversa, ciò non è vero per la SOMMA DI SOTTOINSIEME, la PARTIZIONE e lo ZAINO, che sono invece debolmente NP-completi.

Formalmente, un algoritmo che risolve un certo problema P, per qualsiasi dato I d'ingresso, in tempo $p(\#, d)$, con p funzione polinomiale in # e d, ha complessità *pseudo-polinomiale*. L'algoritmo visto per risolvere il problema SOMMA DI SOTTOINSIEME ha complessità pseudo-polinomiale. Vale il seguente risultato:

TEOREMA 1
Nessun problema fortemente NP-completo può essere risolto da un algoritmo pseudo-polinomiale, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

DIMOSTRAZIONE Se P è fortemente NP-completo, allora il problema ristretto P_p è NP-completo per qualche polinomio p. Se esistesse un algoritmo pseudo-polinomiale A per P, allora A risolverebbe P_p in tempo polinomiale, che è assurdo, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

Il più famoso problema fortemente NP-completo, in cui numeri interi hanno un ruolo rilevante, è la cosiddetta 3-PARTIZIONE, che generalizza appunto il problema della PARTIZIONE:

3-PARTIZIONE (3-PARTITION) Dati $3n$ interi $\{a_1, ..., a_{3n}\}$, esiste una partizione in n triple $T_1, ..., T_n$, tale che la somma dei tre elementi di ogni T_j è la stessa, per $1 \leq j \leq n$?

Al contrario della PARTIZIONE, della SOMMA DI SOTTOINSIEME e dello ZAINO, che sono tutti risolubili in tempo pseudo-polinomiale, la 3-PARTIZIONE non lo è.

Per dimostrare l'NP-completezza forte di un problema P, non è sempre necessario dimostrare l'NP-completezza di un sottoproblema P_p. Infatti, è sufficiente provare l'esistenza di un altro problema Q, di cui si conosce già l'NP-completezza forte, che si può ridurre a P con una trasformazione che richiede tempo pseudo-polinomiale. Tale trasformazione deve inoltre garantire che $\#_P$, il più grande intero introdotto nei dati di P, sia limitato superiormente da una funzione $p(d_Q, \#_Q)$ polinomiale nella dimensione d_Q dell'input di Q e nel più grande intero $\#_Q$ che appare in tale input.

19.2
ALGORITMI DI APPROSSIMAZIONE

Molti problemi di ottimizzazione di rilevante interesse pratico sono NP-ardui, poiché gli stessi problemi in forma decisionale sono NP-completi. Per tali problemi, c'è poca speranza di progettare un algoritmo che trovi la soluzione ottima in tempo polinomiale. In molti casi, però, il fatto di non poter ottenere la soluzione ottima non è poi così

drammatico, e ci si può accontentare di algoritmi che, in tempo polinomiale, trovino una soluzione ammissibile del problema che sia abbastanza "vicina" a quella ottima. Se è possibile dimostrare che la soluzione ammissibile fornita dall'algoritmo si discosta, nel caso pessimo o in quello medio, dalla soluzione ottima per uno scarto (o errore) relativo limitato per tutti i possibili dati di ingresso del problema, allora si ha un algoritmo di "approssimazione".

In questa sezione, vedremo algoritmi di approssimazione assoluta per i problemi BIN PACKING e COMMESSO VIAGGIATORE CON DISUGUALIANZE TRIANGOLARI, dove lo scarto relativo è limitato da una costante che dipende solo dall'algoritmo. Vedremo inoltre un algoritmo di ε-approssimazione per la SOMMA DI SOTTOINSIEME, dove lo scarto relativo ε viene fornito in ingresso all'algoritmo stesso e può essere reso piccolo a piacere, al prezzo di un sempre maggiore sforzo computazionale. Vedremo infine come non tutti i problemi NP-ardui ammettano algoritmi di approssimazione.

19.2.1
APPROSSIMAZIONE ASSOLUTA

Dato un problema di ottimizzazione P, con funzione costo non negativa c, un *algoritmo di approssimazione assoluta* fornisce una soluzione ammissibile x' il cui costo si discosta da quello della soluzione ottima x^* per uno scarto relativo garantito e limitato da una costante. Formalmente, occorre che

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq e$$

per una qualche costante e non negativa che dipende dall'algoritmo e per tutti i possibili dati d'ingresso. In altri termini, l'algoritmo garantisce che la percentuale di scostamento dalla soluzione ottima non superi una certa costante. Tale scostamento vale per la medesima costante e per tutti gli input! Ad esempio, se $e = 1$, allora lo scostamento dall'ottimo non supera il 100%, cioè $c(x')$ è al più il doppio di $c(x^*)$, qualora il problema sia di minimizzazione, oppure almeno la metà, nel caso di massimizzazione.

Per iniziare, consideriamo un problema NP-arduo molto noto:

BIN PACKING Dato un insieme $A = \{a_1, ..., a_n\}$ di interi positivi (i "volumi" di n "oggetti") ed un intero positivo k (la "capacità" di una "scatoletta"), si vuole trovare una partizione di $\{1, ..., n\}$ nel minimo numero di sottoinsiemi disgiunti (le "scatolette") tali che $\sum_{i \in S} a_i \leq k$ per ogni insieme S della partizione.

Si consideri l'algoritmo FF ("First-Fit") in cui gli oggetti sono considerati in un ordine qualsiasi e ciascun oggetto è assegnato alla prima scatoletta che lo può contenere, e dimostriamo che FF è un algoritmo di approssimazione assoluta.

Assumiamo che sia $a_i \leq k$ per ciascun i. Se il numero N di scatolette usate da FF è maggiore di uno, allora tale numero deve essere minore di $\left\lceil \frac{2}{k} \sum_{i=1}^{n} a_i \right\rceil$. Questa proprietà deriva dal fatto che non ci possono essere due scatolette riempite al più per metà. Infatti se ci fossero allora il primo oggetto inserito nella scatoletta di indice maggiore avrebbe po-

tuto essere inserito nella scatoletta di indice minore e l'algoritmo FF non l'avrebbe inserito in un'altra scatoletta. D'altro canto, il minimo numero di scatolette N^* non può essere inferiore a $\left\lceil \frac{1}{k}\sum_{i=1}^{n} a_i \right\rceil$. Pertanto, $N < 2N^*$ ed FF non utilizza più del doppio del numero ottimo di scatolette. È interessante rimarcare che è possibile dimostrare per un'altra via, molto più complicata, un risultato migliore per FF, e cioè che $N < \frac{17}{10}N^* + 2$. Un'approssimazione ancora migliore si ottiene con una variante di FF, detta FFD ("First-Fit-Decreasing") dove gli oggetti sono considerati secondo l'ordine non crescente $a_1 \geq \cdots \geq a_n$. Per FFD è stato dimostrato che $N < \frac{11}{9}N^* + 4$.

Vediamo ora un esempio più complesso, ovvero un algoritmo di approssimazione assoluta per una variante semplificata del problema del COMMESSO VIAGGIATORE:

COMMESSO VIAGGIATORE CON DISUGUAGLIANZE TRIANGOLARI (ΔTSP) Date n città e le distanze d_{ij} tra esse, tali che $d_{ij} \leq d_{ik} + d_{kj}$ per $1 \leq i, j, k \leq n$, trovare un percorso che, partendo da una qualsiasi città, attraversi ogni città esattamente una volta e ritorni alla città di partenza, in modo che la distanza complessiva percorsa sia minima.

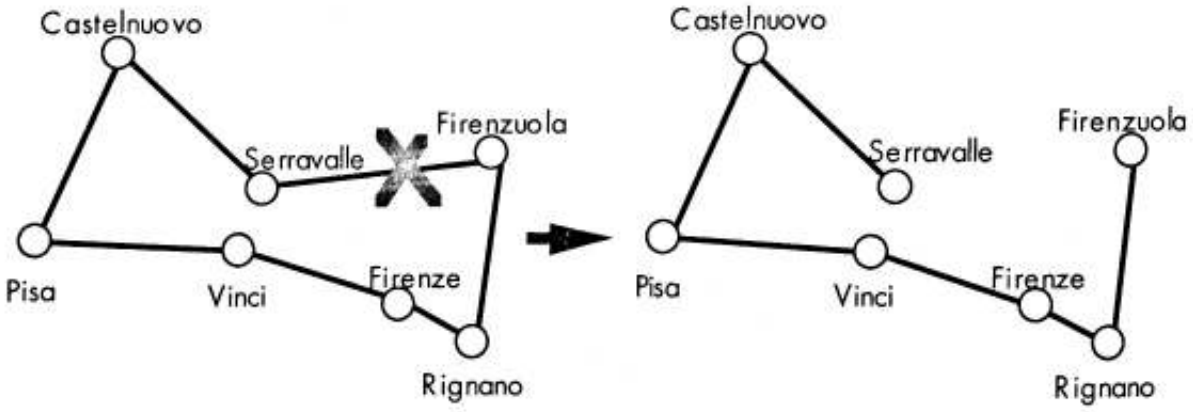


Fig. 19.1 Cancellando un arco da un circuito Hamiltoniano si ottiene una catena Hamiltoniana che è anche un albero di copertura.

La versione decisionale di ΔTSP è NP-completa. Infatti, nella prova di NP-completezza dell'usuale COMMESSO VIAGGIATORE (Teorema 18.6) sono state usate distanze $d_{ij} \in \{1, 2\}$, per $1 \leq i, j \leq n$, che ovviamente verificano la proprietà di triangolarità.

Per semplicità, ΔTSP può essere reinterpretato come il problema di trovare un circuito Hamiltoniano di costo minimo in un grafo K_n di n nodi, non orientato, pesato e completo, cioè formato da una cricca di n nodi.

Come mostrato nella Fig. 19.1, un qualsiasi circuito Hamiltoniano di K_n in cui si cancelli un arco è un particolare albero di copertura per esso e quindi un minimo albero di copertura per K_n non può avere un costo superiore alla soluzione ottima di ΔTSP.

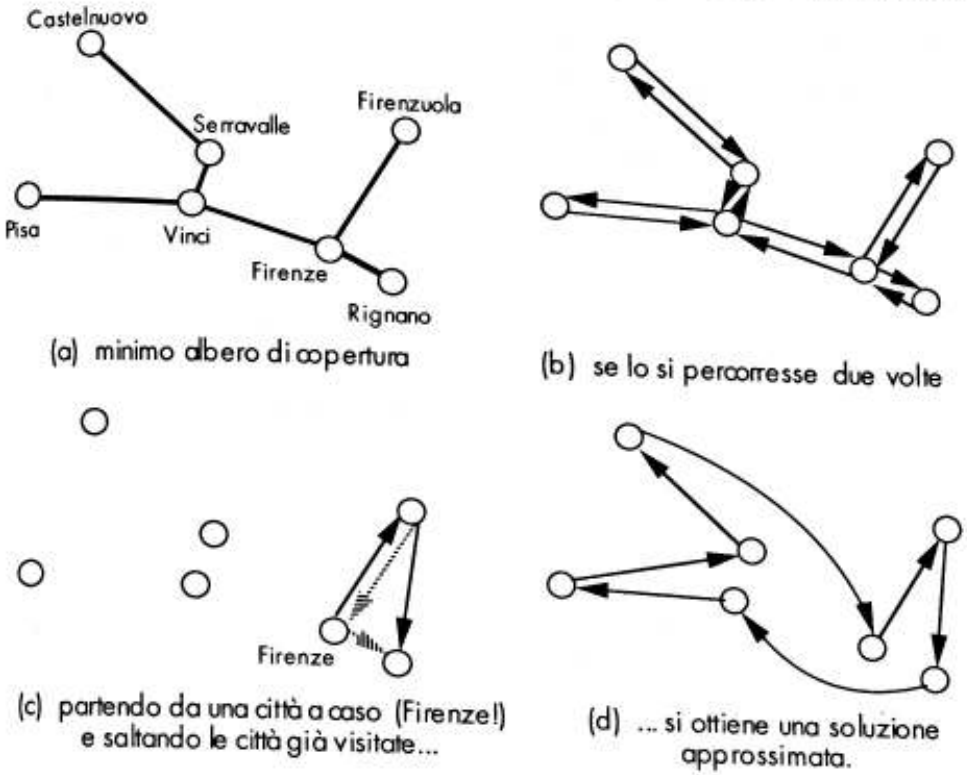


Fig. 19.2 Algoritmo di approssimazione assoluta per ΔTSP.

Se si individua un minimo albero di copertura e se ne percorrono gli archi due volte, la prima volta in un senso e l'altra nel senso opposto, si può visitare ciascuna città almeno una volta e la distanza complessiva percorsa non supera il doppio della distanza ottima di ΔTSP. Visitando le città in modo da "saltare" una città già visitata, si ottiene, per la triangolarità delle distanze, un circuito Hamiltoniano x' il cui costo $c(x')$ non supera il doppio del costo $c(x^*)$ del circuito Hamiltoniano minimo x^* per ΔTSP. Un esempio è fornito nella Fig. 19.2.

Questo algoritmo di approssimazione richiede $O(n^2 \log n)$ tempo, poiché il minimo albero di copertura può essere trovato con l'algoritmo di Kruskal [capitolo 14] e la soluzione x' si può ottenere considerando l'albero ordinato in modo arbitrario e poi visitandone i nodi in ordine anticipato [capitolo 5].

L'algoritmo fornisce una soluzione in cui la distanza complessiva percorsa non supera del 100% quella minima. Infatti, si ha:

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq \frac{|2c(x^*) - c(x^*)|}{|c(x^*)|} = 1.$$

È possibile individuare dati di ingresso particolarmente "perversi" per i quali l'algoritmo produce soluzioni che raggiungono lo scarto pessimo del 100%. Per esempio, si considerino $2n$ città che si trovino sui vertici di due poligoni regolari concentrici, ciascuno di n lati, con distanze pari alle distanze Euclidee tra tali vertici. Un esempio per $n = 6$ è mostrato nella Fig. 19.3(a), dove r ed $r + d$ sono i raggi dei cerchi circoscritti, rispettivamente, al poligono interno e a quello esterno. Il minimo albero di copertura è strutturato come mostrato nel-

la Fig. 19.3(b) e percorrendo tale albero due volte, si ottiene il percorso illustrato nella Fig. 19.3(c). L'algoritmo di approssimazione può trovare, nel caso pessimo, la soluzione x' mostrata nella Fig. 19.3(d), che è assai diversa da quella ottima x^* della Fig. 19.3(e). Se L ed L' sono i lati, rispettivamente, del poligono interno e di quello esterno, allora:

$$c(x') = (n-1)(L+L') + 2d \quad \wedge \quad c(x^*) = n(L+L')/2 + nd.$$

Ricordiamo che, se R è il raggio del cerchio circoscritto ad un poligono regolare di n lati, allora la lunghezza di ogni lato è $2R \operatorname{sen}(\pi/n)$. Assumendo che il poligono interno abbia raggio $r = 1$ ed il poligono esterno abbia raggio $r + d = 1 + 1/n^2$, si ottiene:

$$\begin{aligned} L &= 2\operatorname{sen}(\pi/n), \\ L' &= 2(1+1/n^2)\operatorname{sen}(\pi/n), \\ c(x') &= 2(n-1)(2+1/n^2)\operatorname{sen}(\pi/n) + 2/n^2, \\ c(x^*) &= n(2+1/n^2)\operatorname{sen}(\pi/n) + 1/n. \end{aligned}$$

Essendo:

$$\lim_{n\to\infty} n \operatorname{sen}(\pi/n) = \pi$$

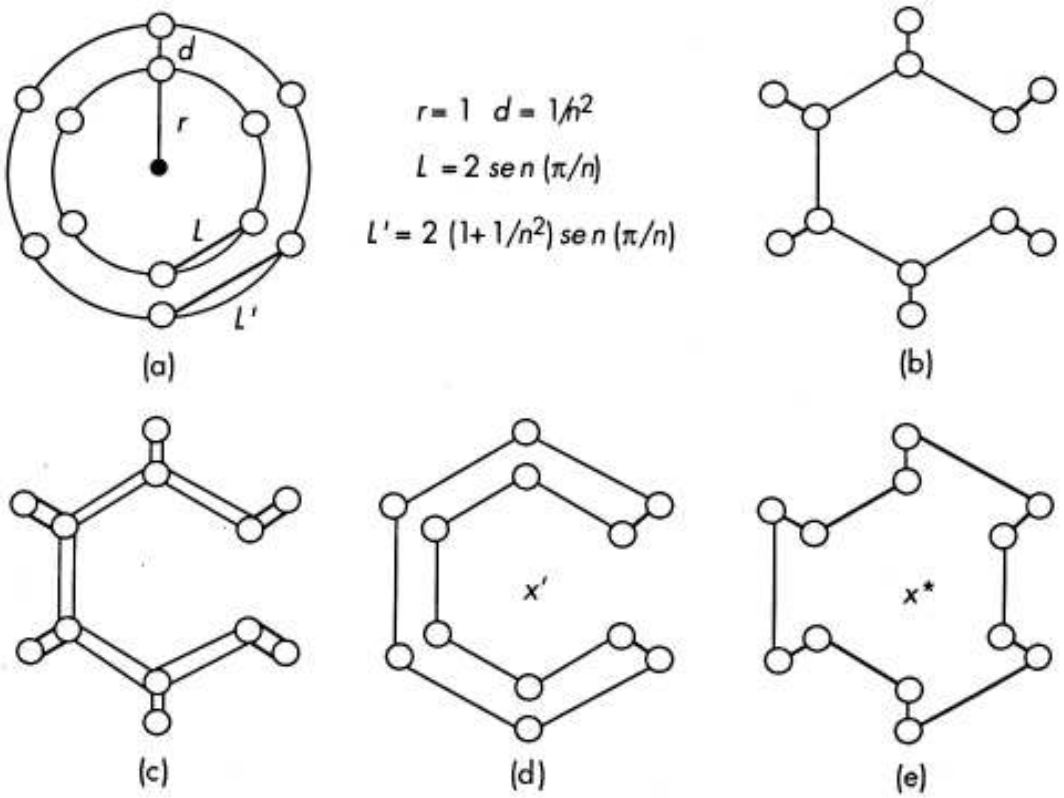


Fig. 19.3 Caso pessimo per Δ_{TSP} (a) Le $2n$ città. (b) Il minimo albero di copertura. (c) L'albero percorso due volte. (d) Soluzione approssimata x', avente costo $c(x') = (n-1)(L+L') + 2d$. (e) Soluzione ottima x^*, avente costo $c(x^*) = n(L+L')/2 + nd$.

si ottengono:

$$\lim_{n\to\infty} c(x') = 4\pi$$
$$\lim_{n\to\infty} c(x^*) = 2\pi$$

e pertanto, al limite, $c(x')$ tende ad essere il doppio di $c(x^*)$.

I particolari dati di ingresso visti precedentemente, mostrano come lo scarto pessimo $|c(x') - c(x^*)| / |c(x^*)| \leq 1$ sia raggiungibile e quindi non migliorabile per il particolare algoritmo di approssimazione esposto. Questo però non deve fare erroneamente credere che tale algoritmo sia il migliore algoritmo di approssimazione possibile per il ΔTSP, poiché si potrebbe dimostrare uno scarto inferiore ad 1 per un altro algoritmo. In effetti, è noto un algoritmo di approssimazione più astuto, dovuto a Christofides (1976), che fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq \frac{3}{2} c(x^*)$, cioè che non si scosta più del 50% da quella minima.

19.2.2

NON APPROSSIMABILITÀ

Purtroppo, per molti problemi di ottimizzazione, trovare una soluzione approssimata è tanto difficile quanto trovarne una ottima! Questo è vero, per esempio, per le versioni di ottimizzazione di CRICCA, COLORAZIONE e COMMESSO VIAGGIATORE (senza diseguaglianze triangolari). A titolo di esempio, vediamo il seguente risultato.

TEOREMA 2

Non esiste alcun algoritmo di approssimazione assoluta per il COMMESSO VIAGGIATORE tale che $c(x') \leq sc(x^*)$, con s intero positivo, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

DIMOSTRAZIONE Per assurdo, si assuma l'esistenza di un algoritmo polinomiale A che fornisca una soluzione x' con $c(x') \leq sc(x^*)$, per un s intero positivo. Si consideri la riduzione CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE utilizzata nel Teorema 18.6 e si ponga:

$$d_{ij} = \begin{cases} 1, & \text{se } [i,j] \in E, \\ sn, & \text{se } [i,j] \notin E. \end{cases}$$

Per costruzione, la soluzione ottima x^* del COMMESSO VIAGGIATORE è tale che $c(x^*) \leq sn$ se e solo se il grafo G ha un circuito Hamiltoniano.

Si possono presentare due casi. Se G ha un circuito Hamiltoniano, allora $c(x^*) = n$, e quindi A fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq sc(x^*) = sn$. Se invece G non ha alcun circuito Hamiltoniano, allora $c(x^*) > sn$, ed A fornisce, in tempo polinomiale, una soluzione x' per cui $c(x') \geq c(x^*) > sn$.

Pertanto, A restituisce una soluzione x' con $c(x') \leq sn$ se e solo se G ha un circuito Hamiltoniano. Questa è chiaramente una contraddizione, perché A risolverebbe in tempo polinomiale un problema NP-completo, il che è impossibile, a meno che $\mathbb{P}$ non coincida con $\mathbb{NP}$.

19.2.3 ε-APPROSSIMAZIONE

In taluni casi, è possibile progettare algoritmi di approssimazione che producono soluzioni approssimate il cui scarto dall'ottimo non è una costante che dipende dall'algoritmo (come nell'approssimazione assoluta), bensì un valore fornito come dato di input che può essere reso arbitrariamente piccolo. In altri termini, comunque si scelga un ε piccolo a piacere, con $0 < \varepsilon \leq 1$, tali algoritmi producono in tempo polinomiale nella dimensione del problema, ma in genere esponenziale in $1/\varepsilon$, una soluzione ammissibile x' tale che:

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq \varepsilon$$

Algoritmi di questo tipo sono detti *algoritmi di ε-approssimazione*. Si consideri ad esempio la versione di ottimizzazione della SOMMA DI SOTTOINSIEME:

SOMMA DI SOTTOINSIEME Dati un insieme $A = \{a_1, ..., a_n\}$ di interi positivi ed un intero positivo k, trovare un sottoinsieme S di $\{1, ..., n\}$ tale che $\sum_{i \in S} a_i \leq k$ e $\sum_{i \in S} a_i$ sia massima.

Sia $\varepsilon = 1/(h + 1)$, con h intero maggiore o uguale ad uno (o, equivalentemente, $h = (1 - \varepsilon)/\varepsilon$), e si supponga di memorizzare l'insieme A in un vettore di interi. Il problema può essere risolto con l'algoritmo ε-approssimato apSubsetSum(), basato sulla tecnica greedy, che genera dapprima nel ciclo (1) tutti i sottoinsiemi S di A di cardinalità al più h, e successivamente considera nel ciclo (2) ciascun elemento a_j di A, $1 \leq j \leq n$, aggiungendolo ad S qualora non sia già presente in S e non venga superata la soglia k.

```
apSubsetSum(integer[] A, integer n, integer k, integer h)
    {ordina A in modo che A[1] ≥ A[2] ≥ ··· ≥ A[n]}
    SET maxSol ← Set()
    integer max ← 0
(1) foreach S : S ⊆ {1, ..., n} and |S| ≤ h do
        integer z ← Σ_{i∈S} A[i]
        if z ≤ k then
(2)         for j ← 1 to n do
                if j ∉ S and z + A[j] ≤ k then
                    S.insert(j)
                    z ← z + A[j]
            if z ≥ max then
                max ← z
                maxSol ← S
```

Poiché, fissato un $m > 0$, ci sono $n(n-1) \cdots (n-m+1)/m! \leq n^m$ sottoinsiemi di $\{1, ..., n\}$ di cardinalità m, il ciclo (1) è ripetuto $\sum_{0 \leq m \leq h} n^m \leq hn^h + 1$ volte. Il ciclo (2), invece, è ripetuto n volte. Pertanto, la complessità dell'algoritmo apSubsetSum() è $O(hn^{h+1})$, che è polinomiale in n, ma esponenziale in h (e quindi in $1/\varepsilon$). Fissato un h, comunque, la complessità diviene polinomiale.

apSubsetSum() fornisce una soluzione $x' = maxSol$ di costo $c(x') = max$ tale che $|c(x') - c(x^*)|/|c(x^*)| \leq \varepsilon$. Per dimostrare questo risultato, sia $x^* = B$ il sottoinsieme ottimo di A con

$$|B| = m, B = \{b_1, b_2, \ldots, b_m\}, b_1 \geq b_2 \cdots \geq b_m, \text{ e } c(x^*) = \sum_{1 \leq i \leq m} b_i.$$

Se $m \leq h$, allora l'insieme B è stato considerato nel ciclo (1) e apSubsetSum() ha fornito la soluzione ottima. Se invece $m > h$, essendo $c(x^*) = \sum_{1 \leq i \leq m} b_i$, si ha che $b_i \leq c(x^*)/i$ per $1 \leq i \leq h$, e quindi $b_i \leq c(x^*)/(h+1)$ per $h < i \leq m$. In tal caso, la procedura apSubsetSum() avrà considerato il sottoinsieme $B' = \{b_1, b_2, \ldots, b_h\}$ nel ciclo (1). Sia

$$b_i = \max\{b \in B : \text{apSubsetSum() non ha aggiunto } b \text{ a } B' \text{ nel ciclo (2)}\}.$$

Se tale b_i non esiste, apSubsetSum() ha trovato la soluzione ottima. Altrimenti,

$$c(x^*) - c(x') \leq b_i \leq \frac{c(x^*)}{h+1}$$

e quindi, dividendo per $c(x^*)$, si ottiene

$$\frac{|c(x^*) - c(x')|}{|c(x^*)|} \leq \frac{1}{h+1} = \varepsilon$$

10.4 ESEMPIO – [Somma di sottoinsieme approssimato]
Siano $n = 8$, $k = 45$, $A = \{23, 19, 13, 12, 11, 8, 7, 5\}$, $h = 1$ (e quindi $\varepsilon = 1/2$). La seguente tabella illustra la computazione di apSubsetSum():

$\|S\|$	insieme costruito in (1)	insieme costruito in (2)	z
0	ø	{23, 19}	42
1	{23}	{23, 19}	42
1	{19}	{23, 19}	42
1	{13}	{23, 13, 8}	44
1	{12}	{23, 12, 8}	43
1	{11}	{23, 11, 8}	42
1	{8}	{23, 13, 8}	44
1	{7}	{23, 13, 7}	43
1	{5}	{23, 13, 5}	41

L'esecuzione di apSubsetSum() dà come risultato $maxSol = x' = \{23, 13, 8\}$ con $max = c(x') = 44$. Si noti che la soluzione ottima $B = x^* = \{19, 13, 8, 5\}$, con $c(x^*) = k = 45$, sarebbe stata trovata dall'algoritmo scegliendo $h = 2$.

Ovviamente, la situazione ideale sarebbe quella di poter sempre progettare un algoritmo A di ε-approssimazione per un problema P che producesse una soluzione ε-approssimata in tempo polinomiale sia nella dimensione del problema sia in $1/\varepsilon$. Purtroppo, tali algoritmi sono noti soltanto per rari problemi di ottimizzazione che coinvolgono numeri, come ad esempio la SOMMA DI SOTTOINSIEME, per i quali esistono già algoritmi che trovano la

soluzione ottima in tempo pseudo-polinomiale. Un forte risultato negativo, a questo riguardo, è dato dal seguente teorema, che diamo senza dimostrazione.

TEOREMA 3
Sia P un problema di ottimizzazione tale che:

(1) la versione decisionale di P è fortemente NP-completa;

(2) per ogni insieme I di dati di input per P il valore $c(x^*)$ della soluzione ottima è limitato superiormente da $p(\#)$, dove p è un polinomio e # è il più grande numero che appare in I,

allora, a meno che $\mathbb{P}$ non coincida con $\mathbb{NP}$, non esiste alcun algoritmo di ε-approssimazione per P che richieda tempo polinomiale sia nella dimensione del problema che in $1/\varepsilon$.

19.3 ALGORITMI BRANCH-&-BOUND

Per risolvere un problema di ottimizzazione NP-arduo, si può modificare la procedura enumerazione(), vista nel capitolo 16, in modo da evitare di effettuare certe sequenze di scelte che si rivelino incapaci di generare la soluzione ottima.

Si assuma che il problema da risolvere sia di minimo, che ogni sequenza di scelte abbia costo non negativo, e che ogni scelta, aggiunta alle scelte già effettuate, non faccia diminuire il costo della soluzione parziale così costruita. Si supponga di avere a disposizione una opportuna funzione "lower bound" lb(S, i), che dipenda dalla sequenza di scelte fatte $S[1, ..., i]$ e garantisca che tutte le soluzioni ammissibili generabili aggiungendo nuove scelte a tale sequenza abbiano costo maggiore o uguale a lb(S, i). Allora, è possibile mantenersi la migliore soluzione ammissibile *minSol* tra quelle finora esaminate, insieme al suo costo *minCost*, e valutare lb(S, i) ogniqualvolta è effettuata una nuova scelta $S[i]$ in un nodo dell'albero delle scelte. Se lb(S, i) è maggiore o uguale a *minCost*, allora si può evitare di generare ed esplorare il sottoalbero delle scelte radicato in tal nodo (si veda la Fig. 19.4).

Questo metodo, detto branch-&-bound, non migliora la complessità della procedura enumerazione(), che resta superpolinomiale nel caso pessimo, ma ne abbassa drasticamente il tempo di esecuzione in pratica. Ciò vale in special modo se si riesce ad individuare una "buona" funzione lb, che sia cioè il più vicino possibile al costo della soluzione ottima.

Supponiamo per semplicità che il numero di scelte per individuare una soluzione sia limitato da n, e che al passo i-esimo sia possibile calcolare l'insieme di possibili scelte C a partire dalle scelte passate $S[1, ..., i-1]$ tramite la funzione choices(). Sia $c(S, i)$ il costo della sequenza di scelte $S[1, ..., i]$. Inoltre, siano *minSol* e *minCost* parametri globali inizializzati a $\emptyset$ e $+\infty$. Allora, la procedura branch&bound() descrive una possibile struttura generale, da chiamarsi con parametro $i = 1$.

Ovviamente, se $|C| \leq m$, per tutti i passi $1 \leq i \leq n$, ed il calcolo di lb() richiede $O(f(n))$ tempo, allora la complessità di branch&bound() è $O(m^n f(n))$.

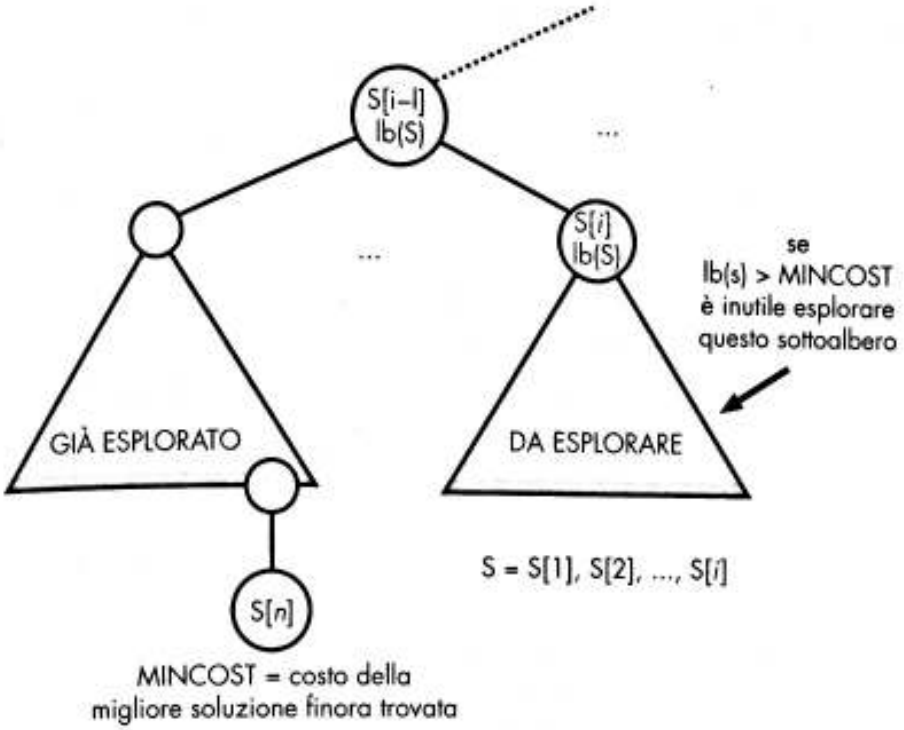


Fig. 19.4 Esplorazione "intelligente" dell'albero delle scelte.

```
branch&bound(ITEM[] S, integer n, integer i, ...)
SET C ← choices(S, n, i, ...)          % Determina C in funzione di S[1, ..., i – 1]
foreach c ∈ C do
    S[i] ← c
    integer lb ← lb(S, i)
    if lb < minCost then
        if i < n then
            branch&bound(S, n, i + 1, ...)
        else
            if c(S, i) < minCost then
                minSol ← S
                minCost ← c(S, i)
```

In pratica, il tempo di computazione di una procedura branch-&-bound è influenzato da svariati fattori:

- *Soluzione di partenza*. Anziché partire con $minCost = +\infty$, è in genere più conveniente determinare il costo di una soluzione ammissibile del problema ed inizializzare *minCost* al valore di tale costo;
- *Branch*. Prestazioni migliori si possono ottenere variando la cosiddetta regola di branch, ovvero il criterio per determinare le scelte da effettuare e generare i figli di un nodo nell'albero delle scelte;
- *Bound*. La "migliore" regola per calcolare lb() è quella che dà il lower bound più stretto, cioè più vicino possibile al costo della soluzione ottima, ma che sia nel contempo calcolabile velocemente;

- *Ordine di visita.* L'ordine più usato per generare e visitare l'albero delle scelte è senza dubbio quello anticipato (LIFO) precedentemente esposto, perché richiede meno spazio; infatti, utilizzando una pila, il massimo numero di "record di attivazione" è proporzionale alla lunghezza di un percorso radice-foglia, che è $O(n)$; talvolta, sono usati criteri diversi, in cui l'albero è visitato per livelli (FIFO) o utilizzando una priorità (p.e. considerando prima quel figlio avente il più promettente lower bound);
- *Scarto dall'ottimo.* Anziché raggiungere la soluzione ottima a tutti i costi, l'esecuzione può essere interrotta quando lo scarto dall'ottimo scenda sotto una certa soglia prestabilita, cioè quando $(minCost - \mathsf{lb}^*)/\mathsf{lb}^* \leq e$, con e costante prefissata ed lb^* uguale al più piccolo lower bound calcolato per i nodi generati ma non ancora visitati dell'albero delle scelte.

Come esempio di tecnica branch-&-bound, risolviamo il problema del COMMESSO VIAGGIATORE (senza diseguaglianze triangolari), per il quale sappiamo che non può esistere né un algoritmo ottimo pseudo-polinomiale né un algoritmo di approssimazione assoluta polinomiale (a meno che $\mathbb{P} = \mathbb{NP}$). Cerchiamo dunque di specializzare la procedura branch&bound() vista precedentemente.

19.3.1
COMMESSO VIAGGIATORE

Sia n il numero di città e $d[h, k]$ la distanza, intera non negativa, tra le città h e k, $1 \leq h, k \leq n$. Si assuma che al passo i-esimo siano state fatte le scelte $S[1, ..., i]$ prese dall'insieme $\{1, ..., n\}$. Ovviamente, un qualsiasi percorso ammissibile del commesso viaggiatore generabile "espandendo" $S[1, ..., i]$ deve attraversare le città $S[1]$, $S[2]$, ..., $S[i]$ in quest'ordine, passare da $S[i]$ ad una qualsiasi delle rimanenti $n - i$ città, attraversare queste ultime città in un ordine qualsiasi, e da una di queste ritornare ad $S[1]$. Si pongano:

$$
\begin{aligned}
A &= \min_h \{d[S[1], h] : h \notin S\}, \\
B &= \min_h \{d[S[i], h] : h \notin S\}, \\
C[i] &= C[i-1] + d[S[i-1], S[i]], \\
C[1] &= 0, \\
D[h] &= \min_{p,q} \{d[h, p] + d[h, q] : h \neq p \neq q\}, \text{ per ogni } h \notin S.
\end{aligned}
$$

Allora, un lower bound $\mathsf{lb}(S)$ è dato dalla seguente espressione:

$$
\begin{aligned}
\mathsf{lb}(S) &= C[i] + \left\lceil (A + B + \sum_{h \notin S} D[h])/2 \right\rceil, && \text{se } i < n, \\
\mathsf{lb}(S) &= C[i] + d[S[i], S[1]], && \text{se } i = n.
\end{aligned}
$$

Infatti, $C[i]$ è la distanza complessiva per attraversare, nell'ordine, le città $S[1]$, $S[2]$, ..., $S[i]$. Invece, A è un lower bound della distanza tra una qualsiasi delle rimanenti $n - i$ città

ed $S[1]$, mentre B è un lower bound della distanza tra $S[i]$ ad una qualsiasi di tali $n - i$ città. Infine, $D[h]$ è un lower bound della distanza percorsa per attraversare una qualsiasi di queste ultime $n - i$ città, provenendo da (e dirigendosi verso) un'altra di queste $n - i$ città.

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

distanze $d[h, k]$

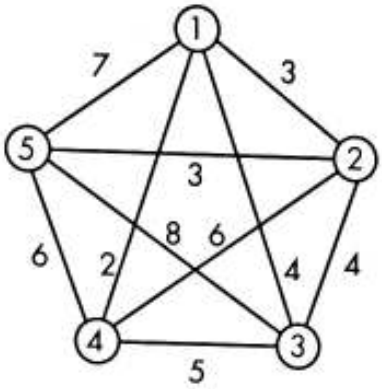


Fig. 19.5 Dati di ingresso per il COMMESSO VIAGGIATORE.

19.5 ESEMPIO – [Lower bound per il COMMESSO VIAGGIATORE]
Si consideri la matrice delle distanze fornita nella Fig. 19.5 per $n = 5$ città. Siano $i = 2$, $S[1] = 1$ e $S[2] = 3$. Allora

$A = \min\{d[1, 2], d[1, 4], d[1, 5]\} = \min\{3, 2, 7\} = 2$
$B = \min\{d[3, 2], d[3, 4], d[3, 5]\} = \min\{4, 5, 8\} = 4$
$C[2] = C[1] + d[1, 3] = 4$
$D[2] = 3 + 3 = 6$
$D[4] = 2 + 5 = 7$
$D[5] = 3 + 6 = 9$
$\text{lb}(S) = 4 + \lceil(2+4+6+7+9)/2\rceil = 18.$

Per non generare più volte lo stesso percorso a partire da tutte le città che lo compongono, la prima scelta può essere fatta una volta per tutte nel programma chiamante. Assumendo per esempio $S[1] = 1$, la procedura ricorsiva bbTsp() può essere richiamata direttamente con parametro $i = 2$, dopo aver inizializzato $C[1]$ ed lb a zero e R a $\{2, 3, ..., n\}$.

```
bbTsp(ITEM[] S, integer[] C, SET R, integer n, integer i)
  foreach c ∈ R do
      S[i] ← c
      R.remove(c)
      C[i] ← C[i − 1] + d[S[i − 1], S[i]]
      {calcola A, B, e D[h] per ogni h ∈ R}
      integer lb ← C[i] + iif(i < n, ⌈(A + B + Σ_{h∉S} D[h])/2⌉, d[S[i], S[1]])
      if lb < minCost then
          if i < n then
              bbTsp(S, C, R, n, i + 1, ...)
          else
              C[n] ← lb
              minSol ← S
              minCost ← C[n]
      R.insert(c)
```

Per risparmiare tempo, anziché partire con $minCost = +\infty$, si può partire inizializzando *minCost* al valore di una qualsiasi soluzione ammissibile, cioè di una permutazione di $\{1, ..., n\}$. Una tale permutazione può essere generata casualmente in tempo $O(n)$ avvalendosi della funzione random() introdotta nel capitolo 17. Infatti, si consideri un vettore $V[1 ... n]$ inizializzato in modo che $V[i] = i$ per $i = 1, 2, ..., n$. Allora, una permutazione delle n città può essere posta in un secondo vettore $P[1 ... n]$ come segue:

```
for i ← 1 to n do
    integer r ← random(i, n)
    P[i] ← V[r]
    V[r] ← V[i]
```

19.6 ESEMPIO – [bbTsp()]

Sia 4, 3, 2, 5, 1 la permutazione casuale generata per $n = 5$ città. Questa corrisponde al circuito 1, 4, 3, 2, 5, 1 che, per i dati della Fig. 19.5, ha costo 2 + 5 + 4 + 3 + 7 = 21. Si assuma pertanto di aver inizializzato *minCost* a 21. Allora, applicando la procedura bbTsp() ai dati della Fig. 19.5, si genera l'albero delle scelte mostrato nella Fig. 19.6. Il confronto "lb < *minCost*" permette di "potare" ben 42 nodi sui 65 possibili, con un notevole risparmio del tempo totale di esecuzione.

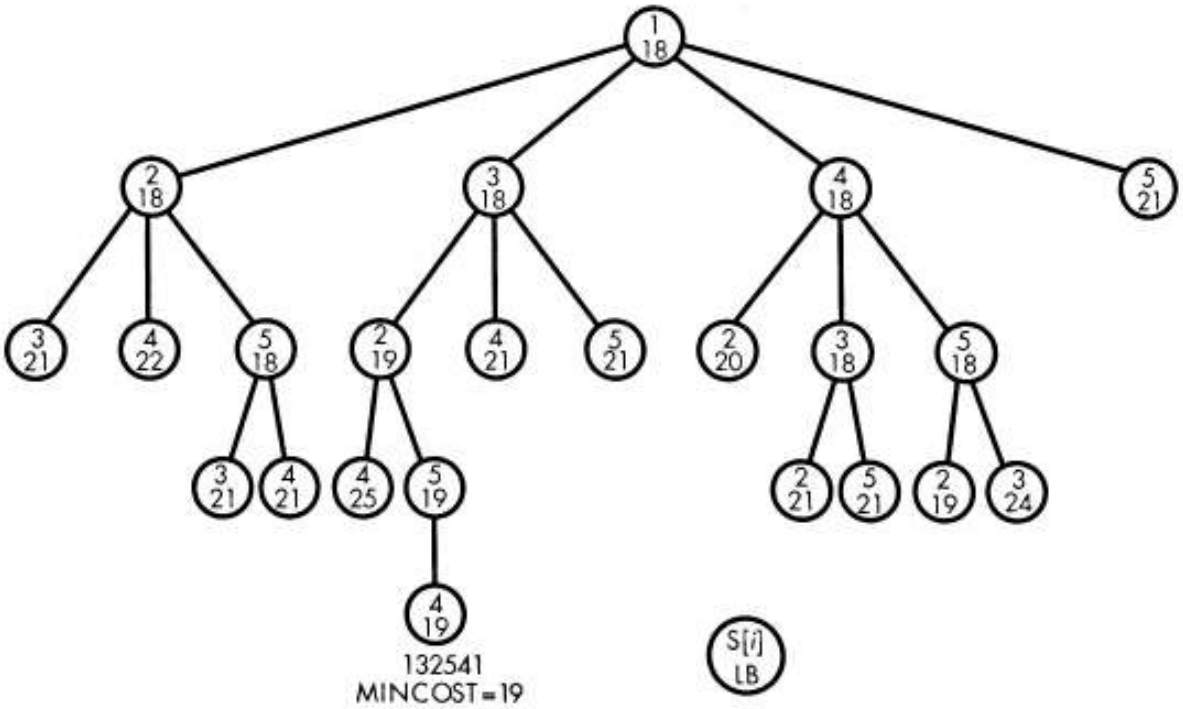


Fig. 19.6 Albero delle scelte generato dalla procedura bbTsp() per i dati della FIG. 19.5 con *minCost* inizializzato a 21.

È bene rimarcare come la procedura bbTsp() appena descritta non sia che un semplice esempio (e sicuramente non il migliore) di impiego della tecnica branch-&-bound per il COMMESSO VIAGGIATORE. In effetti, il branch-&-bound è una tecnica così generale che si possono progettare molte procedure diverse per lo stesso problema variandone le regole

di branch, le funzioni lower bound, e l'ordine di visita dell'albero delle scelte. Per esempio, ciascun nodo dell'albero può riguardare la scelta di un arco (anziché un nodo) del grafo K_n e avere così due soli figli, corrispondenti all'inclusione oppure all'esclusione dell'arco, mentre può essere considerato l'arco la cui esclusione massimizza il lower bound. Una funzione alternativa per il lower bound può essere ottenuta con un *1-albero*, dove un 1-albero per K_n è formato da un albero di copertura per i nodi 2, 3, ..., n più due archi incidenti nel nodo 1. È immediato osservare che ogni ciclo Hamiltoniano per K_n è un 1-albero, ma non viceversa, per cui il peso di un 1-albero minimo è un lower bound al costo della soluzione ottima (un 1-albero minimo si può facilmente trovare in tempo polinomiale). In pratica, uno dei migliori lower bound per il COMMESSO VIAGGIATORE è stato ottenuto da Held e Karp (1971) modificando ulteriormente la nozione di 1-albero. Infine, i nodi dell'albero delle scelte possono essere considerati per priorità, per esempio considerando per primo il nodo col lower bound più piccolo.

19.4 ALGORITMI EURISTICI

Quando si è presi dalla disperazione a causa della enorme difficoltà di un problema di ottimizzazione NP-arduo, si può ricorrere ad algoritmi "euristici" che forniscono una soluzione ammissibile, non necessariamente ottima né approssimata. Tali euristiche sono di solito basate su tecniche greedy [capitolo 14] oppure di ricerca locale [capitolo 15].

19.4.1 GREEDY

Iniziamo progettando un algoritmo euristico di tipo greedy per il problema del COMMESSO VIAGGIATORE. Consideriamo l'input come grafo non orientato completo e pesato. Ordiniamone gli archi per pesi non decrescenti e proviamo ad aggiungere archi alla soluzione seguendo questo ordine finché non sono stati aggiunti $n - 1$ archi, dove n è il numero di nodi. Per ogni nodo, contiamo quanti archi incidenti in quel nodo sono stati aggiunti alla soluzione. Per poter aggiungere un arco, occorre verificare che per ciascuno dei suoi nodi non siano stati già scelti due archi e che non si formino circuiti. Per verificare quest'ultima proprietà, si utilizza un MFSET, terminando di aggiungere archi quando l'MFSET è formato da una sola componente. A questo punto, si è trovata una catena Hamiltoniana e si chiude il circuito aggiungendo l'arco tra i due nodi estremi della catena, ciascuno dei quali ha un solo arco incidente. Questa euristica richiede tempo $O(n^2 \log n)$ e si può riassumere nella procedura greedyTsp().

19.7 ESEMPIO – [Commesso viaggiatore "ingordo"]
Applicando l'algoritmo euristico greedyTsp() ai dati di ingresso della Fig. 19.6, si inseriscono in S, nell'ordine, gli archi [1,4], [1,2], [2,5], [3,4], [3,5] che producono il circuito 1, 4, 3, 5, 2, 1 il cui costo è 21. Tale soluzione può essere utilizzata come soluzione di partenza per poi applicare o un algoritmo branch-&-bound come bbTsp() (dopo aver inizializzato *minCost* a 21) o un'ulteriore euristica di ricerca locale [sottoparagrafo 19.4.2], che di solito per il COMMESSO VIAGGIATORE si comporta meglio di greedy.

```
SET greedyTsp(GRAPH G)
  SET S ← Set()
  MFSET M ← Mfset(G.n)
  integer[] in ← new integer[1 ... n]
  for[] i ← 1 to G.n do in [i] ← 0
  {ordina gli archi per peso non decrescente}
  foreach [u, v] ∈ G.E do
      if in[u] < 2 and in[v] < 2 and M.find(u) ≠ M.find(v) then
          S.insert([u, v])
          in[u] ← in[u] + 1
          in[v] ← in[v] + 1
          M.merge(u, v)
  integer u ← 1; while in[u] ≠ 1 do u ← u + 1
  integer v ← u + 1; while in[v] ≠ 1 do v ← v + 1
  S.insert([u, v])
  return S
```

Come secondo esempio, consideriamo ora il problema della BISEZIONE [capitolo 18] in forma di ottimizzazione:

BISEZIONE Dato un grafo non orientato $G = (V, E)$ di n nodi, trovare una partizione di V in due sottoinsiemi V_0 ed V_1 di $n/2$ nodi ciascuno tale che il numero di archi con un estremo in V_0 e l'altro estremo in V_1 sia minimo.

Dato G con n pari, una semplice euristica greedy può essere basata sulle considerazioni seguenti. Si inizia selezionando casualmente un nodo che è posto in V_0 ed un altro nodo che è posto in V_1. Si prosegue aggiungendo alternatamente un nodo a V_0 e il nodo successivo a V_1, fino all'esaurimento dei nodi. Come mostrato nella Figura 19.7, il nodo u da aggiungere in ciascun passo a V_k è scelto in modo da minimizzare il numero di archi $[u, v]$ con $v \in V_{1-k}$. Nel caso ci siano più nodi che soddisfano tale criterio, però, si sceglie tra questi il nodo u che massimizza il numero di archi $[u, v]$ con $v \in V_k$. Se ci sono più nodi che soddisfano anche il secondo criterio, allora il nodo u è scelto casualmente tra questi. Tale euristica, detta *min-max-greedy*, si comporta molto bene, proprio grazie al secondo criterio di scelta. Infatti, gli archi $[u, v]$ con $v \in V_k$ non andranno ad aumentare il costo della partizione, poiché u è aggiunto proprio a V_k. Pertanto, più archi di questo tipo sono aggiunti subito, minore diviene la possibilità che nei passi successivi si sia costretti a scegliere un nodo che, introducendo parecchi archi tra V_k e V_{1-k}, faccia aumentare di molto il costo della partizione.
L'euristica *min-max-greedy* può essere realizzata in modo da avere complessità $O(n^2)$ nel caso pessimo, utilizzando una apposita struttura di dati detta "a cestini" (cfr. Esercizio 19.16). Prove sperimentali hanno mostrato che in pratica *min-max-greedy* fornisce "buone" soluzioni. Anche in questo caso, è opportuno ripetere l'esecuzione dell'euristica un numero prefissato di volte, p.e. 100, scegliendo ogni volta, in modo casuale, una nuova coppia di nodi di partenza da inserire in V_0 ed V_1, e selezionando alla fine la migliore delle 100 soluzioni individuate.

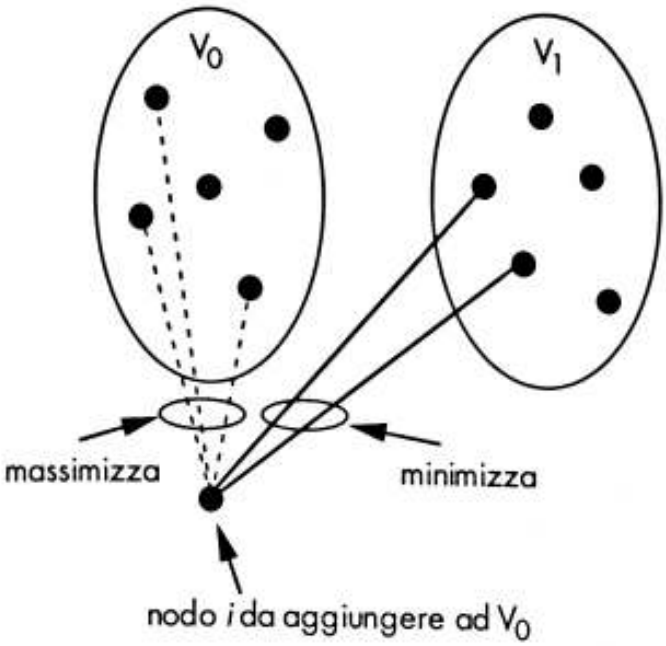


Fig. 19.7 Euristica *min-max-greedy* per la BISEZIONE: situazione prima di aggiungere un nuovo nodo a V_0.

19.4.2
RICERCA LOCALE

Per introdurre le tecniche euristiche basate su ricerca locale, si consideri ancora il problema del COMMESSO VIAGGIATORE e sia π un circuito Hamiltoniano del grafo completo non orientato K_n. Un'euristica ragionevole di ricerca locale può essere progettata sulla base del seguente intorno:

$$I_2(\pi) = \{\pi' : \pi' \text{ è ottenuto da } \pi \text{ cancellando due archi non consecutivi del circuito e sostituendoli con due archi esterni al circuito}\}.$$

Si noti che $|I_2(\pi)| = n(n-1)/2 - n$. Infatti, cancellati due archi non consecutivi, c'è un unico modo per riconnettere i due spezzoni in un circuito con archi che non fanno parte di π, mentre il numero di tutte le possibili coppie di archi non consecutivi di π è dato da quello di tutte le $n(n-1)/2$ coppie meno le n coppie di archi consecutivi. Pertanto, in un'euristica di ricerca locale, il passaggio da una soluzione π ad una $\pi' \in I_2(\pi)$ richiede $O(n^2)$ tempo.

19.8 ESEMPIO – [Ricerca locale per COMMESSO VIAGGIATORE]
Si consideri un'euristica di ricerca locale con intorno I_2 e la si esegua sui dati della Fig. 19.5. Come illustrato nella Fig. 19.8, partendo dal circuito iniziale $\pi = 1, 2, 3, 4, 5, 1$ con costo $c(\pi) = 25$, e scambiando gli archi [1, 5] e [3, 4] con gli archi [1, 4] e [3, 5], si ottiene la nuova soluzione $\pi' = 1, 2, 3, 5, 4, 1$ di costo $c(\pi') = 23$. Scambiando ulteriormente gli archi [1, 2] e [3, 5] con [1, 3] e [2, 5], si ottiene la soluzione ottima $\pi'' = 1, 3, 2, 5, 4, 1$ di costo $c(\pi'') = 19$.

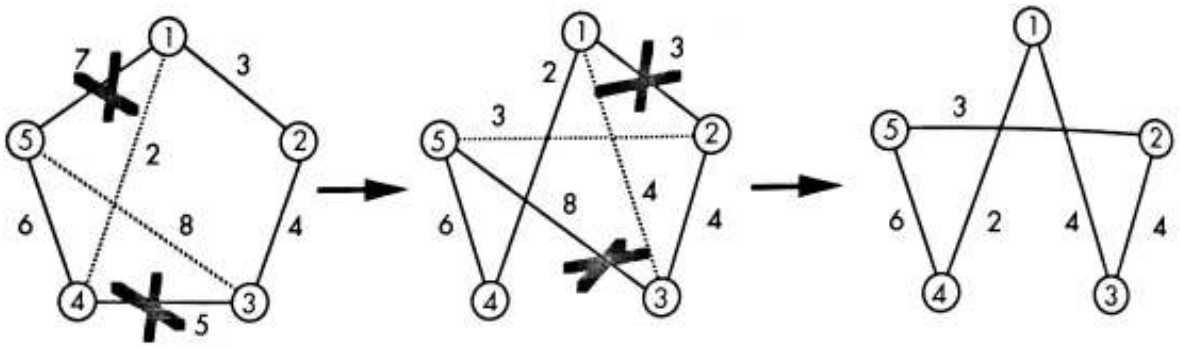


Fig. 19.8 Euristica di ricerca locale con intorno I_2 per il COMMESSO VIAGGIATORE.

Ovviamente, una soluzione iniziale può essere generata casualmente oppure costruita tramite l'euristica greedyTsp(). Purtroppo, il numero complessivo di soluzioni π esaminate dall'euristica di ricerca locale può essere esponenziale. Inoltre, non vi è alcuna garanzia che la soluzione trovata sia ottima. Infatti, come già è stato osservato nel capitolo 15, un algoritmo di ricerca locale si arresta su un ottimo locale, cioè su una soluzione π il cui costo $c(\pi)$ è minore di $c(\pi')$ per ogni $\pi' \in I_2(\pi)$. In generale, niente assicura che tale soluzione sia anche un ottimo globale, poiché può esistere una soluzione $\pi^* \notin I_2(\pi)$ con costo $c(\pi^*)$ minore di $c(\pi)$, per esempio una ottenibile scambiando 3 o 4 archi anziché due soli archi.

Generalizzando, si può definire, per $2 \leq k \leq n$, un intorno $I_k(\pi)$ dato da tutte le soluzioni π' ottenibili da π cancellando k archi del circuito e sostituendoli con k archi esterni al circuito. Purtroppo, è possibile dimostrare che l'unico intorno che garantisce sempre che l'ottimo locale trovato sia anche l'ottimo globale si ottiene solo dall'unione di $I_2(\pi)$, $I_3(\pi)$, ..., $I_n(\pi)$. Comunque, prove sperimentali hanno mostrato che per l'intorno dato dall'unione di $I_2(\pi)$ e $I_3(\pi)$ si ottiene una euristica di ricerca locale che fornisce in pratica "buone" soluzioni. Ovviamente, tale euristica ha complessità esponenziale nel caso pessimo e non vi è alcuna garanzia che la soluzione prodotta sia vicina a quella ottima! In tal caso, è pratica comune ripetere l'esecuzione dell'euristica di ricerca locale un numero prefissato di volte, p.e. 100, scegliendo ogni volta, in modo casuale, una soluzione ammissibile di partenza, e selezionando alla fine la migliore delle 100 soluzioni (ottimi locali) così individuate.

Anche per il problema della BISEZIONE si possono definire euristiche di ricerca locale, che magari iniziano proprio a partire dalla soluzione prodotta da *min-max-greedy* e poi continuano scambiando ripetutamente tra loro un nodo di V_0 con uno di V_1. Purtroppo, mentre per il problema del COMMESSO VIAGGIATORE le euristiche di ricerca locale risultano nettamente superiori a quelle greedy, per la BISEZIONE invece accade esattamente il contrario, e per ottenere soluzioni migliori occorre considerare tecniche euristiche più raffinate della ricerca locale.

La ricerca locale, infatti, tentando sempre di migliorare il costo della soluzione, presenta l'inconveniente di bloccarsi su un minimo locale. Per superare tale inconveniente, sono state introdotte tecniche più sofisticate di ricerca che permettono anche di peggiorare il costo della soluzione, nella speranza di sfuggire da un minimo locale. Tali tecniche devono però "evitare" la possibilità di effettuare troppo presto "mosse" inverse di "miglioramento", per non ricadere subito sul medesimo minimo locale. A tal fine, vengono introdotte delle "proibizioni" delle mosse che devono valere per un certo lasso di tempo, e per questo tali tecniche sono dette di "ricerca tabù".

19.5 REALITY CHECK

Il GIOCO DEL QUINDICI è un rompicapo inventato da Samuel Loyd nel 1878. Il gioco consiste in un griglia quadrata di $n \times n$ posizioni (nella forma classica, $n = 4$), con $n^2 - 1$ tessere quadrate di lato unitario (nel caso di $n = 4$, il numero di tessere è 15, da cui il nome), numerate da 1 a $n^2 - 1$, che coprono tutte le caselle della griglia, tranne una casella che rimane vuota. Una "mossa" consiste nello spostare orizzontalmente o verticalmente una tessera adiacente alla casella vuota, coprendo la casella vuota ma scoprendo la casella dove si trovava la tessera. Lo scopo del gioco è riordinare le tessere in modo crescente dopo averle "mescolate" in modo casuale.

19.10 ESEMPIO – [Gioco del quindici]
La quattro tabelle in Fig. 19.10 corrispondono ad (a) una configurazione iniziale; (b) una mossa a partire da (a), spostando il 7 nella casella vuota; (c) una mossa a partire da (b), spostando l'8 nella casella vuota; (d) la configurazione finale.

È possibile risolvere questo problema attraverso un algoritmo branch-&-bound. Per ogni configurazione delle tessere, possiamo definire il lower bound come la somma delle "distanze" di una tessera dal suo posto, ovvero il numero di caselle attraverso le quali la tessera deve comunque transitare per portarsi al proprio posto. Per esempio, nella configurazione (a) mostrata nella Fig. 19.10 le tessere 4, 7, 8, 9, 12, 13, 14 sono al loro posto, mentre le tessere 2, 3, 1, 5, 6, 10, 11, 15 si trovano in caselle adiacenti ai rispettivi posti di competenza e quindi ognuna di esse contribuisce con distanza 1; pertanto il lower bound per la configurazione (a) è pari a 8. Si noti che per la configurazione (b) il lower bound è 9, mentre per la configurazione (c) il lower bound è 10, quindi non sono certo mosse da seguire.

(a)

2	3		4
1	5	7	8
9	6	10	12
13	14	11	15

(b)

2	3	7	4
1	5		8
9	6	10	12
13	14	11	15

(c)

2	3	7	4
1	5	8	
9	6	10	12
13	14	11	15

(d)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Fig. 19.10 Il gioco del quindici. (a) Una configurazione iniziale; (b) una mossa a partire da (a); (c) una mossa a partire da (b); (d) la configurazione finale.

La radice dell'albero delle scelte corrisponde alla configurazione iniziale (la (a) nel nostro esempio), mentre la regola di branch prevede di generare tutti i figli di un nodo che

corrispondono alle configurazioni ottenibili con una mossa a partire dalla configurazione di tale nodo. Per esempio, il nodo (a) avrà 3 figli: uno è (b), gli altri si ottengono spostando le tessere 4 e 3. Ogni nodo dell'albero delle scelte viene caratterizzato dal suo livello (distanza dalla radice) che ci dà il numero di mosse effettuate (costo) per ottenere la corrispondente configurazione. L'ordine di visita dell'albero delle scelte si svolge per priorità, utilizzando una coda con priorità: si visita per primo quel nodo non ancora visitato per il quale è minima la somma del suo livello e del suo lower bound.

19.1 [Partizione] Si progetti un algoritmo di programmazione dinamica con complessità pseudo-polinomiale per la PARTIZIONE.

19.2 [Scheduling multiprocessore] Fornire un algoritmo di programmazione dinamica con complessità pseudo-polinomiale per il problema dello SCHEDULING MULTIPROCESSORE definito nel capitolo 18.

19.3 [Algoritmo di McNaughton] Si dimostri che SCHEDULING MULTIPROCESSORE in forma di ottimizzazione è risolubile in tempo $O(n)$ se è possibile frazionare l'esecuzione di un programma in più intervalli (aventi lunghezze reali), evitando ovviamente che due intervalli dello stesso programma siano eseguiti nello stesso istante da più di un processore (Suggerimento: si consideri $d = \max\{\max_i \{t_i\}, (1/m) \sum_i t_i\}$).

19.4 [I "soliti ignoti"] La scorsa notte, i "soliti ignoti" hanno svuotato l'oreficeria "Bella Gioia", usando la tecnica del "buco". Dopo il fattaccio, i tre componenti della banda si sono riuniti per spartire il bottino in parti uguali. Il valore del bottino è stimato in m Euro, con m multiplo di 3, sommando i prezzi riportati sui cartellini degli n monili. Dopo lunghe discussioni, il capo della banda vi ha incaricato di scrivere una procedura per effettuare la spartizione usando la tecnica della programmazione dinamica.

19.5 [Minima copertura con nodi] Dato un grafo non orientato G, il problema della MINIMA COPERTURA CON NODI chiede di trovare un sottoinsieme S di nodi avente minima cardinalità tale che ciascun arco abbia almeno un estremo in S. Si progetti un algoritmo di approssimazione assoluta.

19.6 [ΔTSP euclideo] Si assuma che le n città del ΔTSP siano punti nel piano e che le distanze tra le città siano le distanze Euclidee tra i punti. Si dimostri che un percorso ottimo del commesso viaggiatore non incrocia mai se stesso.

19.7 [Zaino, ε-approssimato] Si fornisca un algoritmo ε-approssimato per il problema dello ZAINO in forma di ottimizzazione.

19.8 [Cricca di peso massimo] Si progetti, usando la tecnica del branch-&-bound, un algoritmo per il problema della CRICCA DI PESO MASSIMO: Dato un grafo non orientato $G = (V, E)$ con pesi interi non negativi sui nodi, trovare un sottoinsieme S dei nodi tale che esista un arco tra ogni coppia di nodi in S e la somma dei pesi dei nodi in S sia la più grande possibile.

ESERCIZI

19.9 [1-albero minimo] Si progetti un algoritmo di complessità polinomiale per trovare un 1-albero minimo di un grafo non orientato e pesato. Si riesegua poi l'algoritmo bbTsp(), utilizzando un 1-albero minimo per calcolare il lower bound, sui dati dell'Esempio 19.7.

19.10 [Riduzione matrice delle distanze] Si progetti, usando la tecnica del branch-&-bound, un algoritmo per il COMMESSO VIAGGIATORE utilizzando i seguenti criteri: (1) Scelta: inclusione/esclusione di un arco; (2) Regola di branch: considerare l'arco la cui esclusione massimizza il lower bound; (3) Visita: per priorità (cioè per lower bound più piccolo); (4) Lower bound: metodo di "riduzione" sulla matrice delle distanze (con lb inizializzato a 0):

(1) calcolare $r_i = \min_j\{d_{ij}\}$ per ogni riga i della matrice;

(2) se $r_i > 0$, sottrarre r_i a tutti gli elementi d_{ij} della riga i e sommare r_i ad lb;

(3) calcolare $c_j = \min_j\{d_{ij}\}$ per ogni colonna j della matrice;

(4) se $c_j > 0$, sottrarre c_j a tutti gli elementi d_{ij} della colonna j e sommare c_j ad lb;

Si esegua l'algoritmo sui dati dell'Esempio 19.8.

19.11 [Zaino, B&B] Si progetti un algoritmo branch-&-bound per il problema dello ZAINO in forma di ottimizzazione.

19.12 [Minima superstringa comune] Date n stringhe $P_1, ..., P_n$, il problema della MINIMA SUPERSTRINGA COMUNE chiede di trovare la più corta stringa T che è una superstringa di ciascuna P_i. Dato che questo problema è NP-arduo, si progetti una euristica di tipo greedy modificando quella per il COMMESSO VIAGGIATORE vista in questo capitolo.

19.13 [Ottimo locale per COMMESSO VIAGGIATORE] Si costruisca un grafo K_6 per il COMMESSO VIAGGIATORE per il quale esistano due soluzioni π e π^* tali che π sia un ottimo locale per l'intorno $I_2(\pi)$, mentre π^* sia un ottimo globale che non appartiene ad $I_2(\pi)$.

19.14 [Cardinalità di I_k] Si dimostri che la cardinalità di $I_k(\pi)$, per il COMMESSO VIAGGIATORE è $O(n^k)$. Per $k > 2$, esiste un unico modo di ricomporre il circuito dopo aver tolto k archi?

19.15 [Partizione] Si progetti un algoritmo euristico di ricerca locale per il problema della PARTIZIONE in forma di ottimizzazione, dove si vuole minimizzare lo scarto tra la somma degli elementi della prima parte e la somma di quelli della seconda parte.

19.16 [min-max-greedy] Le operazioni richieste dall'euristica *min-max-greedy* possono essere realizzate usando una struttura di dati detta "a cestini". Vengono utilizzate due matrici *cestino*[0 ... 1, 0 ... *n*, 0 ... *n*] e *dimensione*[0 ... 1, 0 ... *n*]. In generale, *cestino*[*k*, *h*, *d*] contiene un nodo *u* tale che il numero di archi [*u*, *v*] con $v \in V_k$ è uguale ad *h*, mentre *dimensione*[*k*, *h*] contiene il numero di tali nodi. Inoltre, sono mantenuti in *min*[*k*] e *max*[*k*] i valori del minimo e massimo "cestino" non vuoto (si veda la Fig. 19.12, notando come ogni nodo compaia in entrambi gli insiemi). Si considerino le seguenti operazioni:

- insert(*u*), che inserisce il nodo *u* nel cestino;
- remove(*u*), che cancella il nodo *u* dal cestino;
- inc(*k*, *u*), che aumenta di 1 il numero di archi [*u*, *v*] con $v \in V_k$.

Si scrivano procedure efficienti per realizzare tali operazioni, aggiornando opportunamente i valori *min*[*k*] e *max*[*k*]. Si mostri poi come realizzare efficientemente la scelta del nodo *i* da aggiungere alla partizione.

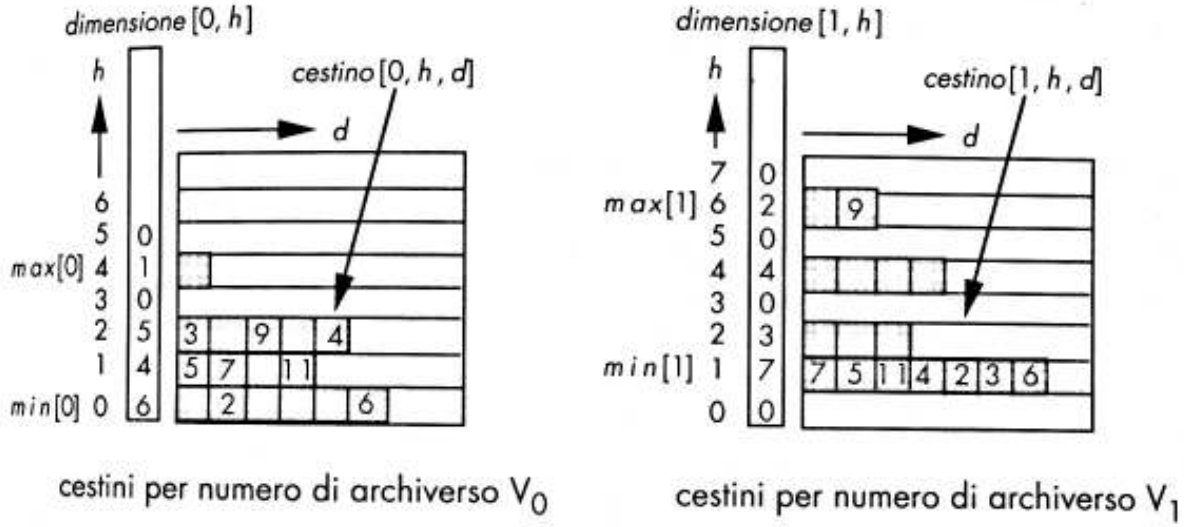


Fig. 19.12 Struttura di dati "a cestini".

19.17 [Allineamento globale multiplo] Generalizzando il problema dell'allineamento globale visto nell'Esercizio 13.6 ad un numero arbitrario *k* di stringhe $P_1, ..., P_k$, si ottiene il cosiddetto problema dell'ALLINEAMENTO GLOBALE MULTIPLO che però è NP-arduo. Si progetti un algoritmo euristico per l'allineamento globale multiplo basato sull'algoritmo per l'allineamento globale tra 2 stringhe.

19.18 [Colorazione] Si progetti un algoritmo euristico per il problema della COLORAZIONE in forma di ottimizzazione.

19.4 La procedura bottino() utilizza una matrice booleana a tre dimensioni $S[1 \dots n, -m \dots m/3, -m \dots m/3]$. $S[i, j, h]$ = **true** se i primi i monili sono spartiti dando valore j al primo ladro, h al secondo, ed il rimanente al terzo. La spartizione è possibile se e solo se $S[n, m/3, m/3]$ = **true**, e la soluzione si può ricavare a ritroso a partire da $S[n, m/3, m/3]$. La procedura ha complessità pseudo-polinomiale $O(nm^2)$.

```
bottino(integer[] P, integer n, integer m, integer[][][] S)
for i ← 1 to n do
    for j ← −m to m/3 do
        for h ← −m to m/3 do
            S[i, j, h] ← false
S[1, P[1], 0] ← true
S[1, 0, P[1]] ← true
S[1, 0, 0] ← true
for i ← 2 to n do
    for j ← −m to m/3 do
        for h ← −m to m/3 do
            S[i, j, h] ← S[i − 1, j − P[i], h] or S[i − 1, j, h − P[i]] or S[i − 1, j, h]
```

19.5 Si parte con una soluzione S vuota e si scandiscono uno dopo l'altro tutti gli archi di G. Se si incontra un arco $e = [u, v]$ per il quale entrambi i nodi estremi u e v non sono stati ancora selezionati, allora si aggiungono i due nodi u e v alla soluzione S. Tale algoritmo adotta una strategia *greedy*, poiché la scelta permette di coprire non solo l'arco e, ma anche tutti gli archi incidenti nei nodi u e v. Inoltre garantisce di coprire tutti gli archi poiché l'arco e non è selezionato se almeno uno dei suoi nodi estremi è già stato incluso in S. La complessità è $O(n + m)$ qualora si memorizzi il grafo con liste di adiacenza. Si osservi che il sottografo indotto dall'insieme S consiste di $|S|/2$ archi nessuna coppia dei quali ha un nodo estremo in comune. D'altro canto, per coprire tale sottografo occorrono almeno $|S|/2$ nodi e quindi una copertura ottima S^* di G ha cardinalità non inferiore ad $|S|/2$. Ma allora $|S|$ non supera il doppio della cardinalità minima $|S^*|$.

19.42 Si assuma per semplicità e senza perdere in generalità che nessuna stringa sia una sottostringa di un'altra. Costruiamo un grafo completo orientato e pesato G (detto grafo delle *sovrapposizioni*) i cui nodi corrispondono alle stringhe e i cui archi orientati corrispondono a coppie ordinate di stringhe. Il peso c_{ij} dell'arco (i, j) è uguale al numero massimo di caratteri identici tra un suffisso di P_i ed un prefisso di P_j, ma cambiato di segno (per esempio, se P_i = ACGA e P_j = GACA, allora $c_{ij} = -2$, mentre $c_{ji} = -1$). Trovare la minima superstringa comune corrisponde a trovare il cammino Hamiltoniano di peso minimo nel grafo G. L'euristica greedy è simile alla procedura greedyTsp(), con la differenza che per ogni nodo i occorrono due contatori $in[i]$ e $out[i]$ inizializzati a 0, che un arco orientato (i, j) è aggiunto alla soluzione se $out[i] < 1$ **and** $in[j] < 1$ **and** S.find(i) $\neq$ S.find(j), e che ovviamente si evita di richiudere il ciclo non aggiungendo alcun arco nella conclusione.

19.13 Si considerino i dati mostrati nella Fig. 19.12(a), dove si assume che le distanze non esplicitate siano tutte uguali a 5. La soluzione π^* illustrata nella Fig. 19.12(b) ha costo $c(\pi^*) = 11$ ed è un ottimo globale. La soluzione π della Fig. 19.12(c) ha costo $c(\pi) = 12$ ed è un ottimo locale, come si può verificare considerando i costi di tutte le soluzioni $\pi' \in I_2(\pi)$ mostrate nella Fig. 19.12(d). L'ottimo globale π^* non appartiene a $I_2(\pi)$, poiché non è ottenibile da π scambiando due archi, ma scambiandone tre.

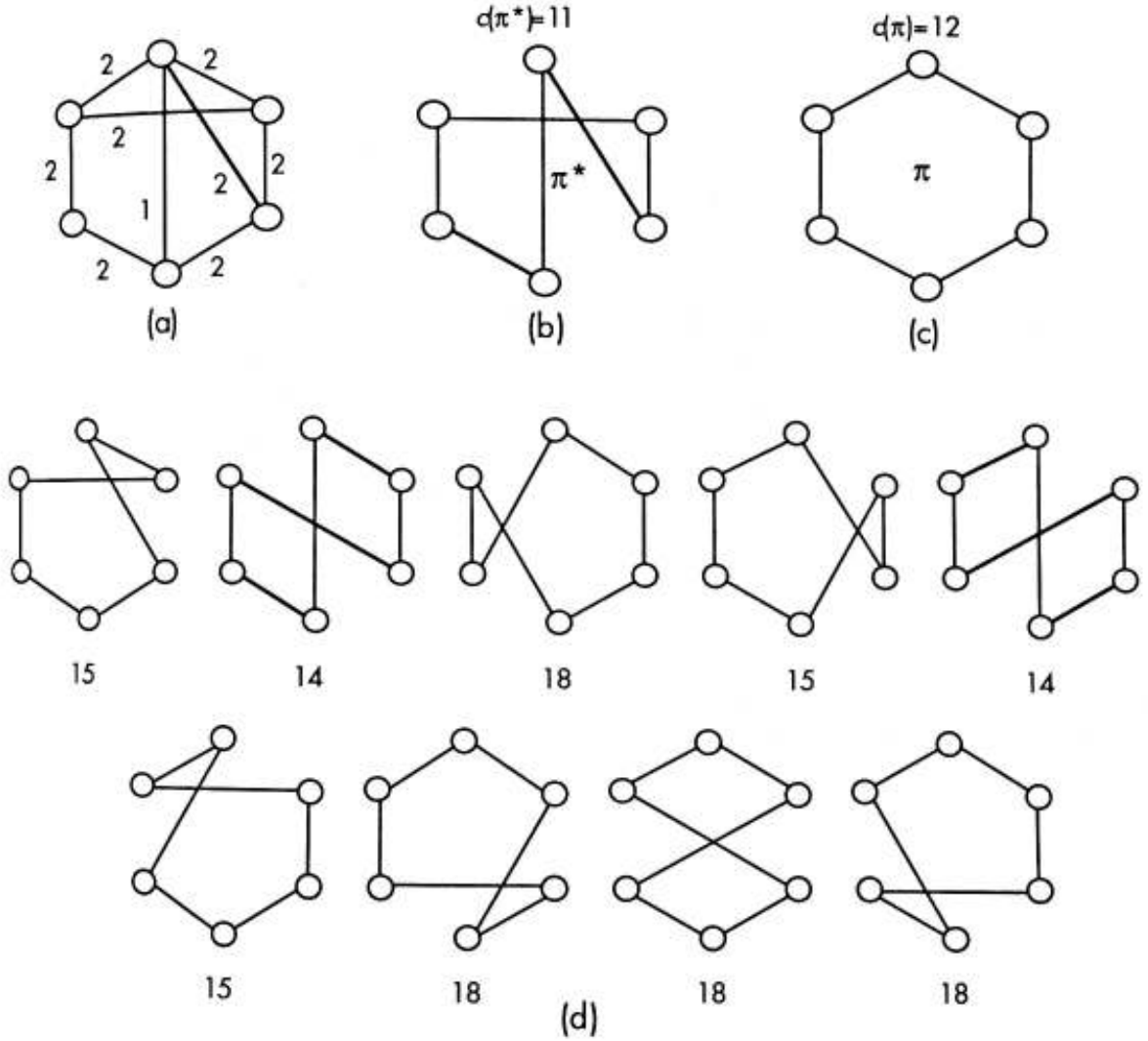


Fig. 19.12 (a) Dati per il COMMESSO VIAGGIATORE (le distanze non indicate valgono 5). (b) Ottimo globale π^*. (c) Un ottimo locale π. (d) Tutte le soluzioni π' che stanno in $I_2(\pi)$.

19.15 Dato che una soluzione ammissibile S per il problema consiste in una partizione dell'insieme A in due parti, un intorno $I(S)$ si può definire come l'insieme di tutte le soluzioni S' ottenibili da S o spostando un elemento da una parte all'altra o scambiando tra loro due elementi tra le due parti. La nuova soluzione S' che sostituisce S si sceglie tra quelle che diminuiscono lo scarto tra la somma degli elementi che stanno in una parte e la somma di quelli che stanno nell'altra parte. La cardinalità di $I(S)$ è $O(n^2)$. Per

ciascuna mossa possibile (spostamento o scambio), la differenza tra gli scarti delle soluzioni S ed S' si può calcolare in tempo costante. Pertanto ogni iterazione di ricerca locale costa tempo $O(n^2)$.

19.17 La semplice euristica seguente, nota come *star alignment*, è molto usata in biologia. Si risolvono tutti i problemi di allineamento globale tra tutte le coppie di stringhe usando $\frac{k(k-1)}{2}$ volte l'algoritmo di programmazione dinamica visto nella soluzione dell'Esercizio 13.7. Sia c_{ij} il punteggio ottenuto per la coppia di stringhe P_i e P_j. Si calcola $s_i = \sum_{j=1}^{k} c_{ij}$, per ogni $1 \leq i \leq k$ e si seleziona $s_h = \max_{1 \leq i \leq k} \{s_i\}$. Si sceglie come soluzione dell'allineamento globale multiplo quella data dai $k-1$ allineamenti globali ottenuti tra P_h e ciascuna delle altre stringhe. Tale euristica ha complessità $O(k^2 n^2)$, dove n è la lunghezza della stringa più lunga tra $P_1, \ldots, P_k$.

19.18 Descriviamo una euristica molto nota, chiamata *saturation degree*. Si assuma che i colori siano indicati con interi positivi. L'euristica colora un nodo u alla volta, assegnandogli il primo colore disponibile (cioè l'intero più piccolo) che non è stato ancora assegnato ad alcun nodo del suo insieme di adiacenza adj(u). Ad ogni passo, si colora il nodo u per il quale è massimo il numero di colori distinti già assegnati ai nodi in adj(u). In caso di parità tra più nodi, si colora il nodo u per il quale è massimo il numero di nodi già colorati in adj(u). L'idea di base è che questi nodi hanno una minore possibilità di scelta del colore e quindi un maggior rischio che tutti i colori disponibili siano assegnati ai nodi adiacenti nei passi successivi. In caso di ulteriore parità tra più nodi, se ne colora uno di questi scelto in modo casuale. Di solito, l'euristica è rieseguita un prefissato numero di volte. Infatti ad ogni esecuzione le scelte casuali che risolvono le situazioni di parità saranno presumibilmente diverse, e quindi in genere si otterranno colorazioni diverse, tra le quali viene selezionata quella che usa il minor numero di colori.

CONCLUSIONE

Giunti alla fine di questo testo, cerchiamo di ricavarne "la morale". Nel cercare di risolvere un problema, lo si affronta da due lati. Da una parte, si cerca di trovare costruttivamente un algoritmo che lo risolva, fornendo così una limitazione superiore alla complessità del problema. D'altro lato, si cerca di dimostrare matematicamente una limitazione inferiore alla complessità intrinseca del problema, che valga per qualsiasi algoritmo. Infine, se possibile, si cerca di individuare un algoritmo ottimo, dimostrando limitazioni inferiori via via più elevate e/o progettando algoritmi via via più veloci. In quest'ottica, le dimostrazioni di NP-completezza servono a provare rigorosamente limitazioni inferiori superpolinomiali della complessità.

In pratica, i due aspetti interagiscono e vengono entrambi affrontati dal progettista. Infatti, dati di ingresso particolarmente sfavorevoli per il comportamento di un certo algoritmo possono suggerire la strada da seguire in una dimostrazione di NP-completezza. D'altro canto, una prova di NP-completezza fallita può aiutare ad individuare particolari situazioni da affrontare nella progettazione di un algoritmo.

Una caratterizzazione matematica della soluzione spesso suggerisce un semplice algoritmo di risoluzione, di solito non molto efficiente. Applicando opportune tecniche di progettazione, delle quali le più usate sono il divide-et-impera e la programmazione dinamica, e organizzando i dati in strutture di dati appropriate, si può riuscire a migliorare notevolmente l'efficienza dell'algoritmo proposto. Queste tre fasi (caratterizzazione della soluzione, applicazione di tecniche di progettazione, strutturazione dei dati) possono essere riconsiderate più di una volta, e in un ordine qualsiasi, ottenendo versioni sempre più efficienti dell'algoritmo, la cui complessità può magari raggiungere in ordine di grandezza una limitazione inferiore dimostrata precedentemente. Trovato un algoritmo efficiente in ordine di grandezza, si prova a modificarlo "limando" se possibile le costanti coinvolte nella funzione di complessità ed effettuando prove sperimentali su dati "reali", in modo da ottenere una versione dell'algoritmo che sia efficiente non solo in teoria ma anche in pratica. Per esempio, è ben noto che il Quick Sort e l'Algoritmo del Simplesso, pur aven-

do complessità superiore in ordine di grandezza ad altri algoritmi che risolvono i problemi dell'ORDINAMENTO e della PROGRAMMAZIONE LINEARE, sono efficientissimi in pratica.

Se invece il problema è "intrattabile", si possono seguire diverse strade. Innanzitutto, si può vedere se c'è qualche caso speciale del problema che sia risolubile in tempo polinomiale. Se il problema generale è di ottimizzazione, si può tentare di individuare un algoritmo di approssimazione assoluta che abbia complessità polinomiale e fornisca una soluzione vicina a quella ottima. Se il problema non è "fortemente" NP-arduo, si può cercare un algoritmo esatto di complessità pseudo-polinomiale oppure uno ε-approssimato di complessità polinomiale. Si possono progettare algoritmi esponenziali di tipo branch-&-bound, o ricercare algoritmi euristici, di solito basati su tecniche greedy o di ricerca locale, che siano particolarmente facili da programmare, abbiano se possibile complessità polinomiale, e forniscano sperimentalmente "buone" soluzioni su dati "reali". A tal fine, sono spesso disponibili via "internet" sia particolari dati di ingresso significativi del problema, sia il codice eseguibile e/o i risultati di vari algoritmi per lo stesso problema. È così possibile valutare la "bontà" del proprio algoritmo confrontandone, sugli stessi dati di ingresso, sia il valore della soluzione sia il tempo di calcolo con quelli di altri algoritmi già noti.

	Problemi "facili" (polinomiali)	*Problemi "difficili" (NP-ardui)*
1.	DOMINO LINEARE	DOMINO LIMITATO
2.	2-SODDISFATTIBILITÀ	3-SODDISFATTIBILITÀ
3.	PROGRAMMAZIONE LINEARE	PROGRAMMAZIONE LINEARE 0/1
4.	ABBINAMENTO 2-DIMENSIONALE	ABBINAMENTO 3-DIMENSIONALE
5.	CIRCUITO EULERIANO	CIRCUITO HAMILTONIANO
6.	ZAINO REALE	ZAINO INTERO
7.	MINIMO ALBERO DI COPERTURA	MINIMO ALBERO DI COPERTURA CON k CONNESSIONI
8.	CAMMINI MINIMI	CAMMINI MINIMI SENZA NODI RIPETUTI
9.	2-COLORAZIONE	3-COLORAZIONE

Tabella 1 Coppie di problemi apparentemente simili ma con complessità diversa.

Molto importante è l'analisi dei "casi speciali" del problema. Spesso si è vittime di una inutile generalizzazione, che magari rende NP-arduo un problema altrimenti risolubile in tempo polinomiale! Se si è interessati solo ad alcuni casi speciali, l'intrattabilità del problema generalizzato è praticamente irrilevante.

Per esempio, la SODDISFATTIBILITÀ è NP-completa se in ogni clausola compaiono al più tre variabili (affermate o negate), ma è risolubile in tempo polinomiale se ne compaiono al più due! La CRICCA, l'INSIEME INDIPENDENTE, la COLORAZIONE, e il CIRCUITO HAMILTONIANO diventano polinomiali se formulati su un grafo di intervalli, i cui nodi rappresentano intervalli della retta reale e i cui archi rappresentano l'intersezione di due intervalli [capitolo 9]. Viceversa, il MINIMO ALBERO DI COPERTURA diventa NP-completo se si richiede che ogni nodo dell'albero sia collegato ad al più k nodi, con k dato di input del problema. La PROGRAMMAZIONE LINEARE INTERA (anche solo 0/1) è NP-completa, ma la PROGRAMMAZIONE LINEARE, in cui le variabili x_i sono reali, è risolubile in tempo polinomiale.

La 3-COLORAZIONE è NP-completa, ma la 2-COLORAZIONE è risolubile in tempo polinomiale. L'ABBINAMENTO 3-DIMENSIONALE è NP-completo, ma quello 2-dimensionale (che è equivalente all'usuale ABBINAMENTO) è risolubile in tempo polinomiale.

Problemi	*Complessità e autori*
1. PROGRAMMAZIONE LINEARE	$O(n^{3.5}L^2)$ $L = \log \lvert \prod a_{ij} \prod b_i \rvert$ Karmarkar (1984)
2. FLUSSO MASSIMO	$O(mn \log (n^2/m))$ Goldberg-Tarjan (1986)
3. ABBINAMENTO	$O(m\sqrt{n})$ Micali-Vazirani (1980)
4. MINIMO ALBERO DI COPERTURA	$O(min\{m + n \log n, m\beta\})$ $\beta = min\{i: \log^{(i)} n \leq m/n\}$ Fredman-Tarjan (1987)
5. CAMMINI MINIMI TRA TUTTE LE COPPIE	$O(mn + n^2 \log n)$ Fredman-Tarjan (1987)
6. MOLTIPLICAZIONE DI MATRICI	$O(n^{2.376})$ Coppersmith-Winograd (1987)
7. MOLTIPLICAZIONE DI POLINOMI	$O(n \log n)$ Cooley-Tukey (1965)
8. PRIMALITÀ	$O(\log^{6+\epsilon} n)$ Lenstra-Pomerance (2005)

Tabella 2 Complessità dei migliori algoritmi conosciuti per alcuni problemi.

È istruttivo osservare come molti problemi, solo in apparenza simili, abbiano complessità diversa. Per esempio, il CIRCUITO HAMILTONIANO, in cui si chiede se esiste una catena chiusa che includa ogni nodo una e una sola volta, è NP-completo, mentre il CIRCUITO EULERIANO, in cui si chiede se esiste una catena chiusa che includa ogni arco una e una sola volta, è risolubile in tempo polinomiale. Analogamente, l'INSIEME INDIPENDENTE, dove si cerca un insieme di nodi a due a due non connessi da archi, è NP-arduo, mentre l'ABBINAMENTO, dove si cerca un insieme di archi a due a due non incidenti su uno stesso nodo, è risolubile in tempo polinomiale.

La Tabella 1 riassume 9 coppie di problemi apparentemente simili: quelli nella colonna di sinistra sono "facili" (risolubili in tempo polinomiale), mentre quelli nella colonna di destra sono "difficili" (NP-ardui). Alcune delle soluzioni viste in questo testo sono state scelte per la loro efficacia didattica, ma non sono gli algoritmi più efficienti in assoluto. Nella Tabella 2 riportiamo quindi alcuni di questi problemi, assieme alla complessità dei migliori algoritmi conosciuti e agli autori degli algoritmi stessi.

Infine, è bene rimarcare che lo studio dell'algoritmica non finisce qui: esistono innumerevoli altri problemi "classici", che non è stato possibile affrontare per motivi di spazio. Per alcuni di questi problemi, è possibile trovare fascicoli supplementari sul sito associato al libro.

BIBLIOGRAFIA

La presente bibliografia è suddivisa per capitoli. Per ogni capitolo, sono elencati sia libri di testo, specificando se si tratta di testi introduttivi o specialistici, sia articoli di riviste scientifiche. I testi introduttivi sono adatti sia a studenti del Corso di Laurea in Informatica che a lettori che abbiano una preparazione matematica a livello di scuola superiore. I testi specialistici, alcuni dei quali sono adottati come libri di testo nell'ambito di corsi universitari avanzati, sono invece adatti a studenti del successivo Corso di Laurea Magistrale in Informatica, o comunque a lettori più maturi. Gli articoli di riviste scientifiche sono articoli di ricerca che rappresentano la fonte originale degli argomenti trattati nel presente testo oppure che forniscono materiale per ulteriori approfondimenti e complementi. Non essendo stati generalmente scritti a fini didattici, presuppongono spesso la conoscenza di risultati specifici, e sono per questo più adatti a studenti del Dottorato di Ricerca. Tali articoli sono stati inseriti ugualmente perché è opinione degli autori che tutti si debbano pian piano abituare a leggere direttamente le fonti originali, con spirito critico e costruttivo.

1) INTRODUZIONE

A. Aho, J. Hopcroft e J. Ullman (1983), *Data Structures and Algorithms*, Addison-Wesley, Reading, MA. Ottimo testo introduttivo dedicato alle strutture di dati e alla progettazione e all'analisi di algoritmi.

S. Baase e A. Van Gelder (2000), *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, MA. Buon testo introduttivo, breve e chiaro, su progettazione e analisi di algoritmi.

T.H. Cormen, C.E. Leiserson, R.L. Rivest e C. Stein (2001), *Introduction to Algorithms*, McGraw-Hill, New York. Testo enciclopedico rivolto alla progettazione e all'analisi di algoritmi ed alle strutture di dati. Descrive e analizza anche gli "heap binomiali" e gli "heap di Fibonacci", atti ad effettuare l'operazione di unione tra due heap.

D. Harel (1987), *Algorithmics: The Spirit of Computing*, Addison-Wesley, Reading, MA. Eccellente opera introduttiva che affronta tutti gli aspetti dell'algoritmica.

E. Horowitz, S. Sahni e D. Mehta (1995), *Fundamental of Data Structures in C++*, Computer

Science Press, New York. Opera specialistica a carattere enciclopedico sulle strutture di dati, con procedure in C++.

E. Horowitz, S. Sahni e S. Rajasekaran (1997), *Computer Algorithms*, Computer Science Press, New York. Opera enciclopedica su progettazione e analisi di algoritmi.

J. Kleinberg e E. Tardos (2006), *Algorithm Design*, Addison-Wesley, Reading, MA. Ottimo testo pedagogico per apprendere il processo di progettazione degli algoritmi, partendo dalla formulazione di problemi reali e giungendo all'analisi degli algoritmi progettati.

D. Knuth (1968, 1969, 1973), *The Art of Computer Programming* (3 volumi), Addison-Wesley, Reading, MA. Un'opera enciclopedica fondamentale sull'algoritmica, in particolare, e sull'intera informatica, in generale.

R. Sedgewick (1988), *Algorithms*, Addison-Wesley, Reading, MA. Ottimo testo introduttivo contenente algoritmi, scritti in Pascal, per i principali problemi computazionali. Ne esistono anche edizioni successive in cui gli algoritmi sono scritti in C, C++ e Java.

S. Skiena (2008), *The Algorithm Design Manual*, Springer-Verlag, London, UK. Ottimo testo di riferimento sui principali problemi algoritmici e la loro risoluzione. Contiene anche riferimenti a librerie open-source e commerciali che realizzano tali algoritmi.

2) ANALISI DI ALGORITMI

J.L. Bentley, D. Haken e J.B. Saxe (1980), "A General Method for Solving Divide-and-Conquer Recurrences", SIGACT *News* 12, pp. 36-44. Propone un metodo generale per risolvere relazioni di ricorrenza.

R. Sedgewick e P. Flajolet (1996), *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA. Testo specialistico sulle tecniche di analisi della complessità di algoritmi, comprese le cosiddette "funzioni generatrici".

R.E. Tarjan (1985), "Amortized Computational Complexity", *SIAM Journal on Algebraic and Discrete Methods* 6, pp. 306-318. Inventa l'analisi ammortizzata.

3) TIPI DI DATO E STRUTTURE DI DATI

E. Lodi e G. Pacini (1990), *Introduzione alle Strutture di Dati*, Boringhieri, Torino. Buona trattazione in italiano su specifica e algebra di strutture di dati.

D. Stubbs e N. Webre (1987), *Data Structures with Abstract Data Types and Modula-2*, Brooks/Cole, Monterey, CA. Introduce le strutture di dati specificandole come tipi di dato e utilizzando il linguaggio Modula-2, che è particolarmente adatto allo scopo.

4) STRUTTURE DI DATI ELEMENTARI, 5) ALBERI

D. Knuth (1968), *The Art of Computer Programming; Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA. Contiene numerosissimi riferimenti sulle origini delle strutture elementari di dati, come vettori, liste, pile, code e alberi.

6) ALBERI BILANCIATI DI RICERCA, 7) TABELLE HASH, 8) INSIEMI E DIZIONARI

G.M. Adelson-Velskii e E.M. Landis (1962), "An Algorithm for the Organization of Information", *Soviet Mathematics Doklady* 3, pp. 1259-1263. Introduce gli alberi bilanciati AVL, che prendono il nome dalle iniziali degli autori.

A. Aho, J. Hopcroft e J. Ullman (1983), *Data Structures and Algorithms*, Addison-Wesley, Reading, MA. Tratta tra l'altro anche gli alberi 2-3 e 2-3-4.

A. Andersson (1993), "Balanced Search Trees Made Simple", *Lecture Notes in Computer Science*, pp. 60-71. Presenta una realizzazione semplice ed efficiente dei B-alberi binari.

R. Bayer (1972), "Symmetric Binary B-Trees: Data Structures and Maintainance Algorithms", *Acta Informatica* 1, pp. 290-306. Propone i B-alberi binari.

R. Bayer e E. McCreight (1972), "Organization and Maintainance of Large Ordered Indices", *Acta Informatica* 1, pp. 173-189. Sono introdotti per la prima volta i B-alberi (dove forse la B è l'iniziale di "bilanciati" oppure di Bayer stesso).

L. Guibas e R. Sedgewick (1978), "A Dichromatic Framework for Balanced Trees", *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pp. 8-21. Propone l'uso di due colori (rosso e nero) per i B-alberi binari di Bayer e ne studia le proprietà di bilanciamento.

D. Knuth (1973), *The Art of Computer Programming; Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA. Testo di riferimento per l'analisi delle tabelle hash.

9) GRAFI

C. Berge (1973), *Graphs and Hypergraphs*, North-Holland, Amsterdam. Testo classico su grafi e ipergrafi, trattati da un punto di vista squisitamente matematico.

R. Diestel (2005), *Graph Theory*, Springer-Verlag, Heidelberg. Recente libro di testo adatto per auto-apprendere i fondamenti matematici della teoria dei grafi, liberamente scaricabile da http://diestel-graph-theory.com/index.html

M.C. Golumbic (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York. Testo introduttivo su algoritmi per grafi "perfetti", che includono i grafi di intervalli come caso particolare.

T. Nishizeki e N. Chiba (1988), *Planar Graph: Theory and Algorithms*, North-Holland, Amsterdam. Testo su teoria ed algoritmi per grafi "planari".

F. Roberts (1978), *Graph Theory and its Applications to Problems of Society*, SIAM, Philadelphia, PA. Buon testo introduttivo sulle innumerevoli applicazioni dei grafi nei problemi della vita quotidiana.

10) CODE CON PRIORITÀ E INSIEMI DISGIUNTI

M.R. Brown (1978), "Implementation and Analysis of Binomial Queue Algorithms", *SIAM Journal on Computing* 7, pp. 298-319. Studia in dettaglio le proprietà degli "heap binomiali" introdotti da Vuillemin.

M.L. Fredman e R.E. Tarjan (1987), "Fibonacci Heaps and Their Use in Improved Network Optimization Problems", *Journal of the ACM* 34, pp. 596-615. Propone gli "heap di Fibonacci"

per realizzare code con priorità. È ammessa anche l'operazione union(), che richiede $O(1)$ tempo ammortizzato.

J. van Leeuwen e R.E. Tarjan (1984), "Worst-case Analysis of Set Union Algorithms", *Journal of the ACM* 7, pp. 245-281. Analizza svariate tecniche per realizzare un MFSET.

J. Vuillemin (1978), "A Data Structure for Manipulating Priority Queues", *Communications of the ACM* 21, pp. 309-315. Introduce gli "heap binomiali" per realizzare code con priorità. È ammessa anche l'operazione union(), che richiede $O(\log n)$ tempo nel caso pessimo.

J.W.J. Williams (1964), "Algorithm 232 (Heapsort)", *Communications of the ACM* 7, pp. 347-348. Propone per la prima volta sia la realizzazione di una coda con priorità con uno heap che l'algoritmo Heapsort.

11) STRUTTURE DI DATI E PROGETTAZIONE DI ALGORITMI

R.E. Bellman (1958), "On a Routing Problem", *Quarterly Applied Mathematics* 16, pp. 87-90. Introduce le condizioni di ottimalità per i CAMMINI MINIMI e fornisce un algoritmo generale che è stato scoperto anche da Ford e da Moore.

A.A. Bertossi e A. Gori (1988), "Total Domination and Irreduncance in Weighted Interval Graphs", *SIAM Journal on Discrete Mathematics* 1, pp. 317-327. Formula il problema dell'INSIEME DOMINANTE su un grafo di intervalli come problema di CAMMINI MINIMI e lo risolve usando una struttura di dati simile ad uno heap.

E.W. Dijkstra (1959), "A Note on Two Problems in Connection with Graphs", *Numerische Mathematik* 1, pp. 269-271. Fornisce l'algoritmo per risolvere il problema dei CAMMINI MINIMI per grafi con lunghezze non negative sugli archi.

M.L. Fredman e R.E. Tarjan (1987), "Fibonacci Heaps and Their Use in Improved Network Optimization Problems", *Journal of the ACM* 34, pp. 596-615. Introduce gli heap di Fibonacci e li usa per risolvere il problema dei CAMMINI MINIMI.

G. Gallo e S. Pallottino (1988), "Shortest Paths Algorithms", *Annals of Operations Research* 13, pp. 3-79. Buon articolo di rassegna su algoritmi e strutture di dati per il problema dei CAMMINI MINIMI.

J.E. Hopcroft e R.E. Tarjan (1974), "Efficient Planarity Testing", *Journal of the ACM* 21, pp. 549-568. Presenta un algoritmo di complessità ottima per verificare la planarità di un grafo.

D.B. Johnson (1977), "Efficient Algorithms for Shortest Paths in Sparse Networks", *Journal of the ACM* 24, pp. 1-13. Rende più efficiente l'algoritmo di Dijkstra per grafi sparsi.

U. Pape (1974), "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem", *Mathematical Programming* 7, pp. 212-222. In pratica l'algoritmo di CAMMINI MINIMI più efficiente per grafi sparsi planari.

12) DIVIDE-ET-IMPERA

J.W. Cooley e J.W. Tukey (1965), "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation* 19, pp. 297-301. Algoritmo ottimo per la TRASFORMATA DISCRETA DI FOURIER, che permette di moltiplicare, con la stessa complessità, due polinomi.

C.A.R. Hoare (1962), "Quicksort", *Computer Journal* 5, pp. 10-15. Presenta per la prima volta il Quicksort, suggerendo anche tutte le principali varianti.

A. Karatsuba e Y. Ofman (1962), "Multiplication of Many-Digital Numbers by Automatic Com-

puters". *Proceedings of the USSR Academy of Sciences* 145, pp. 293-294. Mostra per la prima volta che il limite inferiore per la moltiplicazione di due numeri non è $\Omega(n^2)$.

D. Knuth (1973), *The Art of Computer Programming; Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA. Un trattato enciclopedico sugli algoritmi di ordinamento.

R. Sedgewick (1978), *Quicksort*, Garland, New York. Libro interamente dedicato al "quicksort"; include l'analisi di complessità e gli effetti pratici di numerose varianti.

V. Strassen (1969), "Gaussian Elimination is not Optimal", *Numerische Matematik* 14, pp. 354-356. Il primo algoritmo per la MOLTIPLICAZIONE DI MATRICI con complessità inferiore ad $O(n^3)$.

13) PROGRAMMAZIONE DINAMICA

R. Bellman (1957), *Dynamic Programming*, Princeton University Press. Testo scritto dall'inventore stesso della "programmazione dinamica".

R. Floyd (1962), "Algorithm 92 (Shortest Path)", *Communications of the ACM* 5, pp. 345. Propone l'algoritmo per i CAMMINI MINIMI tra tutte le coppie, basandosi su un teorema di Warshall pubblicato nello stesso anno.

T.C. Hu e M.T. Shing (1980), "Some Theorems about Matrix Multiplication", *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, pp. 28-35. Fornisce l'algoritmo ottimo per moltiplicare una catena di matrici.

Z. Galil e R. Giancarlo (1988), "Data Structures and Algorithms for Approximate String Matching", *Journal of Complexity* 4, pp. 33-72. Articolo di rassegna sullo STRING MATCHING APPROSSIMATO.

G.M. Landau e U. Vishkin (1988), "Fast String Matching with k Differences", *Journal of Computer and System Sciences* 37, pp. 63-78. Risolve molto efficientemente una variante dello STRING MATCHING.

14) GREEDY

E.L. Lawler (1976), *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York. Include un'ampia trattazione dei matroidi, strutture per le quali la tecnica greedy garantisce di trovare sempre l'ottimo globale.

J.B. Kruskal (1956), "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem", *Proceedings of the AMS* 7, pp. 48-50. Introduce l'algoritmo per il MINIMO ALBERO DI COPERTURA che porta il suo nome.

J.M. Moore (1968), "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs", *Management Science* 15, pp. 102-109. Introduce l'algoritmo per il problema dei PROGRAMMI IN RITARDO.

R.C. Prim (1957), "Shortest Connection Networks and Some Generalizations", *Bell System Technical Journal* 36, pp. 1389-1401. Introduce un algoritmo per il MINIMO ALBERO DI COPERTURA molto simile all'algoritmo di Dijkstra.

15) RICERCA LOCALE

G.B. Dantzig (1963), *Linear Programming and Extensions*, Princeton University Press. Testo scritto da chi ha inventato nel 1947 l'algoritmo del "simplesso" per la PROGRAMMAZIONE LINEARE.

J. Edmonds e R.M. Karp (1972), "Theoretical Improvements in the Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM* 19, pp. 248-264. Viene descritto l'algoritmo per il flusso massimo che utilizza una visita in ampiezza per trovare un cammino aumentante.

L.R. Ford e D.R. Fulkerson (1962), *Flows in Networks*, Princeton University Press. Propone l'algoritmo per il flusso massimo che usa un cammino aumentante qualsiasi.

V.M. Malhotra, M.P. Kumar e S.N. Maheshwari (1978), "An $O(n^3)$ Algorithm for Finding Maximum Flows in Networks", *Information Processing Letters* 7, pp. 277-278. Introduce il cosiddetto algoritmo dei "3 indiani" (dalla nazionalità degli autori), basandosi su risultati precedenti di Dinic e Karzanov.

C. Papadimitriou e K. Steiglitz (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ. Ottimo testo su algoritmi di ottimizzazione, dedicato alla PROGRAMMAZIONE LINEARE e a problemi su grafi, come FLUSSO MASSIMO, CAMMINI MINIMI, e ABBINAMENTO.

D.L. Shell (1959), "A High-speed Sorting Procedure", *Communications of the ACM* 2, pp. 30-32. Inventa l'algoritmo Shellsort.

R.E. Tarjan (1983), *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA. Testo specialistico su strutture di dati e algoritmi per problemi di ottimizzazione su grafi, inclusi il FLUSSO MASSIMO e i CAMMINI MINIMI.

16) BACKTRACK

R.L. Graham (1972), "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set", *Information Processing Letters* 1, pp. 132-133. Presenta l'algoritmo di complessità ottima per l'INVILUPPO CONVESSO.

F. Preparata e M. Shamos (1985), *Computational Geometry: An Introduction*, Springer-Verlag, New York. Ottimo libro su algoritmi per problemi di geometria computazionale, incluso l'INVILUPPO CONVESSO.

17) ALGORITMI PROBABILISTICI

M. Agrawal, N. Kayal, N. Saxena (2004), *PRIMES is in P*, Annals of Mathematics 160 (2): 781–793. L'articolo originale che dimostra che il problema della PRIMALITÀ può essere risolto in tempo polinomiale.

M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest e R.E. Tarjan (1973), "Time Bounds for Selection", *Journal of Computer and System Sciences* 7, pp. 448-461. È proposto un algoritmo di complessità lineare nel caso pessimo per la SELEZIONE.

P. Ferragina e F. Luccio (2001), *Crittografia*, Bollati Boringhieri, Torino. Ottimo testo introduttivo in italiano sulla crittografia, con taglio squisitamente algoritmico.

C.A.R. Hoare (1961), "Algorithm 63 (Partition) and Algorithm 65 (Find)", *Communications of the ACM* 4, pp. 321-322. Propone l'algoritmo di complessità lineare nel caso medio per la SELEZIONE.

H.W. Lenstra jr. e Carl Pomerance (2011), *Primality testing with Gaussian periods*. Rapporto di ricerca, Università di Darthmouth, 2011. L'articolo che migliora i risultati di Agrawal, Kayal e Saxena sulla primalità.

S.K. Park e K.W. Miller (1988), "Random Number Generators: Good Ones Are Hard to Find",

Communications of the ACM 31, pp. 1192-1201. Consigli utili per progettare un buon generatore di numeri pseudo-casuali.

M.O. Rabin (1976), "Probabilistic Algorithms", in: *Algorithms and Complexity* (a cura di J. Traub), Academic Press, New York, pp. 21-39. Inventa gli algoritmi probabilistici.

M.O. Rabin (1980), "Probabilistic Algorithm for Testing Primality", *Journal of Number Theory* 12, pp. 128-138. Presenta l'algoritmo probabilistico per la PRIMALITÀ.

R.L. Rivest, A. Shamir e L.M. Adleman (1978), "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM* 21, pp. 120-126. Introduce il sistema crittografico RSA.

18) TEORIA DELLA NP-COMPLETEZZA

J. Cohen (1979), "Non-Deterministic Algorithms", *ACM Computing Surveys* 11, pp. 79-94. Articolo divulgativo sugli algoritmi non deterministici.

S. Cook (1971), "The Complexity of Theorem Proving Procedures", *Proceedings of the 27th Annual Symposium on Foundation of Computer Science*, pp. 151-158. Introduce per la prima volta, indipendentemente da Levin, la nozione di NP-completezza e dimostra che la SODDISFATTIBILITÀ è NP-completa.

J. Edmonds (1965), "Paths, Trees, and Flowers", *Canadian Journal of Mathematics* 17, pp. 449-467. Introduce le classi $\mathbb{P}$ ed $\mathbb{NP}$, congetturando $\mathbb{P} \neq \mathbb{NP}$.

M.R. Garey e D.S. Johnson (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA. Il primo e miglior testo enciclopedico sull'NP-completezza, contenente un'appendice con centinaia di problemi NP-completi.

R.M. Karp (1972), "Reducibility among Combinatorial Problems", in: *Complexity of Computer Computations* (a cura di R.E. Miller e J.W. Thatcher), Plenum Press, New York, pp. 85-103. Prosegue il lavoro di Cook, mostrando l'NP-completezza di svariati problemi con riduzioni dalla SODDISFATTIBILITÀ.

R.E. Ladner (1975), "On the Structure of Polynomial Time Reducibility", *Journal of the ACM* 22, pp. 155-171. Prova l'esistenza di problemi NP-intermedi, qualora $\mathbb{P} \neq \mathbb{NP}$.

L. Levin (1972), "Universal Sorting Problems", *Problemy Peredachi Informatsii* 9, pp. 265-266 (in russo). Definisce, indipendentemente da Cook, la nozione di NP-completezza e dimostra che il DOMINO LIMITATO è NP-completo.

C.H. Papadimitriou (1995), *Computational Complexity*, Addison-Wesley, Reading, MA. Ottimo testo specializzato sulla complessità computazionale.

19) TECNICHE RISOLUTIVE PER PROBLEMI INTRATTABILI

N. Christofides (1976), "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem", Technical Report GSIA, Carnagie Mellon University, Pittsburgh, PA. Propone l'algoritmo con scarto 50% dal minimo per il ΔTSP.

M.R. Garey e D.S. Johnson (1978), "Strong NP-Completeness Results: Motivations, Examples, and Implications", *Journal of the ACM* 25, pp. 499-508. Definisce la nozione di NP-completezza forte.

F. Glover (1989), "Tabu Search: Part I", *ORSA Journal on Computing* 1, pp. 190-260. Inventa la ricerca tabù.

M. Held e R.M. Karp (1971), "The Travelling Salesman Problem and Minimum Spanning Trees: Part II", *Mathematical Programming* 1, pp. 6-25. Articolo sull'utilizzo di 1-alberi modificati

in algoritmi branch-&-bound per il TSP (È il seguito della "Parte I" pubblicata l'anno precedente su *Operations Research* 18, pp. 1138-1162).

O.H. Ibarra e C.E. Kim (1975), "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems", *Journal of the ACM* 22, pp. 463-468. Propone algoritmi di approssimazione per la SOMMA DI SOTTOINSIEME e lo ZAINO.

D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey e R.L. Graham (1974), "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms", *SIAM Journal on Computing* 3, pp. 299-325. Sono dimostrate approssimazioni per vari algoritmi di BIN PACKING, incluso "First-Fit-Decreasing".

E.L. Lawler e D.E. Wood (1966), "Branch-and-Bound Methods: A Survey", *Operations Research* 14, pp. 699-719. Articolo di rassegna sul branch-&-bound con numerosi riferimenti "storici".

C. Papadimitriou e K. Steiglitz (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ. Tratta anche il branch-&-bound, sia per la PROGRAMMAZIONE LINEARE INTERA che per problemi su grafi.

D.J. Rosenkrantz, R.E. Stearns e P.M. Lewis (1977), "An Analysis of Several Heuristics for the Travelling Salesman Problem", *SIAM Journal on Computing* 6, pp. 563-581. Tratta anche l'algoritmo con scarto 100% dal minimo per il ΔTSP.

S. Sahni e T. Gonzalez (1976), "P-complete Approximation Problems", *Journal of the ACM* 23, pp. 555-565. Dimostra che il COMMESSO VIAGGIATORE non è approssimabile.

CONCLUSIONE

D. Coppersmith e S. Winograd (1987), "Matrix Multiplication Via Arithmetic Progression", *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pp. 1-6. In teoria il miglior algoritmo per la MOLTIPLICAZIONE DI MATRICI, ma non utilizzabile in pratica a causa delle enormi costanti coinvolte nell'ordine di grandezza.

M.L. Fredman e R.E. Tarjan (1987), "Fibonacci Heaps and Their Use in Improved Network Optimization Problems", *Journal of the ACM* 34, pp. 596-615. I migliori algoritmi per i CAMMINI MINIMI e il MINIMO ALBERO DI COPERTURA.

A.V. Goldberg e R.E. Tarjan (1986), "A New Approach to the Maximum Flow Problem", *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pp. 136-146. Miglior algoritmo per il FLUSSO MASSIMO.

N. Karmarkar (1984), "A New Polynomial-Time Algorithm for Linear Programming", *Combinatorica* 4, pp. 373-395. In teoria il miglior algoritmo per la PROGRAMMAZIONE LINEARE, ma l'algoritmo del "simplesso" resta competitivo in pratica.

S. Micali e V.V. Vazirani (1980), "An $O(m\sqrt{n})$ Algorithm for Finding Maximum Matching in General Graphs", *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science*, pp. 17-27. Miglior algoritmo per l'ABBINAMENTO in un grafo qualsiasi (non necessariamente bipartito).

INDICE ANALITICO

2-Colorazione, 386-87
3-Colorazione, 386-87
3-Partition *vedi* 3-Partizione
3-Partizione, 360

Abbinamento, 301-303, 341-42, 386-387
  2-dimensionale, 386-87
  3-dimensionale, 348, 351, 386-87
  bipartito, 301
ABR *vedi* Albero bilanciato di ricerca
Agglomerato, 133
Agglomerazione
  primaria, 138
  secondaria, 138
Albero
  bilanciato di ricerca, 107
  binario, 97
  binario di ricerca, 107
  dei suffissi, 251
  delle scelte, 98, 336-37
  di cammini, 211
  di copertura, 162, 171
    BFS, 171
    DFS, 172
    limitato, 354
    minimo, 202, 271, 274, 282, 386
  genealogico, 62
  k-bilanciato, 112
  libero, 162
  ordinato, 62, 88
  perfettamente bilanciato, 112
  radicato, 89, 162
  realizzato con puntatori, 108, 128
  realizzato con vettore dei padri, 95
  Red-Black, 113
Algoritmica, XI
Algoritmo, XI, 5
  branch-&-bound, 368
  dei tre indiani, 294
  deterministico, 341
  di approssimazione, 360
  di Bellman-Ford-Moore, 216
  di Dijkstra, 214
  di Edmonds-Karp, 294
  di Euclide, 323
  di Floyd-Warshall, 259
  di Ford-Fulkerson, 294
  di Fredman-Tarjan, 216
  di Graham, 308
  di Johnson, 215
  di Karatsuba, 234
  di Kruskal, 274
  di McNaughton, 379
  di Pape-D'Esopo, 219
  di Prim, 276, 281
  di Rabin, 323
  di Strassen, 236
  enumerativo, 338
  euristico, 373
  non deterministico, 335
  ottimo, 370, 385
  polinomiale, 27
  probabilistico, 319
  pseudo-polinomiale, 357
  randomizzato *vedi* probabilistico
  superpolinomiale, 26

Allineamento
  globale, 266
    multiplo, 381
  locale, 266
  semiglobale, 266
Altezza, 89
Analisi di algoritmi, 20
  ammortizzata, 42
  per livelli, 36
  per tentativi, 41
and, 12
Antenato, 89
Appartenenza (ad un insieme), 60
Approssimazione assoluta, 361
Arco, 159
  all'indietro, 174
  di attraversamento, 174
  in avanti, 174
  incidente, 159
  non orientato, 160
  orientato,159
Array, 12
Azione elementare, 7

Backtrack, 305
Bin packing, 361
Bisection *vedi* Bisezione
Bisezione, 349, 374-76
Bit, 8
Blocco, 11
boolean, 12
Branch-&-bound, 368
Bucket *vedi* Cestino

Cammino, 160
  aumentante, 292
  ha miltoniano, 354
  massimo, 354
  minimo, 181, 211, 257-58, 296
    fra tutte le coppie, 257, 265, 387
    senza nodi ripetuti, 386
Campo, 12
Capacità, 291
  residua, 292
Caratterizzazione della soluzione, 209, 212, 306, 385
Cardinalità (di un insieme), 60
Catena, 161
  hamiltoniana, 354
Certificato polinomiale, 334
Cestino, 381
Chiavi, 61
Child *vedi* Figlio
choice, 335
Ciclo, 160
  euleriano, 300
  hamiltoniano, 354
Circuito, 161
  euleriano, 386-87
  hamiltoniano, 349, 350, 362, 375, 386-87
Classe di complessità, 25
Classificazione di un problema, 209
Clique *vedi* Cricca
Coda, 74
  con priorità, 185
    realizzata con alberi bilanciati, 187
    realizzata con heap, 187, 203, 215
    realizzata con liste, 167, 186
  realizzata con vettore circolare, 77
Codifica
  concisa, 339
  di Huffman, 279
Codominio, 61
Coefficiente binomiale, 247
Collisione, 132
Colorazione, 128, 332-33, 344, 365, 381, 386-87
  di intervalli, 281
Coloring *vedi* Colorazione
Commesso viaggiatore, 332-36, 338, 344, 350, 360-62, 365, 370-73, 375-76, 380
  con disuguaglianze triangolari, 362
Complessità *vedi* Tempo di calcolo
Componente, 161
  connessa, 161, 173
  fortemente connessa, 161
Compostezza, 323-24
Compressione dei percorsi, 194
Condizioni di Bellman, 212
Connessione, 161
  forte, 161
  co-$\mathbb{NP}$, 340
Convex Hull *vedi* Inviluppo convesso
Copertura esatta di insiemi, 348
Costo delle operazioni, 20
Counting Sort, 35
Cricca, 332-334, 336-39, 344-46, 349, 359, 362, 365, 379, 386
  di peso massimo, 379
Crittografia a chiave pubblica, 325

Dato, 6, 55
  di input, 6
  di output, 6
delete,13
ΔTSP, 362-65, 379
Dequeue, 219
Diametro di un grafo, 181
Dictionary *vedi* Dizionario

Differenza (insiemistica), 60
Dimensione cricca massima, 339
Dimensione dell'input, 26
Discendente, 89
Distanza, 211
disuguaglianze triangolari, 361
Divide-et-impera, 18, 39, 210, 225
Dizionario, 61, 107, 149
realizzato con alberi bilanciati, 154
realizzato con tabelle hash, 153
do, 11
Dominio, 61
Domino 335
limitato 331, 332, 342-47, 349, 351, 359, 386
lineare, 342, 386

ε-approssimazione, 361, 366
else, 11
Enumerazione, 338
tramite backtrack, 311
Esplorazione *vedi* Visita
Espressione Polinomiale Nulla, 320
Euristica, 373
del rango, 198
Eventi contabili, 33
Exact cover *vedi* Copertura esatta di insiemi

failure, 335
Fattore di carico, 141
FIFO, 74
Figlio, 89
destro, 97
sinistro, 97
First-Fit, 362
Decreasing, 362
Flusso, 288
massimo, 288, 291, 341, 387
saturante, 295
Foglia, 89
for, 9, 11
foreach, 11
Fratello, 89
Funzione, 14
hash, 134
Gioco del quindici, 377
Grado
entrante, 159
uscente, 159
Grafo
aciclico, 175, 177, 218
bipartito, 180
connesso, 161
di intervalli, 164, 180, 386
fortemente connesso, 161
non orientato, 160
orientato, 159
orientato aciclico, 182
pesato, 166
planare, 219
realizzato con matrice, 165
realizzato con vettori, 167
sparso, 166
Graph isomorphism *vedi* Isomorfismo tra grafi
Greedy, 270

Hash, 131
Hashing doppio, 137, 139, 142
Heap, 187
Heapsort, 192

if, 11
iif, 11
Inclusione (in un insieme), 60
Indice, 9
Insertion Sort, 28
Insieme, 60, 149
dominante, 221, 341
totale, 221
indipendente, 221, 349, 354, 360, 386
di intervalli, 273
di intervalli pesati, 251
realizzato con alberi bilanciati, 154
realizzato con alberi Red-Black, 112
realizzato con liste non ordinate, 151
realizzato con liste ordinate, 152
realizzato con tabelle hash, 153
realizzato con vettore booleano, 149
vuoto, 338
integer, 61
Interpolazione
per ricerca di un vettore, 226
Intersezione (insiemistica), 60
Intorno, 285
Invariante, 16
Inversione, 238, 242, 288
Inviluppo convesso, 305, 342
Invisita, 92, 305

Java Collections, XII, 65, 82, 123, 145, 154

Leaf *vedi* Foglia
LIFO, 74
Limitazione
inferiore, 25
superiore, 25
Lista
bidirezionale, 70
circolare, 71

con sentinella, 71
di adiacenza, 167
di trabocco, 137
monodirezionale, 70
realizzata con puntatori, 70
realizzata con vettori, 72
Livello
di dettaglio, 8
di un nodo di un albero, 94, 162
Longest Common Subsequence *vedi* Sottosequenza comune massimale
Lunghezza
di un cammino, 114, 160
di una catena, 161

Map *vedi* Dizionario
Massima sottostringa comune, 265
Matching *vedi* Abbinamento
Matrice, 12
di adiacenza, 165
nodi-nodi, 165
di incidenza nodi-archi, 166
Mediana, 320
Memoria secondaria, 124, 144, 236
Merge Sort, 29, 31, 41
polifase, 236
Merge-Find Set, 195
realizzato con foresta, 196
realizzato con lista, 196
Metodo di generazione hash
della divisione, 135
della moltiplicazione, 135
MFSET *vedi* Merge-Find Set
min-max-greedy, 374, 381
Minimo, 6, 9, 16-17
Moltiplicazione, 387
catena di matrici, 253
di matrici, 234
di polinomi, 232

new, 13
nil, 13
Nodo, 159
adiacente, 159
Non determinismo, 335
not, 12
$\mathbb{NP}$, 339-41
NP-completezza, 331, 344
forte, 359
Numero
di Fibonacci, 255
pseudo-casuale, 319

Ω, 24

O, 24
or, 12
Oracolo, 33
Ordinamento, 27, 185, 192-94, 226, 236
Counting Sort, 35
Heapsort, 192
in-place, 236
Insertion sort, 28
limitazione inferiore, 25, 35, 104
Merge Sort, 29
Pigeonhole Sort, 85
Quicksort, 227
Selection Sort, 27, 236
Shell Sort, 288
stabile, 236
Ordine
di grandezza, 24
classificazione, 26
di visita di un albero, 102
topologico, 176
Ottimo
globale, 286-87, 376
locale, 286-87, 376
Overflow, 44, 136

$\mathbb{P}$, 339-41
Padre, 89
Pagina di memoria secondaria, 124
Parametro
attuale, 14
formale, 14
Parent *vedi* padre
Partition *vedi* Partizione
Partizione, 350-51, 356, 360
Partizione di Eulero, 302
Pigeonhole Sort, 85
Pila, XII, 74
e ricorsione, 81
realizzata con vettore, 76
Poligono convesso, 305
Portata, 297
Postvisita, 92
Pozzo, 291
precondition, 15
Precondizione, 15
Previsita, 92
Primalità, 322-23, 387
Primality *vedi* Primalità
Priorità,185
crescente, 185
decrescente, 185
Problema
computazionale, 6
decisionale, 335

di ottimizzazione, 209, 248, 270-71, 274, 279, 285, 288
di ricerca, 145, 339
fortemente NP-completo, 359-60
NP-completo, 344, 347, 351, 354
Procedura, 14
Progettazione di algoritmi, 18, 209
Programmazione
dinamica, 210, 247
lineare, 166, 287, 332, 344, 347, 349
intera, 386
0/1 332, 344, 347, 349, 368, 386
Pseudo-polinomiale, 357, 386
Pseudocodice, XII, 8
sintassi, 10
Puntatori, 13, 63-64, 94, 99, 101

Queue *vedi* Coda
Quicksort, 227

Radice, 62, 89
random, 319
real, 12
Realizzazione, 64
Record, 12
Regola di Horner, 136
Relazioni di ricorrenza, 22
con partizione bilanciata, 40
lineari, 38
Rete di flusso, 291
residua, 292
return, 15
Ricerca, 6, 7, 17
binaria 9, 41, 107-8, 226-27, 238
con insuccesso, 141-43
con successo, 141-43
dicotomica *vedi* Ricerca binaria
locale, 285
per interpolazione, 226
tabù, 376
Ricorsione, 15, 232, 250
di coda, 15
Riducibilità polinomiale, 341
Ristrutturazione, 144
Root *vedi* Radice
RSA, 325

Satisfiability *vedi* Soddisfattibilità
Saturation degree, 384
Scansione
di una tabella hash, 137
esterna, 137
interna, 137
lineare, 138
pseudo casuale, 139
quadratica, 138
Scheduling, 180, 300, 349
multiprocessore, 349-50, 379
Selection *vedi* Selezione
Selection Sort, 27
Selezione, 320-21
Sentinella, 71
Sequenza, 57
unimodale, 239
Set *vedi* Insieme
Shell Sort, 288
Sibling *vedi* Fratello
Soddisfattibilità, 327, 333, 344, 346-47, 349, 351, 386
Soluzione
ammissibile, 209
ottima, 209
Somma di sottoinsieme, 348, 350, 358, 360-61, 366
Sorgente, 291, 300
Sorting *vedi* Ordinamento
Sottosequenza comune massimale, 262
Sottostruttura ottima, 248
Sovrapposizione di intervalli, 180
Specifica
semantica, 57
sintattica, 57
Spigolo *vedi* Arco
Stack *vedi* Pila
Standard Template Library, 66, 124, 145
Star alignment, 384
String matching, 249, 332
approssimato, 249
Struttura di dati, 56
realizzazione, 64
specifica, 57
Subset sum *vedi* Somma di sottoinsieme
success, 335-37
Sudoku, 314

Tabella
dei simboli, 132
hash, 36, 64, 133
Tecniche di progettazione, 207
Tempo di calcolo, 20
ammortizzato, 23, 275
nel caso medio, 23, 131, 141, 230
nel caso pessimo, 23, 141
Teorema
del flusso massimo/taglio minimo, 292
delle ricorrenze lineari
con partizione bilanciata, 40
di ordine costante, 38

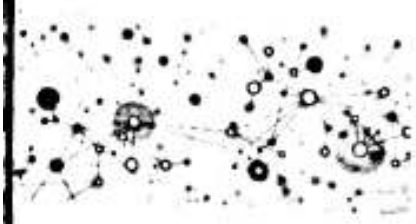
di Bellman, 212
di Cook-Levin, 343
di Eulero, 302
di Fermat, piccolo, 323
Testimone di compostezza, 323
then, 11
Θ, 24
this, 57
Tipo di dato, 55
astratto, 13, 55
composto, 13
primitivo, 57
Torri di Hanoi, 39, 79, 351
triangolo di Tartaglia, 248
TSP *vedi* Commesso viaggiatore

Unione (insiemistica), 60
Unione di insiemi disgiunti, 195

Valutazione di un polinomio, 136
Vertice *vedi* Nodo
Vettore, 12, 56
dei padri, 95
di adiacenza, 167
Visita
anticipata, 94
BFS, 169
DFS, 169
in ampiezza, 92, 169
in profondità, 92, 169
per livelli, 169
posticipata, 94
simmetrica, 94

while, 11

Zaino, 260-2, 271, 281, 283, 348, 350, 359-60, 386
0-1, *vedi* Zaino
intero, *vedi* Zaino
reale, 271, 386



CittàStudi
EDIZIONI

L'algoritmica è quella branca dell'informatica che riguarda la definizione e la progettazione degli algoritmi, l'analisi della loro correttezza ed efficienza; la dimostrazione delle loro limitazioni e complessità e lo studio dei dati da essi elaborati.
Algoritmi e strutture di dati, grazie all'accurato lavoro degli autori, alla loro pluriennale esperienza didattica e scientifica e alla rigorosa ed esauriente trattazione della materia, si pone come il punto di riferimento dell'algoritmica.
Il ricco apparato didattico (fatto di numerosissimi teoremi dimostrati, di una folta serie di esercizi e di molti esempi svolti che completano e integrano ogni capitolo) e la funzionale veste grafica, infine, permettono agli studenti una verifica costante del livello del proprio studio.

Alan Bertossi è professore ordinario presso il Dipartimento di Informatica - Scienza e Ingegneria (DISI) dell'Università di Bologna.

Alberto Montresor è professore associato presso il Dipartimento di Ingegneria e Scienza dell'Informazione dell'Università di Trento.

Alla pagina web www.cittastudi.it sono disponibili le slide di supporto per i docenti e, per gli studenti, algoritmi scritti in C++ e Java, un programma di visualizzazione di tutti gli algoritmi presenti nel testo e ulteriori esercizi aggiuntivi. Per accedere alle risorse web, basta inserire il codice ISBN del libro nel campo «Cerca le risorse web del tuo libro».

In copertina: © iStockphoto.

ISBN 978-88-251-7395-6

€ 31,00