Pierluigi Crescenzi • Giorgio Gambosi
Roberto Grossi • Gianluca Rossi

STRUTTURE DI DATI E ALGORITMI

Progettazione, analisi e programmazione

Seconda edizione

Sommario

Prefazione VII

Introduzione **1**

Capitolo 1 Array, liste e alberi **7**

1.1 Sequenze lineari **8**
1.1.1 Modalità di accesso 8
1.1.2 Allocazione della memoria 9
1.1.3 Array di dimensione variabile 10

1.2 Opus libri: scheduling della CPU **12**
1.2.1 Ordinamento per selezione 14
1.2.2 Ordinamento per inserimento 16

1.3 Gestione di liste **18**
1.3.1 Inserimento e cancellazione 19
1.3.2 Liste doppie 21

1.4 Alberi **23**
1.4.1 Alberi binari 23
1.4.2 Alberi cardinali e ordinali 25

1.5 Esercizi **28**

Capitolo 2 Pile e code **31**

2.1 Pile **32**
2.1.1 Implementazione di una pila mediante un array 32
2.1.2 Implementazione di una pila mediante una lista 34

2.2 Opus libri: Postscript e notazione postfissa **35**

2.3 Code **42**
2.3.1 Implementazione di una coda mediante un array 42
2.3.2 Implementazione di una coda mediante una lista 44

2.4 Code con priorità: heap **45**
2.4.1 Definizione di heap 46
2.4.2 Implementazione di uno heap implicito 48
2.4.3 Heapsort 53

2.5 Esercizi **57**

Capitolo 3 Divide et impera 59

3.1 Ricorsione e paradigma del divide et impera 60

3.2 Relazioni di ricorrenza e teorema fondamentale 63

3.3 Ricerca di una chiave 65

3.4 Ordinamento e selezione per distribuzione 69

3.5 Moltiplicazione veloce di due numeri interi 73

3.6 Opus libri: grafica e moltiplicazione di matrici 77

3.6.1 Moltiplicazione veloce di due matrici 81

3.7 Opus libri: il problema della coppia più vicina 83

3.8 Algoritmi ricorsivi su alberi binari 87

3.8.1 Visite di alberi 91

3.8.2 Alberi completamente bilanciati 92

3.8.3 Nodi cardine di un albero binario 95

3.9 Esercizi 97

Capitolo 4 Dizionari 101

4.1 Dizionari 102

4.2 Liste e dizionari 103

4.3 Opus libri: funzioni hash e peer-to-peer 105

4.3.1 Tabelle hash: liste di trabocco 108

4.3.2 Tabelle hash: indirizzamento aperto 110

4.4 Opus libri: kernel Linux e alberi binari di ricerca 114

4.4.1 Alberi binari di ricerca 114

4.4.2 AVL: alberi binari di ricerca bilanciati 118

4.5 Opus libri: liste invertite e trie 124

4.5.1 Trie o alberi digitali di ricerca 131

4.5.2 Trie compatti 139

4.6 Esercizi 144

Capitolo 5 Casualità e ammortamento 147

5.1 Ordinamento randomizzato per distribuzione 148

5.1.1 Alternativa al teorema fondamentale delle ricorrenze 150

5.2 Dizionario basato su liste randomizzate 152

5.3 Unione e appartenenza a liste disgiunte 158

5.4 Liste ad auto-organizzazione 162

5.5 Tecniche di analisi ammortizzata 167

5.6 Esercizi 170

Capitolo 6 Programmazione dinamica 171

6.1 Il paradigma della programmazione dinamica 172

6.2 Problema del resto 174

6.3 Opus libri: sotto-sequenza comune più lunga 179

6.4 Partizione di un insieme di interi 185

6.5 Problema della bisaccia 188

6.6 Massimo insieme indipendente in un albero 193

6.7 Alberi di ricerca ottimi 196

6.8 Pseudo-polinomialità e programmazione dinamica 200

6.9 Esercizi 201

Capitolo 7 Grafi 203

7.1 Grafi 204

7.1.1 Alcuni problemi su grafi 210

7.1.2 Rappresentazione di grafi 212

7.1.3 Cammini minimi, chiusura transitiva e prodotto di matrici 216

7.2 Opus libri: Web crawler e visite di grafi 218

7.2.1 Visita in ampiezza di un grafo 219

7.2.2 Visita in profondità di un grafo 225

7.3 Applicazioni delle visite di grafi 229

7.3.1 Grafi diretti aciclici e ordinamento topologico 229

7.3.2 Componenti (fortemente) connesse 232

7.4 Opus libri: routing su Internet e cammini minimi 240

7.4.1 Problema della ricerca di cammini minimi su grafi 242

7.4.2 Cammini minimi in grafi con pesi positivi 244

7.4.3 Cammini minimi in grafi pesati generali 250

7.5 Opus libri: data mining e minimi alberi ricoprenti 257

7.5.1 Problema della ricerca del minimo albero di ricoprimento 259

7.5.2 Algoritmo di Kruskal 261

7.5.3 Algoritmo di Jarník-Prim 264

7.6 Esercizi 267

Capitolo 8 NP-completezza e approssimazione **271**

8.1 Problemi intrattabili **272**

8.2 Classi P e NP **272**

8.3 Riducibilità polinomiale **276**

8.4 Problemi NP-completi **281**

8.5 Teorema di Cook-Levin **283**

8.6 Problemi di ottimizzazione **284**

8.7 Generazione esaustiva e backtrack **285**

8.8 Esempi e tecniche di NP-completezza **287**

8.8.1 Tecnica di sostituzione locale 288

8.8.2 Tecnica di progettazione di componenti 289

8.8.3 Tecnica di similitudine 292

8.8.4 Tecnica di restrizione 293

8.9 Come dimostrare risultati di NP-completezza **294**

8.10 Algoritmi di approssimazione **296**

8.11 Opus libri: il problema del commesso viaggiatore **298**

8.11.1 Problema del commesso viaggiatore su istanze metriche 300

8.11.2 Paradigma della ricerca locale 303

8.12 Esercizi **307**

Indice analitico **309**

Prefazione

Essenziale e diretto agli studenti che intendono programmare efficientemente i computer: potremmo definire in questo modo sintetico l'obiettivo didattico di questo testo sugli algoritmi e sulle strutture dei dati, allineandosi a come sono organizzati gli insegnamenti introduttivi di algoritmi e programmazione nei corsi di studio universitari italiani. Per fornire un quadro completo, osserviamo che esistono principalmente due approcci interessanti allo studio degli algoritmi e altrettante tipologie di testi molto validi per studiarli.

Il primo approccio si basa sulla teoria degli algoritmi e presenta le idee in forma distillata studiandone le proprietà matematiche di correttezza e complessità, cercando anche di individuare i limiti computazionali dei problemi studiati. La descrizione degli algoritmi è fatta a parole e mediante lo pseudo-codice, per evidenziare meglio le qualità salienti della computazione senza perdersi nei dettagli di un particolare codice di programmazione. Questo approccio viene indirizzato allo studente che sia sufficientemente abile nella programmazione, in grado di prendere una descrizione astratta dell'algoritmo e di tradurla in codice funzionante e ottimizzato.

Il secondo approccio si basa sulla pratica dell'ingegnerizzazione degli algoritmi (*algorithm engineering*), finalizzata a sfruttare al massimo le caratteristiche del sistema computazionale sottostante per ottenere la massima efficienza per gli algoritmi e le strutture di dati sviluppati. Le proprietà matematiche vengono qui utilizzate per definire algoritmi molto efficienti anche "in pratica", validando così, anche a livello sperimentale, la teoria che li studia, o anche per ottenere algoritmi che risultino efficienti sotto opportune condizioni pratiche, anche se non lo sono necessariamente dal punto di vista teorico. In questo caso, ci si rivolge allo studente più esperto, non solo abile programmatore, ma avvezzo alla teoria di base degli algoritmi e dei sistemi di calcolo, in grado di produrre codice di alta qualità.

Il nostro testo segue un terzo approccio, a metà strada tra i due, rivolto allo studente alle prime armi che sta imparando a programmare e vuole comunque iniziare a studiare problemi algoritmici che forniscano una qualche sfida computazionale: uno studente che, pur essendo in grado di comprendere la teoria, non sa ancora tradurla in programmi. Particolare attenzione è dedicata al codice stesso degli algoritmi, che può essere letto sia come pseudo-codice che ne distilla le idee principali, sia come codice effettivo che può essere trasformato in codice funzionante con minimo sforzo. Della simbiosi così risultante beneficia lo studente, il quale rafforza sia le proprie abilità programmative che le capacità di progettazione e di analisi teorica degli algoritmi. La scelta degli argomenti è stata infatti ispirata da queste linee guida per produrre codice leggibile e sintetico e, al contempo, fornire le basi teoriche della materia. Riteniamo che questo sia uno dei punti di forza di questa nuova edizione.

Un altro punto di forza è la visualizzazione degli algorimi proposti. Il testo contiene numerosi esempi stampati su carta, come da tradizione. Tuttavia si avvale anche della tecnologia di visualizzazione per produrre nuovi esempi che possono essere ottenuti cambiando semplicemente i dati di ingresso: i vari strumenti sono reperibili sul sito web associato al testo e le visualizzazioni sono disponibili in vari formati elettronici che possono essere prodotti e scaricati sul proprio computer, *smartphone* o dispositivo portatile.

Qui subentra un ulteriore punto di forza, ossia la possibilità di accedere al sito web, all'indirizzo **http://hpe.pearson.it/crescenzi**, sia per utilizzare le visualizzazioni che per ottenere ulteriore materiale didattico. La scelta editoriale è stata infatti quella di produrre un libro compatto e snello che copra comunque gli argomenti fondamentali di un insegnamento di Algoritmi e Strutture Dati da 6, 9 o anche 12 crediti universitari, associandolo eventualmente a un laboratorio di Programmazione e *Problem Solving*. Viene quindi utilizzata la pubblicazione elettronica per introdurre ulteriori argomenti: per esempio è disponibile un intero capitolo che fornisce una panoramica su calcolabilità e complessità e può venire utilizzato indipendentemente da questa edizione. Alcuni degli esercizi sono interamente svolti per gli studenti nel testo stesso e altri ancora lo sono nel sito web, mentre per i soli docenti che adottano il testo viene fornito uno spunto di risoluzione per tutti gli esercizi elencati alla fine di ogni capitolo.

Ringraziamenti

Gli autori sono molto riconoscenti a Stefano Maggiolo per aver curato le bozze del libro e ai colleghi Anna Bernasconi, Paolo Ferragina, Fabrizio Luccio e Linda Pagli per aver ideato, per gli esami del corso di Algoritmica dell'Università di Pisa, diversi degli esercizi presenti nel libro. Siamo profondamente grati a tutti gli studenti dei nostri corsi, che con il loro entusiasmo e con le loro osservazioni ci hanno permesso di migliorare la seconda edizione e l'ambiente di visualizzazione. Quest'ultimo non avrebbe potuto essere fruibile senza il fondamentale lavoro di tesi di Carlo Nocentini, a cui va il nostro caloroso ringraziamento. Ringraziamo i seguenti colleghi e amici per aver letto, con spirito critico, versioni preliminari di alcuni capitoli della prima edizione: Anna Bernasconi, Paolo Cignoni, Valentina Ciriani, Andrea Clementi, Miriam Di Ianni, Paolo Ferragina, Gianni Franceschini, Antonio Gullì, Michele Loreti, Fabrizio Luccio, Alessio Malizia, Donatella Merlini, Linda Pagli, Paolo Penna, Nadia Pisanti, Guido Proietti, Geppino Pucci, Romeo Rizzi e Cecilia Verri. Ringraziamo infine il gruppo di lavoro di Pearson Italia e, in particolare, Valentina Favata, Alessandra Piccardo e Carmelo Giarratana, per il loro aiuto durante la stesura della seconda edizione e per averci sempre perdonato le innumerevoli scadenze mancate.

Pierluigi Crescenzi
Giorgio Gambosi
Roberto Grossi
Gianluca Rossi

Introduzione

Ottimi testi su algoritmi presumono che il lettore abbia già sviluppato una capacità di astrazione tale da poter recepire la teoria degli algoritmi con un taglio squisitamente matematico. Altri ottimi testi, ritenendo che il lettore abbia la capacità di intravedere quali siano gli schemi programmativi adatti alla risoluzione dei problemi, danno più spazio agli aspetti implementativi con un taglio pragmatico e orientato alla programmazione. Il nostro libro cerca di combinare questi due approcci, ritenendo che lo studente abbia già imparato i rudimenti della programmazione, ma non sia ancora in grado di astrarre i concetti e di riconoscere gli schemi programmativi da utilizzare. Mirato ai corsi dei primi due anni nelle lauree triennali, il testo segue un approccio costruttivistico che agisce a due livelli, entrambi strettamente necessari per un uso corretto del libro:

- partendo da problemi reali, lo studente viene guidato a individuare gli schemi programmativi più adatti: gli algoritmi presentati sono descritti anche in uno pseudocodice molto vicino al codice reale (ma comunque di facile comprensione);
- sviluppato il codice, ne vengono analizzate le proprietà con un taglio più astratto e matematico, al fine di distillare l'algoritmo corrispondente e studiarne la complessità computazionale.

A supporto di questo approccio costruttivistico, il sito web mette a disposizione degli studenti e dei docenti l'ambiente di visualizzazione ALVIE, con il quale ogni

algoritmo presentato nel testo viene mostrato in azione, rendendo possibile sia eseguirlo su qualunque insieme di dati di esempio sia modificarne il comportamento, se necessario.

Gli argomenti classici dei corsi introduttivi di algoritmi e strutture di dati (come array, liste, alberi e grafi, ricorsione, divide et impera, randomizzazione e analisi ammortizzata, programmazione dinamica e algoritmi golosi) sono integrati con argomenti e applicazioni collegate alle tecnologie più recenti. Uno degli obiettivi del libro è infatti quello di integrare teoria e pratica in modo proficuo per l'apprendimento, fornendo al contempo agli studenti una chiara percezione della significatività dei concetti e delle tecniche introdotte nella risoluzione di problemi attuali. Quest'integrazione tra tecniche algoritmiche e applicazioni dà luogo nel testo a momenti di vera e propria opera di progettazione di strutture di dati e di algoritmi, denominata *opus libri*.[1] L'approccio costruttivistico, a cui abbiamo fatto riferimento in precedenza, viene applicato in tali casi, descrivendo in modo semplice le applicazioni e mostrandone l'impatto nella progettazione efficiente ai fini delle prestazioni ottenute, le quali sono misurate in relazione a un modello di calcolo di riferimento (nel nostro caso, la *Random Access Machine*, come spiegato nel prossimo paragrafo).

La trattazione non è vincolata a un linguaggio di programmazione specifico, ma risulta comprensibile sia agli studenti con maggiore familiarità per i linguaggi strutturati di tipo procedurale, sia a quelli che posseggono una buona conoscenza della programmazione a oggetti. Sebbene l'uso di ALVIE (realizzato in Java), per l'introduzione al suo interno di nuove visualizzazioni, consenta al docente interessato di introdurre le strutture di dati e gli algoritmi su esse operanti usando l'approccio tipico della programmazione a oggetti, il docente stesso può decidere o meno se utilizzare tale paradigma programmativo, senza pregiudicare la fruibilità degli argomenti trattati nel testo.

Rispetto alla prima edizione, questa edizione presenta diverse novità. Anzitutto, il materiale trattato è stato riorganizzato in base a una struttura non più rigidamente orientata alle strutture di dati. In secondo luogo il testo è stato arricchito con molti esempi ed esercizi svolti e il numero di esercizi proposti è significativamente aumentato. Avendo inserito gli esempi di esecuzione di algoritmi, i riferimenti espliciti ad ALVIE sono stati graficamente modificati, riducendoli a un'icona affiancata ai codici descritti nel testo. Infine, siamo pienamente coscienti che non esiste il libro perfetto in grado di soddisfare tutti i docenti: il sapere odierno è sempre più dinamico, variegato e distribuito, e un semplice libro non può catturare le mille sfaccettature di una disciplina scientifica in continua evoluzione. Per

[1] Il termine indica un aspetto di progettazione *hands-on* ricalcando il noto termine *opus dei*. Giochiamo con quest'ultimo termine: come il divino permette di progredire spiritualmente attraverso la sua azione, così un libro permette di progredire mentalmente attraverso l'applicazione dei suoi contenuti.

questo al libro è associato un sito web in cui i docenti possono trovare, oltre all'ambiente di visualizzazione, ulteriore materiale didattico (come integrazioni al testo, esercizi svolti e lucidi in PowerPoint e LaTeX).

Modello RAM e complessità computazionale

La valutazione della complessità di un algoritmo e la classificazione della complessità di un problema fanno solitamente riferimento al concetto intuitivo di **passo elementare**: diamo ora una specifica formale di tale concetto, attraverso una breve escursione nella struttura logica di un calcolatore.

L'idea di memorizzare sia i dati che i programmi come sequenze binarie nella memoria del calcolatore è dovuta principalmente al grande e controverso scienziato ungherese John von Neumann[2] negli anni '50, il quale si ispirò alla macchina universale di Turing. I moderni calcolatori mantengono una struttura logica simile a quella introdotta da von Neumann, di cui il modello RAM (*Random Access Machine* o macchina ad accesso diretto) rappresenta un'astrazione: tale modello consiste in un processore di calcolo a cui viene associata una memoria di dimensione illimitata, in grado di contenere sia i dati che il programma da eseguire. Il processore dispone di un'unità centrale di elaborazione e di due registri, ovvero il contatore di programma che indica la prossima istruzione da eseguire e l'accumulatore che consente di eseguire le seguenti istruzioni elementari:[3]

- operazioni aritmetiche: somma, sottrazione, moltiplicazione, divisione;
- operazioni di confronto: minore, maggiore, uguale e così via;
- operazioni logiche: and, or, not e così via;
- operazioni di trasferimento: lettura e scrittura da accumulatore a memoria;
- operazioni di controllo: salti condizionati e non condizionati.

Allo scopo di analizzare le prestazioni delle strutture di dati e degli algoritmi presentati nel libro, seguiamo la convenzione comunemente adottata di assegnare un **costo uniforme** alle suddette operazioni. In particolare, supponiamo che ciascuna di esse richieda un tempo *costante* di esecuzione, indipendente dal numero dei dati memorizzati nel calcolatore. Il costo computazionale dell'esecuzione di un algoritmo, su una specifica istanza, è quindi espresso in termini di **tempo**, ovvero il numero di istruzioni elementari eseguite, e in termini di **spazio**, ovvero il massimo numero di celle di memoria utilizzate durante l'esecuzione (*oltre* a quelle occupate dai dati in ingresso).

[2] Il saggio *L'apprendista stregone* di Piergiorgio Odifreddi descrive la personalità di von Neumann.

[3] Notiamo che le istruzioni di un linguaggio ad alto livello come C, C++ e JAVA, possono essere facilmente tradotte in una serie di tali operazioni elementari.

Per ogni dato problema, è noto che esistono infiniti algoritmi che lo risolvono, per cui il progettista si pone la questione di selezionare il migliore in termini di **complessità in tempo** e/o di **complessità in spazio**. Entrambe le complessità sono espresse in notazione **asintotica** in funzione della dimensione `n` dei dati in ingresso, ignorando così le costanti moltiplicative e gli ordini inferiori.[4] Ricordiamo che, in base a tale notazione, data una funzione `f`, abbiamo che:

- $O(f(n))$ denota l'insieme di funzioni `g` tali che esistono delle costanti `c`, $n_0 > 0$ per cui vale $g(n) \leq cf(n)$, per ogni $n > n_0$ (l'appartenenza di `g` viene solitamente indicata con $g(n) = O(f(n))$);
- $\Omega(f(n))$ denota l'insieme di funzioni `g` tali che esistono delle costanti `c`, $n_0 > 0$ per cui vale $g(n) \geq cf(n)$, per ogni $n > n_0$ (l'appartenenza di `g` viene solitamente indicata con $g(n) = \Omega(f(n))$);
- $\Theta(f(n))$ denota l'insieme di funzioni `g` tali che $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$ (l'appartenenza di `g` viene solitamente indicata con $g(n) = \Theta(f(n))$);
- $o(f(n))$ denota l'insieme di funzioni `g` tali che $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ (l'appartenenza di `g` viene solitamente indicata con $g(n) = o(f(n))$).

Solitamente, si cerca di minimizzare la complessità asintotica in tempo e, a parità di costo temporale, la complessità in spazio: la motivazione è che lo spazio può essere riusato, mentre il tempo è irreversibile.[5]

Nella complessità al **caso pessimo** o **peggiore** consideriamo il costo massimo su tutte le possibili istanze di dimensione `n`, mentre nella complessità al **caso medio** consideriamo il costo mediato tra tali istanze. La maggior parte degli algoritmi presentati in questo libro saranno analizzati facendo riferimento al caso pessimo, ma saranno mostrati anche alcuni esempi di valutazione del costo al caso medio.

Diamo ora una piccola guida per valutare al caso pessimo la complessità in tempo di alcuni dei costrutti di programmazione più frequentemente usati nel libro (come ogni buona catalogazione, vi sono le dovute eccezioni che saranno illustrate di volta in volta).

- Le singole operazioni logico-aritmetiche e di assegnamento hanno un costo costante.
- Nel costrutto condizionale

```
IF (guardia) { blocco1 } ELSE { blocco2 }
```

  uno solo tra i rami viene eseguito, in base al valore di `guardia`. Non potendo prevedere in generale tale valore e, quindi, quale dei due blocchi sarà eseguito, il costo di tale costrutto è pari a

```
costo(guardia) + max{costo(blocco1), costo(blocco2)}
```

- Nel costrutto iterativo

```
FOR (i = 0; i < m; i = i + 1) { corpo }
```

  sia t_i il costo dell'esecuzione di `corpo` all'iterazione `i` del ciclo (come vedremo in seguito, non è detto che `corpo` debba avere sempre lo stesso costo a ogni iterazione). Il costo risultante è proporzionale a

$$m + \sum_{i=0}^{m-1} t_i$$

  (assumendo che `corpo` non modifichi la variabile `i`).

- Nei costrutti iterativi

```
WHILE (guardia) { corpo }
DO { corpo } WHILE (guardia);
```

  sia `m` il numero di volte in cui `guardia` è soddisfatta. Sia t'_i il costo della sua valutazione all'iterazione `i` del ciclo, e t_i il costo di `corpo` all'iterazione `i`. Poiché `guardia` viene valutata una volta in più rispetto a `corpo` nel primo costrutto e lo stesso numero di volte nel secondo costrutto, in entrambi i casi abbiamo che, assumendo che i costi t'_i e t_i siano positivi, il costo totale è al più

$$\sum_{i=0}^{m} (t'_i + t_i)$$

  (notiamo che, di solito, la parte difficile rispetto alla valutazione del costo per il ciclo `FOR`, è fornire una stima del valore di `m`).

- Il costo della chiamata a funzione è dato da quello del corpo della funzione stessa più quello dovuto al calcolo degli argomenti passati al momento dell'invocazione (come vedremo, nel caso di funzioni ricorsive, la valutazione del costo sarà effettuata in seguito mediante la risoluzione delle relative equazioni di ricorrenza).
- Infine, il costo di un blocco di istruzioni e costrutti visti sopra è pari alla somma dei costi delle singole istruzioni e dei costrutti, secondo quanto appena discusso.

Osserviamo che la valutazione asintotica del costo di un algoritmo serve a identificare algoritmi chiaramente inefficienti *senza* il bisogno di implementarli e sperimentarli. Per gli algoritmi che risultano invece efficienti (da un punto di vista di analisi della loro complessità), occorre tener conto del particolare sistema che

[4] Un nostro collega ama usare la seguente metafora: un miliardario rimane tale sia che possegga un miliardo di euro che ne possegga nove, o che possegga ulteriori diversi milioni (le costanti moltiplicative negli ordini di grandezza e gli ordini inferiori scompaiono con le notazioni asintotiche O, Ω e Θ).

[5] In alcune applicazioni, come vedremo, lo spazio è importante quanto il tempo, per cui cercheremo di minimizzare entrambe le complessità con algoritmi più sofisticati.

intendiamo usare (piattaforma *hardware* e livelli di memoria, sistema operativo, linguaggio adottato, compilatore e così via). Questi aspetti sono volutamente ignorati nel modello RAM per permettere una prima fase di selezione ad alto livello degli algoritmi promettenti, che però necessitano di un'ulteriore indagine sperimentale che dipende anche dall'applicazione che intendiamo realizzare: come ogni modello, anche la RAM non riesce a catturare le mille sfaccettature della realtà.

Limiti superiori e inferiori

Per un dato problema computazionale Π, consideriamo un qualunque algoritmo `A` di risoluzione. Se `A` richiede `t(n)` tempo per risolvere una generica istanza di Π di dimensione `n`, diremo che $O(t(n))$ è un **limite superiore** alla complessità in tempo del problema Π. Lo scopo del progettista è quello di riuscire a trovare l'algoritmo `A` con il migliore tempo `t(n)` possibile.

A tal fine, quando riusciamo a dimostrare con argomentazioni combinatorie che *qualunque* algoritmo `A'` richiede *almeno* tempo `f(n)` per risolvere Π su un'istanza generica di dimensione `n`, asintoticamente per infiniti valori di `n`, diremo che $\Omega(f(n))$ è un **limite inferiore** alla complessità in tempo del problema Π. In tal caso, nessun algoritmo può richiedere asintoticamente meno di $O(f(n))$ tempo per risolvere Π.

Ne deriva che l'algoritmo `A` è **ottimo** se $t(n) = O(f(n))$, ovvero se la complessità in tempo di `A` corrisponde dal punto di vista asintotico al limite inferiore di Π. Ciò ci permette di stabilire che la **complessità computazionale del problema** è $\Theta(f(n))$. Notiamo che spesso la complessità computazionale di un problema combinatorio Π viene confusa con quella di un suo algoritmo risolutore `A`: in realtà ciò è corretto se e solo se `A` è ottimo.

Per quanto riguarda la complessità in spazio, `s(n)`, possiamo procedere analogamente al tempo nella definizione di limite superiore e inferiore, nonché di ottimalità. Da ricordare che lo spazio `s(n)` misura il numero di locazioni di memoria necessarie a risolvere il problema Π, *oltre* a quelle richieste dai dati in ingresso. Per esempio, gli algoritmi **in loco** sono caratterizzati dall'usare soltanto spazio $s(n) = O(1)$.

CAPITOLO

1 Array, liste e alberi

Il modo più semplice per aggregare dei dati elementari consiste nel disporli uno di seguito all'altro a formare una sequenza lineare, identificando ciascun dato con la posizione occupata. In questo capitolo studieremo tale disposizione descrivendo due diversi modi di realizzarla, l'accesso diretto e l'accesso sequenziale, che riflettono l'allocazione della sequenza nella memoria del calcolatore. Studieremo, inoltre, il problema dell'ordinamento mostrando due semplici algoritmi quadratici per risolvere tale problema. Infine, descriveremo come organizzare i dati in strutture gerarchiche introducendo le nozioni di albero binario, albero cardinale e albero ordinale.

1.1 Sequenze lineari
1.2 Opus libri: scheduling della CPU
1.3 Gestione di liste
1.4 Alberi
1.5 Esercizi

1.1 Sequenze lineari

Una **sequenza lineare** è un insieme finito di elementi disposti consecutivamente in cui ognuno ha associato un indice di posizione in modo univoco. Seguendo la convenzione di enumerare gli elementi a partire da 0, indichiamo una sequenza lineare di n elementi con la notazione $a_0, a_1, ..., a_{n-1}$, dove la posizione j contiene il $(j+1)$-esimo elemento rappresentato da a_j (per $0 \le j \le n-1$).

Nel disporre gli elementi in una sequenza, viene ritenuto importante il loro ordine relativo: quindi, la sequenza ottenuta invertendo l'ordine di due qualunque elementi è generalmente diversa da quella originale. Per esempio, consideriamo la parola `algoritmo`, vista come una sequenza di $n = 9$ caratteri $a_0, a_1, ..., a_8$ = a, l, g, o, r, i, t, m, o. Se invertiamo gli elementi a_0 e a_1, otteniamo la parola `lagoritmo` che nel corrente dizionario italiano non ha alcun significato. Se a partire da quest'ultima parola invertiamo gli elementi a_1 e a_3, otteniamo la parola `logaritmo`, che indica una nota funzione matematica.

ESEMPIO 1.1

Un esempio dell'importanza dell'ordine relativo in una sequenza lineare sono le sequenze di transazioni bancarie. Supponiamo che esistano solo due tipi di transazioni, ovvero prelievi e depositi, e che non sia possibile prelevare una somma maggiore del saldo. A partire da un saldo nullo, consideriamo la seguente sequenza di transazioni: deposita 10, deposita 10, preleva 15, preleva 5. Se invertiamo le ultime due transazioni, la nuova sequenza non genera errori in quanto il saldo è sempre maggiore oppure uguale alla quantità che viene prelevata. Se, invece, invertiamo la seconda e la terza transazione, otteniamo una sequenza che genera un errore in quanto stiamo cercando di prelevare 15, quando il saldo è pari a 10.

1.1.1 Modalità di accesso

L'operazione più elementare su una sequenza lineare consiste certamente nell'accesso ai suoi singoli elementi, specificandone l'indice di posizione. Per esempio, nella sequenza $a_0, a_1, \ldots, a_8$ = a, l, g, o, r, i, t, m, o, tale operazione restituisce a_7 = m nel momento in cui viene richiesto l'elemento in posizione 7.

L'accesso agli elementi di una sequenza lineare a viene generalmente eseguito in due modalità. In quella ad **accesso diretto**, dato un indice i, accediamo direttamente all'elemento a_i della sequenza senza doverla attraversare. In altre parole, l'accesso diretto ha un costo computazionale uniforme, indipendente dall'indice di posizione i. Nel seguito chiameremo **array** le sequenze lineari ad accesso diretto e, coerentemente con la sintassi dei più diffusi linguaggi di programmazione, indicheremo con a[i] il valore dell'(i+1)-esimo elemento di un array a. L'altra modalità consiste nel raggiungere l'elemento desiderato attraversando la sequenza a partire da un suo estremo, solitamente il primo elemento. Tale modalità, detta ad **accesso sequenziale**, ha un costo O(i) proporzionale alla posizione i dell'elemento cui si desidera accedere: d'ora in poi chiameremo **liste** le sequenze lineari ad accesso sequenziale. Notiamo però che, una volta raggiunto l'elemento a_i, il costo di accesso ad a_{i+1} è O(1). Generalizzando, il costo è O(k) per accedere ad a_{i+k} partendo da a_i.

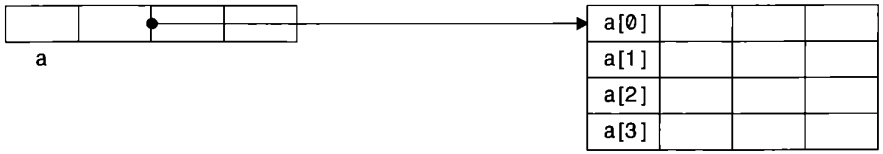
I due modi di realizzare l'accesso agli elementi di una sequenza lineare non devono assolutamente essere considerati equivalenti, vista la differenza di costo computazionale. Entrambe le modalità presentano pro e contro per cui non è possibile dire in generale che una sia preferibile all'altra: tale scelta dipende dall'applicazione che vogliamo realizzare o dal problema che dobbiamo risolvere.

1.1.2 Allocazione della memoria

La descrizione degli algoritmi che fanno uso di array e di liste dovrebbe prescindere dalla specifica allocazione dei dati nella memoria fisica del calcolatore. Tuttavia, una breve digressione su questo argomento permette di comprendere meglio la differenza di costo quando accediamo a un elemento di un array rispetto a un elemento di una lista. Gli array e le liste corrispondono a due modi diversi di allocare la memoria di un calcolatore. Nel caso degli array, le locazioni di memoria associate a elementi consecutivi sono contigue. Il nome dell'array corrisponde a un indirizzo che specifica dove si trova la locazione di memoria contenente l'inizio dell'array (tale inizio viene identificato con il primo elemento dell'array a[0]). Per accedere all'elemento a[i] è dunque sufficiente sommare a tale indirizzo i volte il numero di byte necessari a memorizzare un singolo elemento. Ciò giustifica l'affermazione fatta in precedenza che, nel caso di accesso diretto, il costo dell'operazione di accesso è O(1), in quanto è indipendente dall'indice di posizione dell'elemento desiderato (assumendo, naturalmente, che le operazioni di somma e moltiplicazione effettuate richiedano tutte tempo costante).

ESEMPIO 1.2

Consideriamo l'array a di 4 elementi di 4 byte ciascuno mostrato nella seguente figura.



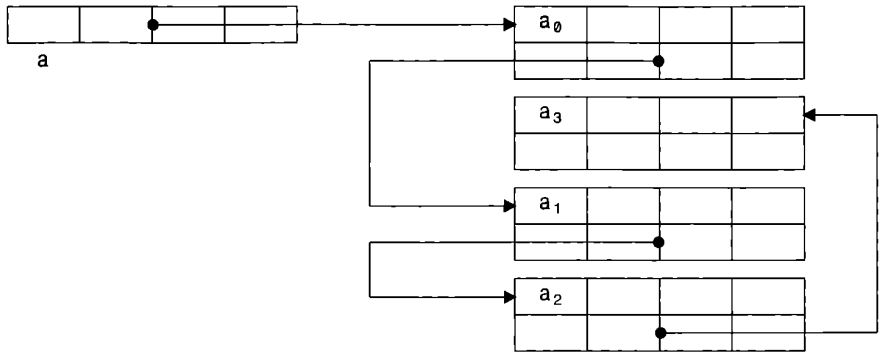
Se 512 è l'indirizzo contenuto in a, allora a[0] (rispettivamente, a[1], a[2] e a[3]) si trova nella locazione di memoria con indirizzo 512 (rispettivamente, 516, 520 e 524).

Differentemente dagli array, gli elementi delle liste sono allocati in locazioni di memoria non necessariamente contigue. Quest'allocazione deriva dal fatto che la memoria per le liste viene gestita dinamicamente durante la computazione, quando i vari elementi sono inseriti e cancellati (in un modo che non è possibile prevedere prima della computazione stessa). Per questo motivo, ogni elemento deve memorizzare, oltre al proprio valore, anche l'indirizzo dell'elemento suc-

cessivo. Il nome della lista corrisponde a un indirizzo che specifica dove si trova la locazione di memoria contenente il primo elemento della lista. Per accedere ad a_i è necessario partire dal primo elemento e scandire uno dopo l'altro tutti quelli che precedono a_i nella lista: quindi, come detto in precedenza, l'accesso a un elemento di una lista ha un costo proporzionale all'indice della sua posizione (in generale, se partiamo da a_i, l'accesso ad a_{i+k} richiede $O(k)$ passi).

ESEMPIO 1.3

Consideriamo la lista a di 4 elementi di 4 byte ciascuno mostrata nella seguente figura.



Per accedere al terzo elemento, ovvero ad a_2, è necessario partire dall'inizio della lista, ovvero da a, per accedere ad a_0, poi ad a_1 e quindi ad a_2.

Le liste, d'altra parte, ben si prestano a implementare sequenze dinamiche: ciò a differenza degli array per i quali, come vedremo nel prossimo paragrafo, è necessario adottare particolari accorgimenti.

1.1.3 Array di dimensione variabile

Volendo utilizzare un array per realizzare una sequenza lineare **dinamica**, è necessario apportare diverse modifiche che consentano di effettuare il suo ridimensionamento: diversi linguaggi moderni, come C++, C# e JAVA usano tecniche simili per fornire array la cui dimensione può variare dinamicamente con il tempo. Prenderemo in considerazione l'inserimento e la cancellazione in fondo a un array a di n elementi per illustrarne la gestione del ridimensionamento. Allocare un nuovo array (più grande o più piccolo) per copiarvi gli elementi di a a ogni variazione della sua dimensione può richiedere $O(n)$ tempo per ciascuna operazione, risultando particolarmente oneroso in termini computazionali, sebbene sia ottimale in termini di memoria allocata. Con qualche piccolo accorgimento, tuttavia, possiamo fare meglio pagando tempo $O(n)$ cumulativamente per ciascun gruppo di $\Omega(n)$ operazioni consecutive, ovvero un costo medio costante per operazione. Inoltre possiamo garantire che il numero di celle allocate sia sempre proporzionale al numero degli elementi contenuti nell'array.

Sia d la taglia dell'array a ovvero il numero di elementi correntemente allocati in memoria e $n \le d$ il numero di elementi effettivamente contenuti nella sequenza memorizzata in a. Ogni qual volta un'operazione di inserimento viene eseguita, se vi è spazio sufficiente ($n + 1 \le d$), aumentiamo n di un'unità. Altrimenti, se $n = d$, allochiamo un array b di taglia 2d, raddoppiamo d, copiamo gli n elementi di a in b e poniamo a = b. Analogamente, ogni qualvolta un'operazione di cancellazione viene eseguita, diminuiamo n di un'unità. Quando $n = d/4$, dimezziamo l'array a: allochiamo un array b di taglia d/2, dimezziamo d, e copiamo gli n elementi di a in b (ponendo a = b). Queste operazioni di verifica ed eventuale ridimensionamento dell'array sono mostrate nel Codice 1.1.

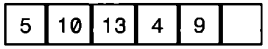
Codice 1.1 Operazioni di ridimensionamento di un array dinamico.

```
1  VerificaRaddoppio( ):        ⟨pre: a è un array di lunghezza d con n elementi⟩
2    IF (n == d) {
3      b = NuovoArray( 2 x d );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7      d = 2 x d;
8  }
```

```
1  VerificaDimezzamento( ):     ⟨pre: a è un array di lunghezza d con n elementi⟩
2    IF ((d > 1) && (n == d/4)) {
3      b = NuovoArray( d/2 );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7      d = d/2;
8  }
```

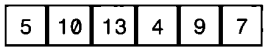
ESEMPIO 1.4

L'array a di taglia d = 6 contiene n = 5 elementi.



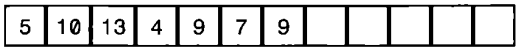
d = 6, n = 5

L'elemento 7 viene inserito in posizione d – 1 in tempo costante.



d = 6, n = 6

Poiché n = d, l'inserimento di un altro elemento richiede la creazione di un nuovo array di dimensione 2d e la copia del vecchio array nelle prime posizioni del nuovo.



d = 12, n = 7

Osserviamo che non è più possibile causare un ridimensionamento di a (raddoppio o dimezzamento) al costo di $O(n)$ tempo per ciascuna operazione: il prossimo teorema mostra che in effetti l'implementazione di tali operazioni mediante un array dinamico è molto efficiente. Ciò è dovuto al fatto che il costo di un ridimensionamento può essere concettualmente ripartito tra le operazioni che lo hanno preceduto, incrementando la loro complessità di un costo costante.

Teorema 1.1 *L'esecuzione di* n *operazioni di inserimento e cancellazione in un array dinamico richiede tempo* $O(n)$. *Inoltre* $d = O(n)$.

Dimostrazione Dopo un raddoppio, ci sono $m = d + 1$ elementi nel nuovo array di $2d$ posizioni. Occorrono almeno $m - 1$ richieste di inserimento per un nuovo raddoppio e almeno $m/2$ richieste di cancellazione per un dimezzamento. In modo simile, dopo un dimezzamento, ci sono $m = d/4$ elementi nel nuovo array di $d/2$ elementi, per cui occorrono almeno $m + 1$ richieste di inserimento per un raddoppio e almeno $m/2$ richieste di cancellazione per un nuovo dimezzamento. In tutti i casi, il costo di $O(m)$ tempo richiesto dal ridimensionamento può essere virtualmente distribuito tra le $\Omega(m)$ operazioni che lo hanno causato (a partire dal precedente ridimensionamento). Infine, supponendo che al momento della creazione dell'array si abbia $n = 1$ e $d = 1$, il numero di elementi nell'array è sempre almeno un quarto della sua taglia, pertanto $d = O(n)$. ❑

Esercizio svolto 1.1 Supponiamo che il dimezzamento di un array dinamico abbia luogo quando $n = d/2$, anziché quando $n = d/4$. Dimostrare che il Teorema 1.1 non è più vero.

Soluzione Per dimostrare che il teorema non è più verificato, definiamo una sequenza di $n = 2^k$ operazioni di inserimento e cancellazione la cui esecuzione richieda tempo $\Theta(n^2)$. Tale sequenza inizia con $n/2 = 2^{k-1}$ inserimenti: al termine della loro esecuzione l'array avrà dimensione $d = n/2$ e sarà, quindi, pieno. A questo punto la sequenza prosegue alternando un inserimento a una cancellazione: ciascuna di queste operazioni causa, rispettivamente, un raddoppio e un dimezzamento della dimensione dell'array e, quindi, richiede tempo $\Theta(n)$. In totale, quindi, l'esecuzione della seconda metà della sequenza ha un costo temporale pari a $\Theta(n^2)$.

1.2 Opus libri: scheduling della CPU

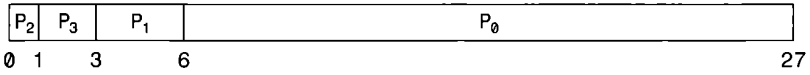
I moderni sistemi operativi sono ambienti di multi-programmazione, ovvero consentono che più programmi possano essere in esecuzione simultaneamente. Questo non vuol dire che una singola unità centrale di elaborazione (CPU, da *Central Processing Unit*) esegua contemporaneamente più programmi, ma semplicemente che nei periodi in cui un programma non ne fa uso, la CPU può dedicarsi ad altri programmi. A tal fine, la CPU ha a disposizione una sequenza di porzioni di programma da dover eseguire, ciascuna delle quali è caratterizzata da un tempo

(stimato) di utilizzo della CPU in millisecondi (per semplicità identifichiamo nel seguito le porzioni con i loro programmi). La decisione di eseguire i programmi in una determinata sequenza si chiama *scheduling*. La CPU può decidere di eseguire i programmi nell'ordine in cui essi appaiono nella sequenza, che di solito coincide con l'ordine di arrivo. Per questo motivo, tale politica viene chiamata *First Come First Served* (FCFS).

ESEMPIO 1.5

Supponiamo che vi siano quattro programmi P_0, P_1, P_2 e P_3 in esecuzione sulla stessa CPU e che in un certo istante di tempo la sequenza dei tempi previsti di utilizzo della CPU da parte loro sia la seguente: 21 ms per P_0, 3 ms per P_1, 1 ms per P_2 e 2 ms per P_3 (ipotizziamo che ciascun programma vada completato prima di passare a elaborarne un altro come accade, per esempio, nella coda di stampa). L'occupazione della CPU da parte dei programmi è quella mostrata nella figura seguente.

I tempi di attesa dei programmi sono pari a 0, 21, 24 e 25 ms, rispettivamente, ottenendo un tempo medio di attesa uguale a $(0 + 21 + 24 + 25)/4 = 17{,}5$ ms. Se la sequenza dei programmi da eseguire fosse giunta in un ordine diverso, applicando la strategia FCFS il tempo medio di attesa risulterebbe diverso. Per esempio, se la sequenza fosse giunta ordinata in modo non decrescente rispetto ai tempi di esecuzione, ovvero fosse P_2, P_3, P_1 e P_0, l'occupazione della CPU sarebbe quella mostrata nella seguente figura.



In questo caso, i tempi di attesa sarebbero 0, 1, 3 e 6 ms, rispettivamente, ottenendo un tempo medio di attesa di $(0 + 1 + 3 + 6)/4 = 2{,}5$ ms, che è significativamente più basso e che tra l'altro è il minimo tempo di attesa medio possibile.

Dall'esempio precedente risulta che il tempo di attesa medio diminuisce se i programmi con tempi di utilizzo minore vengono eseguiti per primi. Per questo motivo, possiamo considerare soluzioni in cui viene sempre eseguita per prima la porzione di programma con tempo di esecuzione più breve: tale politica viene chiamata *Shortest Job First* (SJF). La realizzazione della strategia SJF richiede di poter eseguire l'ordinamento dei tempi previsti di utilizzo della CPU da parte dei diversi programmi. Una situazione analoga occorre nella gestione della coda di attesa di una stampante condivisa, dove i tempi previsti sono proporzionali al numero di pagine da stampare per ogni file nella coda.

L'operazione di ordinamento di una sequenza lineare di elementi è una delle operazioni più frequenti in diverse applicazioni informatiche, e nel seguito ne discuteremo ulteriormente. In questo paragrafo, mostriamo come ottenere un ordinamento mediante due semplici algoritmi, di cui valuteremo la complessità rispetto al numero n di elementi della sequenza. Entrambi gli algoritmi richiedono

un tempo $O(n^2)$ e risultano utili per la loro semplicità quando il valore di n è piccolo. Vedremo più avanti come sia possibile progettare algoritmi di ordinamento più efficienti, la cui complessità temporale è $O(n \log n)$.

1.2.1 Ordinamento per selezione

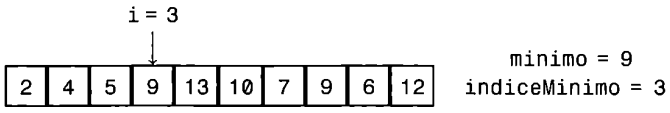
L'algoritmo di ordinamento per selezione, detto *selection sort*, consiste nell'eseguire n passi: al generico passo i = 0, 1, ..., n - 1, viene selezionato l'elemento che occuperà la posizione i della sequenza ordinata. In altre parole, al termine del passo i, gli i + 1 elementi selezionati fino a quel momento coincidono con i primi i + 1 elementi della sequenza ordinata. Per realizzare il passo i, l'algoritmo deve selezionare il minimo (se l'ordinamento da ottenere è non decrescente, altrimenti il massimo) tra gli elementi che si trovano dalla posizione i in avanti, per poi sistemarlo nella posizione corretta i. L'algoritmo di ordinamento per selezione è mostrato nel Codice 1.2, dove il ciclo termina per i = n - 2 in quanto nel passo i = n - 1 c'è un unico elemento rimasto.

Codice 1.2 Ordinamento per selezione di un array a.

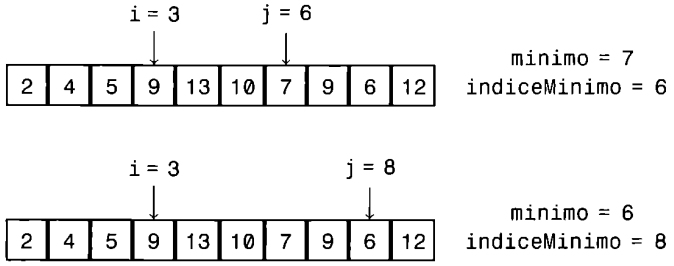
```
1  SelectionSort( a ):                          ⟨pre: la lunghezza di a è n⟩
2    FOR (i = 0; i < n-1; i = i+1) {
3      minimo = a[i];
4      indiceMinimo = i;
5      FOR (j = i+1; j < n; j = j+1) {
6        IF (a[j] < minimo) {
7          minimo = a[j];
8          indiceMinimo = j;
9        }
10     }
11     a[indiceMinimo] = a[i];
12     a[i] = minimo;
13   }
```

ESEMPIO 1.6

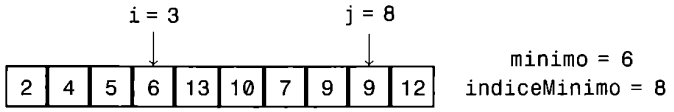
Dopo i + 1 esecuzioni del ciclo esterno dell'algoritmo le prime i + 1 posizioni sono occupate dagli i + 1 elementi più piccoli di a. All'inizio della quarta esecuzione del ciclo esterno (i = 3) la composizione dell'array a è la seguente.



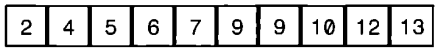
La variabile j del ciclo interno viene utilizzata per cercare il minimo tra gli elementi che si trovano dalla posizione i in poi: ogni volta che viene trovato un elemento minore del minimo attuale si aggiornano le variabili indiceMinimo e minimo. Nel nostro esempio, ciò avviene in corrispondenza di j = 6 e j = 8.



Terminato il ciclo interno avviene lo scambio tra il minimo e l'elemento in posizione i.



Al termine delle n operazioni l'array risulterà ordinato.



Teorema 1.2 *Il selection sort ordina un array di* n *elementi in tempo* $\Theta(n^2)$.

Dimostrazione L'algoritmo esegue n - 1 iterazioni del ciclo esterno del Codice 1.2, che va dalla riga 2 alla riga 13. Il costo t_i dell'iterazione i è dato da un numero costante di operazioni di costo $O(1)$, che corrispondono all'inizializzazione del minimo (righe 3 e 4) e al suo posizionamento finale scambiandolo con l'attuale elemento nella posizione i (righe 11 e 12), più il costo del ciclo interno, che corrisponde alla ricerca del minimo e che va dalla riga 5 alla riga 10. Quindi, al passo i, l'algoritmo esegue un numero di operazioni proporzionale a n - i: il numero totale di operazioni al caso pessimo è pertanto proporzionale a

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

Si noti che tale complessità in tempo è sempre raggiunta, per qualunque sequenza iniziale di n elementi: quindi, ogni sequenza è la peggiore, dal punto di vista del costo di computazione. La correttezza dell'algoritmo discende immediatamente osservando che, al termine dell'iterazione i del ciclo esterno, i primi i elementi dell'array coincidono i primi i+1 elementi della sequenza ordinata e che nelle iterazioni successive tali elementi non sono più spostati. ❑

Il teorema precedente afferma che la complessità dell'ordinamento per selezione è sempre quadratica, indipendentemente dalla sequenza iniziale. Nel prossimo paragrafo descriveremo un algoritmo di ordinamento altrettanto semplice, le cui prestazioni possono però essere in alcuni casi significativamente migliori.

1.2.2 Ordinamento per inserimento

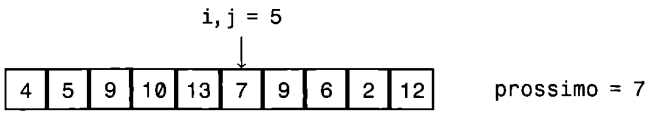
L'algoritmo di ordinamento per inserimento, detto *insertion sort*, consiste anch'esso nell'eseguire n passi: al passo `i` = 1, 2, ..., n – 1, l'elemento in posizione `i` viene inserito al posto giusto tra i primi `i`+1 elementi. In altre parole, al termine del passo `i`, gli `i`+1 elementi sistemati fino a quel momento sono tra di loro ordinati, ma non coincidono necessariamente con i primi `i`+1 elementi della sequenza ordinata. Se `prossimo` denota l'elemento in posizione `i`, per realizzare il passo `i` l'algoritmo confronta `prossimo` con i primi `i` elementi fino a trovare la posizione corretta in cui inserirlo: a tale scopo, procede dalla posizione `i` – 1 verso l'inizio della sequenza, spostando ciascuno degli elementi maggiore di `prossimo` di una posizione in avanti per far posto a `prossimo` stesso. L'algoritmo di ordinamento per inserimento è mostrato nel Codice 1.3.

Codice 1.3 Ordinamento per inserimento di un array a.

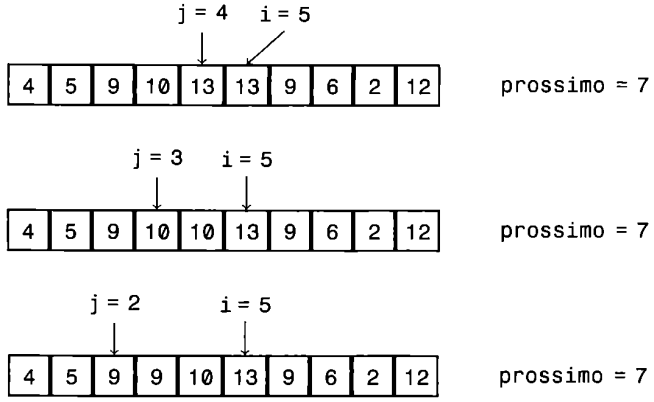
```
1  InsertionSort( a ):                          ⟨pre: la lunghezza di a è n⟩
2    FOR (i = 1; i < n; i = i+1) {
3      prossimo = a[i];
4      j = i;
5      WHILE ((j > 0) && (a[j-1] > prossimo)) {
6        a[j] = a[j-1];
7        j = j-1;
8      }
9      a[j] = prossimo;
10   }
```

ESEMPIO 1.7

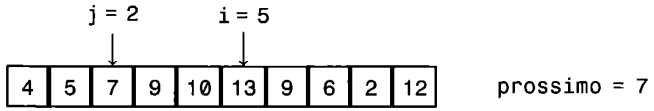
Sia a = {5, 10, 13, 4, 9, 7, 9, 6, 2, 12}. Dopo `i` + 1 esecuzioni del ciclo esterno dell'algoritmo i primi `i` + 1 elementi dell'array risultano ordinati. Al passo successivo si cerca la posizione corretta dell'elemento in posizione `i` (memorizzato nella variabile `prossimo`) all'interno della sequenza composta dai primi `i` + 1 elementi di a. All'inizio della sesta esecuzione del ciclo esterno la situazione è la seguente.



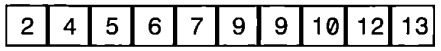
La variabile `j` viene decrementata fintanto che `a[j – 1]` contiene un elemento maggiore di `prossimo`. Inoltre il valore `a[j – 1]` viene copiato alla sua destra.



Quando `a[j-1]` risulta essere minore o uguale a `prossimo` (oppure quando si raggiunge la prima posizione dell'array), il valore di `prossimo` viene copiato in `a[j]`.



Al termine delle n operazioni l'array risulterà ordinato.



Teorema 1.3 *L'insertion sort ordina un array di* n *elementi in tempo* $O(n^2)$.

Dimostrazione L'algoritmo esegue un doppio ciclo, di cui quello più esterno (dalla riga 2 alla 10 del Codice 1.3) corrisponde agli n – 1 passi dell'algoritmo. Il ciclo `while` interno (dalla riga 5 alla 8), esamina le posizioni `i` – 1, `i` – 2, ..., fino a trovare un elemento non superiore a `prossimo` o fino a raggiungere l'inizio dell'array, determinando dunque il punto in cui `prossimo` deve essere inserito: man mano che esamina gli elementi in tali posizioni, questi vengono spostati di una posizione in avanti. Al termine del ciclo interno (riga 9), avviene l'effettiva operazione di inserimento di `prossimo` nella posizione corretta. Pertanto, al passo `i` l'algoritmo di ordinamento per inserimento esegue un numero di operazioni proporzionale a `i`+1 al caso pessimo, e il numero totale di operazioni eseguite è limitato superiormente e asintoticamente dalla seguente sommatoria:

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

La correttezza dell'algoritmo discende immediatamente osservando che, al termine dell'iterazione `i+1` del ciclo esterno, i primi `i` elementi dell'array sono ordinati: al termine dell'ultima iterazione, quindi, l'intero array è ordinato. ❑

Anche l'insertion sort è pertanto un algoritmo di ordinamento con complessità quadratica rispetto al numero di elementi da ordinare. Tuttavia, a differenza del selection sort, può richiedere un tempo significativamente inferiore per certe sequenze di elementi: se la sequenza iniziale è già in ordine o soltanto un numero costante di elementi è fuori ordine, l'insertion sort richiede $O(n)$ tempo mentre la complessità del selection sort rimane comunque $\Theta(n^2)$.

Esercizio svolto 1.2 Fornite una sequenza di `n` elementi per cui l'algoritmo insertion sort esegua $\Theta(n^2)$ operazioni.

Soluzione Consideriamo una sequenza ordinata in modo opposto, tale che $a_0 > a_1 > \ldots > a_{n-1}$. L'inserimento di a_i al posto giusto tra i primi `i + 1` elementi della sequenza richiede in tal caso di raggiungere sempre la posizione all'inizio dell'array: in altre parole, l'iterazione `i` del ciclo esterno esegue un numero di passi proporzionale a `i+1`. Pertanto, il numero totale di operazioni eseguite è proporzionale a n^2.

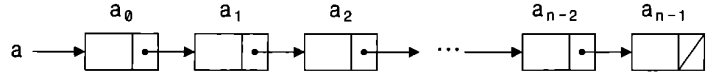
1.3 Gestione di liste

Abbiamo già visto come l'organizzazione sequenziale dei dati può, in generale, essere realizzata in due diverse modalità: quella ad accesso diretto e quella ad accesso sequenziale. Nel primo caso, il tipo di dati utilizzato è l'array, i cui pregi e difetti sono già stati analizzati. In questo paragrafo, invece, ci concentriamo sul tipo di dati lista, che realizza l'organizzazione dei dati con la modalità di accesso sequenziale. Ricordiamo che la caratteristica essenziale di questa realizzazione consiste nel fatto che i dati non risiedono in locazioni di memoria contigue e, pertanto, ciascun dato deve includere, oltre all'informazione vera e propria, un riferimento al dato successivo.

Sia dato un elemento `x` di una lista in posizione `i`; nel seguito indicheremo con `x.dato` l'informazione associata a tale elemento e con `x.succ` il riferimento che lo collega all'elemento nella posizione `(i + 1)`-esima. A tale proposito, presumiamo l'esistenza di un valore `null`, utilizzato per indicare un riferimento "nullo", vale a dire un riferimento a nessuna locazione di memoria, e che per l'ultimo elemento `x` della lista sia `x.succ = null`. Nel seguito, inoltre, denoteremo con `a` il riferimento al primo elemento della lista, ovvero alla locazione di memoria che lo contiene: evidentemente, nel caso in cui la lista sia vuota, tale riferimento avrà valore `null`.

ESEMPIO 1.8

Nella seguente figura rappresentiamo una lista di `n` elementi.



Nella figura, i riferimenti sono rappresentati in modo sintetico, senza evidenziare le relative locazioni di memoria, e il valore `null` è indicato con il simbolo /.

Le istruzioni seguenti mostrano come accedere all'elemento in posizione `i` di una lista `a` in tempo `O(i)`, dove nella variabile `p` viene memorizzato, al termine del ciclo `while`, il riferimento a tale elemento oppure `null` se tale elemento non esiste:

```
p = a;
j = 0;
WHILE ((p != null) && (j < i)) {
  p = p.succ;
  j = j+1;
}
```

Osserviamo che questo codice può essere facilmente modificato in modo da realizzare la ricerca di una chiave `k` all'interno di una lista: è sufficiente infatti modificare la condizione di terminazione della scansione della lista, la quale viene determinata dalla verifica del raggiungimento dell'ultimo elemento della lista oppure dall'aver trovato la chiave desiderata (ovvero `p.dato` uguale a `k`).

1.3.1 Inserimento e cancellazione

A differenza degli array, le liste si prestano molto bene a gestire sequenze lineari dinamiche, in cui il numero degli elementi presenti può variare nel tempo a causa di operazioni di inserimento e di cancellazione. In effetti, l'inserimento di un nuovo elemento all'interno di una lista può consistere semplicemente nel porlo in cima alla lista stessa, eseguendo le seguenti istruzioni, in cui ipotizziamo che `x` indichi il riferimento all'elemento da inserire (si veda la Figura 1.1):

```
x.succ = a;
a = x;
```

L'inserimento dopo l'elemento indicato da un riferimento `p` $\neq$ `null` è una semplice variazione dell'operazione precedente, in cui `p.succ` sostituisce la variabile `a`, come mostrato nelle seguenti istruzioni:

```
x.succ = p.succ;
p.succ = x;
```

Leggermente più complicata è la cancellazione di un elemento, operazione che dovrà rendere l'elemento e la lista mutuamente non raggiungibili attraverso i relativi riferimenti. Avendo a disposizione il riferimento `x` all'elemento da cancellare,

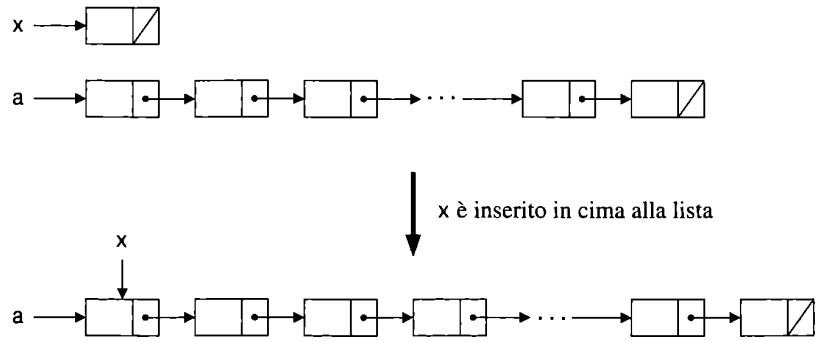


Figura 1.1 Inserimento in testa a una lista.

un caso particolare è rappresentato dalla situazione in cui x coincida con a, vale a dire in cui l'elemento da cancellare sia il primo della lista. In tal caso la cancellazione viene effettuata modificando il riferimento iniziale alla lista in modo da andare a "puntare" al secondo elemento, come mostrato nelle istruzioni seguenti:

```
a = x.succ;
x.succ = null;
```

Per la cancellazione di un elemento diverso dal primo è necessario non solo avere a disposizione il suo riferimento x, ma anche un riferimento p all'elemento che lo precede. In questo modo, possiamo cancellare l'elemento desiderato creando un "ponte" tra il suo predecessore e il suo successore (Figura 1.2). Questo ponte può essere realizzato mediante le seguenti istruzioni:

```
p.succ = x.succ;
x.succ = null;
```

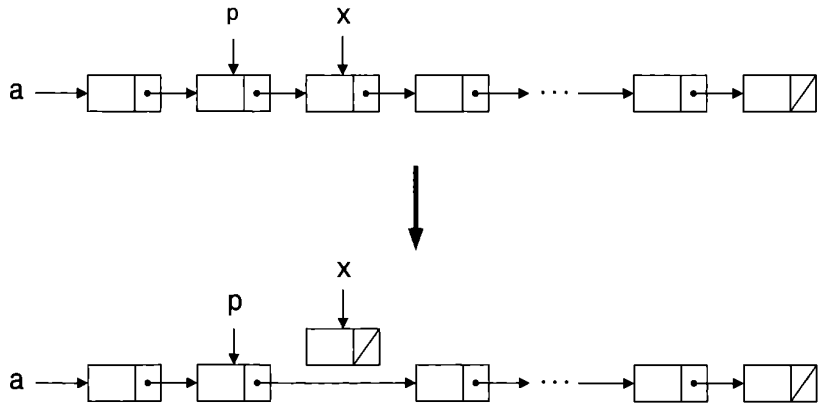


Figura 1.2 Cancellazione di un elemento da una lista.

Ignorando il costo di allocazione e deallocazione e quello per determinare i riferimenti x e p nella lista a, in quanto esso dipende dall'applicazione scelta, le operazioni di inserimento e cancellazione appena descritte presentano un costo computazionale di $O(1)$ tempo e spazio, indipendente cioè dalla dimensione della lista. Ricordiamo a tale proposito come le medesime operazioni su un array richiedano tempo lineare $\Theta(n)$.

1.3.2 Liste doppie

La struttura di una lista può essere modificata in modo tale da effettuare più efficientemente determinate operazioni. In questo paragrafo introdurremo una delle più diffuse variazioni di questo tipo: la lista doppia.

In una **lista doppia** l'elemento x in posizione i ha, oltre al riferimento x.succ all'elemento in posizione i + 1, un riferimento x.pred all'elemento in posizione i - 1, con x.pred uguale a null se i = 0. Tale estensione consente di spostare in tempo $O(1)$ un riferimento x sia all'elemento successivo (con l'istruzione x = x.succ) che al precedente (con l'istruzione x = x.pred). La Figura 1.3 fornisce un esempio di lista doppia: in questa figura non sono evidenziati, per semplicità, i campi relativi ai riferimenti.

Figura 1.3 Lista doppia.

L'aggiunta del riferimento "all'indietro", sebbene complichi leggermente l'operazione di inserimento di un nuovo elemento, semplifica in modo sostanziale quella di cancellazione, in quanto consente di accedere, a partire dall'elemento da cancellare, ai due elementi circostanti, il cui contenuto va modificato nell'operazione di cancellazione. Al contrario, in una lista semplice la cancellazione di un elemento richiede un riferimento all'elemento precedente. Nello specifico, detto x il riferimento all'elemento da cancellare, possiamo considerare i seguenti quattro casi che determinano l'insieme delle istruzioni necessarie per eseguire la cancellazione.

Caso 1. x fa riferimento al primo elemento della lista, cioè x è uguale ad a. In questo caso, non avendo un predecessore, la cancellazione determina la modifica del riferimento pred del successore x.succ; inoltre, è necessario aggiornare il riferimento iniziale a, come mostrato nelle seguenti istruzioni:

```
x.succ.pred = null;
a = x.succ;
x.succ = null;
```

Caso 2. x fa riferimento all'ultimo elemento della lista, cioè x.succ è uguale a null. In questo caso, non avendo un successore, la cancellazione richiede la

modifica del riferimento succ del predecessore x.pred, come mostrato nelle seguenti istruzioni:

```
x.pred.succ = null;
x.pred = null;
```

Caso 3. x è l'unico elemento nella lista, cioè x è uguale ad a e x.succ è uguale a null. In questo caso, l'effetto della cancellazione è quello di rendere la lista vuota, e quindi di assegnare al riferimento iniziale il valore null, mediante la seguente istruzione:

```
a = null;
```

Caso 4. x fa riferimento a un elemento "interno" della lista. In questo caso, vanno aggiornati sia il riferimento succ del predecessore che il riferimento pred del successore, come mostrato nelle seguenti istruzioni:

```
x.succ.pred = x.pred;
x.pred.succ = x.succ;
x.succ = null;
x.pred = null;
```

Notiamo che tutti i casi appena discussi garantiscono correttamente che x.succ = x.prec = null dopo una cancellazione, evitando così di lasciare un riferimento pendente (*dangling pointer*). L'assegnamento dei riferimenti a null è contestuale al linguaggio di programmazione usato. Nei linguaggi dove il recupero di celle di memoria inutilizzate è eseguito automaticamente tramite un *garbage collector* (per esempio, JAVA), ciascuna delle zone di memoria allocate dinamicamente ha un contatore di riferimenti in entrata: quando tale contatore è pari a zero, la zona può essere liberata dal garbage collector. In tale contesto, porre a null il valore dei riferimenti non più utilizzati, anche se non sempre strettamente necessario per la logica del codice, ha come beneficio collaterale il recupero di zone di memoria non più necessarie all'esecuzione del programma.

Esercizio svolto 1.3 Si descriva un algoritmo per l'inversione di una lista semplice. Ad esempio, con input la lista 1, 2, 3, 4 l'algoritmo deve trasformarla nella lista 4, 3, 2, 1. L'algoritmo non deve fare uso di memoria aggiuntiva.

Soluzione L'operazione di inversione di una lista può essere realizzata facendo uso dell'algoritmo ricorsivo descritto nel seguente codice.

```
InvertiLista( p, x ):
  IF (x.succ != null) {
    a = Inverti( x, x.succ );
  } ELSE {
    a = x;
  }
  x.succ = p;
  RETURN a;
```

Per invertire una lista non vuota riferita dalla variabile a, dovremo invocare la funzione InvertiLista con i parametri null e a.

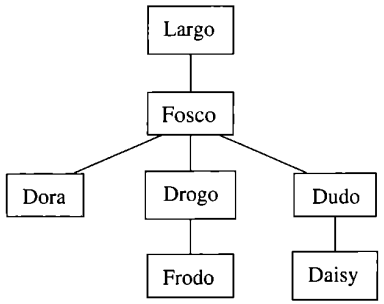
1.4 Alberi

Gli alberi rappresentano una generalizzazione delle liste nel senso che, mentre ogni elemento di una lista ha al più un successore, ogni elemento di un albero, anche detto **nodo**, può avere più di un successore. Ogni nodo dell'albero ha pertanto associata la lista (eventualmente vuota) dei **figli** ad esso collegati da un **arco**: utilizzando una terminologia che è un misto di genealogia e botanica, chiamiamo **foglie** i nodi senza figli e **nodi interni** i rimanenti nodi. Allo stesso tempo, l'albero associa a tutti i nodi, eccetto la **radice** (ovvero il nodo di partenza dell'albero), un unico genitore, detto **padre**: i nodi figli dello stesso padre sono detti **fratelli**. Osserviamo che, ad ogni nodo interno, è anche associato il **sottoalbero** di cui tale nodo è radice. Se un nodo u è la radice di un sottoalbero contenente un nodo v, diciamo che u è un **antenato** di v e che v è un **discendente** di u.

La **distanza** tra due nodi dell'albero è il numero di archi che separano i due nodi. L'**altezza** dell'albero è data dalla massima distanza di una foglia dalla radice dell'albero. Infine, la **profondità** di un nodo è la sua distanza dalla radice.

ESEMPIO 1.9

Come vedremo, gli alberi sono solitamente utilizzati per rappresentare partizioni ricorsive di insiemi e strutture gerarchiche: un tipico utilizzo di alberi per rappresentare gerarchie è fornito dagli alberi genealogici, in cui ciascun nodo dell'albero rappresenta una persona della famiglia i cui figli sono ad esso collegati da un arco ciascuno. Ad esempio, nella figura seguente, è mostrata una parte dell'albero genealogico della famiglia Baggins di Hobbiville, quella relativa ai discendenti di Largo (corrispondente alla radice dell'albero), il cui unico figlio è Fosco, i cui nipoti sono Dora, Drogo e Dudo, e i cui pronipoti sono Frodo e Daisy.



Il sottoalbero associato al nodo Drogo contiene, oltre a se stesso, il suo unico figlio Frodo. Il padre di Drogo è Fosco e i suoi fratelli sono Dora e Drudo. Infine, Drogo è discendente di Largo, che è suo antenato.

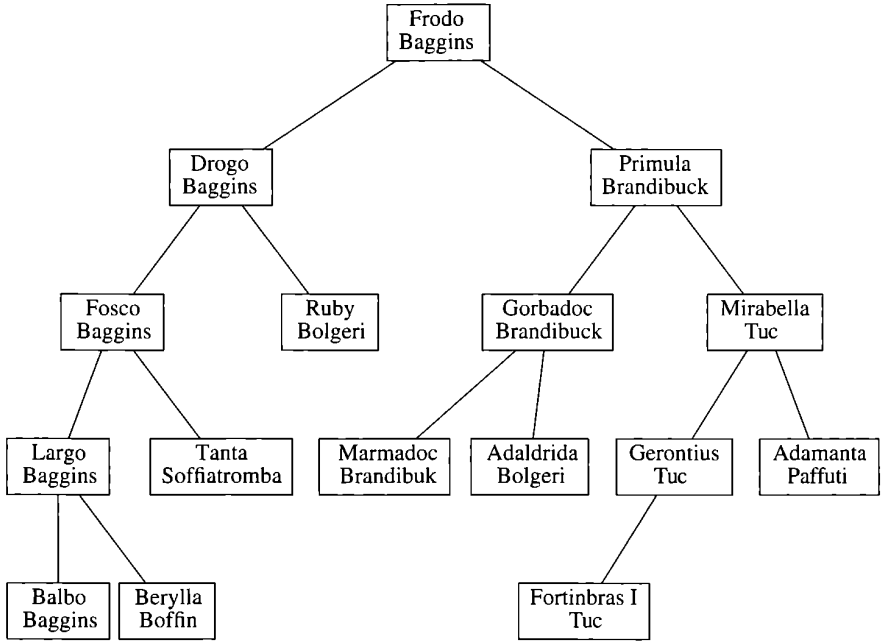
1.4.1 Alberi binari

Un particolare tipo di albero è costituito dagli **alberi binari** che, rispetto a quelli generali, presentano due principali caratteristiche: ogni nodo ha al più due figli e ogni figlio ha un ruolo ben determinato che dipende dall'essere il **figlio sinistro**

oppure il **figlio destro**. Un albero binario può essere definito ricorsivamente nel modo seguente: l'albero vuoto è un albero binario che non contiene alcun nodo e che viene indicato con `null`, analogamente a quanto fatto con la lista vuota. Un albero binario (non vuoto) contenente `n` elementi è costituito dalla radice `r`, che memorizza uno di questi elementi mettendolo "a capo" degli altri; i rimanenti `n - 1` elementi sono divisi in due gruppi disgiunti, ricorsivamente organizzati in due sottoalberi binari distinti, etichettati come sinistro e destro e radicati nei due figli r_S e r_D della radice. Notiamo che uno o entrambi i nodi r_S e r_D possono essere `null`, a rappresentare sottoalberi vuoti, e che i figli di una foglia sono entrambi uguali a `null`, così come il padre della radice dell'albero.

ESEMPIO 1.10

Gli alberi genealogici sono spesso utilizzati anche per rappresentare l'insieme degli antenati di una persona, anziché quello dei suoi discendenti: in tali alberi i figli di un nodo rappresentano, in modo apparentemente contraddittorio, i suoi genitori. Pertanto, tali alberi sono alberi binari in cui il figlio sinistro indica il padre mentre il figlio destro rappresenta la madre. Nella figura seguente sono, per esempio, rappresentati gli antenati (noti agli autori) di Frodo Baggins.

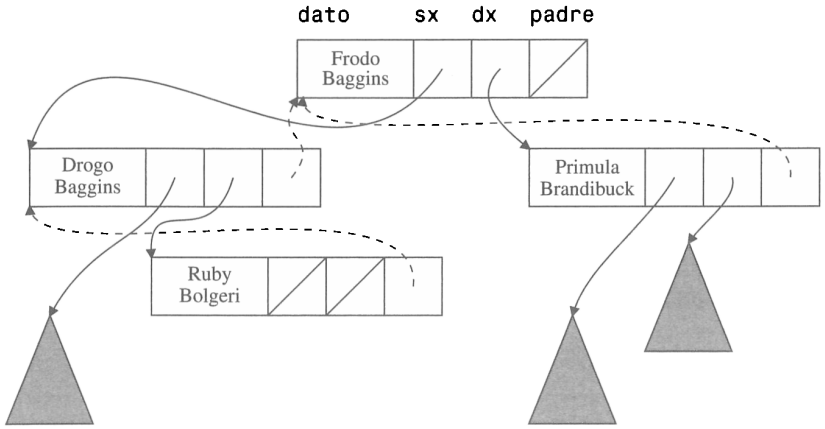


Ad esempio, il figlio sinistro di Drogo Baggins è il padre Fosco Baggins, mentre il figlio destro è la madre Ruby Bolgeri. Osserviamo che in alcuni casi uno solo dei due figli può essere specificato, come nel caso di Gerontius Tuc di cui non è nota la madre.

Un albero binario viene generalmente rappresentato nella memoria del calcolatore facendo uso di tre campi. In particolare, dato un nodo `u`, indichiamo con `u.dato` il contenuto del nodo, con `u.sx` il riferimento al figlio sinistro e con `u.dx` il riferimento al figlio destro (talvolta ipotizzeremo che sia anche presente un riferimento `u.padre` al padre del nodo).

ESEMPIO 1.11

Nella figura seguente (in cui tre sottoalberi sono solo tratteggiati e non esplicitamente disegnati) mostriamo come viene rappresentata la parte superiore dell'albero genealogico illustrato nella figura dell'esempio precedente: osserviamo, tuttavia, che nel seguito preferiremo sempre fare riferimento alla rappresentazione grafica semplificata di quest'ultima figura.



Nella figura i riferimenti `u.padre` sono rappresentati con linee tratteggiate per distinguerli dai riferimenti ai figli.

Osserviamo come i due campi `u.sx` e `u.dx` di Ruby Bolgeri siano entrambi uguali a `null`, in quanto tale nodo è una foglia dell'albero.

1.4.2 Alberi cardinali e ordinali

Gli alberi binari sono un caso particolare degli **alberi cardinali** o **k-ari**, caratterizzati dal fatto che ogni nodo ha `k` riferimenti ai figli, i quali sono numerati da `0` a `k - 1`. Precisamente, un nodo `u` di un albero `k`-ario ha i campi `u.dato` e `u.padre` come negli alberi binari, mentre i riferimenti ai suoi figli sono memorizzati in un array `u.figlio` di dimensione `k`, dove `u.figlio[i]` è il riferimento (eventualmente uguale a `null`) al figlio `i` (`0` ≤ `i` ≤ `k - 1`): per `k = 2`, abbiamo che `u.figlio[0]` corrisponde a `u.sx` mentre `u.figlio[1]` corrisponde a `u.dx`. Per `k = 3, 4, ...`, si ottengono alberi ternari, quaternari e così via, che possono essere definiti ricorsivamente come gli alberi binari.

Gli **alberi ordinali** si differenziano da quelli cardinali, in quanto ogni nodo memorizza soltanto la lista *ordinata* dei riferimenti *non nulli* ai suoi figli. Il numero di tali figli è variabile da nodo a nodo e viene chiamato **grado**: assumendo che `n` sia il numero di nodi dell'albero, il grado è un intero compreso tra `0` (nel caso di una foglia) e `n - 1` (nel caso di una radice che ha i rimanenti nodi come figli), e un nodo di grado `d > 0` ha `d` figli che sono numerati consecutivamente da `0` a `d - 1`. Un esempio di albero ordinale è quello mostrato nella figura dell'Esempio 1.9 dove il grado massimo (denominato **grado dell'albero**) è `d = 3`.

Osserviamo che gli alberi cardinali e gli alberi ordinali sono due strutture di dati *differenti*, nonostante l'apparente somiglianza. Gli alberi nella Figura 1.4 sono distinti se considerati come alberi cardinali, in quanto il nodo D è il figlio destro del nodo B nel primo caso ed è il figlio sinistro nel secondo caso, mentre tali alberi sono indistinguibili come alberi ordinali in quanto D è il primo (e unico) figlio di B. Nel caso degli alberi ordinali, inoltre, non è possibile preallocare un nodo in modo da poter ospitare il massimo numero di figli, essendo il suo grado variabile. Usiamo quindi la **memorizzazione binarizzata** dell'albero, introducendo nodi in cui, oltre al campo `u.dato`, sono presenti anche i campi `u.padre` per il riferimento al padre, `u.primo` per il riferimento al *primo figlio* (con numero `0`) e `u.fratello` per il riferimeno al *successivo fratello* nell'ordine filiare.

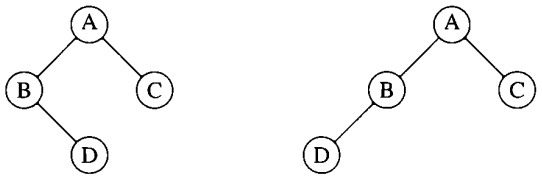
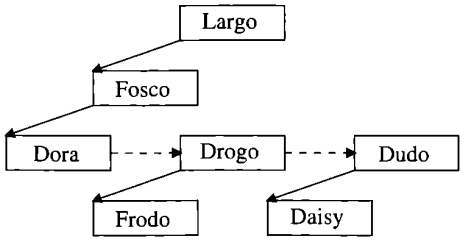


Figura 1.4 Due alberi binari distinti che risultano indistinguibili come alberi ordinali.

ESEMPIO 1.12

Nella figura seguente mostriamo la memorizzazione binarizzata dell'albero ordinale mostrato nella figura dell'Esempio 1.9.



Nella figura, i riferimenti `u.fratello` sono rappresentati come frecce tratteggiate per distinguerli dai riferimenti `u.primo`.

Osserviamo che, a differenza degli alberi cardinali in cui l'accesso a un qualunque figlio richiede sempre tempo costante, negli alberi ordinali (memorizzati in modo binarizzato) per raggiungere il figlio `i` (con `i > 0`) è necessario `O(i)` tempo per scandire la lista dei figli con il seguente frammento di codice:

```
p = u.primo;
j = 0;
WHILE ((p != null) && (j < i)) {
  p = p.fratello;
  j = j+1;
}
```

Quindi la memorizzazione binarizzata dei nodi in un albero ordinale, facendo uso di due riferimenti, è analoga alla rappresentazione degli alberi binari, ma la semantica delle due rappresentazioni è ben diversa! Allo stesso tempo, la memorizzazione binarizzata permette di stabilire l'esistenza di una *corrispondenza biunivoca* tra gli alberi binari e gli alberi ordinali: ogni albero binario di `n` nodi con radice `r` è la memorizzazione binarizzata di un distinto albero ordinale di `n+1` nodi in cui viene introdotta una nuova radice fittizia il cui primo figlio è `r`. Tale corrispondenza identifica il campo `u.sx` degli alberi binari con il campo `u.primo` della memorizzazione binarizzata degli alberi ordinali e il campo `u.dx` con il campo `u.fratello`, e viene esemplificata nella Figura 1.5, dove la radice fittizia è mostrata come un pallino ed è introdotta ai soli fini della presente discussione (un altro esempio deriva dai due alberi binari mostrati nella Figura 1.4, quando questi sono interpretati come la memorizzazione binarizzata di due distinti alberi ordinali: nel primo caso i nodi B e D sono fratelli, mentre nel secondo caso D è il primo e unico figlio di B).

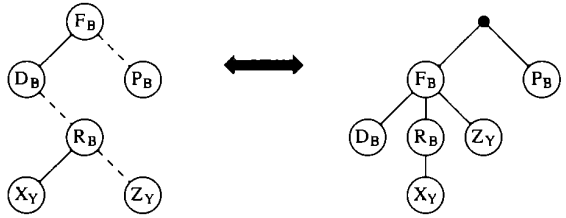


Figura 1.5 Corrispondenza tra un albero binario di n nodi e un albero ordinale di n+1 nodi.

Esercizio svolto 1.4 Un **nodo unario** di un albero è un nodo interno con un solo figlio. Dimostrare che in un albero `T` non vuoto che non abbia nodi unari il numero `n` di nodi interni è strettamente minore del numero `f` delle foglie.

Soluzione Dimostriamo l'asserto per induzione su n. Se $n = 0$, abbiamo che $f = 1$ (essendo l'albero non vuoto) e, quindi, $n < f$. Supponiamo che l'asserto sia vero per ogni $k < n$ e dimostriamolo per $n > 0$. Sia x un nodo interno i cui figli sono tutte foglie (tale nodo deve necessariamente esistere) e costruiamo un nuovo albero T' cancellando da T i figli di x. Il numero di nodi interni di T' è uguale a $n' = n-1$, mentre il numero di foglie di T' è uguale a $f' = f-c+1$ dove c indica il numero di figli di x. Per l'ipotesi fatta su T, abbiamo che $c \geq 2$, per cui $f' \leq f-1$. Per ipotesi induttiva, abbiamo che $n' < f'$, ovvero $n-1 < f-1$ e, quindi, $n < f$, come volevasi dimostrare.

1.5 Esercizi

1.1 Dato un array di n elementi con ripetizioni, sia k il numero di elementi distinti in esso contenuti. Progettare e analizzare un algoritmo che restituisca un array di k elementi contenente una e una sola volta gli elementi distinti dell'array originale.

1.2 Descrivere un algoritmo che, dati due array a e b di n e m elementi, rispettivamente, con $m \leq n$, determini se esiste un segmento di a uguale a b in tempo $O(nm)$.

1.3 Descrivere un algoritmo di complessità lineare che, dato un array di elementi interi, determini la sottosequenza più lunga di elementi contigui dell'array non decrescente (ad esempio, se l'array è 5, 10, 12, 4, 9, 11, 2, 3, 6, 7, 8, 1, 13, allora la sottosequenza più lunga è 2, 3, 6, 7, 8).

1.4 Un algoritmo di ordinamento è **stabile** se, in presenza di elementi uguali, ne mantiene la posizione relativa nella sequenza d'uscita (quindi gli elementi uguali appaiono contigui ma non sono permutati tra di loro): dire se gli algoritmi di ordinamento discussi nel capitolo sono stabili.

1.5 Descrivere un algoritmo che ordini in tempo lineare un array contenente solo 0 e 1. Generalizzare tale algoritmo al caso l'array contenga solo elementi interi compresi tra 0 e c e valutarne la complessità.

1.6 Si consideri il seguente frammento di codice.

```
BubbleSort( a ):                    ⟨pre: la lunghezza di a è n⟩
  FOR (i = 0; i < n; i = i+1) {
    FOR (j = 0; j < n-i; j = j+1) {
      IF (a[j] > a[j+1]) {
        t = a[j+1];
        a[j+1] = a[j];
        a[j] = t;
      }
    }
  }
```


Dimostrare che la funzione BubbleSort ordina l'array a in tempo $O(n^2)$. Descrivere, poi, come modificare il codice precedente in modo che, nel caso di certe sequenze di elementi, la complessità temporale sia lineare.

1.7 Una sequenza è palindroma se rimane uguale a se stessa quando viene letta da destra a sinistra. Per esempio, la sequenza abbabba è palindroma. Dato un intero m e un array a di n interi, progettare e analizzare un algoritmo che verifichi se esiste un segmento di a di m elementi che forma una sequenza palindroma.

1.8 Mostrare come modificare il codice di ricerca di un elemento in una lista semplice, utilizzando un ulteriore riferimento q in modo tale che valga la seguente invariante, necessaria a implementare l'operazione di cancellazione in una lista semplice: se entrambi p e q puntano allo stesso elemento, questo è il primo della lista; altrimenti, p e q.succ puntano allo stesso elemento nella lista, e tale elemento è diverso dal primo elemento della lista.

1.9 Mostrare le istruzioni necessarie a inserire un nuovo elemento in cima a una lista doppia.

1.10 Descrivere un'implementazione dell'algoritmo insertion sort che utilizzi liste anziché array, identificando il tipo di lista adatto a ottenere, per ogni sequenza di n dati in ingresso, un costo computazionale uguale a quello dell'implementazione basata su array.

1.11 Sia a una lista doppia contenente solo 0 e 1 e tale che $a_0 = 1$: sia n il numero intero rappresentato in binario da a, assumendo che a_0 sia la cifra più significativa. Scrivere un frammento di codice che modifichi la lista producendo la rappresentazione binaria di n + 1.

1.12 Svolgere l'Esercizio svolto 1.3 senza fare uso della ricorsione e utilizzando una quantità di memoria aggiuntiva costante (ovvero indipendente dalla dimensione della lista).

1.13 Una lista circolare è una lista in cui l'ultimo elemento last, invece di avere last.succ = null, ha last.succ = first, dove first è il primo elemento della lista, puntato, come al solito, dal puntatore iniziale a. Mostrare le istruzioni necessarie a inserire ed eliminare un elemento da una lista circolare.

1.14 Una lista circolare doppia è una lista circolare in cui ogni elemento x ha riferimenti sia al successore (x.succ) che al predecessore (x.prec) nella lista, con first.prec = last. Mostrare le istruzioni necessarie a inserire ed eliminare un elemento da una lista circolare doppia.

1.15 Dato un albero binario, dove ogni nodo ha un puntatore al padre, e dati due nodi x e y dell'albero, progettare un algoritmo che determini il loro minimo antenato comune.

CAPITOLO

2 Pile e code

In questo capitolo, definiamo e analizziamo due strutture di dati comunemente utilizzate in contesti informatici (e non solo) per la gestione di sequenze lineari dinamiche, ovvero le pile e le code. Per ciascuna di esse, descriviamo due diversi possibili modi di implementarla e forniamo poi alcuni esempi significativi di applicazione.

2.1 Pile
2.2 Opus libri: Postscript e notazione postfissa
2.3 Code
2.4 Code con priorità: heap
2.5 Esercizi

2.1 Pile

Una **pila** è una collezione di elementi in cui le operazioni disponibili, come l'estrazione di un elemento, sono ristrette unicamente a quello più recentemente inserito. Questa politica di accesso, detta LIFO (*Last In First Out*), comporta che l'ordine con cui gli elementi sono estratti dalla pila è opposto rispetto all'ordine dei relativi inserimenti e, per esempio, riflette quanto avviene per una pila di vassoi, in cui il vassoio che possiamo prendere è sempre quello in cima alla pila, che è anche l'ultimo a essere stato riposto.

L'insieme delle operazioni caratteristiche di una pila è composto da tre operazioni, due delle quali, rispettivamente, inseriscono ed estraggono l'elemento in cima alla pila, mentre la terza restituisce tale elemento senza estrarlo. In particolare, la prima operazione prende il nome di `Push` e inserisce un nuovo elemento in cima alla pila; la seconda è detta `Pop` ed estrae l'elemento in cima alla pila restituendo l'informazione in esso contenuta; la terza è detta `Top` e restituisce l'informazione contenuta nell'elemento in cima alla pila senza estrarlo. In alcune applicazioni è utile avere anche l'operazione `Empty` che verifica se la pila è vuota o meno.

Come vedremo, ogni operazione invocata su di una pila può essere eseguita in tempo costante, indipendentemente dal numero di elementi contenuti nella pila stessa: che ciò sia possibile può essere verificato immediatamente considerando il caso della pila di vassoi, in quanto riporre o prendere un vassoio richiede lo stesso tempo, indipendentemente da quanti siano i vassoi sovrapposti. In effetti, se gli elementi nella pila sono mantenuti ordinati secondo l'istante di inserimento, tutte e tre le operazioni agiscono su un'estremità (la cima della pila) della sequenza. Basta quindi avere la possibilità di accedere direttamente a tale estremità per effettuare le operazioni in tempo indipendente dalla dimensione della pila: in questo paragrafo proponiamo due specifiche implementazioni della struttura di dati pila, che consentono effettivamente di fare ciò.

2.1.1 Implementazione di una pila mediante un array

Una pila può essere implementata utilizzando un array. In particolare, gli elementi della pila sono memorizzati in un array di dimensione iniziale pari a una costante predefinita. Successivamente, la dimensione dell'array viene raddoppiata o dimezzata per garantire che sia proporzionale al numero di elementi effettivamente contenuti nella pila: l'analisi del metodo di ridimensionamento di un array (discussa nel Paragrafo 1.1.3) mostra che occorre un tempo costante ammortizzato per operazione.

Gli elementi della pila sono memorizzati in sequenza nell'array a partire dalla locazione iniziale, inserendoli man mano nella prima locazione disponibile: ciò comporta che la "cima" della pila corrisponde all'ultimo elemento di tale

sequenza. Basterà quindi tenere traccia dell'indice della locazione che contiene l'ultimo elemento della sequenza per implementare le operazioni `Push`, `Pop`, `Top` e `Empty` in modo che richiedano tempo costante ammortizzato, come mostrato nel Codice 2.1, in cui ipotizziamo che la pila sia rappresentata per mezzo di un array `pilaArray` di dimensione variabile (gestito mediante le funzioni, definite nel Paragrafo 1.1.3, `VerificaRaddoppio` e `VerificaDimezzamento`).

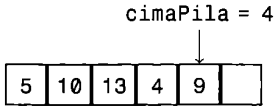
La cima della pila corrisponde all'elemento dell'array il cui indice è memorizzato nella variabile `cimaPila`, inizialmente posta uguale a –1. Facendo uso di tale informazione, le operazioni di accesso alla pila sono molto semplici da realizzare. Infatti, l'elemento in cima alla pila sarà sempre `pilaArray[cimaPila]`. L'operazione `Push` incrementa `cimaPila`, dopo avere verificato che l'array non sia pieno (nel qual caso la sua dimensione andrà raddoppiata). L'operazione `Pop` richiede di verificare se la pila non è vuota invocando la funzione `Empty` e, in tal caso, di decrementare il valore di `cimaPila`, verificando che l'array non sia poco popolato (nel qual caso la sua dimensione andrà dimezzata). Osserviamo come, nel caso di un'operazione `Pop`, il contenuto dell'elemento dell'array che si trova nella posizione specificata da `cimaPila` non debba essere necessariamente azzerato, in quanto, nel momento in cui faremo di nuovo accesso a tale elemento, il suo contenuto sarà stato modificato dalla corrispondente operazione `Push`.

Codice 2.1 Implementazione di una pila mediante un array: le funzioni `VerificaRaddoppio` e `VerificaDimezzamento` seguono l'approccio del Paragrafo 1.1.3.

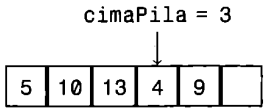
```
Push( x ):
  VerificaRaddoppio( );
  cimaPila = cimaPila + 1;
  pilaArray[ cimaPila ] = x;

Pop( ):
  IF (!Empty( )) {
    x = pilaArray[ cimaPila ];
    cimaPila = cimaPila - 1;
    VerificaDimezzamento( );
    RETURN x;
  }

Top( ):
  IF (!Empty( )) RETURN pilaArray[ cimaPila ];

Empty( ):
  RETURN (cimaPila == -1);
```

ESEMPIO 2.1

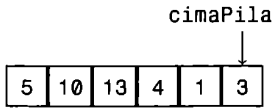
La pila illustrata di seguito contiene elementi di tipo intero.



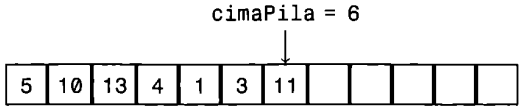
L'operazione `Top()` restituisce il valore 9 mentre `Empty()` restituisce FALSE. L'esecuzione dell'operazione `Pop()` restituisce il valore 9 e trasforma la pila come segue.



Dopo l'esecuzione delle sequenza di operazioni `Push(1); Push(3)` la pila apparirà nel modo seguente.



Un'ulteriore operazione di inserimento (`Push(11)`) causerà il ridimensionamento dell'array eseguito dalla funzione `VerificaRaddoppio()`.



2.1.2 Implementazione di una pila mediante una lista

Una pila può essere implementata anche utilizzando una lista i cui elementi sono mantenuti ordinati in base al loro tempo di inserimento decrescente. In tal modo, la "cima" della pila corrisponde all'inizio della lista, e le operazioni agiscono tutte sull'elemento iniziale della lista stessa. Nel Codice 2.2, il riferimento all'elemento in cima alla pila è memorizzato nella variabile `cimaPila` e ciascun elemento contiene oltre all'informazione un riferimento all'elemento successivo (`cimaPila` è `null` nel caso in cui la pila sia vuota). Facendo uso di tali riferimenti, le operazioni di accesso alla pila sono quindi altrettanto semplici da realizzare come quelle esaminate nel paragrafo precedente. La modalità di allocazione di un nodo nella lista (riga 2 in `Push`) dipende dal linguaggio di programmazione adottato.

Codice 2.2 Implementazione di una pila mediante una lista.

```
1  Push( x ):
2    u = NuovoNodo( );
3    u.dato = x;
4    u.succ = cimaPila;
5    cimaPila = u;
```

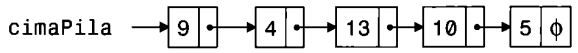
```
1  Pop( ):
2    IF (!Empty( )) {
3      x = cimaPila.dato;
4      cimaPila = cimaPila.succ;
5      RETURN x;
6    }
```

```
1  Top( ):
2    IF (!Empty( )) RETURN cimaPila.dato;
```

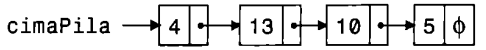
```
1  Empty( ):
2    RETURN (cimaPila == null);
```

ESEMPIO 2.2

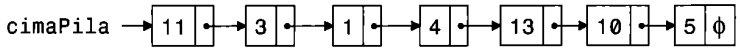
L'implementazione mediante lista della pila dell'esempio precedente può essere rappresentata graficamente come segue.



L'operazione `Pop()`, oltre a restituire `cimaPila.dato = 9`, trasforma la pila nel seguente modo.



Infine ecco lo stato della pila dopo la sequenza di operazioni `Push(1)`, `Push(3)` e `Push(11)`.



2.2 Opus libri: Postscript e notazione postfissa

Postscript è un linguaggio di programmazione per la grafica che viene eseguito da un interprete che utilizza una pila e la notazione postfissa o polacca inversa definita di seguito. Se un'operazione in Postscript ha k argomenti, questi ultimi si trovano nelle k posizioni in cima alla pila, ovvero l'esecuzione di k operazioni `Pop` fornisce gli argomenti all'operazione in Postscript, il cui risultato viene po-

sto sulla pila tramite un'operazione Push. Grazie alla sua ampia diffusione, il linguaggio Postscript è diventato uno degli standard tipografici principalmente adottati, insieme alla sua evoluzione PDF (*Portable Document Format*), per la stampa professionale (inclusa quella del presente libro). Il principio del suo funzionamento basato sulla pila è intuitivo e possiamo illustrarlo usando le espressioni aritmetiche come esempio.

È uso comune in matematica scrivere l'operatore tra gli operandi, come in A + B, piuttosto che dopo gli operandi, come in AB+ (nel seguito, supponiamo che gli operatori siano tutti binari). La prima forma si chiama **notazione infissa**, mentre la seconda si chiama **postfissa** o **polacca inversa**, dalla nazionalità del matematico Łukasiewicz che ne studiò le proprietà.

La notazione postfissa ha alcuni vantaggi rispetto a quella infissa. Anzitutto, le espressioni scritte in notazione postfissa non hanno bisogno di parentesi (l'ordine degli operandi viene preservato rispetto all'infissa). In secondo luogo, non è necessario specificare una priorità, talvolta arbitraria, degli operatori (per esempio, il fatto che A + B × C sia equivalente ad A + (B × C) è dovuto al fatto che la moltiplicazione ha, in base a una definizione arbitraria, priorità superiore alla somma).

Infine, tali espressioni si prestano a essere valutate semplicemente, da sinistra a destra, mediante l'uso di una pila applicando le seguenti regole in base al simbolo letto. Supponiamo di avere le seguenti operazioni binarie:

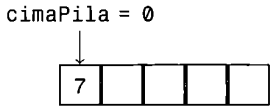
- operando: viene eseguita la Push di tale operando;
- operatore: vengono eseguite due Pop, l'operatore viene applicato ai due operandi prelevati dalla pila (nel giusto ordine) e viene eseguita la Push del risultato.

ESEMPIO 2.3

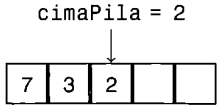
Si vuole calcolare il valore della seguente espressione in notazione postfissa

7 3 2 + ×

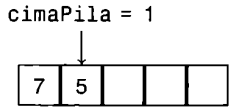
L'espressione viene scandita da sinistra a destra: il primo elemento è un operando e quindi viene inserito in una pila inizialmente vuota.



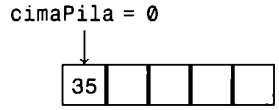
Anche i due elementi che seguono (3 e 2) sono operandi quindi vengono aggiunti in cima alla pila.



Per valutare l'operatore + si prelevano dalla pila due operandi 2 e 3 con due istruzioni Pop() e il risultato della loro somma viene inserito in cima alla pila con l'operazione Push(5).



L'ultimo elemento della sequenza è ancora un operatore (×) e quindi, come nel passo precedente, con due Pop() si ottengono i due operandi necessari, viene eseguita l'operazione di moltiplicazione e il risultato di questa viene inserito in cima alla pila con una Push().



Terminata la scansione dell'espressione, la cima della pila contiene il risultato finale.

Oltre che per la valutazione di espressioni algebriche in notazione polacca inversa, il tipo di dati pila può essere usato anche per convertire un'espressione algebrica in forma infissa in un'espressione algebrica equivalente in forma postfissa.

Supponiamo per semplicità che le parentesi siano comunque esplicitate, per cui un'espressione è data da una coppia di parentesi al cui interno ci sono due espressioni (ricorsivamente definite) separate da un operatore. A questo punto la trasformazione avviene leggendo l'espressione infissa da sinistra a destra e applicando le seguenti regole in base al simbolo letto:

- parentesi aperta: viene ignorata;
- operando: viene accodato direttamente in fondo all'espressione postfissa in costruzione senza passare per la pila;
- operatore: viene eseguita la Push di tale operatore;
- parentesi chiusa: viene eseguita la Pop per riprendere l'operatore che viene accodato in fondo all'espressione postfissa in costruzione.

In realtà, non abbiamo bisogno di imporre le parentesi nell'espressione infissa quando non sono necessarie. Per verificare tale affermazione, ipotizziamo che l'espressione infissa sia composta dei seguenti simboli: variabili e costanti (ovvero, lettere alfanumeriche), operatori binari (ovvero, +, −, ×, /) e parentesi (ovvero, (e)). Supponiamo inoltre che l'espressione infissa sia sintatticamente corretta (per esempio, non sia A + × B) e sia terminata dal simbolo speciale $. L'esecuzione delle azioni inizia ponendo una copia del simbolo $ nella pila vuota, e ha termine quando l'espressione infissa diviene vuota.

	Simbolo corrente dell'espressione infissa				
Cima della pila	$	+ oppure –	× oppure /	(	)
$	4	1	1	1	
+ oppure –	2	2	1	1	2
× oppure /	2	2	2	1	2
(		1	1	1	3

Figura 2.1 Regole per la conversione di un'espressione infissa in una postfissa.

Durante la conversione, se il simbolo corrente nell'espressione infissa è un operando (una variabile o una costante), esso viene accodato direttamente in fondo all'espressione postfissa in costruzione senza passare per la pila. Altrimenti, il simbolo corrente è un operatore, una parentesi oppure il simbolo $, e le regole per elaborare tale simbolo sono rappresentate succintamente nella Figura 2.1, dove il numero contenuto all'incrocio di una riga e di una colonna rappresenta una delle seguenti azioni, da intraprendere quando il simbolo di riga è sulla cima della pila e il simbolo di colonna è quello attualmente letto nell'espressione infissa:

1. viene eseguita la `Push` del simbolo corrente dell'espressione infissa e si passa al simbolo successivo della sequenza infissa;
2. viene eseguita una `Pop` e l'operatore così ottenuto viene accodato in fondo all'espressione postfissa in costruzione;
3. il simbolo corrente viene ignorato nell'espressione infissa e viene eseguita una `Pop` (ignorando il simbolo restituito) e si passa al simbolo successivo della sequenza infissa;
4. la conversione ha avuto termine, il simbolo corrente viene cancellato dall'espressione infissa e viene eseguita una `Pop` (ignorando il simbolo restituito).

Usando le regole nella tabella della Figura 2.1, possiamo trasformare anche espressioni che hanno delle parentesi implicitamente definite dall'associatività a sinistra e dalla precedenza degli operatori, come nel caso di `6 + (5 - 4) × (1 + 2 + 3)`. Il costo computazionale dell'algoritmo di conversione e di valutazione è `O(n)` tempo per un'espressione di `n` simboli, ipotizzando che il costo di valutazione di un singolo operatore sia costante e quindi non dipenda dalla lunghezza dell'espressione.

Teorema 2.1 *L'algoritmo di conversione da notazione infissa a postfissa le cui regole sono definite nella tabella della Figura 2.1 è corretto.*

Dimostrazione Consideriamo l'espressione infissa $e = e_1 \alpha e_2$ dove α è un operatore ed e_1 ed e_2 espressioni infisse. Per prima cosa osserviamo che nel momento in cui α viene messo nella pila, nessun operatore di e_1 è nella pila: infatti se la pila contenesse un operatore α' di e_1, al momento in cui viene considerato il simbolo α della sequenza di input verrebbe applicata la regola 2, ovvero α' verrebbe estratto dalla pila e accodato alla sequenza di output. Solo nei passi successivi, quando ogni operatore di e_1 è stato tolto dalla pila, l'operatore α vi verrebbe inserito utilizzando la regola 1: questo succederebbe solo quando la cima della pila fosse occupata da $ o (oppure un operatore con priorità minore di α, ovvero non appartenente a e_1.

Indichiamo con $|e|$ il numero di operatori che compongono l'espressione e. Dimostreremo, per induzione su $n = |e|$, che quando tutti gli operatori di e vengono estratti dalla pila, in output l'algoritmo ha prodotto la versione postfissa dell'espressione e che denoteremo $p(e)$.

Se $n = 0$ l'espressione e è composta da un solo operando che viene appeso in output senza passare per la pila.

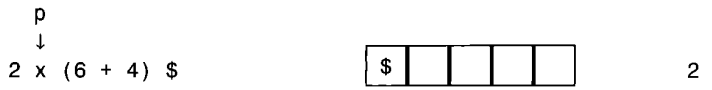
Se $n > 0$, al momento in cui viene analizzato il simbolo di input α e questo viene messo in coda tutti gli $|e_1| < n$ operatori di e_1 sono stati estratti dalla coda e, per ipotesi induttiva, la sequenza $p(e_1)$ è stata accodata alla sequenza di output. L'operatore α viene messo nella pila e si passa alla scansione della sequenza e_2. Gli operatori che la compongono vengono inseriti nella pila fino a raggiungere il carattere $ o) oppure un altro operatore di priorità minore. Questi caratteri denotano la fine dell'espressione e_2 e inducono una serie di applicazioni della regola 2: ovvero, tutti gli operatori di e_2 vengono estratti dalla pila. Poiché $|e_2| < n$, per ipotesi induttiva $p(e_2)$ viene accodato in output. Infine sulla cima della pila troviamo l'operatore α che verrà estratto dalla pila e accodato alla sequenza postfissa producendo come risultato $p(e_1)p(e_2)\alpha$. ❑

ESEMPIO 2.4

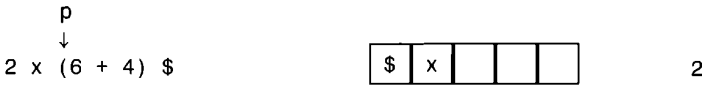
Applichiamo l'algoritmo di trasformazione sull'espressione infissa 2 × (6 + 4). Supponiamo che la sequenza sia memorizzata in un array in cui ogni elemento della sequenza occupa un elemento dell'array. Scandiamo la sequenza utilizzando una variabile intera p inizializzata a zero, che indica la posizione dell'elemento della sequenza attualmente in esame.

D'ora in poi, a sinistra viene visualizzata la sequenza postfissa di input evidenziando la posizione di p, al centro lo stato della pila e a destra l'effetto dell'operazione corrente sull'espressione postfissa parziale inizialmente vuota.

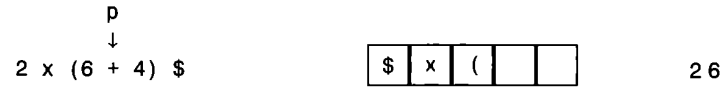
Il primo elemento della sequenza è un operatore, questo viene accodato all'espressione postfissa parziale e p viene spostato di una posizione a destra.



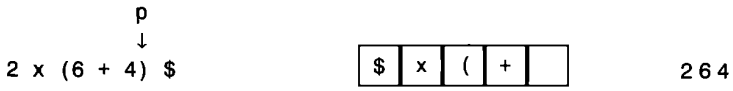
La variabile p punta a un operatore mentre in cima alla pila troviamo il simbolo $, pertanto viene applicata la regola 1 della tabella nella Figura 2.1 (ovvero `Push(×)`) e p viene spostato di una posizione a destra.



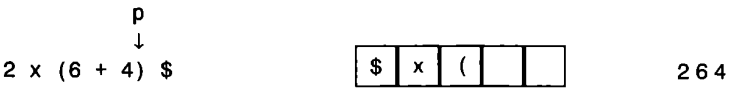
Viene ripetuta la stessa operazione del passo precedente per i simboli (e x rispettivamente sulla sequenza di input e in cima alla pila. Il prossimo simbolo dell'espressione infissa è un operando, il quale viene accodato alla sequenza di output senza passare per la pila e viene incrementata p.



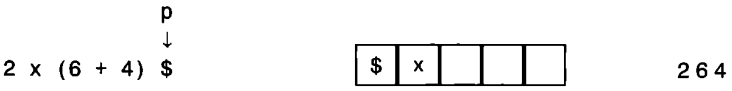
Si applica ancora la regola 1 per il carattere + in input. Al passo successivo l'operando 4 viene accodato in output.



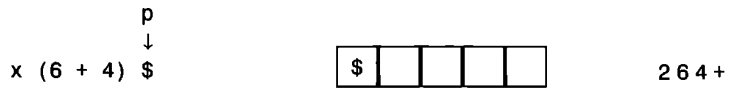
Ora si applica le regola 2, ovvero con `Pop()` si accede all'elemento in cima alla pila (+) che viene accodato alla sequenza postfissa parziale ed eliminato dalla pila; p non viene incrementata.



Il simbolo) sulla sequenza di input e (sulla pila comporta l'esecuzione di una operazione `Pop()` ignorando il simbolo restituito e l'incremento di p (regola 3).



Segue un'applicazione della regola 2: x, il risultato di `Pop()`, viene aggiunto all'espressione postfissa.



Infine, non avendo spostato p, il simbolo $ appare sia come elemento attuale della sequenza di input che in cima alla pila: viene applicata la regola 4 restituendo l'output 2 6 4 + x. La conversione ha così termine.

Esercizio svolto 2.1 Un intervallo `[a, b]` di interi rappresenta l'insieme {a, a + 1, a + 2, ..., b}, dove a ≤ b. Per esempio, `[15, 17]` rappresenta l'insieme {15, 16, 17}. L'unione tra intervalli può essere vista come l'unione tra i corrispondenti insiemi di interi. Per esempio, l'unione di `[8, 9]`, `[10, 10]`, `[15, 17]`, `[16, 19]` e `[18, 18]` rappresenta l'insieme {8, 9, 10, 15, 16, 17, 18, 19}.

Vale la proprietà che l'unione di intervalli può essere sempre rappresentata in modo univoco come una lista ordinata L di intervalli *disgiunti e non adiacenti*, in quanto gli intervalli adiacenti del tipo `[a, b]` e `[b + 1, c]` possono essere sempre visti come un unico intervallo `[a, c]`. Per esempio, l'unione degli intervalli `[8, 9]`, `[10, 10]`, `[15, 17]`, `[16, 19]` e `[18, 18]` è rappresentato da L = `[8,10]`, `[15, 19]`. Dati in ingresso 2n interi a_0, b_0, a_1, b_1, ..., a_{n-1}, b_{n-1}, interpretati come n intervalli $[a_0, b_0]$, $[a_1, b_1]$, ..., $[a_{n-1}, b_{n-1}]$, si vuole stampare la lista L che rappresenta la loro unione. Progettare un algoritmo che risolva il problema in tempo $O(n \log n)$ secondo il seguente schema: (1) ordina gli intervalli in base al valore dei loro estremi sinistri a_i (ipotizzando che siano tutti distinti); (2) scandisce gli intervalli secondo l'ordine di cui sopra e li fonde utilizzando una struttura di dati "pila".

Soluzione Vediamo prima una soluzione $O(n^2)$ per catturare l'idea algoritmica di base. L'approccio segue uno schema induttivo per l'intervallo $[a_i, b_i]$. Passo base $i = 0$: $L = [a_0, b_0]$. Passo induttivo $i > 0$, che elabora $[a_i, b_i]$ in $O(n)$ tempo. Sia $L = [c_0, d_0], ..., [c_{k-1}, d_{k-1}]$ la lista attualmente in uso:

- se $[a_i, b_i]$ è contenuto in un intervallo di L, salta al passo $i + 1$;
- altrimenti:
  - elimina da L tutti gli intervalli $[c_j, d_j]$ tali che $[c_j, d_j] \subseteq [a_i, b_i]$ ottenendo la lista L′;
  - trova, se esiste, l'unico intervallo $[c', d']$ in L′ che interseca $[a_i, b_i]$ oppure è adiacente a esso, dove $c' < a_i$; altrimenti, poni $c' = a_i$;
  - trova, se esiste, l'unico intervallo $[c'', d'']$ in L′ che interseca $[a_i, b_i]$ oppure è adiacente a esso, dove $b_i < d''$; altrimenti, poni $d'' = b_i$;
  - aggiorna $L = L' \cup [c', d'']$.

Infine, restituisci $L = [c_0, d_0], ..., [c_{n-1}, d_{n-1}]$, la lista ottenuta dopo il passo $i = n - 1$.

Per ridurre il costo computazionale a $O(n \log n)$ tempo, sia $[a_0, b_0]$, $[a_1, b_1]$, ..., $[a_{n-1}, b_{n-1}]$ la sequenza di intervalli ordinati risultanti dal passo (1) e che quindi richiede $O(n \log n)$ tempo. La scansione degli intervalli richiede $O(n)$ tempo. Inizialmente, metti $[a_0, b_0]$ nella pila. Per il passo $i > 0$, sia `[a, b]` l'intervallo in cima alla pila e $[a_i, b_i]$ l'intervallo corrente:

- se $a_i \leq b + 1$, fai `Pop` di `[a, b]` e `Push` di $[a, \max\{b, b_i\}]$. Deve valere $a < a_i$ per il punto (1);
- se $a_i > b + 1$, fai `Push` di $[a_i, b_i]$.

Infine, stampa il contenuto della pila alla fine dell'ultimo passo.

2.3 Code

Analogamente a una pila, una coda è una collezione di elementi in cui le operazioni disponibili sono definite da una specifica politica di accesso: mentre in una pila l'accesso è consentito solo all'ultimo elemento inserito, in una coda estraiamo il primo elemento, in "testa" alla coda, essendo presente da più tempo, mentre inseriamo un nuovo elemento in fondo alla coda, perché è più recente. Ciò corrisponde a quanto avviene in molte situazioni quotidiane come, per esempio, nel pagare un pedaggio autostradale, nel fare acquisti in un negozio e, in generale, nel ricevere una serie di eventi o richieste da elaborare. Una politica del tipo suddetto viene detta FIFO (*First In First Out*) in quanto il primo elemento a essere inserito nella coda è anche il primo a essere estratto.

Le operazioni principali definite su una coda permettono di inserire un nuovo elemento nella coda e di estrarre un elemento dalla coda stessa in tempo costante: `Enqueue` inserisce un nuovo elemento in fondo alla coda, `Dequeue` estrae l'elemento dalla testa della coda e restituisce l'informazione in esso contenuta e `First` restituisce l'informazione contenuta nell'elemento in testa alla coda, senza estrarre tale elemento. Infine, l'operazione `Empty` verifica se la coda è vuota. Mentre in una pila c'è un unico punto di accesso su cui le operazioni vanno a incidere (quello corrispondente alla "cima" della pila), nel caso di una coda ne esistono due, ovvero le estremità della coda stessa, in quanto `First` e `Dequeue` vanno a operare sulla "testa", mentre `Enqueue` incide sul "fondo".

2.3.1 Implementazione di una coda mediante un array

L'implementazione di una coda mediante un array consiste nel memorizzare gli elementi della coda in un array di dimensione variabile. Gli elementi della coda sono memorizzati in sequenza nell'array a partire dalla locazione associata all'inizio della coda, inserendoli man mano nella prima locazione disponibile: ciò comporta che la fine della coda corrisponde all'ultimo elemento di tale sequenza. Basterà quindi tenere traccia dell'indice (indicato con `testaCoda`) della locazione che contiene il primo elemento della sequenza e di quello (indicato con `fondoCoda`) della locazione in cui poter inserire il prossimo elemento per implementare le operazioni `First`, `Enqueue` e `Dequeue` in tempo costante ammortizzato. Tuttavia, per poter sfruttare al meglio lo spazio a disposizione e non dover spostare gli elementi ogni volta che un oggetto viene estratto dalla coda, la gestione dei due indici `testaCoda` e `fondoCoda` avviene in modo "circolare" rispetto alle locazioni disponibili nell'array. In altre parole, se uno di questi due indici supera la fine dell'array, allora esso viene azzerato facendo in modo che indichi la prima locazione dell'array stesso.

Nel Codice 2.3, utilizziamo tre interi `cardCoda`, `testaCoda` e `fondoCoda` che sono stati inizializzati rispettivamente con i valori `0`, `0` e `-1`: inizialmente la coda è vuota (quindi, contiene `cardCoda = 0` elementi e `testaCoda` vale `0`) e il prossimo elemento potrà essere inserito nella prima locazione dell'array (quindi, `fondoCoda` vale `-1` perché viene prima incrementato). Il metodo `Empty` si limita a verificare se il valore di `cardCoda` è uguale a 0. L'operazione di incremento di `testaCoda` e `fondoCoda` è circolare, per cui adoperiamo il modulo della divisione intera a tal fine (riga 4 in `Enqueue` e riga 5 in `Dequeue`, nelle quali supponiamo che la lunghezza dell'array sia memorizzata nella variabile `lunghezzaArray`). Inoltre, osserviamo che, quando `arrayCoda` deve essere raddoppiato o dimezzato, le funzioni `VerificaRaddoppio` e `VerificaDimezzamento` ricollocano ordinatamente gli elementi della coda nelle prime celle dell'array e pongono `testaCoda` a `0` e `fondoCoda` a `cardCoda - 1`.

Codice 2.3 Implementazione di una coda mediante un array: le funzioni `VerificaRaddoppio` e `VerificaDimezzamento` seguono l'approccio del Paragrafo 1.1.3 e aggiornano anche i valori di `testaCoda` e di `fondoCoda`.

```
1  Enqueue( x ):
2    VerificaRaddoppio( );
3    cardCoda = cardCoda + 1;
4    fondoCoda = (fondoCoda + 1) % lunghezzaArray;
5    codaArray[ fondoCoda ] = x;
```

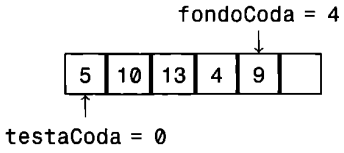
```
1  Dequeue( ):
2    IF (!Empty( )) {
3      cardCoda = cardCoda - 1;
4      x = codaArray[ testaCoda ];
5      testaCoda = (testaCoda + 1) % lunghezzaArray;
6      VerificaDimezzamento( );
7      RETURN x;
8    }
```

```
1  First( ):
2    IF (!Empty( )) RETURN codaArray[ testaCoda ];
```

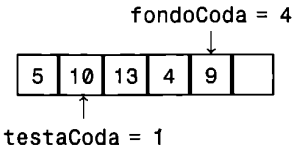
```
1  Empty( ):
2    RETURN (cardCoda == 0);
```

ESEMPIO 2.5

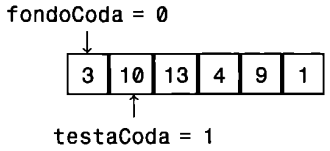
La coda illustrata di seguito contiene elementi di tipo intero.



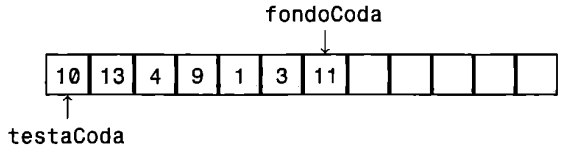
L'operazione `First()` restituisce il valore 5 mentre `Empty()` restituisce FALSE. L'esecuzione dell'operazione `Dequeue()` restituisce il valore 5 e trasforma la coda come segue.



Dopo l'esecuzione delle sequenza di operazioni `Enqueue(1); Enqueue(3)` la coda apparirà nel modo seguente.



Un'ulteriore operazione di inserimento `Enqueue(11)` causerà il ridimensionamento dell'array eseguito dalla funzione `VerificaRaddoppio()`.



2.3.2 Implementazione di una coda mediante una lista

L'implementazione più naturale di una coda mediante una struttura con riferimenti consiste in una sequenza di nodi concatenati e ordinati in modo crescente secondo l'istante di inserimento. In tal modo, il primo nodo della sequenza corrisponde alla "testa" della coda ed è il nodo da estrarre nel caso di una `Dequeue`, mentre l'ultimo nodo corrisponde al "fondo" della coda ed è il nodo al quale concatenare un nuovo nodo, inserito mediante `Enqueue`. Lasciamo al lettore il compito di definire tale implementazione sulla falsariga di quanto fatto per la pila nel Codice 2.2 e per le liste doppie nel Paragrafo 1.3.2.

2.4 Code con priorità: heap

La struttura di dati **coda con priorità** memorizza una collezione di elementi in cui a ogni elemento è associato un valore, detto **priorità**, appartenente a un insieme totalmente ordinato (solitamente l'insieme degli interi positivi). La coda con priorità può essere vista come un'estensione della coda (Paragrafo 2.3) e, infatti, le operazioni disponibili sono le stesse della coda: `Empty`, `Enqueue`, `First` e `Dequeue`. L'unica e sostanziale differenza rispetto a tale tipo di dati è che le ultime due operazioni devono restituire (ed estrarre, nel caso della `Dequeue`) l'elemento di priorità massima (nel seguito, considereremo sempre il caso in cui la coda con priorità restituisca il massimo, ma le stesse considerazioni possono essere applicate *mutatis mutandis* al caso in cui dobbiamo restituire il minimo). Ai fini della discussione, ipotizziamo che ciascun elemento `e` memorizzato in una coda con priorità contenga due campi, ossia un campo `e.prio` per indicarne la priorità e un campo `e.dato` per indicare il dato a cui associare quella priorità.

La necessità di estrarre gli elementi dalla coda in funzione della loro priorità, ne rende l'implementazione più complessa rispetto a quella della semplice coda. Per convincerci di ciò consideriamo la semplice implementazione di una coda con priorità mediante una lista dei suoi elementi. Nel caso in cui decidiamo di implementare in tempo costante l'operazione `Enqueue`, inserendo i nuovi elementi in corrispondenza a un estremo della lista, ne deriva che la lista non sarà ordinata: per l'implementazione di `Dequeue` e di `First` è necessario individuare l'elemento di priorità massima all'interno della lista e, quindi, tale operazione richiederà tempo `O(n)`, dove `n` è la lunghezza della lista. Se decidiamo, invece, di mantenere gli elementi della lista ordinati rispetto alla loro priorità (per esempio, in ordine non crescente), ne deriva che `Dequeue` e `First` richiederanno `O(1)` tempo, in quanto l'elemento di priorità massima sarà sempre il primo della lista. Al tempo stesso, però, in corrispondenza a ogni `Enqueue` dovremo utilizzare tempo `O(n)` per inserire il nuovo elemento nella giusta posizione della lista, corrispondente all'ordinamento degli elementi nella lista stessa.

Facendo uso di una soluzione più sofisticata è possibile implementare una coda con priorità in modo più efficiente, attraverso un bilanciamento del costo di esecuzione delle operazioni `Dequeue` e `Enqueue`, ottenuto per mezzo di un'organizzazione dell'insieme degli elementi in cui questi siano ordinati solo in parte. Inizialmente descriveremo tale soluzione facendo riferimento a una particolare organizzazione ad albero degli elementi stessi: mostreremo poi come tale organizzazione possa essere riportata in termini di array facendo uso di una diversa rappresentazione dell'albero. Negli esempi che seguiranno, inoltre, indicheremo sempre solo le priorità degli elementi contenuti nei nodi, senza rappresentare mai esplicitamente il contenuto dei campi `dato`.

2.4.1 Definizione di heap

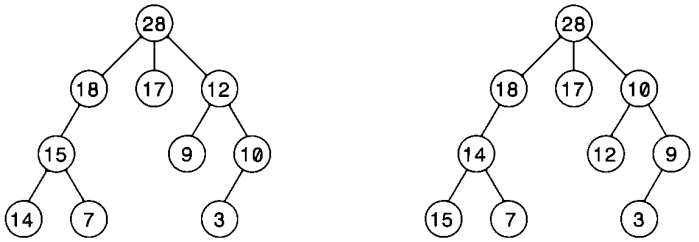
Uno **heaptree** è un albero vuoto oppure un albero H che soddisfa la seguente proprietà di heap, dove r indica la radice di H e v_0, v_1, ..., v_{k-1} i suoi figli:

1. la priorità dell'elemento contenuto nella radice di H è maggiore o uguale di quelle dei figli della radice, ovvero r.prio ≥ v_i.prio, per 0 ≤ i < k;
2. l'albero radicato in v_i è uno heaptree per 0 ≤ i < k.

Come corollario immediato della definizione, abbiamo che la radice di uno heaptree contiene un elemento di priorità massima dell'insieme. Di conseguenza, l'effettuazione dell'operazione First nel caso di una coda con priorità implementata con uno heaptree richiede O(1) tempo.

ESEMPIO 2.6

L'albero nella parte sinistra della seguente figura è uno heaptree, a differenza di quello nella parte destra in cui, per esempio, il nodo con priorità 14 è padre del nodo con priorità 15 e il nodo con priorità 10 è padre del nodo con priorità 12.



L'ordine gerarchico esistente tra i nodi di un albero permette di classificarli in base alla loro **profondità**. La radice r ha profondità 0, i suoi figli hanno profondità 1 (se diversi da null), i nipoti hanno profondità 2 e così via: in generale, se la profondità di un nodo è pari a p, allora i suoi figli non vuoti (ovvero diversi da null) hanno profondità p + 1. L'**altezza** h di un albero è data dalla massima profondità raggiunta dalle sue foglie: quindi, l'altezza misura la massima distanza di una foglia dalla radice dell'albero, in termini del numero di archi attraversati. Per esempio, l'altezza dell'albero mostrato nella figura dell'Esempio 1.9 è 3, mentre quella dell'albero mostrato nella figura dell'Esempio 1.10 è 4.

Un albero binario è **completo** se ogni nodo interno ha esattamente due figli non vuoti. L'albero è **completamente bilanciato** se, oltre a essere completo, tutte le foglie hanno la stessa profondità. Un albero binario di altezza h è **completo a sinistra** se i nodi di profondità minore di h formano un albero completamente bilanciato, e se i nodi di profondità h sono tutti accumulati a sinistra.

Lemma 2.1 L'altezza h di un albero completamente bilanciato con n nodi è uguale a log(n + 1) - 1.

Dimostrazione Un albero completamente bilanciato di altezza h ha $2^h - 1$ nodi interni e 2^h foglie: ne deriva che la relazione tra l'altezza h e il numero di nodi è la seguente: $n = 2^h - 1 + 2^h = 2^{h+1} - 1$. Quindi, h = log(n + 1) - 1. ❑

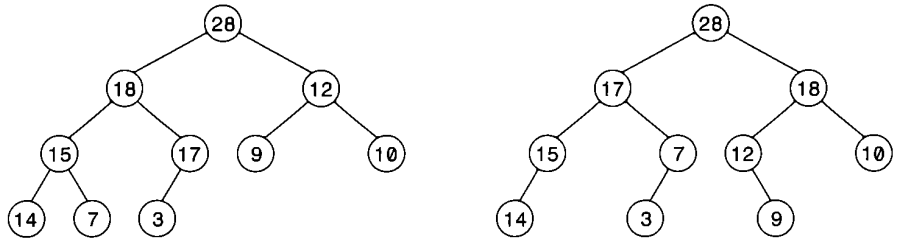
Corollario 2.1 Se h è l'altezza di un albero completo a sinistra con n nodi, allora h = O(log n).

Dimostrazione Dal Lemma 2.1 segue che, se m è il numero di nodi a profondità minore di h, allora h = log(m + 1) < log(n + 1) = O(log n). ❑

Uno **heap** è uno heaptree con i vincoli aggiuntivi di essere binario e completo a sinistra: dal corollario precedente, segue che uno heap con n nodi ha altezza pari a h = O(log n).

ESEMPIO 2.7

L'albero binario nella parte sinistra della seguente figura è uno heap, mentre quello nella parte destra, pur essendo uno heaptree, non è uno heap in quanto non è completo a sinistra.



Il fatto che l'altezza di uno heap sia logaritmica nel numero di nodi ci consente di effettuare in tempo O(log n) le operazioni Enqueue e Dequeue, di cui forniamo uno schema algoritmico generale, mostrando successivamente come realizzare tali algoritmi facendo uso di un array. L'operazione Enqueue di un elemento e in uno heap H prevede, ad alto livello, l'esecuzione dei seguenti passi.

1. Inseriamo un nuovo nodo v contenente e come foglia di H in modo da mantenere H completo a sinistra.
2. Iterativamente, confrontiamo la priorità di e con quella dell'elemento f contenuta nel padre di v e, se e.prio > f.prio, scambiamo i due nodi: l'iterazione termina quando v diventa la radice oppure quando e.prio ≤ f.prio (notiamo che in questo modo manteniamo la proprietà di uno heaptree).

Come possiamo vedere, l'operazione `Enqueue` opera inserendo un elemento nella sola posizione che consente di preservare la completezza a sinistra dello heap, facendo poi risalire l'elemento nello heap, di figlio in padre, fino a trovare una posizione che soddisfi la proprietà di uno heaptree. Pertanto, tale operazione richiede tempo `O(logn)`: a tal fine, ci basta osservare che il numero di passi effettuati è proporzionale al numero di elementi con `i` quali `e` viene confrontato e che tale numero è al massimo pari all'altezza `h = O(logn)` dello heap (in quanto `e` viene confrontato con al più un elemento per ogni livello dello heap).

Passiamo a considerare ora l'operazione `Dequeue`, che può essere effettuata, sempre ad alto livello, su uno heap `H` nel modo seguente.

1. Estraiamo la radice di `H` in modo da restituire l'elemento in essa contenuto alla fine dell'operazione.
2. Rimuoviamo l'ultima foglia di `H` (quella più a destra nell'ultimo livello), per inserirla come radice `v` (al posto di quella estratta), al fine di mantenere `H` completo a sinistra.
3. Iterativamente, confrontiamo la priorità dell'elemento in `v` con quelle degli elementi nei suoi figli e, se il massimo fra le priorità non è quella di `v`, il nodo `v` viene scambiato con il figlio contenente un elemento di priorità massima: l'iterazione termina se `v` diventa una foglia o se contiene un elemento la cui priorità è maggiore o uguale a quelle degli elementi contenuti nei suoi figli.

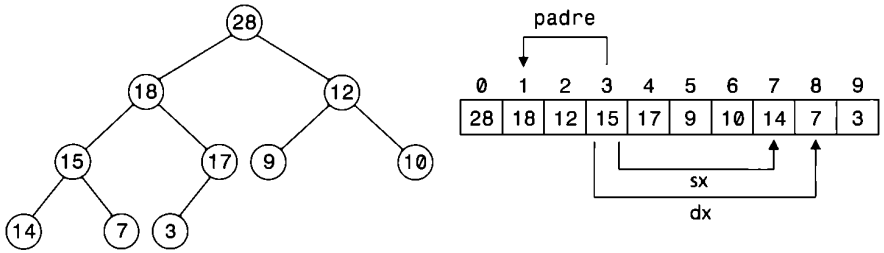
Al contrario dell'operazione `Enqueue`, l'operazione `Dequeue` opera facendo scendere un elemento, impropriamente posto come radice, all'interno dello heap, fino a soddisfare la proprietà di uno heaptree. Applicando le stesse considerazioni effettuate nel caso dell'operazione `Enqueue`, possiamo verificare che l'operazione `Dequeue` implementata su uno heap richiede anch'essa tempo `O(logn)`. Nel seguito mostriamo come sia possibile rappresentare in modo compresso la struttura di uno heap, ovvero i riferimenti da ciascun nodo ai suoi figli, mediante un array e in modo da garantire che la simulazione di tali riferimenti richieda tempo costante.

2.4.2 Implementazione di uno heap implicito

Senza fare uso di memoria aggiuntiva, a parte quella necessaria ai dati e a un numero costante di variabili locali, la relazione tra i nodi di un albero completo a sinistra può essere rappresentata in modo **implicito** utilizzando un array di `n` posizioni, dove `n` indica il numero di nodi dell'albero, e applicando la seguente semplice regola di posizionamento: la radice occupa la posizione `i = 0`; se un nodo occupa la posizione `i`, allora il suo figlio sinistro (se esiste) occupa la posizione `2i+1` e il suo figlio destro (se esiste) occupa la posizione `2i+2`.

ESEMPIO 2.8

La rappresentazione implicita dello heap nella parte sinistra della seguente figura è mostrata nella parte destra della figura.



Osserviamo come, per esempio, il figlio sinistro (rispettivamente, destro) del nodo con priorità 15, che occupa la posizione 3, si trovi nella posizione 7 (rispettivamente, 8).

In base a tale rappresentazione, la navigazione nell'albero da un nodo ai suoi figli, e viceversa, richiede tempo costante come nella rappresentazione esplicita. In particolare, osserviamo che il padre di un nodo che occupa la posizione `i` occupa la posizione $\lfloor (i-1)/2 \rfloor$ e che se $2i+1 \geq n$, allora `u.sx = null`, se $2i+2 \geq n$, allora `u.dx = null` e se `i = 0`, allora `u.padre = null`. Osserviamo inoltre che, nell'ottica del risparmio di memoria, la rappresentazione implicita è preferibile perché usa soltanto `O(1)` celle di memoria aggiuntive, oltre allo spazio necessariamente richiesto per la memorizzazione dei campi `u.dato`.

Uno heap `H` è un albero completo a sinistra e pertanto può essere implementato in modo implicito per mezzo di un array dinamico, che indichiamo con `heapArray`. Come effetto di questa rappresentazione, se `H` ha `n` nodi, i corrispondenti elementi sono memorizzati nelle prime `n` posizioni di `heapArray`: pertanto, nell'implementazione è necessario prevedere anche una variabile intera `heapSize`, che indichi il numero di elementi attualmente presenti nello heap (in altre parole, `heapSize` è l'indice della prima posizione libera di `heapArray`). Tale implementazione delle quattro operazioni fornite da una coda con priorità è riportata nei Codici 3.4 e 3.5, che seguono lo schema algoritmico descritto nel paragrafo precedente.

Teorema 2.2 *L'esecuzione di* `k` *operazioni* `Empty`, `Enqueue`, `First` *o* `Dequeue` *su uno heap contenente inizialmente* `m` *elementi richiede tempo* `O(klogn)` *dove* `n = m + k`.

Dimostrazione Possiamo osservare, per prima cosa, che la funzione `Empty` restituisce il valore `true` se e solo se `heapSize` è uguale a 0 mentre, se lo heap non è vuoto, la funzione `First` restituisce il primo elemento di `heapArray` che, per la rappresentazione implicita sopra illustrata, corrisponde alla radice dello heap. La complessità di queste due operazioni è quindi chiaramente costante.

Codice 2.4 Implementazione di uno heap mediante un array: le funzioni `VerificaRaddoppio` e `VerificaDimezzamento` seguono l'approccio del Paragrafo 1.1.3.

```
1  Empty( ):
2    RETURN heapSize == 0;
```

```
1  First( ):
2    IF (!Empty( )) RETURN heapArray[0];
```

```
1  Enqueue( e ):
2    VerificaRaddoppio( );
3    heapArray[heapSize] = e;
4    heapSize = heapSize + 1;
5    RiorganizzaHeap( heapSize - 1 );
```

```
1  Dequeue( ):
2    IF (!Empty( )) {
3      massimo = heapArray[0];
4      heapArray[0] = heapArray[heapSize - 1];
5      heapSize = heapSize - 1;
6      RiorganizzaHeap( 0 );
7      VerificaDimezzamento( );
8      RETURN massimo;
9    }
```

L'operazione `Enqueue` verifica se l'array debba essere raddoppiato (riga 2): sappiamo già, in base al Teorema 1.1, che il costo ammortizzato di questa verifica è costante. Successivamente, inserisce il nuovo elemento nella prima posizione libera dell'array e, quindi, come foglia dello heap per mantenerne la completezza a sinistra (riga 3). Dopo aver aggiornato il valore di `heapSize` (riga 4), per mantenere la proprietà di uno heaptree viene invocata la funzione `RiorganizzaHeap` (riga 5). Pertanto la complessità ammortizzata dell'operazione `Enqueue` è una costante più la complessità della funzione `RiorganizzaHeap`, di cui posponiamo la discussione.

Se lo heap contiene almeno un elemento, l'operazione `Dequeue` determina quello con la massima priorità, ovvero quello che si trova nella posizione iniziale dell'array (riga 3): quindi, per mantenere la completezza a sinistra, copia nella prima posizione l'elemento che si trova nella posizione finale e che corrisponde all'ultima foglia dello heap, e aggiorna il valore di `heapSize` (righe 4 e 5). Per mantenere la proprietà di uno heaptree, invoca `RiorganizzaHeap` e, infine, verifica se l'array debba essere dimezzato (righe 6 e 7). Anche in questo caso, quindi, la complessità ammortizzata dell'operazione `Dequeue` è una costante più la complessità della funzione `RiorganizzaHeap`.

Questa funzione, che è riportata nel Codice 2.5, ripristina la proprietà di uno heaptree in un albero completo a sinistra e rappresentato in modo implicito, con l'ipotesi che l'elemento `e = heapArray[i]` sia eventualmente l'unico a violare la proprietà di heaptree. Il primo ciclo viene eseguito quando `e` deve risalire lo heap perché `e.prio` è maggiore della priorità di suo padre: dobbiamo quindi scambiare `e` con il padre e iterare (righe 2-5). Il secondo ciclo viene eseguito quando `e` deve scendere perché `e.prio` è minore della priorità di almeno uno dei suoi figli: dobbiamo quindi scambiare `e` con il figlio avente priorità massima, in modo da far risalire tale figlio e preservare la proprietà di uno heaptree (righe 6-10). Tale eventualità è segnalata nella riga 6 dal fatto che `MigliorePadreFigli` non restituisce `i` come posizione dell'elemento di priorità massima tra `e` ed i suoi figli (questo vuol dire che un figlio ha priorità strettamente maggiore poiché, a parità di priorità, viene restituito `i`). Il codice relativo a `MigliorePadreFigli` tiene conto dei vari casi al contorno che si possono presentare (per esempio, che `e` abbia un solo figlio, il quale deve essere sinistro e deve essere l'ultima foglia). Notiamo che, per ogni posizione `i` nello heap e ogni elemento `e = heapArray[i]`, non può mai accadere che vengano eseguiti entrambi i cicli di `RiorganizzaHeap`: in altre parole, `e` sale con il primo ciclo o scende con il secondo, oppure è già al posto giusto e, quindi, nessuno dei cicli viene eseguito. Ne deriva che il costo di `RiorganizzaHeap` è proporzionale all'altezza dello heap e, quindi, richiede tempo logaritmico nel numero di elementi attualmente presenti.

Codice 2.5 Riorganizzazione di uno heap per mantenere la proprietà di uno heaptree.

```
1  RiorganizzaHeap(i):  ⟨pre: heapArray è uno heap tranne che in posizione i⟩
2    WHILE (i>0 && heapArray[i].prio > heapArray[Padre(i)].prio) {
3      Scambia( i, Padre( i ) );
4      i = Padre( i );
5    }
6    WHILE (Sinistro(i) < heapSize && i != MigliorePadreFigli(i)) {
7      migliore = MigliorePadreFigli( i );
8      Scambia( i, migliore );
9      i = migliore;
10   }
```

```
1  MigliorePadreFigli(i):      ⟨pre: il nodo in posizione i ha almeno un figlio⟩
2    j = k = Sinistro(i);
3    IF (k+1 < heapSize) k = k+1;
4    IF (heapArray[k].prio > heapArray[j].prio) j = k;
5    IF (heapArray[i].prio >= heapArray[j].prio) j = i;
6    RETURN j;
```

```
1  Padre( i ):
2    RETURN (i-1)/2;
```

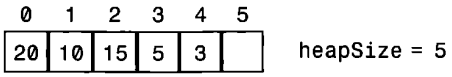
```
1  Sinistro( i ):
2    RETURN 2 x i + 1;
```

```
1  Scambia( i, j ):
2    tmp=heapArray[i];
3    heapArray[i]=heapArray[j];
4    heapArray[j]=tmp;
```

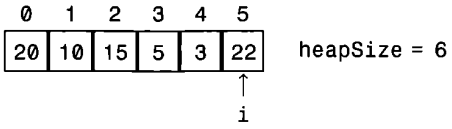
In conclusione, l'esecuzione di ciascuna delle k operazioni richiede un tempo ammortizzato logaritmico nel numero di elementi attualmente presenti nello heap e, quindi, il tempo totale è $O(k \log n)$. ❑

ESEMPIO 2.9

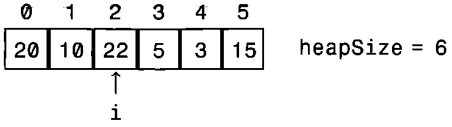
Consideriamo l'operazione Enqueue(22) nello heap rappresentato nella figura.



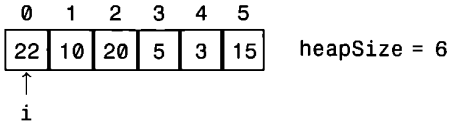
Dopo aver verificato che l'array ha ancora posizioni libere (con VerificaRaddoppio), si posiziona il nuovo elemento in fondo all'array e si incrementa la variabile heapSize.



Viene invocata la funzione RiorganizzaHeap che scambia l'elemento appena inserito nella posizione i = 5 con il padre, cioè quello nella posizione (i-1)/2 ovvero 15 in quanto quest'ultimo è più piccolo di 22.

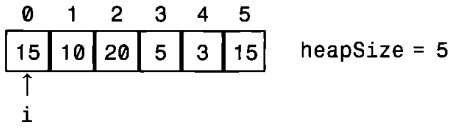


L'elemento nella posizione i è minore dell'elemento nella posizione (i-1)/2 = 0 quindi, ancora una volta, i due elementi vanno scambiati.

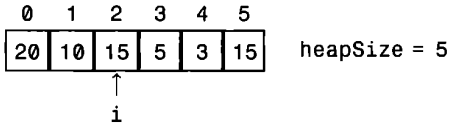


Siamo arrivati alla radice (i = 0), la funzione RiorganizzaHeap ha termine.

L'operazione Dequeue(), dopo aver memorizzato il massimo dello heap nella variabile massimo, procede alla eliminazione di questo: copia l'ultimo elemento dello heap, quello nella posizione heapSize - 1, nella posizione 0 e decrementa heapSize.



L'esecuzione della funzione RiorganizzaHeap(0) scambia l'elemento nella posizione i = 0 con il massimo dei suoi figli che si trovano nella posizione 2i + 1 = 1 e 2i + 2 = 2. La variabile i assume il valore della nuova posizione di 15.



Poiché 15 non ha figli (dato che 2i+1 ≥ heapSize) la funzione termina restituendo il valore della variabile massimo.

2.4.3 Heapsort

L'utilizzo di una coda con priorità fornisce un semplice algoritmo di ordinamento: per ordinare un array di n elementi avendo a disposizione una coda con priorità PQ è sufficiente, infatti, dapprima inserire gli elementi in PQ, effettuando n operazioni Enqueue. A questo punto, heapSize = n e quindi possiamo estrarre gli elementi uno dopo l'altro, in ordine non crescente, mediante n operazioni Dequeue: tali operazioni possono essere semplificate scambiando la radice (il massimo corrente) con l'ultima foglia e riducendo lo heap di un elemento. Quest'idea si realizza nel Codice 2.6: l'algoritmo di ordinamento così ottenuto è detto **heapsort**.

Codice 2.6 Ordinamento mediante heap di un array a.

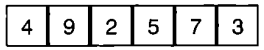
```
1  HeapSort( heapArray ):          ⟨pre: la lunghezza di heapArray è n⟩
2    heapSize = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      Enqueue( heapArray[i] );
5    }
6    WHILE (heapSize > 0) {
7      Scambia( heapSize - 1, 0 );
8      heapSize = heapSize - 1;
9      RiorganizzaHeap( 0 );
10   }
```

Teorema 2.3 *Lo heapsort ordina un array di* n *elementi in tempo* O(n log n).

Dimostrazione Come possiamo vedere nel Codice 2.6, l'algoritmo, dato heapArray da ordinare, prima riposiziona gli elementi di heapArray in modo tale da costruire uno heap, e poi elimina iterativamente il massimo elemento nello heap (nella posizione 0), la cui dimensione è decrementata di 1, costruendo al tempo stesso la sequenza ordinata degli elementi di heapArray a partire dal fondo. In particolare, all'iterazione j del ciclo WHILE, l'elemento massimo estratto dallo heap (che a quel punto ha dimensione heapSize = n - j ed è rappresentato dagli elementi in heapArray[0, n-j-1]) viene inserito nella posizione n-j-1 di heapArray. Al termine di tale iterazione, abbiamo che gli elementi in heapArray[0,n-j-2] formano uno heap e sono tutti di priorità minore o uguale a quelli ordinati in heapArray[n-j-2, n-1]: quando j = n, risultano tutti in ordine. Per quanto riguarda la complessità dell'algoritmo, osserviamo che, nel corso del primo ciclo, sono eseguite n operazioni Enqueue a partire da uno heap vuoto (notiamo che heapSize è inizialmente uguale a 0 e che, quando Enqueue deve inserire l'elemento heapArray[i] nel ciclo FOR, le precedenti posizioni heapArray[0, i-1] formano uno heap con heapSize = i, per i > 0). In base al teorema precedente, il costo di queste operazioni è O(n log n). Successivamente, l'algoritmo esegue n scambi, ciascuno seguito da una riorganizzazione dello heap: dalla dimostrazione del teorema precedente segue che ciascuna riorganizzazione richiede tempo logaritmico nel numero di elementi rimasti nello heap. Quindi, anche in questo caso il costo di queste operazioni è O(n log n). ❑

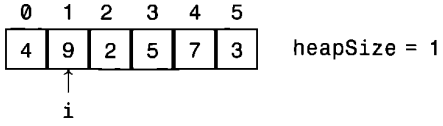
ESEMPIO 2.10

Applichiamo lo heapsort all'array heapArray mostrato nella figura.

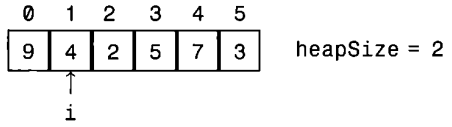


Nella prima fase, costituita dal primo ciclo, vengono inseriti gli elementi dell'array heapArray nella coda con priorità. Per la rappresentazione della coda con priorità si utilizza una porzione dello stesso heapArray. In particolare durante l'i-esimo passo del ciclo l'array è diviso in due parti: i primi i elementi sono quelli della coda con priorità, mentre gli altri sono quelli ancora non inseriti nello heap.

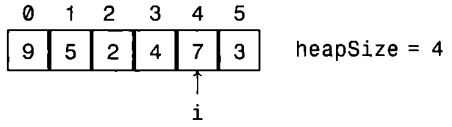
Dopo la prima esecuzione del primo ciclo l'unico elemento nello heap è 4.



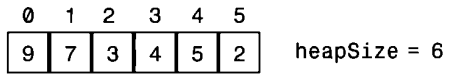
Il prossimo elemento da inserire, heapArray[i] = 9, viene scambiato con il massimo dello heap.



L'inserimento dell'elemento in posizione 2 non provoca scambi, mentre 5 deve essere scambiato con 4. All'inizio del passo successivo del ciclo la situazione è questa.

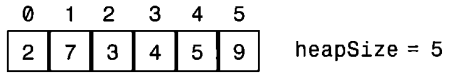


Infine 7 viene scambiato con 5 e 3 con 2 ottenendo lo heap illustrato.

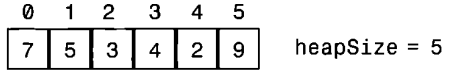


Nel secondo ciclo, per i che va da heapSize - 1 a 1, si considera lo heap racchiuso tra gli indici 0 e i. Il massimo di questo heap viene scambiato con l'elemento in posizione i e si riorganizza lo heap racchiuso tra gli indici 0 e i - 1. Ecco cosa avviene in dettaglio nel nostro esempio.

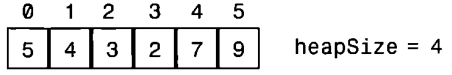
Si scambia 2 con 9 e si decrementa heapSize.



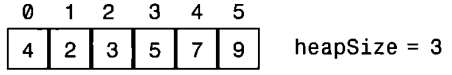
Quindi si invoca la funzione RiorganizzaHeap(0) che fa sì che il 2 venga prima scambiato con il massimo tra 7 e 3 e poi con il massimo tra 4 e 5.



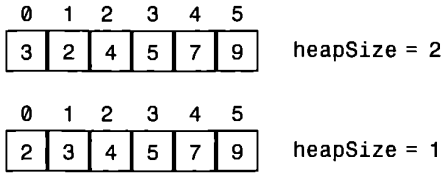
Si scambia 2 con 7 e si riorganizza lo heap contenuto tra le posizioni 0 e 3.



Si prosegue con la coppia formata da 2 e 5.



E così via.



Quando heapSize = 1 l'array risulta ordinato.

Notiamo che lo heapsort opera in loco, ovvero non fa uso di memoria aggiuntiva a parte un numero costante di variabili ausiliarie: ciò era vero anche nel caso dei due algoritmi di ordinamento che abbiamo visto nel capitolo precedente. Tuttavia, in termini di tempo, lo heapsort, che richiede tempo $O(n \log n)$, è drasticamente migliore del selection sort e dell'insertion sort, che richiedevano tempo $O(n^2)$: il prossimo risultato afferma che non è possibile fare di meglio.

Teorema 2.4 *Ogni algoritmo di ordinamento basato su confronti di elementi richiede* $\Omega(n \log n)$ *confronti al caso pessimo.*

Dimostrazione Usiamo un semplice approccio combinatorio basato sulla teoria dell'informazione. Sia A un qualunque algoritmo di ordinamento che usa confronti tra coppie di elementi. Ogni confronto dà luogo a tre possibili risposte in $\{<, =, >\}$, quindi, dopo avere effettuato t confronti, A può discernere al più 3^t situazioni distinte. Poiché il numero di possibili ordinamenti di n elementi è pari a $n!$, ovvero al numero delle loro permutazioni, viene richiesto all'algoritmo di discernere tra $n!$ possibili situazioni: pertanto, deve valere $3^t \geq n!$ perché altrimenti l'algoritmo certamente non sarebbe corretto. Dalla disuguaglianza

$$n! = n(n-1)\cdots 1 > n(n-1)\cdots\left(\frac{n}{2}+1\right) > \underbrace{\frac{n}{2}\cdots\frac{n}{2}}_{n/2 \text{ volte}} = (n/2)^{n/2}$$

deriva che $3^t \geq (n/2)^{n/2}$ e che quindi occorrono $t \geq (n/2)\log_3(n/2) = \Omega(n \log n)$ confronti, come volevasi dimostrare. ❑

Il teorema precedente mostra che $\Omega(n \log n)$ è un limite inferiore per il problema dell'ordinamento per confronti. D'altra parte l'analisi dello heapsort fatta nella dimostrazione del Teorema 3.3 dimostra che un limite superiore alla complessità di tale problema è $O(n \log n)$. Poiché i limiti superiore e inferiore coincidono asintoticamente, abbiamo che la complessità del problema è $\Theta(n \log n)$ e che lo heapsort è un algoritmo ottimo.

Esercizio svolto 2.2 Scrivere un frammento di codice che trasformi la rappresentazione esplicita di un albero completo a sinistra nella corrispondente rappresentazione implicita.

Soluzione Il codice esegue quanto richiesto esaminando i nodi dell'albero per livelli e inserendoli uno dopo l'altro in fondo all'array nella prima posizione disponibile.

```
Implicita( u, nodo ):
    ⟨pre: u radice di albero con n nodi e nodo array di n elementi⟩
  ultimo = attuale = 0;
  nodo[attuale] = u;
  WHILE (attuale <= ultimo) {
    u = nodo[attuale];
    IF (u.sx != null) {
      nodo[ultimo+1] = u.sx;
      ultimo = ultimo + 1;
    }
    IF (u.dx != null) {
      nodo[ultimo+1] = u.dx;
      ultimo = ultimo + 1;
    }
    attuale = attuale + 1;
  }
```

2.5 Esercizi

2.1 Scrivere il codice per realizzare la conversione e l'interprete per le espressioni polacche inverse. Usare un spazio bianco per separare variabili e costanti.

2.2 Modificare la tabella di conversione da un'espressione infissa a una postfissa (Figura 2.1) in modo da riconoscere quando l'espressione infissa fornita in ingresso è sintatticamente incorretta (per esempio, A + × B).

2.3 Scrivere il codice per implementare una coda mediante una lista, secondo quanto descritto nel Paragrafo 2.3.2.

2.4 Un *min-max heap* è una struttura di dati che implementa una coda con priorità sia rispetto al minimo che rispetto al massimo. In uno heap di questo tipo, nella radice è contenuto il minimo e, in ognuno dei suoi due figli, il massimo tra gli elementi presenti nel relativo sottoalbero. In generale, i nodi nell'i-esimo livello dall'alto (dove assumiamo che la radice è a livello 1) contengono gli elementi minimi nei relativi sottoalberi, se i è dispari, e gli elementi massimi, se i è pari. Implementare tale struttura con le operazioni Enqueue, ExtractMin, ExtractMax, che inseriscono un elemento ed estraggono il minimo o il massimo, rispettivamente.

2.5 Spiegare perché il limite inferiore sull'ordinamento enunciato nel Teorema 3.4 non è in contraddizione con la complessità lineare dell'algoritmo proposto nell'Esercizio 2.5.

2.6 Una sequenza *bilanciata* di parentesi può essere definita ricorsivamente come segue: la sequenza vuota è bilanciata; se `S` e `T` sono sequenze bilanciate, allora `(S)T` e `S(T)` sono sequenze bilanciate. Per esempio, `()(()(()))` e `(()()())(())` sono sequenze bilanciate, mentre `()((()))())` e `)())(())` non lo sono. Progettare un algoritmo efficiente che verifichi se una sequenza di parentesi è bilanciata.

2.7 Il minimo di un heap risiede in una foglia dell'albero. Sfruttando questa proprietà, progettare un algoritmo che restituisca l'elemento minimo di uno heap implicito e calcolarne la complessità.

2.8 Dimostrare che non può esistere un'implementazione della coda con priorità in cui la complessità delle operazioni `First` e `Empty` è costante e quella delle operazioni `Enqueue` e `Dequeue` è $o(\log n)$, dove `n` è la dimensione della coda.

2.9 Data una coda di priorità `C` in cui gli `n` elementi possono essere soltanto le stringhe `algo`, `logo`, `prog` (e quindi ci sono elementi ripetuti per `n > 3`). Scrivere un programma che in tempo $O(n)$ costruisca la coda `C` in cui tutte le stringhe `algo` sono considerate minori delle stringhe `logo`, e queste ultime sono considerate minori delle stringhe `prog`.

2.10 Dato un array di `n` elementi con ripetizioni, sia `k` il numero di elementi distinti. Progettare un algoritmo che richieda $O(n \log k)$ tempo per identificare i `k` elementi distinti.

2.11 Dati due heap impliciti di `n` elementi, progettare un algoritmo per stabilire se i due heap contengono gli stessi elementi (si noti che non è detto che i loro array siano necessariamente uguali).

2.12 Considerare la variante della riorganizzazione di uno heap che procede esclusivamente verso i figli, indicata con `RiorganizzaHeapFigli` e ottenuta prendendo soltanto le righe 6-10 (ossia il solo secondo ciclo) di `RiorganizzaHeap` nel Codice 2.5. Mostrare che il seguente codice costruisce correttamente uno heap dal basso verso l'alto e che il tempo richiesto è $O(n)$. Viene migliorata in tal caso la complessità dell'algoritmo `HeapSort` discusso nel Paragrafo 2.4.3?

```
CreaHeapMigliorato( ):
  heapSize = n;
  FOR (i = n/2-1; i >=0; i = i - 1) {
    RiorganizzaHeapFigli(i);
  }
```

CAPITOLO

3 Divide et impera

In questo capitolo, introduciamo una delle tecniche più note per lo sviluppo di algoritmi efficienti, ovvero il paradigma del divide et impera. Per analizzare la complessità di algoritmi sviluppati facendo uso di tale tecnica, dimostreremo il teorema delle ricorrenze, che fornisce uno strumento potente per la ricerca della soluzione di relazioni di ricorrenza. Gli esempi che mostreremo in questo capitolo spazieranno dalla ricerca all'interno di un array, all'ordinamento, all'algebra di numeri interi e di matrici, alla geometria computazionale.

3.1 Ricorsione e paradigma del divide et impera
3.2 Relazioni di ricorrenza e teorema fondamentale
3.3 Ricerca di una chiave
3.4 Ordinamento e selezione per distribuzione
3.5 Moltiplicazione veloce di due numeri interi
3.6 Opus libri: grafica e moltiplicazione di matrici
3.7 Opus libri: il problema della coppia più vicina
3.8 Algoritmi ricorsivi su alberi binari
3.9 Esercizi

3.1 Ricorsione e paradigma del divide et impera

Il paradigma del divide et impera è uno dei più utilizzati nella progettazione di algoritmi ricorsivi e si basa su un principio ben noto a chiunque abbia dovuto scrivere programmi di complessità non elementare. Tale principio consiste nel suddividere un problema in due o più sotto-problemi, nel risolvere tali sotto-problemi, eventualmente applicando nuovamente il principio stesso, fino a giungere a problemi "elementari" che possano essere risolti in maniera diretta, e nel combinare le soluzioni dei sotto-problemi in modo opportuno così da ottenere una soluzione del problema di partenza. Quello che contraddistingue il paradigma del divide et impera è il fatto che i sotto-problemi sono istanze dello stesso problema originale, ma di dimensioni ridotte: il fatto che un sotto-problema sia elementare o meno dipende, essenzialmente, dal fatto che la dimensione dell'istanza sia sufficientemente piccola da poter risolvere il sotto-problema in maniera diretta.

Il paradigma del **divide et impera** può, dunque, essere strutturato nelle seguenti tre fasi che lo caratterizzano.

Decomposizione: identifica un numero piccolo di sotto-problemi dello stesso tipo, ciascuno definito su un insieme di dati di dimensione inferiore a quello di partenza.

Ricorsione: risolvi ricorsivamente ciascun sotto-problema fino a ottenere insiemi di dati di dimensioni tali che i sotto-problemi possano essere risolti direttamente.

Ricombinazione: combina le soluzioni dei sotto-problemi per fornire una soluzione al problema di partenza.

ESEMPIO 3.1

Consideriamo l'algoritmo di ordinamento per fusione (in inglese *mergesort*), che opera in tempo $O(n \log n)$ secondo il paradigma divide et impera.

Decomposizione: se la sequenza ha almeno due elementi, dividila in due sotto-sequenze uguali (o quasi) in lunghezza (nel caso in cui abbia meno di due elementi non vi è nulla da fare).

Ricorsione: ordina ricorsivamente le due sotto-sequenze.

Ricombinazione: fondi le due sotto-sequenze ordinate in un'unica sequenza ordinata.

L'algoritmo di ordinamento per fusione è descritto nel Codice 3.1. Per implementare l'algoritmo è però necessario specificare come le due sotto-sequenze ordinate possano essere fuse. Esistono diversi modi per far ciò, sia mediante l'uso di memoria addizionale che operando in loco: poiché in quest'ultimo caso l'algoritmo di fusione risulta piuttosto complicato, preferiamo fornire la soluzione che fa uso di un array aggiuntivo. Tale soluzione si ispira al metodo utilizzato per fondere due mazzi di carte ordinati in modo crescente. In tal caso, a ogni passo, per determinare la carta di valore minimo nei due mazzi è sufficiente confrontare le due carte in cima ai mazzi stessi: tale carta può essere quindi inserita in fondo al nuovo mazzo. Rimandiamo il lettore all'Esercizio svolto 3.1.

Tornando alla descrizione dell'algoritmo di ordinamento per fusione mostrato nel Codice 3.1, osserviamo che le prime tre istruzioni della struttura `IF` (che vengono eseguite solo se vi sono almeno due elementi da ordinare) corrispondono alla fase di decomposizione del problema (riga 4) e a quella di soluzione ricorsiva dei due sotto-problemi (righe 5 e 6). L'invocazione della funzione `Fusione` realizza la fase di ricombinazione (riga 7), fondendo in un unico segmento ordinato i due segmenti `a[sinistra, centro]` e `a[centro + 1, destro]` ordinati prodotti dalla ricorsione.

Consideriamo una chiamata su n elementi. L'esecuzione della funzione `Fusione` richiede $O(n)$ e le altre operazioni richiedono $O(1)$ tempo, per cui esistono delle costanti c_0, $n_0 > 0$ tale che il costo non supera $c_0 n$ per $n \geq n_0$. Nell'analisi manca però il costo delle chiamate ricorsive, per cui non è possibile fornire immediatamente una formula chiusa. Utilizziamo una relazione di ricorrenza, indicando con $T(n)$ il costo dell'intera funzione ricorsiva `MergeSort` che vogliamo analizzare: poiché ci sono due chiamate ricorsive sulla metà degli elementi, possiamo limitare superiormente il costo di tali chiamate con $2T(n/2)$. A questo punto, possiamo affermare che il numero totale di passi eseguiti dal Codice 3.1 su un array di n elementi può essere limitato superiormente dalla seguente relazione per $n \geq n_0$:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(n/2) + cn & \text{altrimenti} \end{cases} \tag{3.1}$$

dove c_0 e c sono due costanti positive. Potremmo risolvere direttamente la relazione (3.1) espandendo ricorsivamente la parte $T(n/2)$, ottenendo così $T(n) = O(n \log n)$. Tuttavia, vedremo nel Paragrafo 3.2 che tale relazione rientra in una famiglia più generale, per cui forniremo una tecnica generale di risoluzione. Osservando che uno stesso vettore d'appoggio di n elementi può essere riutilizzato per tutte le operazioni di fusione, possiamo concludere che l'algoritmo richiede $O(n)$ spazio aggiuntivo.

Notiamo, infine, che la scansione sequenziale dei dati da fondere rende l'ordinamento per fusione uno dei principali metodi di ordinamento per grandi quantità dati che risiedono nella memoria secondaria del calcolatore (per esempio, un disco rigido), notoriamente con accesso più lento della memoria principale (che non può ospitare tutti i dati in tale scenario).

Codice 3.1 Ordinamento per fusione di un array a.

```
1  MergeSort( a, sinistra, destra ):
2                                   ⟨pre: 0 ≤ sinistra ≤ destra ≤ n - 1⟩
3    IF (sinistra < destra) {
4      centro = (sinistra+destra)/2;
5      MergeSort( a, sinistra, centro );
6      MergeSort( a, centro+1, destra );
7      Fusione( a, sinistra, centro, destra );
8    }
```

Esercizio svolto 3.1 Sia a un array e siano a[sx, cx] e a[cx + 1, dx] due segmenti adiacenti di a, dove sx ≤ cx < dx, ciascuno ordinato in modo non decrescente. Progettare un algoritmo che richieda tempo lineare per fondere i due segmenti e ottenere che anche a[sx, dx] sia ordinato.

Soluzione Per ottenere che l'intero segmento a[sx, dx] sia ordinato, possiamo utilizzare un array b d'appoggio che viene riempito nel modo seguente. Partendo da i = sx e j = cx + 1, memorizziamo il minimo tra a[i] e a[j] nella prima posizione libera di b: se tale minimo è a[i], allora incrementiamo di 1 il valore di i, altrimenti incrementiamo di 1 il valore di j. Ripetiamo questo procedimento fino a quando i diviene maggiore di cx oppure j diviene maggiore di dx: nel primo caso, memorizziamo i rimanenti elementi del segmento a[cx + 1, dx] (se ve ne sono) nelle successive posizioni libere di b, mentre, nel secondo caso, memorizziamo i rimanenti elementi del segmento a[sx, cx] (se ve ne sono) nelle successive posizioni libere di b. Al termine di questo ciclo, b conterrà gli elementi del segmento a[sx, dx] ordinati in modo non decrescente, per cui sarà sufficiente ricopiare b all'interno di tale segmento.

La fusione di due sequenze ordinate appena descritta è realizzata nel Codice 3.2. Poiché a ogni iterazione del ciclo WHILE l'indice i (che parte da sx e arriva al massimo a cx) oppure l'indice j (che parte da cx + 1 e arriva al massimo a dx) aumenta di 1, tale ciclo può essere eseguito al più (cx - sx + 1) + (dx - cx - 1) = dx - sx volte. Uno solo dei due cicli FOR successivi (righe 16-18) viene eseguito, per al più cx - sx + 1 e dx - cx iterazioni, rispettivamente. Infine, l'ultimo ciclo FOR verrà eseguito dx - sx + 1 volte: pertanto, la fusione dei due segmenti richiede un numero di passi O(dx - sx), ovvero linearmente proporzionale alle somme delle lunghezze dei due segmenti da fondere.

Codice 3.2 Fusione di due segmenti adiacenti ordinati.

```
1  Fusione( a, sx, cx, dx ):                    ⟨pre: 0 ≤ sx ≤ cx < dx ≤ n - 1⟩
2    i = sx;
3    j = cx+1;
4    k = 0;
5    WHILE ((i <= cx) && (j <= dx)) {
6      IF (a[i] <= a[j]) {
7        b[k] = a[i];
8        i = i+1;
9      } ELSE {
10       b[k] = a[j];
11       j = j+1;
12     }
13     k = k+1;
14   }
15   FOR ( ; i <= cx; i = i+1, k = k+1)
16     b[k] = a[i];
17   FOR ( ; j <= dx; j = j+1, k = k+1)
18     b[k] = a[j];
19   FOR (i = sx; i <= dx; i = i+1)
20     a[i] = b[i-sx];
```


3.2 Relazioni di ricorrenza e teorema fondamentale

La formulazione ricorsiva di un algoritmo di tipo divide et impera necessita di un nuovo strumento analitico per valutarne la complessità temporale, facendo uso di **relazioni di ricorrenza**, ovvero di espressioni matematiche che esprimono una funzione T(n) sugli interi come una combinazione dei valori T(i), con 0 ≤ i < n. Al fine di derivare una formulazione in forma chiusa di T(n) utilizzeremo il **teorema fondamentale delle ricorrenze** (*master theorem*), che consente di fornire un limite superiore in forma chiusa a relazioni di ricorrenza di questo tipo e, quindi, alla complessità degli algoritmi (ricordando che esistono teoremi ancora più generali che permettono di risolvere molti più casi di quelli presentati nel libro).

Teorema 3.1 *Sia* $f(n)$ *una funzione non decrescente e siano* α, β, n_0, c_0 *e* c *delle costanti tali che* $\alpha \geq 1$, $\beta > 1$ *e* n_0, c_0, $c > 0$, *per la relazione di ricorrenza*

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ \alpha T(n/\beta) + cf(n) & \text{altrimenti} \end{cases} \tag{3.2}$$

(*dove* n/β *va interpretato come* $\lfloor n/\beta \rfloor$ *o* $\lceil n/\beta \rceil$). *Se esistono due costanti positive* γ e n_0' tali che $\alpha f(n/\beta) \leq \gamma f(n)$ *per ogni* $n \geq n_0'$, *allora la relazione di ricorrenza ha i seguenti limiti superiori*:

1. $T(n) = O(f(n))$ *se* $\gamma < 1$;
2. $T(n) = O(f(n)\log_\beta n)$ *se* $\gamma = 1$;
3. $T(n) = O(n^{\log_\beta \alpha})$ *se* $\gamma > 1$ e $\alpha > 1$.

Utilizziamo la (3.2) per stabilire i limiti superiori della complessità di algoritmi e problemi. Essa rappresenta la complessità di un algoritmo di tipo divide et impera: applichiamo il caso base quando ci sono al più n_0 elementi; altrimenti, dividiamo gli n elementi in gruppi da n/β elementi ciascuno, effettuiamo α chiamate ricorsive (ciascuna su n/β elementi) e impieghiamo tempo f(n) per il resto del codice (fasi di decomposizione e di ricombinazione).

Dimostrazione Per la dimostrazione dell'enunciato, ipotizziamo per semplicità che $n_0 = n'_0 = 1$ e che n sia una potenza di β, in modo da calcolare la forma chiusa di $T(n)$ per la ricorrenza (3.2): il primo livello di ricorsione contribuisce con $cf(n)$, il secondo con $c\alpha f(n/\beta)$, il terzo con $c\alpha^2 f(n/\beta^2)$ e così via, fino all'ultimo livello dove abbiamo al più $c\alpha^h f(n/\beta^h)$ (in quanto alcune chiamate ricorsive modellate dalla ricorrenza potrebbero essere terminate prima), ottenendo

$$T(n) \le cf(n) + c\alpha f\left(\frac{n}{\beta}\right) + \cdots + c\alpha^i f\left(\frac{n}{\beta^i}\right) + \cdots + c\alpha^h f\left(\frac{n}{\beta^h}\right) = c\sum_{i=0}^{h} \alpha^i f\left(\frac{n}{\beta^i}\right) \qquad (3.3)$$

Osserviamo che il valore di h è tale che $n/\beta^h = n_0 = 1$, implicando così che $h = \log_\beta n$. Inoltre, una semplice induzione su $i \ge 0$ mostra che

$$\alpha^i f\left(\frac{n}{\beta^i}\right) \le \gamma^i f(n) \qquad (3.4)$$

Infatti, il caso base ($0 \le i \le 1$) è immediato. Nel caso induttivo, osserviamo che $\alpha^{i+1} f(n/\beta^{i+1})$ può essere scritto come

$$\alpha^i \left(\alpha f\left(\left(\frac{n}{\beta^i}\right)/\beta\right)\right) \le \alpha^i \left(\gamma f\left(\frac{n}{\beta^i}\right)\right) = \gamma\left(\alpha^i f\left(\frac{n}{\beta^i}\right)\right) \le \gamma(\gamma^i f(n))$$

dove abbiamo usato la proprietà che $\alpha f(x/\beta) \le \gamma f(x)$ nella prima uguaglianza e l'ipotesi induttiva nell'ultima. Ne deriva che, utilizzando la relazione (3.4), possiamo scrivere la ricorrenza (3.3) come

$$T(n) \le c\sum_{i=0}^{h} \gamma^i f(n) = cf(n)\sum_{i=0}^{h} \gamma^i \qquad (3.5)$$

Valutiamo ora la ricorrenza nei tre casi previsti dal teorema, ricordando la forma chiusa della serie $\sum_{i=0}^{h} \gamma^i = \frac{\gamma^{h+1} - 1}{\gamma - 1}$ per $\gamma \ne 1$.

1. Nel primo caso abbiamo $\gamma < 1$, per cui $\frac{\gamma^{h+1} - 1}{\gamma - 1} = \frac{1 - \gamma^{h+1}}{1 - \gamma} < \frac{1}{1 - \gamma} = O(1)$, e quindi $T(n) = O(f(n))$.
2. Nel secondo caso abbiamo $\gamma = 1$, per cui $\sum_{i=0}^{h} \gamma^i = h + 1 = O(\log_\beta n)$; ne deriva che $T(n) = O(f(n)\log_\beta n)$.
3. Nel terzo caso abbiamo $\gamma > 1$ e $\alpha > 1$. Utilizzando la relazione (3.3) con $i = h$, possiamo riscriverla così:

$$T(n) = O(\alpha^h f(n/\beta^h)) = O(\alpha^h f(n_0)) = O(\alpha^{\log_\beta n}) = O(2^{\log \alpha \log n / \log \beta}) = O(n^{\log_\beta \alpha}).$$

I tre casi sopra concludono la dimostrazione del teorema. Rimane da considerare il caso in cui n non sia una potenza di β. Nel caso che si prenda $\lfloor n/\beta \rfloor$, possiamo maggiorarlo con n/β e la relazione (3.3) rimane valida. Nel caso di $\lceil n/\beta \rceil$, basta considerare la parte superiore anche nella condizione del teorema, richiedendo che $\alpha f(\lceil n/\beta \rceil) \le \gamma f(n)$ per ogni $n \ge n'_0$. ❑

ESEMPIO 3.2

Applicando il teorema fondamentale delle ricorrenze all'equazione (3.1), otteniamo $\alpha = 2$, $\beta = 2$ e $f(n) = n$: rientriamo nel secondo caso del teorema, in quanto $\alpha f(n/\beta) = 2(n/2) = n = f(n)$. Quindi, il numero di passi richiesti dall'ordinamento per fusione su un array di n elementi è $O(n \log n)$, e abbiamo pertanto dimostrato che tale algoritmo è ottimo.

3.3 Ricerca di una chiave

Descriviamo ora una semplice applicazione del paradigma divide et impera a uno dei problemi più comuni che sorgono quando si sviluppano sistemi informatici. Uno degli usi più frequenti del calcolatore è quello di cercare una **chiave**, ovvero un valore specificato, tra una mole di dati disponibili in forma elettronica. Data una sequenza lineare di n elementi memorizzata in un array, la **ricerca** di una chiave k consiste nel verificare se la sequenza contiene un elemento il cui valore è uguale a k (nel seguito identificheremo gli elementi con il loro valore). Se la sequenza non rispetta alcun ordine, è necessario esaminare tutti gli elementi con il metodo della **ricerca sequenziale**, descritto nel Codice 3.3. L'algoritmo non fa altro che scandire, uno dopo l'altro, i valori degli elementi contenuti nell'array a: al termine del ciclo `WHILE` (dalla riga 3 alla 5), se la chiave `k` è stata trovata, la variabile `indice` contiene la posizione della sua prima occorrenza. Nel caso pessimo, quando la chiave cercata non è tra quelle nella sequenza, il ciclo scorre tutti gli elementi e pone `indice = -1`, richiedendo un numero di operazioni proporzionale al numero di elementi presenti nella sequenza, e quindi tempo $O(n)$.

Codice 3.3 Ricerca sequenziale di una chiave k in un array a.

```
1  RicercaSequenziale( a, k ):          ⟨pre: la lunghezza di a è n⟩
2  indice = 0;
3  WHILE (indice < n && a[indice] != k) {
     indice = indice + 1;
5  }
6  IF (indice >= n) indice = -1;
7  RETURN indice;
```

È possibile fare di meglio quando la sequenza è ordinata? Usando la **ricerca binaria** (o dicotomica), il costo si riduce a $O(\log n)$ al caso pessimo: invece di scandire miliardi di dati durante la ricerca, ne esaminiamo solo poche decine! Un modo folkloristico di introdurre il metodo si ispira alla ricerca di una parola in un dizionario (o di un numero in un elenco telefonico). Prendiamo la pagina centrale del dizionario: se la parola cercata è alfabeticamente precedente a tale pagina, strappiamo in due il dizionario e ne buttiamo via la metà destra; se la parola è alfabeticamente successiva alla pagina centrale, buttiamo via la metà sinistra.

Altrimenti, l'unica possibilità è che la parola sia nella pagina centrale. Con un numero costante di operazioni possiamo diminuire il numero di pagine da cercare di circa la metà. È sufficiente ripetere il metodo per ridurre esponenzialmente tale numero fino a giungere alla pagina cercata o concludere che la parola non appare in alcuna pagina.

Analogamente, possiamo cercare una chiave k quando l'array a è ordinato in modo non decrescente, basandoci su operazioni di semplice confronto tra due elementi. Confrontiamo la chiave k con l'elemento che si trova in posizione centrale nell'array, a[n/2], e se k è minore di tale elemento ripetiamo il procedimento nel segmento costituito dagli elementi che precedono a[n/2]. Altrimenti, lo ripetiamo in quello costituito dagli elementi che lo seguono. Il procedimento termina nel momento in cui k coincide con l'elemento centrale del segmento corrente (nel qual caso, abbiamo trovato la posizione corrispondente) oppure il segmento diventa vuoto (nel qual caso, k non è presente nell'array).

La ricerca di una chiave all'interno di un array di n elementi ordinati viene quindi realizzata risolvendo il problema della ricerca della chiave in un array di dimensione pari a circa la metà di quella dell'array originale, fino a giungere a un array con un solo elemento, nel qual caso il problema diviene chiaramente elementare. Più formalmente, tale descrizione della ricerca binaria è mostrata nel Codice 3.4, in cui le istruzioni alle righe 3-9 risolvono in maniera diretta il caso elementare, mentre le istruzioni alle righe 12 e 14 riducono il problema della ricerca all'interno del segmento attuale a quello della ricerca nella metà sinistra e destra, rispettivamente, del segmento stesso. Nella chiamata iniziale, il parametro `sinistra` assume il valore `0` e il parametro `destra` assume il valore `n - 1`.

Codice 3.4 Ricerca binaria di una chiave k in un array a con il paradigma del divide et impera.

```
1  RicercaBinariaRicorsiva( a, k, sinistra, destra ):
2                          ⟨pre: a ordinato e 0 ≤ sinistra ≤ destra ≤ n - 1⟩
3    IF (sinistra == destra) {
4      IF (k == a[sinistra]) {
5        RETURN sinistra;
6      } ELSE {
7        RETURN -1;
8      }
9    }
10   centro = (sinistra+destra)/2;
11   IF (k <= a[centro]) {
12     RETURN RicercaBinariaRicorsiva( a, k, sinistra, centro );
13   } ELSE {
14     RETURN RicercaBinariaRicorsiva( a, k, centro+1, destra );
15   }
```


Osserviamo che se il segmento all'interno del quale stiamo cercando una chiave è costituito da un solo elemento, allora il Codice 3.4 esegue un numero costante c_0 di operazioni. Altrimenti, il numero di operazioni eseguite è pari a una costante c più il numero di passi richiesto dalla ricerca della chiave in un segmento di dimensione pari alla metà di quello attuale. Pertanto, il numero totale T(n) di passi eseguiti su un array di n elementi verifica la seguente relazione di ricorrenza:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2)+c & \text{altrimenti} \end{cases} \tag{3.6}$$

Teorema 3.2 *La ricerca binaria in un array ordinato richiede* $O(\log n)$ *passi.*

Dimostrazione Facendo riferimento alla relazione (3.6) che indica la complessità dell'algoritmo di ricerca binaria, abbiamo che $\alpha = 1$, $\beta = 2$, $n_0 = 1$ e $f(n) = 1$ per ogni n: pertanto, scegliendo $\gamma = 1$ e $n'_0 = 1$, rientriamo nel secondo caso del teorema fondamentale delle ricorrenze. Possiamo quindi concludere che la soluzione della relazione di ricorrenza è $O(f(n)\log_\beta n) = O(\log n)$: il teorema risulta quindi essere dimostrato. ❑

Il teorema precedente identifica un limite superiore per il problema della ricerca di una chiave usando confronti tra gli elementi di una sequenza ordinata. Seguendo le argomentazioni del Paragrafo 2.4.3, vogliamo ora stabilire un limite inferiore al problema per dimostrare che l'algoritmo di ricerca binaria è asintoticamente ottimo, stabilendo così anche la complessità del problema della ricerca per confronti in una sequenza ordinata.

Teorema 3.3 Ogni *algoritmo di ricerca per confronti (non soltanto la ricerca binaria) ne richiede* $\Omega(\log n)$ *al caso pessimo.*

Dimostrazione Sia A un qualunque algoritmo di ricerca che usa confronti tra coppie di elementi. L'algoritmo A deve discernere tra n + 1 situazioni: la chiave cercata non appare nella sequenza (`indice = -1`), oppure appare in una delle n posizioni della sequenza ($0 \leq$ `indice` $\leq n - 1$). Dopo t confronti di chiavi (mai esaminate prima), l'algoritmo A può discernere al più 3^t situazioni. Poiché viene richiesto di discernerne n + 1, deve valere $3^t \geq n + 1$. Ne deriva che occorrono $t \geq \log 3(n+1) = \Omega(\log n)$ confronti: ciò rappresenta un limite inferiore per il problema della ricerca per confronti. ❑

In base ai due teoremi precedenti possiamo quindi concludere che la complessità del problema della ricerca di una chiave usando confronti tra gli elementi è $\Theta(\log n)$ e che la ricerca binaria è un algoritmo ottimo.

Modificando semplicemente le righe 4-8 in modo che restituiscano la posizione `sinistra` se `k < a[sinistra]` e la posizione `sinistra + 1` altrimenti (invece che il valore `-1`), la ricerca binaria così modificata restituisce il **rango** `r` di `k` (dove $0 \le r \le n$), definito come il numero di elementi in `a` che sono minori o uguali a `k`. Inoltre notiamo che nel caso che l'array contenga un multi-insieme ordinato, dove sono ammesse le occorrenze multiple delle chiavi, il Codice 3.4 individua la posizione dell'occorrenza più a sinistra della chiave `k`.

Esercizio svolto 3.2 Progettare un algoritmo non ricorsivo che implementi la ricerca binaria all'interno di un array ordinato.

Soluzione Si consideri il seguente codice, che non è equivalente al Codice 3.4.

```
RicercaBinariaIterativa( a, k ):          ⟨pre: a ordinato e di lunghezza n⟩
  sinistra = 0;
  destra = n-1;
  trovato = FALSE;
  indice = -1;
  WHILE ((sinistra <= destra) && (!trovato)) {
    centro = (sinistra+destra)/2;
    IF (a[centro] > k) {
      destra = centro-1;
    } ELSE IF (a[centro] < k) {
      sinistra = centro+1;
    } ELSE {
      indice = centro;
      trovato = TRUE;
    }
  }
  RETURN indice;
```

Tale codice mantiene implicitamente l'invariante, per le due variabili `sinistra` e `destra` che delimitano il segmento in cui effettuare la ricerca, secondo la quale vale $a[sinistra] \le k \le a[destra]$. Inizialmente, queste due variabili sono poste a 0 e $n-1$ (possiamo escludere i casi limite in cui $k < a[0]$ oppure $a[n-1] < k$, per cui presumiamo senza perdita di generalità che $a[0] \le k \le a[n-1]$). A ogni iterazione del ciclo `WHILE`, se la chiave `k` è minore dell'elemento centrale (il cui indice di posizione è dato dalla semisomma delle due variabili `sinistra` e `destra`), la ricerca prosegue nella parte sinistra del segmento: per questo motivo, la variabile `destra` viene modificata in modo da indicare l'elemento immediatamente precedente a quello centrale. Se, invece, la chiave `k` è maggiore del valore dell'elemento centrale, la ricerca prosegue nella parte destra del segmento. Se, infine, la chiave è stata trovata, l'indice di posizione della sua occorrenza viene memorizzato nella variabile `indice` e il ciclo ha termine in quanto la variabile `trovato` assume il valore vero. Quando la chiave non appare, il segmento diventa vuoto poiché `sinistra > destra`.

La strategia dicotomica rappresentata dalla ricerca binaria è utile in tutti quei problemi in cui vale una qualche proprietà monotona booleana $P(x)$ al variare di un parametro intero x: in altre parole, esiste un intero x_0 tale che $P(x)$ è vera per $x \le x_0$ e falsa per $x > x_0$ (o, viceversa, $P(x)$ è falsa per $x \le x_0$ e vera per $x > x_0$). Usando uno schema simile alla ricerca binaria possiamo trovare il valore di x_0.

ESEMPIO 3.3

Ipotizziamo di dover indovinare il valore di un numero $n > 0$ pensato segretamente da un nostro amico. È dispendioso chiedere se $n = 1$, $n = 2$, $n = 3$, e così via per tutta la sequenza di numeri da 1 fino a n (che rimane sconosciuto fino all'ultima domanda), in quanto richiede di effettuare n domande. Usando il nostro schema, possiamo porre concettualmente $x_0 = n$ e, come prima cosa, effettuare domande del tipo "$x \le n$?" per potenze crescenti $x = 1, 2, 4, 8, \ldots, 2^h$ e così via. Pur non conoscendo il valore di n, ci fermiamo non appena $2^{h-1} < n \le 2^h$ (a causa della riposta affermativa del nostro amico), effettuando in tal modo $h = O(\log n)$ domande poiché $h < 1 + \log n$. A questo punto, sappiamo che n appartiene all'intervallo $[2^{h-1} + 1, 2^h]$ e applicando una ricerca binaria all'interno di questo intervallo, con ulteriori $O(\log 2^{h-1}) = O(\log n)$ domande, riusciamo a indovinare il numero n. In totale, abbiamo ridotto il numero di domande da effettuare al nostro amico da n a $O(\log n)$ per indovinare il numero n da lui segretamente pensato.

3.4 Ordinamento e selezione per distribuzione

L'algoritmo di ordinamento per distribuzione (*quicksort*) segue il paradigma del divide et impera e opera nel modo seguente.

Decomposizione: se la sequenza ha almeno due elementi, scegli un elemento **pivot** e dividi la sequenza in due sotto-sequenze eventualmente vuote, dove la prima contiene elementi minori o uguali al pivot e la seconda contiene elementi maggiori o uguali.

Ricorsione: ordina ricorsivamente le due sotto-sequenze.

Ricombinazione: concatena (implicitamente) le due sotto-sequenze ordinate in un'unica sequenza ordinata.

Codice 3.5 Ordinamento per distribuzione di un array a.

```
1  QuickSort( a, sinistra, destra ):
2       ⟨pre: 0 ≤ sinistra, destra ≤ n - 1⟩
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

Il Codice 3.5 implementa l'algoritmo di ordinamento per distribuzione secondo lo schema descritto sopra. Nella riga 4, viene scelta una posizione per il `pivot`: come vedremo, la scelta della posizione è rilevante ma, per adesso, supponiamo di scegliere sempre l'ultima posizione nel segmento `a[sinistra, destra]`. Nella riga 5, gli elementi sono distribuiti all'interno del segmento in modo che l'elemento pivot vada in `a[rango]`, che gli elementi del segmento `a[sinistra, rango - 1]` siano minori o uguali di `a[rango]` e quelli del segmento `a[rango + 1, destra]` siano maggiori o uguali di `a[rango]`: in altre parole, `rango` è la destinazione finale nell'array ordinato dell'elemento pivot. Come possiamo osservare, dopo la ricorsione (righe 6-7) la ricombinazione è nulla.

Il passo fondamentale di distribuzione degli elementi in base al pivot è rappresentato da `Distribuzione`, illustrato nel Codice 3.6. Utilizzando una primitiva `Scambia` per scambiare il contenuto di due posizioni nel segmento, l'elemento pivot viene spostato nell'ultima posizione del segmento (riga 2). Le rimanenti posizioni sono scandite con due indici (righe 3 e 4): una scansione procede in avanti finché non trova un elemento *maggiore* del pivot (righe 6 e 7) mentre l'altra procede all'indietro finché non trova un elemento *minore* del pivot (righe 8 e 9). Un semplice scambio dei due elementi fuori posto (riga 10) permette di procedere con le due scansioni, che terminano quando gli elementi scanditi si incrociano, uscendo di fatto dal ciclo esterno (righe 5-11). Un ulteriore scambio (riga 12) colloca il pivot nella posizione `i` che viene restituita come `rango` nell'algoritmo di ordinamento (riga 13). Poiché le due scansioni sono eseguite in un tempo complessivamente lineare, il costo di `Distribuzione` è $O(n)$ in tempo e $O(1)$ in spazio (in quanto usiamo un numero costante di variabili di appoggio).

Codice 3.6 Distribuzione in loco degli elementi di un segmento `a[sx, dx]` in base alla posizione `px` scelta per il pivot.

```
1  Distribuzione( a, sx, px, dx ):          ⟨pre: 0 ≤ sx ≤ px ≤ dx ≤ n - 1⟩
2    IF (px != dx) Scambia( px, dx );
3    i = sx;
4    j = dx-1;
5    WHILE (i <= j) {
6      WHILE ((i <= j) && (A[i] <= A[dx]))
7        i = i+1;
8      WHILE ((i <= j) && (A[j] >= A[dx]))
9        j = j-1;
10     IF (i < j) Scambia( i, j );
11   }
12   IF (i != dx) Scambia( i, dx );
13   RETURN i;
```



```
1  Scambia( i, j ):                         ⟨pre: sx ≤ i, j ≤ dx⟩
2    temp = a[j]; a[j] = a[i]; a[i] = temp;
```

Proviamo a scrivere la relazione di ricorrenza per il tempo $T(n)$ di esecuzione dell'algoritmo di ordinamento per distribuzione di n elementi. Nel caso base $T(n) \leq c_0$ per $n \leq 1$. Nel passo ricorsivo, ponendo $r = \text{rango} + 1$ dopo l'esecuzione della distribuzione nella riga 5, osserviamo che r è il rango dell'elemento pivot: ci sono $r-1$ elementi a sinistra del pivot e $n-r$ elementi a destra, per cui $T(n) \leq T(r-1) + T(n-r) + cn$, in quanto `Distribuzione` richiede tempo $O(n)$. Non possiamo purtroppo analizzare tale relazione con il teorema fondamentale. Tuttavia, il caso pessimo è quando il pivot è tutto a sinistra ($r = 1$) oppure tutto a destra ($r = n$): in entrambi i casi, la relazione diventa $T(n) \leq T(n-1) + T(0) + cn \leq T(n-1) + c'n$ per un'opportuna costante c' tale che $c'n \geq T(0) + cn$. Quindi $T(n) = O(n^2)$ se l'array è già ordinato: infatti, a ogni chiamata ricorsiva la distribuzione degli elementi è estremamente sbilanciata, risultando sempre $n-1$ elementi ancora da ordinare da una chiamata ricorsiva. Non è difficile vedere che in questa situazione l'analisi del costo è simile a quella dell'ordinamento per selezione o per inserimento.

Esercizio svolto 3.3 Mostrare che la relazione $T(n) \leq T(n-1) + c'n$ fornisce $T(n) = O(n^2)$.

Soluzione Espandendo ripetutamente il termine ricorsivo a sinistra della relazione, otteniamo $T(n) \leq T(n-1) + c'n \leq T(n-2) + c'(n-1) + c'n \leq T(n-3) + c'(n-2) + c'(n-1) + c'n \leq \cdots \leq c' \sum_{k=2}^{n} k + T(1) = \Theta(n^2)$.

Se invece la distribuzione è bilanciata ($r = n/2$), la ricorsione avviene su ciascuna metà e, applicando il teorema delle ricorrenze, possiamo mostrare che il costo è di tempo $O(n \log n)$ (che, come sappiamo, rappresenta il caso migliore che possa capitare). Per il caso medio, è possibile dimostrare che l'algoritmo richiede tempo $O(n \log n)$ perché si possono alternare, durante la ricorsione, situazioni che danno luogo a una distribuzione sbilanciata con situazioni che conducono a distribuzioni bilanciate. Tuttavia, tale costo medio dipende dall'ordine iniziale con cui sono presentati gli elementi nell'array da ordinare.

Come il suo nome suggerisce, l'algoritmo di quicksort è molto veloce in pratica e viene usato diffusamente per ordinare i dati nella memoria principale. La libreria standard del linguaggio C usa un algoritmo di quicksort in cui il caso base della ricorsione avviene per $n \leq n_0$ per una certa costante $n_0 > 1$: terminata la ricorsione, ogni segmento di al più n_0 elementi viene ordinato individualmente, ma basta una singola passata dell'ordinamento per inserzione per ordinare tutti questi segmenti in $O(n \times n_0) = O(n)$ tempo (risparmiando la maggior parte delle chiamate ricorsive).

Possiamo modificare lo schema ricorsivo del Codice 3.5 per risolvere il classico problema della **selezione** dell'elemento con rango `r` in un array `a` di `n` elementi distinti, *senza* bisogno di ordinarli (ricordiamo che `a` contiene `r` elementi minori o uguali di tale elemento): notiamo che tale problema diventa quello di trovare il minimo in `a` quando `r = 1` e il massimo quando `r = n`. Per risolvere il problema per un qualunque valore di `r` con $1 \leq r \leq n$, ricordiamo che la funzione `Distribuzione` del Codice 3.6 permette di trovare il rango del pivot, posizionando tutti gli elementi di rango inferiore alla sua sinistra e tutti quelli di rango superiore alla sua destra. In base a tale osservazione, possiamo modificare il codice di ordinamento per distribuzione considerando che, per risolvere il problema della selezione, è sufficiente proseguire ricorsivamente nel *solo* segmento dell'array contenente l'elemento da selezionare: otteniamo così il Codice 3.7, che determina tale segmento sulla base del confronto tra `r-1` e `rango` (righe 8-14). La ricorsione ha termine quando il segmento è composto da un solo elemento, nel qual caso il codice restituisce tale elemento (notiamo che alcuni elementi dell'array sono stati spostati durante l'esecuzione dell'algoritmo). Il costo medio di tale algoritmo è `O(n)`, quindi minore del costo dell'ordinamento per confronti.

Codice 3.7 Selezione dell'elemento di rango `r` per distribuzione in un array `a`.

```
1  QuickSelect( a, sinistra, r, destra ):
2                                  ⟨pre: 0 ≤ sinistra ≤ r - 1 ≤ destra ≤ n - 1⟩
3    IF (sinistra == destra) {
4      RETURN a[sinistra];
5    } ELSE {
6      scegli pivot nell'intervallo [sinistra...destra];
7      rango = Distribuzione( a, sinistra, pivot, destra );
8      IF (r-1 == rango) {
9        RETURN a[rango];
10     } ELSE IF (r-1 < rango) {
11       RETURN QuickSelect( a, sinistra, r, rango-1 );
12     } ELSE {
13       RETURN QuickSelect( a, rango+1, r, destra );
14     }
15   }
```


3.5 Moltiplicazione veloce di due numeri interi

Un numero intero di `n` cifre decimali, con `n` arbitrariamente grande, può essere rappresentato mediante un array `x` di `n + 1` elementi, in cui `x[0]` è un intero che rappresenta il segno (ossia `+1` o `-1`) e `x[i]` è un intero che rappresenta l'`i`-esima cifra più significativa, dove $0 \leq x[i] \leq 9$ e $1 \leq i \leq n$. Ovviamente, tale rappresentazione è più dispendiosa dal punto di vista della memoria utilizzata, ma consente di operare su numeri arbitrariamente grandi.[1] Vediamo ora come sia possibile eseguire le due operazioni di somma e di prodotto facendo riferimento a tale rappresentazione (nel seguito, senza perdita di generalità, supponiamo che i due interi da sommare o moltiplicare siano rappresentati entrambi mediante `n` cifre decimali, dove `n` è una potenza di due: in caso contrario, infatti, tale condizione può essere ottenuta aggiungendo alla loro rappresentazione una quantità opportuna di `0` nelle posizioni più significative).

Per quanto riguarda la somma, il familiare algoritmo che consiste nell'addizionare le singole cifre propagando l'eventuale riporto, richiede `O(n)` passi ed è quindi ottimo. Non possiamo fare lo stesso discorso per l'algoritmo di moltiplicazione che viene insegnato nelle scuole, in base al quale viene eseguito il prodotto del moltiplicando per ogni cifra del moltiplicatore, eseguendo poi `n` addizioni di numeri di `O(n)` cifre per ottenere il risultato desiderato. Infatti, la complessità di tale algoritmo per la moltiplicazione è $O(n^2)$ tempo.

Facendo uso del paradigma del divide et impera e di semplici uguaglianze algebriche possiamo mostrare ora come ridurre significativamente tale complessità. Osserviamo anzitutto che possiamo scrivere ogni numero intero w di n cifre come $10^{n/2} \times w_s + w_d$, dove w_s denota il numero formato dalle $n/2$ cifre più significative di w e w_d denota il numero formato dalle $n/2$ cifre meno significative. Per moltiplicare due numeri x e y, vale quindi l'uguaglianza

$$xy = (10^{n/2}x_s + x_d)(10^{n/2}y_s + y_d) = 10^n x_s y_s + 10^{n/2}(x_s y_d + x_d y_s) + x_d y_d$$

che ci conduce al seguente algoritmo basato sul paradigma del divide et impera.

Decomposizione: se x e y hanno almeno due cifre, dividili come numeri x_s, x_d, y_s e y_d aventi ciascuno la metà delle cifre.

Ricorsione: calcola ricorsivamente le moltiplicazioni $x_s y_s$, $x_s y_d$, $x_d y_s$ e $x_d y_d$.

Ricombinazione: combina i numeri risultanti usando l'uguaglianza suddetta.

Quindi, indicato con `T(n)` il numero totale di passi eseguiti per la moltiplicazione di due numeri di `n` cifre, eseguiamo quattro moltiplicazioni di due numeri di `n/2`

[1] Il problema di gestire numeri interi arbitrariamente grandi ha diverse applicazioni tra cui i protocolli crittografici: esistono apposite implementazioni per molti linguaggi di programmazione, come, per esempio, la classe `BigInteger` del pacchetto `java.math`.

cifre, dove ciascuna moltiplicazione richiede un costo di $T(n/2)$, e tre somme di due numeri di n cifre. Osserviamo che per ogni $k > 0$, la moltiplicazione per il valore 10^k può essere realizzata spostando le cifre di k posizioni verso sinistra e riempiendo di 0 la parte destra. Pertanto, il costo della decomposizione e della ricombinazione è $O(n)$ in tempo e, maggiorando tale termine con cn per una costante $c > 0$ e indicando il costo per il caso base con la costante $c_0 > 0$, possiamo esprimere $T(n)$ mediante la relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 4T(n/2) + cn & \text{altrimenti} \end{cases} \tag{3.7}$$

Applicando il teorema fondamentale delle ricorrenze alla (3.7), otteniamo $\alpha = 4$, $\beta = 2$ e $f(n) = n$ nella relazione (3.2). Poiché $\alpha f(n/\beta) = 4(n/2) = 2n = 2f(n)$, rientriamo nel terzo caso del teorema con $\gamma = 2$. Tale caso consente di affermare che il numero di passi richiesti è $O(n^{\log 4}) = O(n^2)$, non migliorando quindi le prestazioni del familiare algoritmo di moltiplicazione precedentemente descritto. Tuttavia, facendo uso di un'altra semplice uguaglianza algebrica possiamo migliorare significativamente il costo computazionale dell'algoritmo basato sul paradigma del divide et impera.

Teorema 3.4 *La moltiplicazione di due numeri interi di n cifre decimali può essere eseguita in tempo $O(n^{1,585})$.*

Dimostrazione Ricordiamo che, dati due numeri x e y, vale l'uguaglianza

$$xy = (10^{n/2}x_s + x_d)(10^{n/2}y_s + y_d) = 10^n x_s y_s + 10^{n/2}(x_s y_d + x_d y_s) + x_d y_d$$

Osserviamo che il valore $x_s y_d + x_d y_s$ può essere calcolato facendo uso degli altri due valori $x_s y_s$ e $x_d y_d$ nel modo seguente:

$$x_s y_d + x_d y_s = x_s y_s + x_d y_d - (x_s - x_d) \times (y_s - y_d)$$

Quest'osservazione permette di formulare un nuovo algoritmo di moltiplicazione, descritto nel Codice 3.8, il cui passo di ricombinazione richiede soltanto tre moltiplicazioni (righe 11, 14 e 16) e un numero costante di somme: a tal fine, utilizziamo la funzione `Somma` per calcolare l'addizione di al più tre numeri interi in tempo lineare nel loro numero totale di cifre decimali e, attraverso `Somma`, possiamo ottenere la sottrazione complementando il segno dell'operando sottratto poiché $x - y = x + (-y)$. Seppur concettualmente semplice, l'algoritmo richiede un'implementazione attenta delle varie operazioni, come mostrato nel Codice 3.8, il quale prende in ingresso due numeri con n cifre decimali e ne restituisce il prodotto su $2n$ cifre.

Codice 3.8 Moltiplicazione mediante la tecnica del divide et impera, dove `Somma` calcola l'addizione di al più tre numeri interi rappresentati come array, in tempo lineare. Gli array `prodotto` e `parziale` sono inizializzati a `0`.

```
1  MoltiplicazioneVeloce(x, y, n): ⟨pre: x e y interi di n cifre; n potenza di 2⟩
2    IF (n == 1) {
3      prodotto[1] = (x[1] × y[1]) / 10;
4      prodotto[2] = (x[1] × y[1]) % 10;
5    } ELSE {
6      xs[0] = xd[0] = ys[0] = yd[0] = 1;
7      FOR (i = 1; i <= n/2; i = i + 1) {
8        xs[i] = x[i]; ys[i] = y[i];
9        xd[i] = x[i + n/2]; yd[i] = y[i + n/2];
10     }
11     p1 = MoltiplicazioneVeloce( xs, ys, n/2 );
12     FOR (i = 0; i <= n; i = i+1)
13       { prodotto[i] = p1[i]; }
14     p2 = MoltiplicazioneVeloce( xd, yd, n/2 );
15     xd[0] = yd[0] = -1;
16     p3 = MoltiplicazioneVeloce( Somma(xs,xd), Somma(ys,yd), n/2 );
17     p3[0] = -p3[0];
18     add = Somma( p1, p2, p3 );
19     parziale[0] = add[0];
20     FOR (i = 1; i <= 3 × n/2; i = i+1)
21       { parziale[i] = add[i + n/2]; }
22     prodotto = Somma( prodotto, parziale, p2 );
23   }
24   prodotto[0] = x[0] × y[0];
25   RETURN prodotto;                    ⟨post: prodotto intero di 2n cifre⟩
```

Nel caso base ($n = 1$) effettuiamo il prodotto diretto delle singole cifre, riportandone il risultato su due cifre (righe 3-4) e il relativo segno (riga 24). Nel passo induttivo, calcoliamo x_s, x_d, y_s e y_d prendendo l'opportuna metà delle cifre dal valore assoluto di x e y (righe 6-10). Calcoliamo quindi $p1 = x_s y_s$ ricorsivamente su $n/2$ cifre (laddove $p1$ ne ha n), e poniamo in `prodotto` (che ha $2n$ cifre) il valore di $10^n \times x_s y_s$ (righe 11-13). Procediamo con il calcolo ricorsivo di $p2 = x_d y_d$ (riga 14) e $p3 = (x_s - x_d) \times (y_s - y_d)$ (righe 15-16), entrambi di $2n$ cifre (notiamo che sia $x_s - x_d$ che $y_s - y_d$ richiedono $n/2$ cifre). Poniamo il risultato di $p1 + p2 - p3 = x_s y_d + x_d y_s$ in `add` (righe 17-18), la cui moltiplicazione per $10^{n/2}$ viene memorizzata in `parziale` (righe 19-21). A questo punto, per ottenere il prodotto tra il valore assoluto di x e quello di y, è sufficiente calcolare la somma tra `prodotto` (che contiene $10^n \times x_s y_s$), `parziale` (che contiene $10^{n/2} \times (x_s y_d + x_d y_s)$) e

p2 = $x_d y_d$ (riga 22). Il segno del prodotto viene infine calcolato nella riga 24. Il numero totale di passi eseguiti è quindi pari a

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 3T(n/2) + cn & \text{altrimenti} \end{cases} \qquad (3.8)$$

dove c_0 e c sono due costanti positive. Applicando il teorema fondamentale delle ricorrenze alla (3.8), otteniamo $\alpha = 3$, $\beta = 2$ e $f(n) = n$ nella relazione (3.2): anche questa volta rientriamo nel terzo caso del teorema con $\gamma = \frac{3}{2}$, che consente di affermare che il numero di passi richiesti è $O(n^{\log_2 3}) = O(n^{1,585})$. ❑

ESEMPIO 3.4

Applichiamo l'algoritmo `MoltiplicazioneVeloce` agli interi x = -1425 e y = 2322. Gli interi sono memorizzati in array la cui dimensione è data dal numero di cifre più uno per la cifra di segno memorizzata nella posizione 0. In questo caso il valore di n è 4 quindi gli array x e y hanno dimensione 5 mentre l'array `prodotto` conterrà 8 cifre e avrà dimensione 9. Nelle righe 6-10 vengono costruiti gli array (di 2 cifre) contenenti i valori assoluti delle due metà degli array x e y.

```
xs = +14      xd = +25
ys = +23      yd = +22.
```

Si è esplicitato anche il segno come indicato nella prima posizione degli array.

Gli array p1 e p2 di, rispettivamente, 4 e 8 cifre sono ottenuti calcolando ricorsivamente il prodotto tra gli interi di 2 cifre memorizzati rispettivamente in xs e ys (riga 11) e xd e yd (riga 14).

```
p1 = +0322    p2 = +00000550
```

Il ciclo nella riga 13 copia tutte le cifre di p1 nelle prime posizioni di `prodotto`, le restanti posizioni sono riempite con zeri. Questo equivale a una moltiplicazione per 10^4.

```
prodotto = +03220000.
```

Per il calcolo di p3 (di 8 cifre) prima si inverte il segno di xd e yd per poter calcolare $x_s - x_d$ e $y_s - y_d$ utilizzando la funzione Somma.

```
Somma(+14, -25) = -11        Somma(+23, -22) = +01.
```

Quindi, l'invocazione ricorsiva nella riga 17 calcola p3, di 8 cifre, moltiplicando i risultati delle somme precedenti.

```
p3 = -00000011.
```

Il calcolo di $x_s y_d + x_d y_s$ viene effettuato sommando p1, p2 e p3, quest'ultimo con il segno invertito. Il risultato viene memorizzato in add di 8 cifre.

```
add = +00000883.
```

L'array `parziale` di 8 cifre contiene il valore in add moltiplicato per $100 = 10^{n/2}$, questo viene ottenuto copiando le cifre di add che vanno dalla posizione n/2 + 1 = 3 in poi in `parziale` a partire dalla posizione 1. Le altre posizioni di `parziale` sono riempite di 0 (riga 21).

```
parziale = +00088300.
```


Infine `prodotto` viene calcolato come somma dello stesso `prodotto`, di `parziale` e di p2. Il segno è dato dal prodotto dei segni di x e y.

```
prodotto = -03308850.
```

L'idea alla base del Codice 3.8 può essere ulteriormente sviluppata spezzando i numeri in parti più piccole, per ottenere la moltiplicazione in $O(n \log n \log\log n)$ passi, analogamente a quanto accade nella *trasformata veloce di Fourier*, che è di grande importanza per una grande varietà di applicazioni, che vanno dall'elaborazione di segnali digitali alla soluzione numerica di equazioni differenziali.

3.6 Opus libri: grafica e moltiplicazione di matrici

La definizione di sequenza lineare data all'inizio del capitolo può essere estesa anche al caso in cui consideriamo un'organizzazione degli elementi di un insieme su un **array bidimensionale**, o **matrice**. Anche nel caso degli array bidimensionali, gli elementi della sequenza sono conservati in locazioni contigue della memoria: a differenza degli array monodimensionali, tali locazioni sono organizzate in due dimensioni, ovvero in **righe** e **colonne**. Ogni riga contiene un numero di elementi pari al numero delle colonne dell'array e l'elemento contenuto nella colonna j della riga i di un array A viene indicato con `A[i][j]`. Per esempio, nella Figura 3.1, viene mostrato un array bidimensionale A di 4 righe e 5 colonne (indicato come $A_{4\times5}$) e, per ogni elemento, viene mostrata la notazione con cui esso viene indicato. L'organizzazione bidimensionale delle locazioni di memoria non modifica la principale proprietà degli array, ovvero la possibilità di accedere in modo diretto ai suoi elementi.[2]

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]
A[3][0]	A[3][1]	A[3][2]	A[3][3]	A[3][4]

Figura 3.1 Un array bidimensionale o matrice $A_{4\times5}$.

[2] Un array bidimensionale $A_{r\times c}$ di r righe e c colonne può sempre essere visto come un array monodimensionale b contenente r × c elementi: per ogni i e j con $0 \leq i < r$ e $0 \leq j < c$, l'accesso all'elemento `A[i][j]` corrisponde semplicemente all'accesso a `b[i × c + j]` in tempo $O(1)$. Sebbene in questo libro non ne faremo mai uso, non è difficile immaginare come sia possibile estendere il concetto di array a k dimensioni con k > 2, così che l'accesso a un elemento dell'array avvenga sulla base di k indici in tempo $O(k)$.

Un tipo particolarmente importante di matrici è rappresentato da quelle in cui gli elementi contenuti sono interi o reali: nate per rappresentare sistemi di equazioni lineari nel calcolo scientifico, le matrici sono utilizzate, tra le altre cose, per risolvere problemi su grafi e per classificare l'importanza delle pagine web come vedremo nel seguito del libro. In questo paragrafo, discutiamo la loro importanza nel campo della grafica al calcolatore (*computer graphics*) e della visione artificiale. Una matrice $A = A_{r \times c}$ può modellare i punti luminosi (*pixel*) in cui è discretizzata un'immagine digitale contenuta nella memoria video (*frame buffer*) avente risoluzione $c \times r$ pixel, dove 1024×768, 1280×1024, 1400×1050, 1600×1200 e 1920×1200 sono alcuni dei formati digitali standard (da notare che, nella risoluzione del video, indichiamo prima il numero delle colonne `c` e poi quello delle righe `r`). Il pixel che si trova in corrispondenza della riga `i` e della colonna `j` è rappresentato dall'elemento `A[i][j]`, che specifica (direttamente o indirettamente) il colore e la luminosità del pixel utilizzando uno certo numero di bit (*bit depth*, solitamente pari a 24 o 32).

Quando usiamo un videogioco tridimensionale, stiamo più o meno inconsapevolmente impiegando delle matrici nella rappresentazione delle scene che possiamo osservare muovendoci all'interno dello spazio di gioco. In genere queste scene sono realizzate mediante superfici composte da innumerevoli triangoli disposti nello spazio tridimensionale, i cui vertici di coordinate `(x, y, z)` sono rappresentati mediante array di quattro elementi `[x, y, z, 1]` (l'uso della quarta coordinata pari a 1 sarà chiarito fra breve). Per arrivare a mostrare sul frame buffer del video tali scene in forma digitale (*rendering*) occorre che tutte le primitive geometriche che compongono ciascuna scena attraversino diverse fasi di computazione:

1. una fase di trasformazione per scalare, ruotare e traslare le figure geometriche in base alla vista attuale della scena;
2. una fase di illuminazione per calcolare quanta luce arrivi direttamente su ogni vertice;
3. una fase di trasformazione per dare la prospettiva dell'occhio umano alla vista attuale della scena;
4. una fase di ritaglio (*clipping*) per selezionare solo gli oggetti visibili nella vista attuale;
5. una fase di proiezione della vista attuale in tre dimensioni in un piano bidimensionale (equivalente a un'inquadratura ottica simulata con la grafica vettoriale);
6. una fase di resa digitale (*rastering*) per individuare quali pixel del frame buffer siano infine coperti dalla primitiva appena proiettata.

La realizzazione efficiente di tali fasi di rendering richiede perizia di programmazione e profonda conoscenza algoritmica e matematica (ebbene sì, dobbiamo imparare molta matematica per programmare la grafica dei videogiochi) e si basa sull'impiego massiccio di potenti e specializzate schede grafiche. In particolare, la fase 1 effettua la trasformazione mediante operazioni di somma e prodotto di matrici, definite qui di seguito.

La **somma** $A+B$ di due matrici $A_{r \times s}$ e $B_{r \times s}$ è la matrice $C_{r \times s}$ tale che `C[i][j] = A[i][j] + B[i][j]` per ogni coppia di indici $0 \le i \le r-1$ e $0 \le j \le s-1$: ogni elemento della matrice `C` è quindi pari alla somma degli elementi di `A` e `B` nella medesima posizione.

Il **prodotto** $A \times B$ di due matrici $A_{r \times s}$ e $B_{s \times t}$ è la matrice $C_{r \times t}$ tale che `C[i][j]` $= \sum_{k=0}^{s-1} (A[i][k] \times B[k][j])$ per ogni coppia di indici $0 \le i \le r-1$ e $0 \le j \le t-1$: notiamo che, contrariamente al prodotto tra due interi o reali, tale prodotto *non* è commutativo, ma è associativo. L'elemento `C[i][j]` è anche detto **prodotto scalare** della riga `i` di `A` per la colonna `j` di `B`.

Nella Figura 3.2 illustriamo come scalare, ruotare e traslare una figura mostrando, ai fini della nostra discussione, tali operazioni solo per un punto bidimensionale `[x, y, 1]`.

Per scalare di un fattore α lungo l'asse delle ascisse e di un fattore β lungo quello delle ordinate, effettuiamo la moltiplicazione

$$[x, y, 1] \times \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} = [\alpha x, \beta y, 1]$$

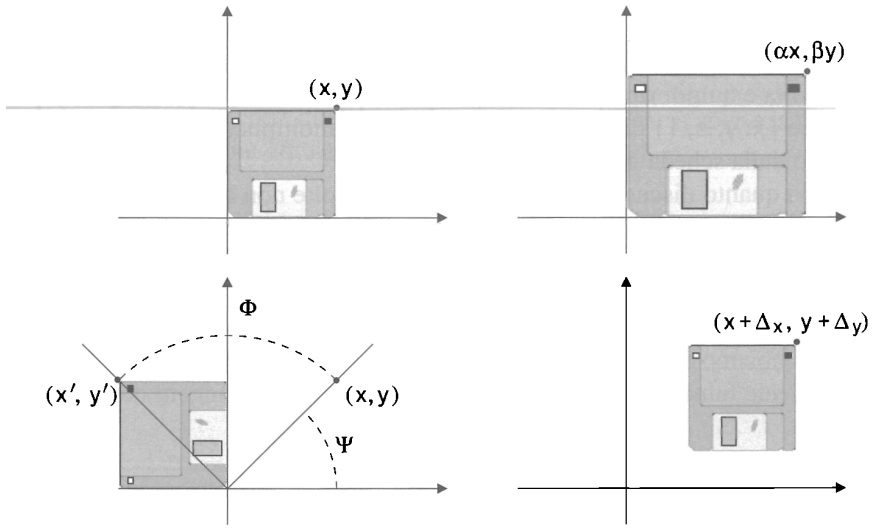


Figura 3.2 Operazioni su matrici per scalare, ruotare e traslare un punto di ascissa x e ordinata y nella fase 1 del rendering di un'immagine digitale.

Per ruotare di un angolo Φ, osserviamo che la nuova posizione $[x', y', 1]$ soddisfa la relazione trigonometrica $x' = x\cos\Phi - y\sin\Phi$ e $y' = x\sin\Phi + y\cos\Phi$, per cui la corrispondente moltiplicazione è la seguente:[3]

$$[x, y, 1] \times \begin{bmatrix} \cos\Phi & \sin\Phi & 0 \\ -\sin\Phi & \cos\Phi & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x', y', 1]$$

Infine, la traslazione per una quantità Δ_x sulle ascisse e per una quantità Δ_y sulle ordinate, necessita della dimensione fittizia per essere espressa come una moltiplicazione

$$[x, y, 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta_x & \Delta_y & 1 \end{bmatrix} = [x + \Delta_x, y + \Delta_y, 1]$$

Non è difficile estendere le suddette trasformazioni al caso tridimensionale, in cui le matrici sono di dimensione 4×4. Altre trasformazioni possono essere espresse mediante la moltiplicazione di opportune matrici come, per esempio, quella per rendere speculare una figura. Il vantaggio di esprimere tutte le trasformazioni in termini di moltiplicazioni risiede nel fatto che così esse possono essere composte in qualunque modo mediante la moltiplicazione delle loro corrispettive matrici. In altri termini, una qualunque sequenza di n trasformazioni applicate a un punto $[x, y, z, 1]$ può essere vista come la moltiplicazione di quest'ultimo per le n matrici $A_0, A_1, \ldots, A_{n-1}$ che rappresentano le trasformazioni stesse:

$$[x, y, z, 1] \times A_0 \times A_1 \times \cdots \times A_{n-1}$$

Tuttavia, dovendo ripetere tale sequenza per tutti i vertici delle figure che si vogliono trasformare è più efficiente calcolare la matrice $A^* = A_0 \times A_1 \times \cdots \times A_{n-1}$ una sola volta e quindi ripetere una *singola* moltiplicazione $[x, y, z, 1] \times A^*$ per ogni vertice $[x, y, z, 1]$ di tali figure (queste ultime moltiplicazioni sono eseguite in parallelo dalla scheda grafica del calcolatore).

Secondo quanto discusso finora, le matrici coinvolte non hanno più di quattro righe e quattro colonne. Tuttavia, per raggiungere un certo grado di realismo nel processo di rendering è necessario simulare con buona approssimazione il comportamento della luce in una scena, calcolando per esempio ombre portate, riflessioni ed effetti di illuminazione indiretta. Nel calcolo dell'illuminazione indiretta, dobbiamo determinare quanta luce arrivi su di un punto, non direttamente da una sorgente luminosa come il sole, ma riflessa dagli altri oggetti della scena.

[3] Se Ψ è l'angolo che il punto (x, y) forma con l'asse delle x e r è la sua distanza dall'origine $(0,0)$, allora, usando le ben note uguaglianze trigonometriche $\cos(\Phi+\Psi) = \cos\Phi\cos\Psi - \sin\Phi\sin\Psi$ e $\sin(\Phi+\Psi) = \sin\Phi\cos\Psi + \cos\Phi\sin\Psi$, otteniamo che $x' = r\cos(\Phi+\Psi) = r\cos\Phi\cos\Psi - r\sin\Phi\sin\Psi$ e $y' = r\sin(\Phi+\Psi) = r\sin\Phi\cos\Psi + r\cos\Phi\sin\Psi$. Poiché $r\cos\Psi = x$ e $r\sin\Psi = y$, abbiamo che vale la relazione utilizzata per la moltiplicazione.

Una delle soluzioni classiche a questo problema, noto come il calcolo della **radiosità**, consiste nel calcolare una matrice $M_{m \times m}$ di vaste dimensioni per una scena composta da m primitive elementari, in cui l'elemento $M[i][j]$ della matrice descrive in percentuale quanta luce che rimbalza sulla superficie i arriva anche sulla superficie j.

Tralasciando come ottenere tali valori, partiamo da un array L_0 di m elementi in cui memorizziamo quanta luce esce da ognuna delle m primitive (ponendo a 0 gli elementi per ogni primitiva eccetto che per le sorgenti luminose). Moltiplicando M per L_0 si ottiene un array L_1 in cui ogni elemento contiene l'illuminazione diretta della primitiva corrispondente. Continuando a moltiplicare per M, otteniamo una serie di array L_i (dove $i > 1$) che rappresentano via via i contributi delle varie "riflessioni" della luce sulle varie superfici, riuscendo così a determinare i contributi dell'illuminazione indiretta per ogni primitiva.

Nel resto del paragrafo ipotizziamo di avere un numero arbitrariamente grande di righe e di colonne e studiamo la complessità dell'algoritmo di moltiplicazione di due matrici.

3.6.1 Moltiplicazione veloce di due matrici

L'algoritmo immediato per calcolare la somma $A + B$ di due matrici A e B, effettua $r \times s$ operazioni di somma (una per ogni elemento di C): poiché dobbiamo sempre esaminare $\Omega(r \times s)$ elementi nelle matrici, abbiamo che la somma di due matrici può essere quindi effettuata in tempo ottimo $O(r \times s)$. Invece, l'algoritmo immediato per il prodotto di due matrici $A_{r \times s}$ e $B_{s \times t}$, mostrato nel Codice 3.9, non è ottimo. Esso richiede di effettuare $O(s)$ operazioni per ognuno degli $r \times t$ elementi di C, richiedendo così un totale di $O(r \times s \times t)$ operazioni. A differenza della somma di matrici, in questo caso esiste una differenza, o **gap di complessità**, con il limite inferiore pari a $\Omega(r \times s + s \times t)$. Restringendoci al caso di matrici *quadrate*, le quali hanno $n = r = s = t$ righe e colonne, osserviamo che il limite superiore è $O(n^3)$, mentre quello inferiore è $\Omega(n^2)$: ciò lascia aperta la possibilità di trovare algoritmi più efficienti per il prodotto di matrici.

Codice 3.9 Algoritmo per la moltiplicazione di due matrici in tempo $O(r \times s \times t)$.

```
ProdottoMatrici( A, B ):          ⟨pre: A e B sono di taglia r×s e s×t⟩
  FOR (i = 0; i < r; i = i+1)
    FOR (j = 0; j < t; j = j+1) {
      C[i][j] = 0;
      FOR (k = 0; k < s; k = k+1)
        C[i][j] = C[i][j] + A[i][k] × B[k][j];
    }
  RETURN C;                       ⟨post: C è di taglia r×t⟩
```

Analogamente alla moltiplicazione veloce tra numeri arbitrariamente grandi (Paragrafo 3.5), l'applicazione del paradigma del divide et impera permette di definire algoritmi per il prodotto di matrici con complessità temporale nettamente inferiore a quella $O(n^3)$ dell'algoritmo descritto nel Codice 3.9. L'**algoritmo di Strassen**, che descriviamo per semplicità nel caso della moltiplicazione di matrici quadrate in cui n è una potenza di due, rappresenta il primo e più diffuso esempio teorico dell'applicazione del divide et impera: esso è basato sull'osservazione che una moltiplicazione di due matrici 2×2 può essere effettuata, nel modo seguente, per mezzo di 7 moltiplicazioni (e 14 addizioni), invece delle 8 moltiplicazioni (e 4 addizioni) del metodo standard.

Consideriamo la seguente moltiplicazione da effettuare:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Se introduciamo i seguenti valori

$$\begin{aligned} v_0 &= (b - d)(g + h) & v_4 &= a(f - h) \\ v_1 &= (a + d)(e + h) & v_5 &= d(g - e) \\ v_2 &= (a - c)(e + f) & v_6 &= (c + d)e \\ v_3 &= (a + b)h \end{aligned}$$

possiamo osservare che la moltiplicazione precedente può essere espressa come

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} v_0 + v_1 - v_3 + v_5 & v_3 + v_4 \\ v_5 + v_6 & v_1 - v_2 + v_4 - v_6 \end{bmatrix}$$

Questa considerazione, che non pare introdurre alcuna convenienza nel caso di matrici 2×2, risulta invece interessante se consideriamo la moltiplicazione di matrici $n \times n$ per $n > 2$. In tal caso, ciascuna matrice di dimensione $n \times n$ può essere considerata come una matrice 2×2, in cui ciascuno degli elementi a, b, ..., h è una matrice di dimensione $n/2 \times n/2$: le relative operazioni di somma e moltiplicazione su di essi sono quindi somme e moltiplicazioni tra matrici.

Indicando con $T(n)$ il costo temporale della moltiplicazione di due matrici $n \times n$ e applicando la considerazione precedente, osserviamo che $T(n)$ è pari al costo di esecuzione di 7 moltiplicazioni tra matrici $\frac{n}{2} \times \frac{n}{2}$, che esprimiamo come $7T\left(\frac{n}{2}\right)$, più il costo di esecuzione di 14 somme di matrici anch'esse $\frac{n}{2} \times \frac{n}{2}$, che possiamo stimare come $O(n^2)$. Da quanto detto deriva la relazione di ricorrenza seguente, dove c_0 e c sono opportune costanti positive:

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 2 \\ 7T\left(\frac{n}{2}\right) + cn^2 & \text{altrimenti} \end{cases} \tag{3.9}$$

Applicando il teorema fondamentale delle ricorrenze alla relazione (3.9), con $\alpha = 7$, $\beta = 2$ e $f(n) = n^2$ nella relazione (3.2), osserviamo che ci troviamo nel terzo caso considerato dal teorema, in quanto $7\left(\frac{n}{2}\right)^2 = \frac{7}{4}n^2$ (quindi, $\gamma' = \frac{7}{4} > 1$): pertanto, $T(n) = O(n^{\log 7}) = O(n^{2,807\cdots})$. È tuttora ignota la complessità della moltiplicazione di due matrici quadrate e la congettura più diffusa è che sia $O(n^\varepsilon)$ per una costante $2 \leq \varepsilon < 2,3727$.

3.7 Opus libri: il problema della coppia più vicina

Un problema classico in *geometria computazionale* è quello di trovare la coppia di punti più vicina tra un insieme di punti del piano. In particolare, dato un insieme P di n punti nel piano cartesiano, si vuole progettare un algoritmo che restituisce la coppia di punti di P la cui distanza euclidea è minima. Chiaramente il problema può essere risolto in tempo $O(n^2)$ calcolando le distanze tra tutti i punti. Mostreremo come, utilizzando la tecnica del divide et impera, il problema può essere risolto in tempo $O(n \log n)$.

Intuitivamente l'idea è la seguente. Se l'insieme ha cardinalità costante usiamo la ricerca esaustiva. Altrimenti lo dividiamo in due parti uguali S e D, per esempio quelli a sinistra e quelli a destra di una fissata linea verticale. Troviamo ricorsivamente le soluzioni per l'istanza per S e quella per D individuando due coppie di punti a distanza minima, d_S e d_D, rispettivamente. La soluzione finale può essere una delle due coppie già individuate oppure può essere formata da un punto in S e uno in D. Quindi se d_{SD} è la minima distanza tra punti aventi estremi in S e D, la soluzione finale è data dalla coppia di punti a distanza $\min\{d_{SD}, d_S, d_D\}$.

Lo scopo è quello di eseguire queste operazioni in tempo $O(n)$. Questo ci permette di esprimere il costo computazionale dell'algoritmo appena descritto per sommi capi con la relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 2 \\ 2T\left(\frac{n}{2}\right) + cn & \text{altrimenti} \end{cases} \tag{3.10}$$

dove c_0, c sono costanti. Per il Teorema 3.1, abbiamo $T(n) = O(n \log n)$.

Nel seguito forniamo i dettagli necessari a realizzare quanto abbozzato sopra. Indichiamo, per ogni punto $p \in P$, con $p.x$ la sua ascissa e $p.y$ la sua ordinata. Ipotizziamo che i punti in P abbiano ascisse distinte: se così non è, possiamo usare le loro ordinate per discriminare.

Il primo passo consiste nell'individuare una retta verticale che taglia l'istanza P in due parti di eguale cardinalità. A tale scopo ordiniamo i punti di P per ascissa non decrescente e sia x l'ascissa del punto in posizione centrale nell'ordinamento. Definiamo $S = \{p \in P : p.x \leq x\}$ e $D = \{p \in P : p.x > x\}$, ovvero S è costituito dai punti a sinistra della retta $L = x$ e D da quelli a destra.

Come è stato già detto, l'idea dell'algoritmo è quella di risolvere il problema per i punti in S e D ottenendo due coppie di punti a distanza d_S e d_D rispettivamente. Queste due coppie di punti sono candidate a essere soluzione finale insieme alla coppia di punti a distanza minima composta da un punto in S e uno in D. L'algoritmo sceglierà la migliore tra le tre soluzioni.

Definito $d = \min\{d_S, d_D\}$, l'algoritmo ricerca una eventuale coppia di punti $p_s \in S$ e $p_d \in D$ tale che la loro distanza sia inferiore a d. Osserviamo subito che p_s e p_d non possono essere a distanza maggiore di d dalla retta L. Quindi possiamo restringere la ricerca di p_s nell'insieme $S' = \{p \in S : x - d \leq p.x \leq x\}$ e di p_d in $D' = \{p \in D : x \leq p.x \leq x + d\}$ (Figura 3.3).

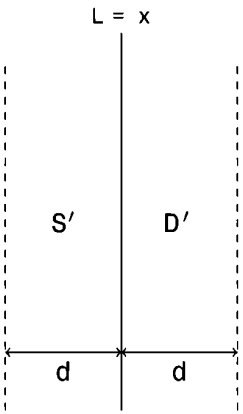


Figura 3.3 La terza coppia di punti candidata a essere soluzione finale ha un estremo in S' e l'altro in D'.

Sia S' che D' potrebbero contenere $\Omega(n)$ punti: tuttavia, fissato un punto $p \in S'$, i punti in D' potenzialmente a distanza inferiore di d da p sono in numero costante.

Lemma 3.1 *Sia P_d la lista dei punti in $S' \cup D'$ ordinati secondo la loro ordinata. Se due punti p e q in $S' \cup D'$ sono a distanza minore di d, allora occorrono come elementi in P_d in due posizioni che distano al più 10.*

Dimostrazione Dividiamo la fascia a distanza d da L in una griglia composta da quadrati di dimensione d/2 (si veda la Figura 3.4), ogni riga è composta da 2 quadrati alla sinistra di L e due alla destra di L. Osserviamo che ogni quadrato contiene al più un punto di $S' \cup D'$: infatti se così non fosse, esisterebbero due punti p_1 e p_2 appartenenti entrambi a S' oppure a D' e a distanza minore di d (contraddicendo la definizione di d).

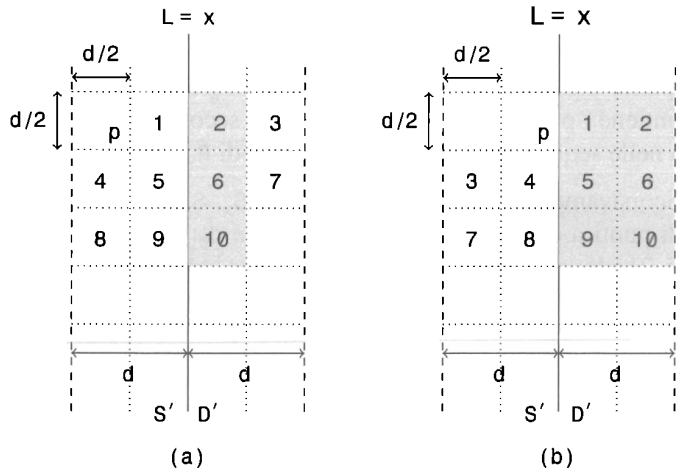


Figura 3.4 Fissato un punto p, le posizioni possibili per un punto q a distanza al più d da p sono quelle in grigio.

Calcoliamo ora il numero massimo di punti in $S' \cup D'$ che separano i punti p e q a distanza minore di d in P_d. Supponiamo, senza perdita di generalità, che p preceda q nell'ordinamento P_d (ovvero che l'ordinata di p sia inferiore a quella di q) e che p sia in S' mentre q sia in D'. Il punto p può trovarsi in una cella con il lato non coincidente con L oppure in una coincidente: nel primo caso (Figura 3.4a) q può trovarsi solo in una delle tre celle annerite della figura in quanto tutte le altre a destra di L hanno una distanza maggiore di d da quella in cui si trova p; nel secondo caso (Figura 3.4b) le celle possibili sono sei. In entrambi i casi, poiché ogni cella contiene esattamente un punto, q è al più 10 posizioni avanti a p. ❑

Dal Lemma 3.1 si deduce che per trovare una coppia di punti in $S' \cup D'$ a distanza minore di d è sufficiente scorrere i punti nell'ordinamento P_d e, per ognuno di questi, calcolarne la distanza dai 10 punti che lo seguono nell'ordinamento. Il costo di questa operazione è lineare in n.

Osserviamo che, affinché valga la relazione di ricorrenza descritta nell'equazione (3.10), dobbiamo essere in grado di dividere l'istanza in due sottoistanze come descritto precedentemente e costruire un ordinamento dei punti rispetto la ordinata in tempo $O(n)$. Questo si ottiene creando due ordinamenti dei punti di P uno rispetto l'ascissa (P_x) e uno rispetto l'ordinata (P_y). Questa operazione viene effettuata preliminarmente ovvero prima dell'invocazione della ricorsione. La parte ricorsiva dell'algoritmo prende in input i due ordinamenti P_x e P_y e prosegue come descritto in seguito (si veda il Codice 3.10).

1. Sia n il numero di punti in input: se l'istanza è sufficientemente piccola, esegui una ricerca esaustiva (righe 3-4).

2. Altrimenti, costruisci gli ordinamenti S_x e D_x selezionando, rispettivamente, i primi n/2 punti di P_x e i secondi n/2 punti di P_x (righe 6-14).
3. Sia p il punto mediano di P_x. Dividi i punti di P_y in due sequenze S_y e D_y: la prima contiene i punti di P_y a sinistra di p e la seconda quelli a destra. L'ordinamento nelle sequenze S_y e D_y rispetta quello di P_y (righe 6-14).
4. Invoca ricorsivamente la funzione con input (S_x, S_y) e (D_x, D_y) ottenendo due coppie di punti a distanza rispettivamente d_S e d_D: definisci d = min{d_S, d_D} (righe 15-21, dove Dist denota la funzione per calcolare la distanza euclidea tra due punti).
5. Dalla sequenza ordinata P_y estrapola i punti a distanza al più d dalla retta verticale passante per p, creando la sequenza P_d (di dimensione m) che rispetta l'ordinamento di P_y (righe 22-26).
6. Trova la coppia (p, q) finale usando come candidate la coppia a distanza al più d trovata nel passo ricorsivo e le coppie di punti in P_d limitandosi a verificare, come descritto precedentemente, la distanza di ogni punto con i 10 punti che lo seguono nella sequenza (righe 27-33).

Codice 3.10 Algoritmo ricorsivo per individuare la coppia di punti più vicini nel piano.

```
CoppiaPiuVicina( Px, Py, n ):
                    ⟨pre: gli array Px e Py sono ordinati e la loro dimensione è n⟩
  IF (n <= 3) {
    RETURN RicercaEsaustiva(Px, Py, n);
  } ELSE {
    p = Px[n/2];
    FOR (i = j = k = 0; i < n/2; i = i + 1) {
      Sx[i] = Px[i]; Dx[i] = Px[i+n/2];
      IF (Py[i].x <= p.x) {
        Sy[j] = Py[i]; j = j+1;
      } ELSE {
        Dy[k] = Py[i]; k = k+1;
      }
    }
    (ps,qs) = CoppiaPiuVicina(Sx, Sy, n/2);
    (pd,qd) = CoppiaPiuVicina(Dx, Dy, n/2);
    IF (Dist(ps, qs) < Dist(pd, qd)) {
      d = Dist(ps, qs); (p, q) = (ps, qs);
    } ELSE {
      d = Dist(pd, qd); (p, q) = (pd, qd);
    }
    FOR (i = m = 0; i < n; i = i + 1) {
      IF (|Py[i].x - p.x| <= d) {
        Pd[m] = Py[i]; m = m+1;
      }
    }
    FOR (i = 0; i < m; i = i + 1) {
      FOR (j = i+1; j <= min{i+10, m}; j = j + 1) {
        IF (Dist(Pd[i], Pd[j]) < d) {
          d = Dist(Pd[i], Pd[j]); (p, q) = (Pd[i], Pd[j]);
        }
      }
    }
    RETURN (p, q);
  }                                  ⟨post: la coppia (p,q) dei punti più vicini⟩
```


Si osservi che ogni passo dell'algoritmo ha un costo al più lineare quindi mettendo insieme l'Equazione (3.10), il Teorema 3.1 e il Lemma 3.1 otteniamo il seguente risultato.

Teorema 3.5 *L'algoritmo è corretto e la sua complessità è* O(n log n).

3.8 Algoritmi ricorsivi su alberi binari

Gli alberi binari, essendo definiti in modo ricorsivo, permettono di progettare naturalmente algoritmi ricorsivi seguendo la metodologia del divide et impera: nel discuterne alcuni esempi, introdurremo anche della terminologia aggiuntiva che, sebbene fornita per semplicità con riferimento agli alberi binari, è in generale applicabile anche ad alberi di tipo diverso.

Un parametro che caratterizza un albero è la sua **dimensione** n, data dal numero di nodi in esso contenuti: chiaramente, un albero di dimensione n ha esattamente n - 1 archi (che collegano un qualunque nodo diverso dalla radice al padre), come possiamo notare nella figura dell'Esempio 1.10 che ha dimensione 16 e contiene 15 archi. Osserviamo che la dimensione di un albero binario può essere definita ricorsivamente nel modo seguente: un albero vuoto ha dimensione 0, mentre la dimensione di un albero non vuoto è pari alla somma delle dimensioni dei suoi sottoalberi, incrementata di 1, per includere la radice. Il Codice 3.11 utilizza tale osservazione per realizzare un algoritmo che determina la dimensione di un albero binario e che si basa sul seguente schema di divide et impera.

Codice 3.11 Algoritmo ricorsivo per il calcolo della dimensione di un albero binario.

```
Dimensione( u ):
  IF (u == null) {
    RETURN 0;
  } ELSE {
    dimensioneSX = Dimensione( u.sx );
    dimensioneDX = Dimensione( u.dx );
    RETURN dimensioneSX + dimensioneDX + 1;
  }
```

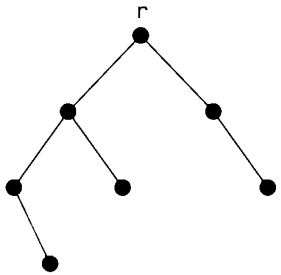
Decomposizione: se l'albero è vuoto, restituisci il valore `0` (riga 3), altrimenti suddividilo nei due sottoalberi radicati nei figli.

Ricorsione: calcola ricorsivamente la dimensione di ciascun sottoalbero (righe 5-6).

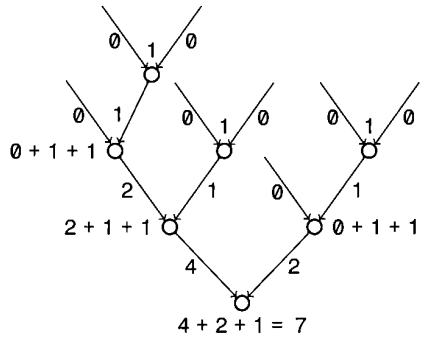
Ricombinazione: restituisci come risultato la somma delle dimensioni dei due sottoalberi incrementata di 1 (riga 7).

ESEMPIO 3.5

Eseguiamo l'algoritmo `Dimensione` sull'albero T nella figura avente radice r.



Nella figura che segue è rappresentato l'albero R delle invocazioni ricorsive di `Dimensione(r)`: i nodi di T e R sono in corrispondenza uno-a-uno, per comodità conviene usare gli stessi nomi per i nodi che sono in corrispondenza nei due alberi; ogni nodo u di R rappresenta la chiamata alla funzione `Dimensione(u)`. C'è un arco diretto tra u e v in R se `Dimensione(v)` ha invocato `Dimensione(u)`; l'etichetta sull'arco indica il valore restituito dalla funzione chiamata a quella chiamante. Infine l'etichetta di ogni nodo u in R rappresenta l'operazione di ricombinazione (riga 7) in cui viene calcolata la dimensione del sottoalbero di T con radice u sommando le dimensioni dei suoi due sottoalberi più uno.



La dimensione dell'albero T si legge nell'etichetta della radice dell'albero R.

Teorema 3.6 *La dimensione* n *di un albero binario può essere calcolata in tempo* O(n).

Dimostrazione Per dimostrare il Teorema 3.6, osserviamo che il problema della dimensione rientra nella famiglia dei problemi su alberi che possono essere risolti con uno schema di tipo divide et impera e che sono chiamati *problemi decomponibili*. La loro computazione ricalca il Codice 3.12, dove `Decomponibile(u)` rappresenta il valore da calcolare relativamente al sottoalbero radicato nel nodo u (che può essere una foglia oppure `null` nel caso base) e `Ricombina` permette di ricombinare i valori calcolati per i figli di u. Per esempio, nel calcolo della dimensione nel Codice 3.11, la funzione `Decomponibile(u)` rappresenta la dimensione del sottoalbero radicato in u e `Ricombina` rappresenta la somma incrementata di 1.

Codice 3.12 Algoritmo di tipo divide et impera per risolvere un problema decomponibile su alberi binari.

```
1  Decomponibile(u):
2    IF (u == null) {
3      RETURN Decomponibile(null);
4    } ELSE {
5      risultatoSX = Decomponibile(u.sx);
6      risultatoDx = Decomponibile(u.dx);
7      RETURN Ricombina(risultatoSX, risultatoDx);
8    }
```

Il vantaggio di avere uno schema di tipo divide et impera per gli alberi è che ci consente di scrivere la relazione di ricorrenza per il costo `T(n)`. Per semplicità, come accade in questo libro, ipotizziamo che il costo di divisione e ricombinazione per ogni nodo dell'albero sia limitato da una costante c. Preso un nodo u e il

suo sottoalbero con n nodi (incluso u), ipotizziamo di avere r - 1 nodi che discendono dal figlio sinistro di u e, quindi, n - r che discendono da quello destro, dove $1 \le r \le n$ e c_0 e c sono costanti positive:

$$T(n) \le \begin{cases} c_0 & \text{se } n \le 1 \\ T(r-1) + T(n-r) + c & \text{altrimenti} \end{cases} \quad (3.11)$$

Teorema 3.7 *La relazione nella* (3.11) *ha soluzione* $T(n) = O(n)$.

Dimostrazione Non potendo applicare il teorema fondamentale, osserviamo che vale $T(0) \le c_0 \le c'$ e dimostriamo per induzione che $T(n) \le 3c'n$ per ogni $n \ge 1$, dove $c' = \max\{c_0, c\}$. Se $n = 1$ (ovvero l'albero contiene un solo nodo), abbiamo che $r = 1$ e, quindi, $T(1) \le 2T(0) + c \le 2c_0 + c \le 3c' = 3c'n$. Supponiamo che l'affermazione sia vera per $1 \le n' < n$. Allora, $T(n) \le T(r-1) + T(n-r) + c$ e, se $1 < r < n$, per ipotesi induttiva abbiamo che $T(n) \le 3c'(r-1) + 3c'(n-r) + c \le 3c'n - 2c < 3c'n$. Se invece $r = 1$, utilizziamo il fatto che $T(0) \le c'$ e applichiamo l'ipotesi induttiva su $T(n-1) \le 3c'(n-1)$, ottenendo $T(n) \le c' + 3c'(n-1) + c \le 3c'n - c < 3c'n$. Lo stesso ragionamento vale se $r = n$. In conclusione, $T(n) = O(n)$ e il teorema risulta essere dimostrato. ❑

Esercizio svolto 3.4 Ricordiamo che l'altezza di un albero misura la massima distanza di una foglia dalla radice dell'albero, in termini del numero di archi attraversati. Progettare un algoritmo ricorsivo che calcoli l'altezza di un albero binario in tempo $O(n)$, dove n denota la dimensione dell'albero.

Soluzione Osserviamo che l'albero composto da un solo nodo ha altezza pari a 0, mentre un albero con almeno due nodi ha altezza pari all'altezza del suo sottoalbero più alto, incrementata di 1 in quanto la radice introduce un ulteriore livello (da cui deriviamo che l'albero vuoto ha altezza pari a -1). Il seguente codice utilizza tale osservazione per realizzare un algoritmo che determina l'altezza di un albero.

```
Altezza( u ):
  IF (u == null) {
    RETURN -1;
  } ELSE {
    altezzaSX = Altezza( u.sx );
    altezzaDX = Altezza( u.dx );
    RETURN max( altezzaSX, altezzaDX ) + 1;
  }                          ⟨post: restituisce -1 se e solo se u è null⟩
```

Come si può notare, abbiamo usato l'accorgimento di considerare come caso base l'albero vuoto (a cui abbiamo assegnato un altezza pari a -1): in tal modo, il codice segue lo stesso schema del Codice 3.11, l'altezza calcolata per le foglie risulta correttamente pari a 0 (in quanto sottoalberi composti da un solo nodo) e, per induzione, è corretta anche l'altezza calcolata per tutti i sottoalberi.

Nello specifico, il codice precedente opera nel modo seguente: se l'albero è vuoto, la sua altezza è pari a -1. Se non lo è, le due chiamate ricorsive calcolano l'altezza dei sottoalberi radicati nei figli: di tali altezze viene restituita come risultato la massima incrementata di 1. L'analisi di complessità del codice ricalca quella del Codice 3.11 mostrata nella dimostrazione del Teorema 3.6.

Rimarchiamo che sia il Codice 3.11 che il codice dell'esercizio precedente hanno un caso base (albero vuoto) e un passo induttivo (albero non vuoto) in cui avvengono le chiamate ricorsive. A parte le differenze sintattiche dovute al fatto che i due codici calcolano quantità differenti, la struttura computazionale è quella dei problemi decomponibili (Codice 3.12): ciascuna invocazione restituisce un valore (la dimensione o l'altezza), che possiamo facilmente dedurre nel caso base di un albero vuoto. Nel passo induttivo, deleghiamo il calcolo delle rispettive quantità alla ricorsione sui due figli (sottoalberi): una successiva fase di combinazione di tali quantità, restituite dalle chiamate ricorsive sui figli, contribuisce a ottenere il risultato per il nodo corrente. Tale risultato va a sua volta restituito mediante l'istruzione RETURN, per far sì che l'induzione si propaghi attraverso la ricorsione: infatti, chi invoca le chiamate ricorsive deve a sua volta trasmettere il risultato così ottenuto. Notiamo che, in base a tale approccio, ogni nodo viene attraversato un numero costante di volte, per cui se il caso base e la regola di ricombinazione richiedono tempo costante, l'esecuzione richiede un tempo totale $O(n)$.

3.8.1 Visite di alberi

Lo schema ricorsivo del paradigma del divide et impera applicato ad alberi binari, permette anche di effettuare una **visita** di un albero binario a partire dalla sua radice. La visita equivale a esaminare tutti i nodi in modo sistematico, una e una sola volta, analogamente alla scansione di sequenze lineari, dove procediamo dall'inizio alla fine o viceversa. Per semplicità, durante la visita facciamo corrispondere l'esame di un nodo all'operazione di stampa del suo contenuto. Tale visita permette di operare varie scelte che dipendono dall'ordine in cui viene esaminato l'elemento memorizzato nel nodo corrente e vengono invocate le chiamate ricorsive nei suoi figli.

Visita anticipata (*preorder*): *stampa l'elemento contenuto nel nodo*; visita ricorsivamente il sottoalbero sinistro; visita ricorsivamente il sottoalbero destro.

Visita simmetrica (*inorder*): visita ricorsivamente il sottoalbero sinistro; *stampa l'elemento contenuto nel nodo*; visita ricorsivamente il sottoalbero destro.

Visita posticipata (*postorder*): visita ricorsivamente il sottoalbero sinistro; visita ricorsivamente il sottoalbero destro; *stampa l'elemento contenuto nel nodo.*

In modo analogo a quanto fatto nella dimostrazione del Teorema 3.6, possiamo dimostrare che il costo di ciascuna delle tre visite è $O(n)$ per un albero di di-

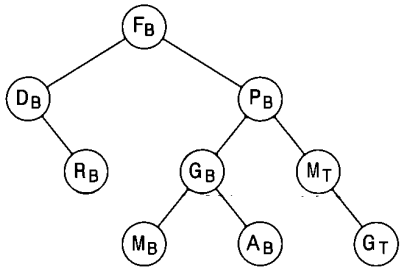
mensione n (cambia soltanto l'ordine in cui l'elemento nel nodo corrente viene stampato). Il codice per tali visite è una semplice variazione del Codice 3.11: per esempio, il Codice 3.13 realizza la visita anticipata. Osserviamo che esso non restituisce alcun valore in questa forma e che può essere trasformato nel codice di una visita simmetrica o posticipata molto semplicemente, spostando l'istruzione di stampa (riga 3).

Codice 3.13 Visita anticipata di un albero binario. Le altre due visite, simmetrica e posticipata, sono ottenute spostando l'istruzione di stampa dalla riga 3 in una delle due righe successive.

```
1  Anticipata( u ):
2    IF (u != null) {
3      print u.dato;
4      Anticipata( u.sx );
5      Anticipata( u.dx );
6    }
```

ESEMPIO 3.6

Per apprezzare la differenza delle tre visite, consideriamo l'esempio mostrato nella parte sinistra della seguente figura.



anticipata:
F_B D_B R_B P_B G_B M_B A_B M_T G_T

simmetrica:
D_B R_B F_B M_B G_B A_B P_B M_T G_T

posticipata:
R_B D_B M_B A_B G_B G_T M_T P_B F_B

ampiezza:
F_B D_B P_B R_B G_B M_T M_B A_B G_T

Nella parte destra della figura, oltre alle tre visite suddette viene illustrato anche il risultato di una quarta visita che illustreremo più avanti.

3.8.2 Alberi completamente bilanciati

Tornando allo schema del Codice 3.11, possiamo notare che esso rappresenta un modo di effettuare una visita posticipata in cui viene raccolta l'informazione necessaria alla computazione di `Dimensione(u)`, a partire dal basso verso l'alto. Per risolvere alcuni problemi su alberi binari, è necessario raccogliere più informazione di quanta ne serva apparentemente: studiamo, per esempio, il caso degli alberi completamente bilanciati.

A tale scopo, ricordiamo che un albero binario è completo se ogni nodo interno ha esattamente due figli non vuoti. L'albero è **completamente bilanciato** se, oltre a essere completo, tutte le foglie hanno la stessa profondità. Un albero completamente bilanciato di altezza h ha quindi $2^h - 1$ nodi interni e 2^h foglie: ne deriva che la relazione tra altezza h e numero di nodi $n = 2^{h+1} - 1$ è $h = \log(n+1) - 1$. Possiamo introdurre la definizione di albero binario **bilanciato**: per un tale albero vale la relazione $h = O(\log n)$, che risulta essere interessante per la complessità delle operazioni fornite da diverse strutture di dati. Notiamo che un albero completamente bilanciato è bilanciato, mentre il viceversa non sempre vale.

Volendo usare lo schema del Codice 3.11 per stabilire se un albero binario è completamente bilanciato, possiamo valutare cosa succede ipotizzando che il valore restituito sia un valore booleano, che risulta `TRUE` se e solo se `T(u)` è completamente bilanciato, dove `T(u)` indica l'albero radicato in u. Indicati come al solito con u_S e con u_D i due figli di u, il fatto che `T(`u_S`)` e `T(`u_D`)` siano completamente bilanciati, non comporta purtroppo che anche `T(u)` lo sia, in quanto i due sottoalberi potrebbero avere altezze diverse: in altre parole, `T(u)` è completamente bilanciato se e solo se `T(`u_S`)` e `T(`u_D`)`, oltre a essere completamente bilanciati, hanno anche la *stessa* altezza.

Nel Codice 3.14 richiediamo che il valore restituito sia una coppia di valori, in cui il primo è `TRUE` se e solo se `T(u)` è completamente bilanciato, mentre il secondo è l'altezza di `T(u)` (calcolata come nel codice dell'Esercizio 3.4). La regola di ricombinazione diventa quindi quella riportata di seguito.

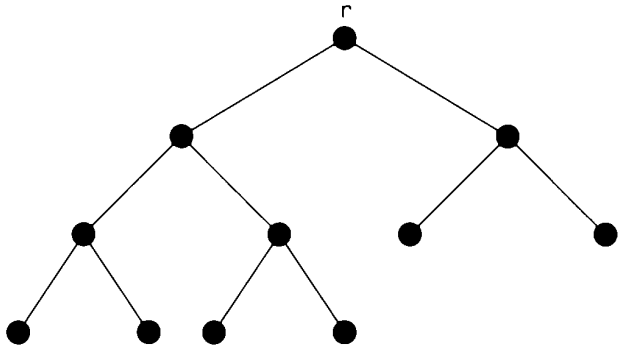
- La prima componente di `CompletamenteBilanciato(u)` è `TRUE` se e solo se lo sono le due prime componenti di `CompletamenteBilanciato(`u_S`)` e di `CompletamenteBilanciato(`u_D`)` sono entrambe `TRUE` e se le due seconde componenti sono uguali (riga 7).
- La seconda componente di `CompletamenteBilanciato(u)` è uguale al massimo tra le due seconde componenti di `CompletamenteBilanciato(`u_S`)` e di `CompletamenteBilanciato(`u_D`)` incrementate di 1 (riga 8).

Codice 3.14 Algoritmo ricorsivo per stabilire se un albero binario è completamente bilanciato.

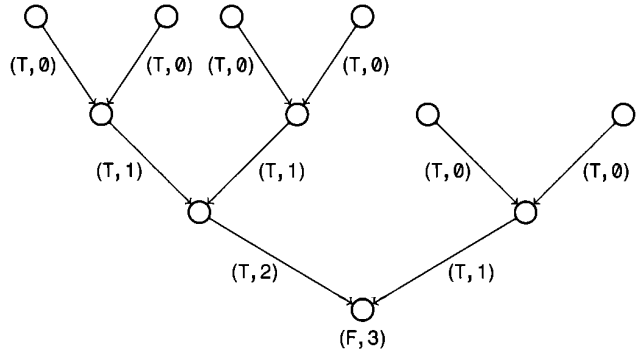
```
1  CompletamenteBilanciato( u ):
2    IF (u == null) {
3      RETURN <TRUE, -1>;
4    } ELSE {
5      <bilSX,altSX> = CompletamenteBilanciato( u.sx );
6      <bilDX,altDX> = CompletamenteBilanciato( u.dx );
7      completamenteBil = bilSX && bilDX && (altSX == altDX);
8      altezza = max(altSX, altDX) + 1;
9      RETURN <completamenteBil,altezza>;
10   }    ⟨post: restituisce TRUE come prima componente ⇔ T(u) è completamente
          bilanciato⟩
```

ESEMPIO 3.7

Eseguiamo l'algoritmo `CompletamenteBilanciato` sull'albero T, avente radice r, nella figura.



Come per l'Esempio 3.5 rappresentiamo R, l'albero delle chiamate ricorsive dell'algoritmo `CompletamenteBilanciato`. Anche in questo caso l'arco diretto tra u e v indica che `CompletamenteBilanciato(u)` restituisce il proprio output a `CompletamenteBilanciato(v)`: l'output è indicato come etichetta dell'arco da una coppia (B, h), dove B ∈ {T, F}, dove T sta per TRUE e F per FALSE.



L'algoritmo restituisce FALSE in quanto i sottoalberi sinistro e destro di r, pur essendo bilanciati, hanno altezze diverse.

Codice 3.15 Algoritmo ricorsivo per individuare i nodi cardine in un albero binario. La chiamata iniziale ha come parametri la radice e la sua profondità pari a 0.

```
1  Cardine( u, p ):                                  ⟨pre: p è la profondità di u⟩
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      altezzaSX = Cardine( u.sx, p+1 );
6      altezzaDX = Cardine( u.dx, p+1 );
7      altezza = max( altezzaSX, altezzaDX ) + 1;
8      IF (p == altezza) print u.dato;
9      RETURN altezza;
10   }                                    ⟨post: stampa i nodi cardine di T(u)⟩
```

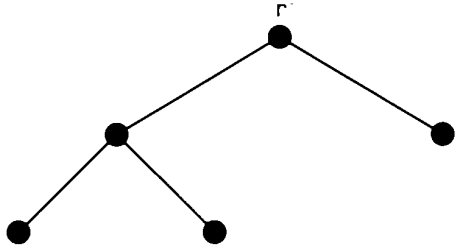

3.8.3 Nodi cardine di un albero binario

Per completare il quadro dello schema generale di sviluppo di algoritmi ricorsivi su alberi binari descritto in questo paragrafo, discutiamo un algoritmo in cui le chiamate non solo raccolgono informazione dai sottoalberi, ma propagano simultaneamente informazione proveniente dagli antenati, passando opportuni parametri alle chiamate. Un problema di questo tipo riguarda l'identificazione dei nodi cardine. Dato un nodo u, sia p_u la sua profondità e h_u l'altezza di `T(u)`. Diciamo che u è un nodo **cardine** se e solo se $p_u = h_u$. Vogliamo progettare un algoritmo ricorsivo che stampi il contenuto di tutti i nodi cardine presenti in un albero binario.

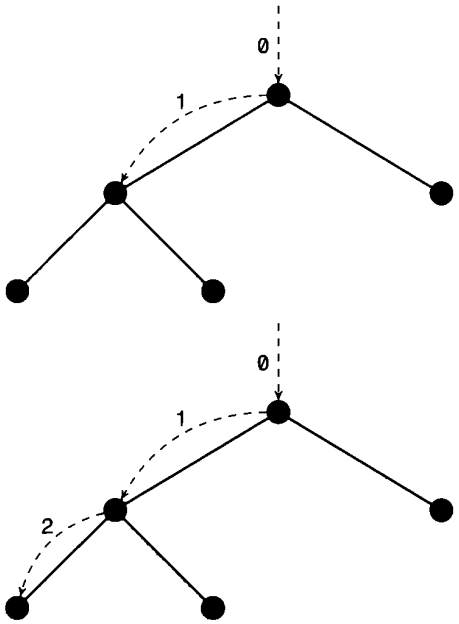
In questo caso, possiamo presumere che il valore restituito dalla chiamata ricorsiva sia h_u, analogamente a quanto fatto nel codice dell'Esercizio 3.4: tuttavia, al momento di invocare la chiamata ricorsiva su u dobbiamo garantire di passare p_u come parametro. Il Codice 3.15 ha quindi due parametri in ingresso per questo scopo: il primo indica il nodo corrente e il secondo la sua profondità. Inizialmente, questi parametri sono la radice r dell'albero e la sua profondità $p_r = 0$. Le successive chiamate ricorsive provvedono a passare i parametri richiesti (righe 5 e 6): ovvero, se il nodo corrente ha profondità p, i figli avranno profondità p + 1. La verifica che la profondità sia uguale all'altezza nella riga 8 stabilisce infine se il nodo corrente è un nodo cardine: in tal caso, la sua informazione viene stampata. Da notare che la complessità temporale dell'algoritmo rimane O(n) in quanto si tratta di una semplice variazione della visita posticipata implicitamente adottata nel Codice 3.11.

ESEMPIO 3.8

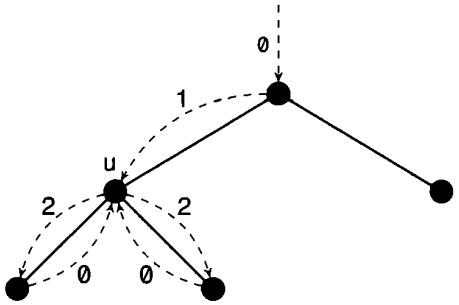
Eseguiamo l'algoritmo `Cardine` con input il nodo r radice dell'albero rappresentato nella figura sottostante.



Rappresentiamo l'invocazione di una chiamata ricorsiva su un nodo u con un arco tratteggiato orientato verso le foglie. L'etichetta dell'arco rappresenta la profondità del nodo u. Ovviamente la prima invocazione è sulla coppia (r, 0).



Arrivati alle foglie inizia il processo di ricostruzione delle altezze dei sottoalberi in quanto per le foglie questo valore è noto.



L'istanza della funzione eseguita con input (u, 1), dopo che le chiamate ricorsive sui nodi figli hanno restituito le altezze di questi (entrambi 0), calcola l'altezza di T(u) (riga 7) e poiché questa risulta uguale alla profondità di u ricevuta in input, stampa il dato contenuto nel nodo. Il procedimento continuerà per tutti gli altri nodi dell'albero.

3.9 Esercizi

3.1 Dato un array ordinato contenente n elementi di tipo intero, progettare un algoritmo che data una chiave k restituisca il numero di occorrenze di k nell'array. L'algoritmo deve avere complessità O(log n).

3.2 Modificare il Codice 3.4 in modo che, nel caso che l'array a memorizzi un multi-insieme ordinato dove gli elementi possono comparire più volte, restituisca la posizione più a *destra* dell'elemento cercato.

3.3 Un array V[0, n – 1] è detto *convesso semplice* se esiste un indice j tale che V[0] = V[1] = V[2] = ··· = V[j] e, per ciascun i > j, vale V[i] < V[i + 1]. Per esempio, l'array V = {3, 3, 3, 5, 8, 14, 15} è convesso semplice, mentre V′ = {3, 3, 3, 5, 8, 8, 14, 15} e V″ = {5, 8, 14, 5} non lo sono. Progettare un algoritmo che, preso in input un array convesso semplice V, trova l'indice j in tempo: (a) O(n) e (b) O(log n).

3.4 Siano date k liste di dimensione totale n, ciascuna contenente interi distinti ordinati in modo crescente. Progettare un algoritmo che esegue l'intersezione delle liste (ossia, stampa gli elementi *comuni* che appaiono in tutte le k liste) utilizzando spazio di appoggio O(k) e complessità in tempo al caso pessimo O(n log k), utilizzando uno heap e l'idea di fusione del mergesort.

3.5 Data la seguente funzione ricorsiva Foo, trovare la corrispondente relazione di ricorrenza e risolverla utilizzando il teorema principale.

```
Foo( n ){
  IF (n < 10) {
    RETURN 1;
  } ELSE IF (n < 1234){
    tmp = n;
    FOR (i = 1; i <= n; i = i+1)
      FOR (j = 1; j <= n; j = j+1)
        FOR (k = 1; k <= n; k = k+1)
          tmp = tmp + i * j * k;
  } ELSE {
    tmp = n;
    FOR (i = 1; i <= n; i = i+1)
      FOR (j = 1; j <= n; j = j+1)
        tmp = tmp + i * j;
  }
  RETURN tmp + Foo( n/2 ) * Foo( n/2 ) + Foo( n/2 );
}
```

3.6 Sia data la funzione ricorsiva:

```
Foo( n ) {
  a = 0;
  FOR( i = 0; i < n; i = i+1 )
    FOR( j = 0; j < n/2; j = j+1 )
      FOR( k = 0; k < 5; k = k+1 )
        a += 1;

  RETURN Foo( n/2 ) + a;
}
```

- Scrivere la relazione di ricorrenza per la complessità in tempo $T(n)$ della funzione Foo.
- Trovare la soluzione per $T(n)$ in forma chiusa.

3.7 Scrivere un algoritmo ricorsivo Foo(n) la cui complessità in tempo al caso pessimo sia modellata dalla seguente relazione di ricorrenza: $T(n) = O(1)$ per $n < 10$, e $T(n) = 3T(n/2) + \Theta(n^2)$ per $n \geq 10$. Infine si risolva anche la relazione suddetta.

3.8 Sia dato un array A di n interi distinti e positivi. Progettare un algoritmo ricorsivo basato sulla tecnica *divide et impera* che conti il numero di elementi che sono minimi relativi del vettore A (escludendo dal conteggio A[0] e A[n - 1]). Si definisce minimo relativo un elemento che è minore dei suoi adiacenti. Per esempio, A = [3, 4, 2, 8, 7, 10, 12, 15, 14, 20] ha tre minimi relativi che sono {2, 7, 14}. Si valuti inoltre la complessità in tempo e spazio al caso pessimo dell'algoritmo proposto.

3.9 L'elemento *mediano* di un insieme S di m elementi (distinti) è quell'elemento che ha esattamente $\lfloor m/2 \rfloor$ elementi minori in S. Per esempio, l'insieme S = {1, 4, 6, 8, 9, 12} ha mediano 8. Siano dati due insiemi A e B di n elementi ciascuno, rappresentati come sequenze ordinate memorizzate negli array $S_A[0, n-1]$ e $S_B[0, n-1]$.

(a) Progettare un algoritmo che trovi il *mediano* di $A \cup B$ in tempo $O(n)$ al caso pessimo.

(b) Progettare un algoritmo che, dato un elemento x di S_A o S_B, calcoli il numero di elementi di $A \cup B$ che sono minori di x in tempo $O(\log n)$.

(c) Utilizzare l'algoritmo del punto precedente per progettare un algoritmo ricorsivo che calcoli il mediano di $A \cup B$ in tempo $O(\log^2 n)$.

3.10 Sia dato un insieme $S = \{s_1, s_2, \ldots, s_n\}$ di n elementi distinti. Si definisce mediano di S, indicato con $\hat{S}$, l'elemento dell'insieme che ha rango $r = \lfloor n/2 \rfloor$ e quindi è l'r-esimo elemento più piccolo di S.

(a) Progettare un algoritmo deterministico che calcola $\hat{S}$ e se ne valuti la complessità in tempo.

(b) Progettare un algoritmo che, ricevuto in ingresso un intero $k < n$, un generico insieme S di n interi e il suo mediano $\hat{S}$, calcoli i k elementi di S più vicini a $\hat{S}$. Se ne valuti la complessità in tempo. (*Suggerimento*: esiste una soluzione in tempo $O(n \log k)$ utilizzando uno heap che contiene i k elementi che sono più vicini a $\hat{S}$ tra quelli esaminati fino a quel momento.)

(c) Progettare un algoritmo deterministico che calcoli $\hat{S}$ in tempo $O(n)$ al caso pessimo nell'ipotesi che gli elementi di S siano interi di valore minore di n^2.

3.11 Nel Paragrafo 3.4 viene menzionato il fatto che il *quicksort* richiede tempo $O(n \log n)$ al caso medio. Estendere il limite inferiore di $\Omega(n \log n)$ confronti per l'ordinamento presentato per il caso pessimo al caso in cui si considera il tempo medio di esecuzione.

3.12 Il Codice 3.7 assume che gli elementi in ingresso siano distinti tra loro. Nel caso di elementi ripetuti in un multi-insieme, estendere la nozione di rango per determinare un unico elemento: per esempio, possiamo individuarlo come l'elemento che occupa la posizione r - 1 nell'array dopo averlo ordinato in modo *stabile* (ossia elementi uguali mantengono la loro posizione relativa). Estendere il Codice 3.7 in modo che possa funzionare anche con un multi-insieme.

3.13 Dato un albero binario, scrivere un algoritmo ricorsivo che richieda tempo lineare nel numero di nodi per restituire il puntatore al nodo u tale che il rapporto tra il numero di nodi discendenti da u (incluso) e l'altezza di u sia massimizzato (in caso di più alberi con tale caratteristica, restituirne uno qualunque). Motivare la correttezza e la complessità dell'algoritmo proposto.

3.14 Sia dato un albero binario T di radice r, in cui ciascun nodo u memorizza un numero intero nel campo dato. Si progetti un algoritmo ricorsivo che stampi i dati contenuti nei nodi u che soddisfano la condizione: la somma dei dati contenuti negli antenati di u (incluso) è uguale alla somma dei dati contenuti nei discendenti di u (incluso). Si determini inoltre la complessità in tempo al caso pessimo dell'algoritmo proposto.

3.15 Dati due nodi u e v in un albero binario, definiamo la loro distanza D(u, v) come il minimo numero di archi (dell'albero) che è necessario attraversare per raggiungere u da v o viceversa. Dato un albero binario T e un intero positivo d, progettare un algoritmo che calcoli quante coppie di nodi u e v soddisfano entrambe le condizioni:

- la loro distanza è D(u, v) = d;
- u è antenato di v o viceversa.

È possibile usare strutture di dati di appoggio e puntatori al padre (anche se questo non è strettamente necessario per risolvere il problema). Analizzare la complessità dell'algoritmo proposto in termini del numero `n` di nodi presenti nell'albero.

3.16 Sia dato un albero binario `T` di radice `r` e si definisca la dimensione di un nodo `u` come il numero dei suoi nodi discendenti, incluso `u` stesso (dove, per convenzione, `u = null` ha dimensione pari a zero). Diciamo che `u` è *pesante* se `u` è la radice `r` oppure soddisfa una delle seguenti due condizioni: (a) la dimensione di `u` è maggiore di quella di suo fratello (l'altro figlio di suo padre); (b) la dimensione è uguale a quella di suo fratello e `u` è figlio sinistro.

(a) Dimostrare che esiste sempre un unico cammino dalla radice `r` a una foglia `f` in cui *tutti* i nodi attraversati da tale cammino sono pesanti.

(b) Scrivere un algoritmo ricorsivo che non usi variabili globali e che, presa in ingresso la radice `r`, restituisca la foglia `f` del cammino suddetto. Ipotizzare che i nodi di `T` contengano solo i puntatori al figlio sinistro, al figlio destro e nessun altro tipo di informazione.

(c) Discutere e motivare la complessità dell'algoritmo proposto.

CAPITOLO

4 Dizionari

In questo capitolo descriviamo la struttura di dati denominata dizionario e le operazioni da essa fornite. Mostriamo quindi come realizzare i dizionari utilizzando le liste doppie, le tabelle hash, gli alberi di ricerca, gli alberi AVL o, infine, i trie o alberi digitali di ricerca e le liste invertite.

4.1 Dizionari
4.2 Liste e dizionari
4.3 Opus libri: funzioni hash e peer-to-peer
4.4 Opus libri: kernel Linux e alberi binari di ricerca
4.5 Opus libri: liste invertite e trie
4.6 Esercizi

4.1 Dizionari

Un dizionario memorizza una collezione di elementi e ne fornisce le operazioni di ricerca, inserimento e cancellazione. I dizionari trovano impiego in moltissime applicazioni: insieme agli algoritmi di ordinamento, costituiscono le componenti fondamentali per la progettazione di algoritmi efficienti. Ai fini della discussione, ipotizziamo che ciascuno degli elementi `e` contenga una **chiave di ricerca**, indicata con `e.chiave`, e che le restanti informazioni in `e` siano considerate dei **dati satellite**, indicati con `e.sat`: come al solito, indicheremo l'elemento vuoto con `null`. Definito il dominio o **universo** `U` delle chiavi di ricerca contenute negli elementi, un **dizionario** memorizza un insieme $S = \{e_0, e_1, \ldots, e_{n-1}\}$ di elementi, dove `n` è la **dimensione** di `S`, e fornisce le seguenti operazioni per una qualunque chiave $k \in U$:

- `Ricerca(k)`: restituisce l'elemento `e` se `k = e.chiave`, oppure il valore `null` se nessun elemento in `S` ha `k` come chiave;
- `Inserisci(e)`: estende l'insieme degli elementi ponendo $S = S \cup \{e\}$, con l'ipotesi che `e.chiave` sia una chiave distinta da quelle degli altri elementi in `S` (se `e` appartiene già a `S`, l'insieme non cambia);
- `Cancella(k)`: elimina dall'insieme l'elemento `e` tale che `k = e.chiave` e pone $S = S - \{e\}$ (se nessun elemento di `S` ha chiave `k`, l'insieme non cambia).

Poiché in diverse applicazioni il campo satellite `e.sat` degli elementi è sempre vuoto e il dizionario memorizza soltanto l'insieme delle chiavi di ricerca distinte, l'operazione di ricerca diventa semplicemente quella di appartenenza all'insieme:

- `Appartiene(k)`: restituisce `TRUE` se e solo se `k` appartiene all'insieme `S`, ovvero `Ricerca(k)` $\neq$ `null`.

Il dizionario è detto **statico** se fornisce soltanto l'operazione di ricerca (`Ricerca`) e viene detto **dinamico** se fornisce anche le operazioni di inserimento (`Inserisci`) e di cancellazione (`Cancella`). Quando una relazione di ordine totale è definita sulle chiavi dell'universo `U` (ovvero, per ogni coppia di chiavi distinte `k` e $k' \in U$ si verifica $k < k'$ o $k' < k$), il dizionario è detto **ordinato**: con un piccolo abuso di notazione, estendiamo gli operatori di confronto tra chiavi di ricerca ai rispettivi elementi, per cui possiamo scrivere che gli elementi di `S` soddisfano la relazione $e_0 < e_1 < \cdots < e_{n-1}$ (intendendo che le loro chiavi la soddisfano) e che, per esempio, vale $k \leq e_i$ (intendendo $k \leq e_i$`.chiave`). Il dizionario ordinato per `S` fornisce le seguenti ulteriori operazioni:

- `Successore(k)`: restituisce l'elemento e_i tale che `i` è il minimo intero per cui $k \leq e_i$ e $0 \leq i < n$, oppure il valore `null` se `k` è maggiore di tutte le chiavi in `S`;
- `Predecessore(k)`: restituisce l'elemento e_i tale che `i` è il massimo intero per cui $e_i \leq k$ e $0 \leq i < n$, oppure `null` se `k` è minore di tutte le chiavi in `S`;

- `Intervallo(k, k')`: restituisce tutti gli elementi $e \in S$ tali che $k \leq e \leq k'$, dove supponiamo $k \leq k'$, oppure `null` se tali elementi non esistono in `S`;
- `Rango(k)`: restituisce l'intero `r` che rappresenta il numero di chiavi in `S` che sono minori oppure uguali a `k`, dove $0 \leq r \leq n$.

Da notare che le prime due operazioni restituiscono lo stesso valore di `Ricerca(k)` quando esiste $e \in S$ tale che `k = e.chiave`, mentre la terza lo ottiene come caso speciale quando $k = k'$. Infine, la quarta operazione può simulare le prime tre se, dato un rango `r`, possiamo accedere all'elemento $e_r \in S$ in modo efficiente.

Quando le operazioni suddette si riferiscono sempre all'insieme `S` di default, adottiamo la convenzione di non specificare ogni volta `S` con la chiamata dell'operazione. Se invece `S` non è l'insieme di default quando utilizziamo una certa operazione, per esempio `Ricerca(k)`, dobbiamo riferirci a `S` esplicitamente: a tal fine adottiamo la notazione `S.Ricerca(k)` per indicare ciò (come accadrà nel Codice 4.3).

4.2 Liste e dizionari

Le liste doppie (Paragrafo 1.3.2) sono un ottimo punto di partenza per l'implementazione efficiente dei dizionari. Nel seguito presumiamo che una lista doppia `L` abbia tre campi, ovvero due riferimenti `L.inizio` e `L.fine` all'inizio e alla fine della lista e, inoltre, un intero `L.lunghezza` contenente il numero di nodi nella lista. Utilizziamo una funzione `NuovaLista` per inizializzare i primi due campi a `null` e il terzo a `0`.

Ricordiamo che ogni nodo `p` della lista `L` è composto da tre campi, `p.pred`, `p.succ` e `p.dato`, e faremo uso della funzione `NuovoNodo` per creare un nuovo nodo quando sia necessario farlo. Il campo `p.dato` contiene un elemento $e \in S$: quindi possiamo indicare con `p.dato.chiave` e `p.dato.sat` i campi dell'elemento memorizzato nel nodo corrente.

L'uso diretto delle liste doppie per implementare i dizionari non è consigliabile per insiemi di grandi dimensioni, in quanto le operazioni richiederebbero $O(n)$ tempo: le liste sono però una componente fondamentale di molte strutture di dati efficienti per i dizionari, per cui riportiamo il codice delle funzioni definite su di esse che saranno utilizzate nel seguito. In particolare, nel Codice 4.1, il ruolo dell'operazione `Inserisci` del dizionario è svolto dalle due operazioni di inserimento in cima e in fondo alla lista, per poterne sfruttare le potenzialità nei dizionari che discuteremo più avanti. Per lo stesso motivo, nel Codice 4.2, l'operazione `Ricerca` restituisce il puntatore `p` al nodo contenente l'elemento trovato (basta prendere `p.dato` per soddisfare le specifiche dei dizionari date nel Paragrafo 5.1).

Osserviamo che le operazioni suddette implementano la gestione delle liste doppie discussa nel Paragrafo 1.3.2. Quindi, la complessità delle operazioni di

inserimento riportate nel Codice 4.1 è costante indipendentemente dalla lunghezza della lista, mentre le operazioni di ricerca e cancellazione riportate nel Codice 4.2 richiedono tempo O(n) dove n è la lunghezza della lista (anche se la cancellazione effettiva richiede O(1) avendo il riferimento al nodo, non utilizzeremo mai questa possibilità).

Codice 4.1 Inserimento in cima e in fondo a una lista doppia, componente di un dizionario.

```
1  InserisciCima( e ):
                    ⟨pre: e non in lista⟩
2    p = NuovoNodo( );
3    p.dato = e;
4    lun = lista.lunghezza;
5    IF (lun == 0) {
6      p.succ = p.pred = null;
7      lista.inizio = p;
8      lista.fine = p;
9    } ELSE {
10     p.succ = lista.inizio;
11     p.pred = null;
12     lista.inizio.pred = p;
13     lista.inizio = p;
14   }
15   lista.lunghezza = lun + 1;
16   RETURN lista;
```

```
1  InserisciFondo( e ):
                    ⟨pre: e non in lista⟩
2    p = NuovoNodo( );
3    p.dato = e;
4    lun = lista.lunghezza;
5    IF (lun == 0) {
6      p.succ = p.pred = null;
7      lista.inizio = p;
8      lista.fine = p;
9    } ELSE {
10     p.succ = null;
11     p.pred = lista.fine;
12     lista.fine.succ = p;
13     lista.fine = p;
14   }
15   lista.lunghezza = lun + 1;
16   RETURN lista;
```

Codice 4.2 Ricerca e cancellazione in una lista doppia, componente di un dizionario.

```
1  Ricerca( k ):
2    p = lista.inizio;
3    WHILE ((p != null) && (p.dato.chiave != k))
4      p = p.succ;
5    RETURN p;
```

```
1  Cancella( k ):
2    p = Ricerca( k );
3    IF (p != null) {
4      IF (lista.lunghezza == 1) {
5        lista.inizio = lista.fine = null;
6      } ELSE IF (p.pred == null) {
7        p.succ.pred = null;
8        lista.inizio = p.succ;
9      } ELSE IF (p.succ == null) {
10       p.pred.succ = null;
11       lista.fine = p.pred;
12     } ELSE {
13       p.succ.pred = p.pred;
14       p.pred.succ = p.succ;
15     }
16     lista.lunghezza = lista.lunghezza - 1;
17   }
18   RETURN lista;
```


Nel seguito descriviamo altri dizionari di uso comune in applicazioni informatiche: notiamo che le operazioni che gestiscono tali dizionari possono essere estese per permettere la memorizzazione di chiavi multiple, ossia la gestione di un multi-insieme di chiavi.

4.3 Opus libri: funzioni hash e peer-to-peer

Il termine inglese **hash** indica un polpettone ottenuto tritando della carne insieme a della verdura, dando luogo a un composto di volume ridotto i cui ingredienti iniziali sono mescolati e amalgamati. Tale descrizione ben illustra quanto succede nelle funzioni $\texttt{Hash}: U \to [0, m-1]$, aventi l'universo U delle chiavi come dominio e l'intervallo di interi da 0 a $m-1$ come codominio (di solito, m è molto piccolo se confrontato con la dimensione di U): una funzione $\texttt{Hash}(k) = h$ trita la chiave $k \in U$ restituendo come risultato un intero $0 \le h \le m-1$. Notiamo che tale funzione non necessariamente preserva l'ordine delle chiavi appartenenti all'universo U.

Alcune funzioni hash sono semplici e utilizzano la codifica binaria delle chiavi (per cui, nel seguito, identifichiamo una chiave k con la sua codifica in binario):

- $\texttt{Hash}(k) = k \,\%\, m$ calcola il modulo di k utilizzando un numero primo m;
- $\texttt{Hash}(k) = k_0 \oplus k_1 \oplus \cdots \oplus k_{s-1}$ spezza la codifica binaria di k nei blocchi k_0, k_1, ..., k_{s-1} di pari lunghezza, dove $0 \le k_i \le m-1$ e l'operazione $\oplus$ indica l'OR esclusivo.[1]

La seconda funzione hash è chiamata **iterativa** in quanto divide la chiave k in blocchi di sequenze binarie k_0, k_1, ..., k_{s-1} e lavora su tali blocchi, di fatto ripiegando la chiave su se stessa (i blocchi hanno la stessa lunghezza e vengono aggiunti dei bit in fondo alla chiave se necessario).

Altre funzioni hash sono più sofisticate, per esempio quelle nate in ambito crittografico come MD5 (*Message-Digest Algorithm* versione 5), inventata dal crittografo Ronald Rivest ideatore del metodo RSA, e SHA-1 (*Secure Hash Algorithm* versione 1), introdotta dalla *National Security Agency* del governo

[1] L'OR esclusivo tra due bit vale 1 se e solo se i due bit sono diversi (0 e 1, oppure 1 e 0): la notazione $a \oplus b = c$ indica l'OR esclusivo *bitwise*, in cui il bit i-esimo di c è l'OR esclusivo del bit i-esimo di a e b.

statunitense. Queste funzioni sono iterative e lavorano su blocchi di 512 bit applicandovi un'opportuna sequenza di operazioni logiche per manipolare i bit (per esempio, l'OR esclusivo o la rotazione dei bit) restituendo così un intero a 128 bit (MD5) o a 160 bit (SHA-1). Per esempio, la valutazione di MD5(`algoritmo`) con la chiave `algoritmo` restituisce la sequenza esadecimale[2]

```
446cead90f929e103816ff4eb92da6c2
```

mentre SHA-1(`algoritmo`) restituisce

```
6f77f39f5ea82a55df8aaf4f094e2ff0e26d2adb
```

La caratteristica di queste funzioni hash è che, cambiando anche leggermente la chiave, l'intero risultante è completamente diverso. Per esempio, MD5(`algoritmi`) restituisce

```
6a8af95d7f185b1a223c5b20cc71eb4a
```

mentre SHA-1(`algoritmi`) restituisce

```
147d401a6a1e3c20e7d6796bcac50a993726d4fa
```

Volendo ottenere un valore hash nell'intervallo `[0, m - 1]` (dove `m` è molto minore di 2^{128}), possiamo utilizzare tali funzioni nel modo seguente

- `Hash(k)` = MD5`(k) % m`
- `Hash(k)` = SHA-1`(k) % m`

anche se va osservato che tali funzioni crittografiche hanno delle ulteriori proprietà per cui il loro uso nei dizionari è forse eccessivo in diverse applicazioni e conviene usare funzioni più semplici da calcolare, come il modulo.

Notiamo che, essendo la dimensione dell'universo `U` molto vasta, esistono sempre due chiavi k_0 e k_1 in `U` tali che $k_0 \neq k_1$ e `Hash(`k_0`)` = `Hash(`k_1`)`: una tale situazione è chiamata **collisione**. Tuttavia, la natura deterministica del calcolo delle suddette funzioni hash, fa sì che se `Hash(`k_0`)` $\neq$ `Hash(`k_1`)` allora $k_0 \neq k_1$. Questa proprietà tipica delle funzioni in generale, coniugata con la robustezza di MD5 e SHA-1 in ambito crittografico (soprattutto la versione più recente SHA-2 che restituisce un intero a 512 bit), trova applicazione anche nei **sistemi distribuiti di condivisione dei file** (*peer-to-peer*).

In tali sistemi distribuiti (come BitTorrent, FreeNet, Gnutella, E-Mule, Napster e così via), l'informazione è condivisa e distribuita tra tutti i *client* o *peer* piuttosto che concentrata in pochi *server*, con un enorme vantaggio in termini di banda passante e tolleranza ai guasti della rete: la partecipazione è su base volontaria e, al momento di scaricare un determinato file, i suoi blocchi sono recuperati da vari punti della rete. Un numero sempre crescente di servizi operanti in ambiente

[2] La codifica esadecimale usa sedici cifre `0`, `1`, ..., `9`, `a`, `b`, ..., `f` per codificare le 2^4 possibili configurazioni di 4 bit.

distribuito usufruisce dei protocolli creati per tali sistemi (per esempio, la telefonia via Internet).

In tale scenario, lo stesso file può apparire con nomi diversi o file diversi possono apparire con lo stesso nome. Essendo la dimensione di ciascun file dell'ordine di svariati megabyte, quando i peer devono verificare quali file hanno in comune, è impensabile che questi si scambino direttamente il contenuto dei file: in tal modo, tutti i peer riceverebbero molti file da diversi altri peer e questo è impraticabile per la grande quantità di dati coinvolti.

Prendiamo per esempio il caso di due peer `P` e `P'` che vogliano capire quali file hanno in comune. Non potendo affidarsi ai nomi dei file devono verificarne il contenuto e l'uso delle funzioni hash in tale contesto è formidabile: per ogni file `f` memorizzato, `P` calcola il valore SHA-1`(f)`, chiamato impronta digitale (*digital fingerprint*), spedendolo a `P'` (solo 160 bit) al posto di `f` (svariati megabyte). Da parte sua, `P'` riceve tali impronte digitali da `P` e calcola quelle dei propri file. Dal confronto delle impronte può dedurre con certezza quali file sono diversi e, con altissima probabilità, quali file sono uguali.

Un altro uso dell'hash in tale scenario è quando `P` decide di scaricare un file `f`. Tale file è distribuito nella rete e, a tale scopo, è stato diviso in blocchi f_0, f_1, ..., f_{s-1}: oltre all'impronta `h` = SHA-1`(f)` dell'intero file, sono disponibili anche le impronte h_i = SHA-1`(`f_i`)` dei singoli blocchi (per `0` $\leq$ `i` $\leq$ `s - 1`). A questo punto, dopo aver recuperato le sole impronte digitali `h`, h_0, h_1, ..., h_{s-1} attraverso un'opportuna interrogazione, `P` lancia le richieste agli altri peer diffondendo tali impronte. Appena ha terminato di ricevere i rispettivi blocchi f_i in modo distribuito, `P` ricostruisce `f` da tali blocchi e verifica che le impronte digitali corrispondano: la probabilità di commettere un errore con tali funzioni hash è estremamente bassa.

Viste le loro proprietà, è naturale utilizzare le funzioni hash per realizzare un dizionario che memorizzi un insieme `S` = $\{e_0, e_1, \ldots, e_{n-1}\}$ di elementi. I dizionari basati sull'hash sono noti come **tabelle hash** (*hash map*) e sono utili per implementare una struttura di dati chiamata **array associativo**, i cui elementi sono indirizzati utilizzando le chiavi in `S` piuttosto che gli indici in `[0, n - 1]`.

La situazione ideale si presenta quando, fissando `m` = O`(n)`, la funzione `Hash` è *perfetta* su `S`, ovvero nessuna coppia di chiavi in `S` genera una collisione: in tal caso, il dizionario è realizzato mediante un semplice array binario `tabella` di `m` bit inizialmente uguali a `0`, in cui ne vengono posti `n` pari a 1 con la regola che `tabella[h] = 1` se e solo se `h = Hash(`e_i`.chiave)` per ciascun elemento e_i dove `0` $\leq$ `i` $\leq$ `n - 1`. Essendo una funzione perfetta, `Hash` non necessita di ulteriori controlli: tuttavia, inserendo o cancellando elementi in `S`, può accadere che `Hash` facilmente perda la proprietà di essere perfetta su `S`.

Da notare che esistono dizionari dinamici basati su famiglie di hash che richiedono tempo O`(1)` al caso pessimo per la ricerca e tempo medio O`(1)` ammortizzato per l'inserimento e la cancellazione.

Il limite di tempo O`(1)` per la ricerca non è in contrasto con quello di Ω`(log n)` confronti per la ricerca (Teorema 3.3): infatti, nel primo caso i bit della chiave `k`

vengono manipolati da una o più funzioni hash per ottenere un indice dell'array `tabella`, mentre nel secondo caso `k` viene soltanto confrontata dalla ricerca binaria con le altre chiavi. In altre parole, il limite di $\Omega(\log n)$ confronti vale supponendo che l'unica operazione permessa sulle chiavi sia il loro confronto diretto con altre (oltre alla loro memorizzazione), mentre tale limite non vale se i bit delle chiavi possono essere usati per calcolare una funzione diversa dal confronto di chiavi.

La costruzione dei suddetti dizionari dinamici basati sull'hash perfetto è piuttosto macchinosa e le loro prestazioni in pratica non sono sempre migliori dei dizionari che fanno uso di funzioni hash non necessariamente perfette. Questi ultimi, pur richiedendo per la ricerca tempo $O(1)$ in media invece che al caso pessimo, sono ampiamente diffusi per la loro efficienza in pratica e per la semplicità della loro gestione, che si riduce a *risolvere* le collisioni prodotte dalle chiavi. Nel seguito descriveremo due semplici modi per fare ciò: mediante liste di trabocco (che oltretutto garantiscono tempo $O(1)$ al caso pessimo per inserire una chiave non presente in `S`) oppure con l'indirizzamento aperto.

4.3.1 Tabelle hash: liste di trabocco

Nelle tabelle hash con liste di trabocco (*chaining*), `tabella` è un array di `m` liste doppie, gestite secondo quanto descritto nel Paragrafo 4.2, e `tabella[h]` contiene le chiavi `e` dell'insieme `S` tali che `h = Hash(e.chiave)`: in altre parole, le chiavi che collidono fornendo lo stesso valore `h` di `Hash` sono memorizzate nella medesima lista, etichettata con `h` (ovviamente tale lista è vuota se nessuna chiave dà luogo a un valore hash `h`).

L'operazione di ricerca scandisce la lista associata al valore hash della chiave, mentre l'operazione di inserimento, dopo aver verificato che la chiave dell'elemento non appare nella lista, inserisce un nuovo nodo con tale elemento in fondo alla lista. La cancellazione verifica che la chiave sia presente e rimuove il corrispondente elemento, e quindi il nodo, dalla lista doppia corrispondente.

Pur avendo un caso pessimo di tempo $O(n)$ (tutte le chiavi danno luogo allo stesso valore hash), ciascuna operazione è molto veloce in pratica se la funzione hash scelta è buona: ovvero se la funzione `Hash` distribuisce *in modo uniformemente casuale* gli `n` elementi di `S` nelle `m` liste di trabocco. In questo modo, la lunghezza media di una qualunque delle liste è $O(n/m)$, dove $n/m = \alpha$ è chiamato **fattore di carico**: quindi, le operazioni sulla tabella richiedono in media un tempo costante $O(1 + \alpha)$ perché il loro costo è proporzionale alla lunghezza della lista acceduta. Mantenendo l'invariante che `m` sia circa il doppio di `n` (Paragrafo 1.1.3), abbiamo che $\alpha = O(1)$, pertanto vale il seguente risultato.

Teorema 4.1 *Il costo medio delle operazioni sulle tabelle hash a liste di trabocco è costante se* $\alpha = O(1)$.

Codice 4.3 Dizionario realizzato mediante tabelle hash con liste di trabocco.

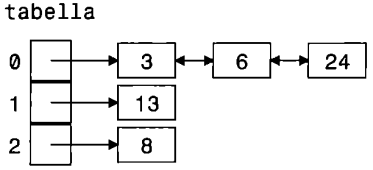
```
1  Ricerca( k ):
2    h = Hash(k);
3    p = tabella[h].Ricerca( k );
4    IF (p != null) RETURN p.dato ELSE RETURN null;
```

```
1  Inserisci( e ):
2    IF (Ricerca( e.chiave ) == null) {
3      h = Hash( e.chiave );
4      tabella[h].InserisciFondo( e );
5    }
```

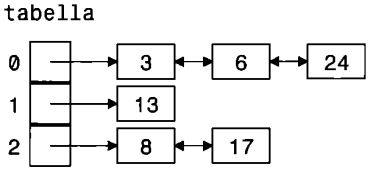
```
1  Cancella( k ):
2    IF (Ricerca( k ) != null) {
3      h = Hash(k);
4      tabella[h].Cancella( k );
5    }
```

ESEMPIO 4.1

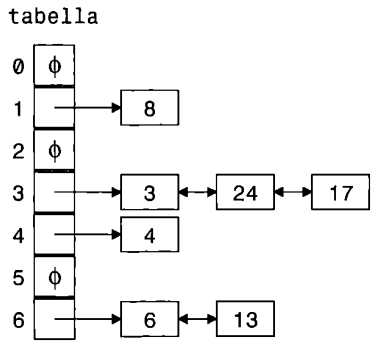
Supponiamo di utilizzare la funzione `Hash(k) = k % m` con `m = 3` e sia {3, 6, 8, 13, 24} l'insieme `S` di `n = 5` elementi. La tabella hash associata è quella mostrata nella figura (sono specificate solo le chiavi).



Eseguiamo l'inserimento di un nuovo elemento avente chiave 17: `Hash(17) = 2` quindi, dopo aver verificato che l'elemento non appartiene alla lista `tabella[2]`, vi viene inserito in fondo (si osservi che per motivi di chiarezza non sono rappresentati graficamente i puntatori `tabella[i].fine`).



Ora `m = 3` e `n = 6`, quindi il successivo inserimento di un elemento avente chiave 4 (non presente) dovrebbe causare il raddoppio della dimensione della tabella. Però, poiché questa dimensione deve essere un numero primo, scegliamo `m = 7`.



Dopo il ridimensionamento della tabella vi possono essere inseriti tutti gli elementi compreso il nuovo.

4.3.2 Tabelle hash: indirizzamento aperto

Nelle tabelle hash a indirizzamento aperto (*open addressing*), `tabella` è un array di `m` celle in cui porre gli `n` elementi, dove `m > n` (quindi il fattore di caricamento è $\alpha = n/m < 1$), in cui usiamo `null` per segnalare che la posizione corrente è vuota. Poiché le chiavi sono tutte collocate direttamente nell'array, usiamo una sequenza di funzioni `Hash[i]` per $0 \le i \le m - 1$, chiamata **sequenza di scansione** (*probing*), tale che i valori `Hash[0](k)`, `Hash[1](k)`, ..., `Hash[m - 1](k)` formino sempre una permutazione delle posizioni `0`, `1`, ..., `m - 1`, per ogni chiave $k \in U$. Tale permutazione rappresenta l'ordine con cui esaminiamo le posizioni di `tabella` durante le operazioni del dizionario.

Per comprendere l'organizzazione delle chiavi nella tabella hash, descriviamo prima l'operazione di inserimento di un elemento. Il Codice 4.4 mostra tale operazione, in cui iniziamo a esaminare le posizioni `Hash[0](k)`, `Hash[1](k)`, ..., `Hash[m - 1](k)` fino a trovare la prima posizione `Hash[i](k)` libera (poiché `n < m`, siamo sicuri che `i < m`): tale procedimento è analogo a quando cerchiamo posto in treno, in cui andiamo avanti fino a trovare un posto libero (ovviamente esaminiamo i posti in ordine lineare piuttosto che permutato). La ricerca di una chiave `k` segue questa falsariga: esaminiamo le suddette posizioni fino a trovare l'elemento con chiave `k` e, nel caso giungessimo in una posizione libera, saremmo certi che la chiave non compare nella tabella (altrimenti l'inserimento avrebbe posto in tale posizione libera un elemento con chiave `k`). La cancellazione di una chiave è solitamente realizzata in modo virtuale, sostituendo l'elemento con una marca speciale, che indica che la posizione è libera durante l'operazione di inserimento di successive chiavi e che la posizione è occupata, ma con una chiave diversa da quella cercata durante le successive operazioni di ricerca: quest'ultima condizione è necessaria poiché una cancellazione che svuotasse la posizione della chiave rimossa potrebbe ingannare una successiva ricerca (non necessariamente della stessa chiave) inducendola a fermarsi erroneamente in quella posizione e

dichiarare che la chiave cercata non è nel dizionario. Se prevediamo di effettuare molte cancellazioni, è quindi più conveniente usare una tabella con liste di trabocco perché si adatta meglio a realizzare dizionari molto dinamici. Nel Codice 4.4 implementiamo una tabella hash a indirizzamento aperto prevedendo di non effettuare mai cancellazioni.

Codice 4.4 Dizionario realizzato mediante tabelle hash con indirizzamento aperto.

```
1  Ricerca( k ):                          ⟨pre: tabella contiene n < m chiavi⟩
2    FOR (i = 0; i < m; i = i+1) {
3      h = Hash[i](k);
4      IF (tabella[h] == null) RETURN null;
5      IF (tabella[h].chiave == k) RETURN tabella[h];
6    }
```

```
1  Inserisci( e ):                        ⟨pre: tabella contiene n < m chiavi⟩
2    IF (Ricerca( e.chiave ) == null) {
3      i = -1;
4      DO {
5        i = i+1;
6        h = Hash[i]( e.chiave );
7        IF (tabella[h] == null) tabella[h] = e;
8      } WHILE (tabella[h] != e);
9    }
```

Per la complessità temporale, osserviamo che il caso pessimo rimane tempo `O(n)` e che ciascuna operazione è molto veloce in pratica se la funzione hash scelta è buona.

Teorema 4.2 Se per ogni chiave `k`, le posizioni in `Hash[0](k)`, `Hash[1](k)`, ..., `Hash[m - 1](k)` formano una delle `m!` permutazioni possibili *in modo uniformemente casuale*, il costo medio delle operazioni su tabelle hash a indirizzamento aperto è costante se $(1 - \alpha)^{-1} = O(1)$.

Dimostrazione Poiché il costo è direttamente proporzionale al valore di `i` tale che `Hash[i](k)` è la posizione individuata per la chiave, indichiamo con `T(n, m)` il valore medio di `i`: in altre parole, `T(n, m)` indica il numero medio di accessi effettuati quando inseriamo una chiave in una tabella di `m` posizioni, contenente già `n` elementi, dove `n < m`. (Il costo della ricerca può essere formulato come il costo di inserimento di quella chiave quando è stata inserita, se la chiave occorre, oppure come il costo di inserimento di una nuova chiave, se non occorre; il costo della cancellazione è pari al costo della ricerca.)

Utilizziamo un'equazione di ricorrenza per definire `T(n, m)`, dove `T(0, m) = 1` in quanto ogni posizione esaminata è sempre libera in una tabella vuota. Per `n > 0`, osserviamo che, essendo occupate `n` posizioni su `m` della tabella, la posizione

esaminata risulta occupata n volte su m (poiché tutte le permutazioni di posizioni sono equiprobabili). Effettuiamo un solo accesso se ci fermiamo su una posizione libera e questo accade con probabilità $\frac{m-n}{m}$. Se invece troviamo la posizione occupata, con probabilità $\frac{n}{m}$, effettuiamo ulteriori $T(n-1, m-1)$ accessi alle rimanenti posizioni (oltre all'accesso alla posizione occupata). Facendo la media pesata di tali costi, otteniamo $\frac{m-n}{m} \times 1 + \frac{n}{m} \times (1 + T(n-1, m-1))$, dando luogo alla seguente relazione di ricorrenza per il costo medio:

$$T(n, m) \leq \begin{cases} 1 & \text{se } n = 0 \\ 1 + \frac{n}{m} T(n-1, m-1) & \text{altrimenti} \end{cases} \qquad (4.1)$$

Proviamo per induzione che la soluzione della (4.1) soddisfa la relazione $T(n, m) \leq \frac{m}{m-n}$ per $m > n \geq 0$. Il caso base per $n = 0$ è immediato. Per $n > 0$ abbiamo che

$$T(n, m) \leq 1 + \frac{n}{m} T(n-1, m-1) < 1 + \frac{n}{m} \times \frac{m}{m-n} = \frac{m}{m-n}$$

Ne consegue che $T(n, m) \leq \frac{m}{m-n} = (1-\alpha)^{-1} = O(1)$ mantenendo l'invariante che m sia circa il doppio di n (Paragrafo 1.1.3). ❑

Si noti che i tempi medi calcolati per le tabelle hash con liste di trabocco e a indirizzamento aperto non sono direttamente confrontabili in quanto l'ipotesi di uniformità della funzione hash nel secondo caso è più forte di quella adottata nel primo caso.

Nella pratica, non possiamo generare una permutazione veramente casuale delle posizioni scandite con `Hash[i]` per $0 \leq i \leq m-1$. Per questo motivo, adottiamo alcune semplificazioni usando una o due funzioni `Hash(k)` e `Hash'(k)` (come quelle descritte nel Paragrafo 4.3), impiegandole come base di partenza per le scansioni della tabella, la cui dimensione m è un numero primo, in uno dei modi seguenti:

- `Hash[i](k) = (Hash(k) + i) % m` (scansione **lineare**): è quella più semplice da realizzare;
- `Hash[i](k) = (Hash(k) + ai² + bi + c) % m` (scansione **quadratica**): occorre scegliere i parametri a, b, c in modo che vengano ottenute tutte le posizioni in `[0, ..., m - 1]` al variare di `i = 0, ..., m - 1`;[3]

[3] Per esempio, la funzione $i^2 + 1$ realizza ciò (a = 1, b = 0, c = 1), mentre la funzione $2i^2$ con m pari non lo garantisce in quanto genera solo numeri pari (a = 2, b = 0, c = 0).

- `Hash[i](k) = (Hash(k) + i × (1 + Hash'(k))) % m` (scansione con **hash doppio**): occorre che `Hash'` sia differente da `Hash` e che per ogni k vengano ottenute tutte le posizioni in `[0, ..., m - 1]` al variare di `i = 0, ..., m - 1`.[4]

Nella scansione lineare, dopo aver calcolato il valore `h = Hash(k)`, esaminiamo le posizioni h, h + 1, h + 2 e così via in modo circolare. Tale scansione crea delle aggregazioni (*cluster*) di elementi, che occupano un segmento di posizioni contigue in `tabella`. Tali elementi sono stati originati da valori hash h differenti, ma condividono lo stesso cluster: la ricerca che ricade all'interno di tale cluster deve percorrerlo tutto nel caso non trovi la chiave. Un più attento esame delle chiavi contenute nel cluster mostra che quelle che andrebbero a finire in una stessa lista di trabocco (Paragrafo 4.3.1) sono tutte presenti nello stesso cluster (e tale proprietà può valere per più liste, le cui chiavi possono condividere lo stesso cluster). Pertanto, quando tali cluster sono di dimensione rilevante (seppure costante), conviene adottare le tabelle hash con liste di trabocco che hanno prestazioni migliori, quando sono associate a una buona funzione hash.

La scansione quadratica non migliora molto la situazione, in quanto i cluster appaiono in altra forma, seguendo l'ordine specificato da `Hash[i](k)`. La situazione cambia usando il doppio hash, in quanto l'incremento della posizione esaminata in `tabella` dipende da una seconda funzione `Hash'`: se due chiavi hanno una collisione sulla prima funzione hash, c'è ancora un'ulteriore possibilità di evitare collisioni con la seconda.

ESEMPIO 4.2

Supponiamo di utilizzare la scansione lineare con la funzione `Hash(k) = k % m` dove `m = 13`. Partendo dalla tabella vuota, l'inserimento delle chiavi 5, 13, 16 e 17 non causa alcuna collisione. La tabella risultante è mostrata nella figura.

	0	1	2	3	4	5	6	7	8	9	10	11	12
tabella	13			16	17	5							

La chiave 3 viene inserita nella posizione 6 essendo occupate le posizioni 3 = (Hash(3) + 0) % 13, 4 = (Hash(3) + 1) % 13 e 5 = (Hash(3) + 2) % 13.

	0	1	2	3	4	5	6	7	8	9	10	11	12
tabella	13			16	17	5	3						

Il segmento di `tabella` tra le posizioni 3 e 6 è un cluster. L'inserimento di una qualsiasi chiave il cui valore hash h ricada all'interno del cluster provoca tante collisioni quanti sono gli elementi del cluster alla destra di h, come già avvenuto inserendo la chiave 3.

[4] Il valore di `1 + Hash'(k)` deve essere sempre primo rispetto a m. Esistono diversi modi per ottenere ciò. Uno è quello di scegliere m come numero primo e garantire che `1 + Hash'(k) < m`. Un altro è quello di fissare m una potenza del due e garantire che `1 + Hash'(k)` restituisca sempre un numero dispari (cioè `Hash'(k)` sia pari).

4.4 Opus libri: kernel Linux e alberi binari di ricerca

Nel sistema operativo GNU/Linux, i processi generati dai vari programmi in esecuzione sono gestiti nel nucleo (*kernel*). In particolare, ogni processo ha a disposizione uno spazio virtuale degli indirizzi di memoria, detto **memoria virtuale**, in cui le celle sono numerate a partire da `0`. La memoria virtuale fa sì che il calcolatore (utilizzando la memoria secondaria) sembri avere più memoria principale di quella fisicamente presente, condividendo quest'ultima tra tutti i processi che competono per il suo uso. La memoria virtuale di un processo è suddivisa in aree di memoria (VMA) di dimensione limitata, ma solo un sottoinsieme di tutte le VMA associate a un processo risultano presenti fisicamente nella memoria principale. Quando un processo accede a un indirizzo di memoria virtuale, il kernel deve verificare che la VMA contenente quell'indirizzo sia presente nella memoria principale: se è così, usa le informazioni associate alla VMA per effettuare la traduzione dell'indirizzo virtuale nel corrispondente indirizzo fisico. In caso contrario, si verifica un *page fault* che costringe il kernel a caricare nella memoria principale la VMA richiesta, eventualmente scaricando nella memoria secondaria un'altra VMA.

La ricerca della VMA deve essere eseguita in modo efficiente: a tale scopo, il kernel usa una strategia mista (tipica di Linux e applicata anche in altri contesti del sistema operativo), che è basata sull'uso di dizionari. Fintanto che il numero di VMA presenti in memoria è limitato (circa una decina), le VMA assegnate a un processo sono mantenute in una lista e la ricerca di una specifica VMA viene eseguita attraverso di essa. Quando il numero di VMA supera un limite prefissato, la lista viene affiancata da una struttura di dati più efficiente dal punto di vista della ricerca: tale struttura di dati è chiamata albero binario di ricerca (fino alla versione 2.2 del kernel, venivano usati gli alberi AVL, mentre nelle versioni successive questi sono stati sostituiti dagli alberi rosso-neri). In effetti, gli alberi binari di ricerca costituiscono uno degli strumenti fondamentali per il recupero efficiente di informazioni e sono pertanto applicati in moltissimi altri contesti, oltre a quello appena discusso.

4.4.1 Alberi binari di ricerca

Un albero binario viene generalmente rappresentato nella memoria del calcolatore facendo uso di tre campi per ogni nodo, come mostrato nel Paragrafo 1.4: dato un nodo `u`, indichiamo con `u.sx` il riferimento al figlio sinistro, con `u.dx` il riferimento al figlio destro e con `u.dato` il contenuto del nodo, ovvero un elemento $e \in S$ nel caso di dizionari (precisamente, faremo riferimento a `u.dato.chiave` e `u.dato.sat` per indicare la chiave e i dati satellite di tale elemento). Volendo impiegare gli alberi per realizzare un dizionario *ordinato* per un insieme

$S = \{e_0, e_1, \ldots, e_{n-1}\}$ di elementi, memorizziamo gli elementi nei loro nodi in modo da soddisfare la seguente proprietà di ricerca per *ogni* nodo `u`:

- tutti gli elementi nel sottoalbero *sinistro* di `u` (riferito da `u.sx`) sono *minori* dell'elemento `u.dato` contenuto in `u`;
- tutti gli elementi nel sottoalbero *destro* di `u` (riferito da `u.dx`) sono *maggiori* di `u.dato`.

Un **albero binario di ricerca** è un albero binario che soddisfa la suddetta proprietà di ricerca: una conseguenza della proprietà è che una visita *anticipata* dell'albero fornisce la sequenza *ordinata* degli elementi in `S`, in tempo $O(n)$.

La ricerca di una chiave `k` in tale albero segue la falsariga della versione ricorsiva della ricerca binaria (Paragrafo 3.1). Il Codice 4.5 ricalca tale schema ricorsivo: partiamo dalla radice dell'albero e confrontiamo il suo contenuto con `k` e, nel caso sia diverso, se `k` è minore proseguiamo la ricerca a sinistra, altrimenti la proseguiamo a destra.

Codice 4.5 Algoritmi ricorsivi per la ricerca dell'elemento con chiave `k` e l'inserimento di un elemento `e` in un albero di ricerca con radice `u`.

```
1  Ricerca( u, k ):
2    IF (u == null) RETURN null;
3    IF (k == u.dato.chiave) {
4      RETURN u.dato;
5    } ELSE IF (k < u.dato.chiave) {
6      RETURN Ricerca( u.sx, k );
7    } ELSE {
8      RETURN Ricerca( u.dx, k );
9    }
```

```
1  Inserisci( u, e ):
2    IF (u == null) {
3      u = NuovoNodo();
4      u.dato = e;
5      u.sx = u.dx = null;
6    } ELSE IF (e.chiave < u.dato.chiave) {
7      u.sx = Inserisci( u.sx, e );
8    } ELSE IF (e.chiave > u.dato.chiave) {
9      u.dx = Inserisci( u.dx, e );
10   }
11   RETURN u;                    ⟨post: se k appare già in u, non viene memorizzata⟩
```

Per l'inserimento osserviamo che, quando raggiungiamo un riferimento vuoto, lo sostituiamo con un riferimento a un nuovo nodo (righe 3-5), che restituiamo per farlo propagare verso l'alto (riga 11) attraverso le chiamate ricorsive (abbiamo

discusso tali schemi ricorsivi nel Paragrafo 3.8): tale propagazione avviene notando che le chiamate ricorsive alle righe 7 e 9 sovrascrivono il campo relativo al riferimento (figlio sinistro o destro) su cui sono state invocate. Infatti se k è minore della chiave nel nodo corrente, l'inseriamo ricorsivamente nel figlio sinistro; se k è maggiore, l'inseriamo ricorsivamente nel figlio destro; altrimenti, abbiamo un duplicato e non l'inseriamo affatto. Il costo della ricerca e dell'inserimento è pari all'altezza h dell'albero, ovvero tempo O(h).

La cancellazione dell'elemento con chiave k presenta più casi da esaminare, in quanto essa può disconnettere l'albero che va, in tal caso, opportunamente riconnesso per mantenere la proprietà di ricerca, come descritto nel Codice 4.6, il cui schema ricorsivo è simile a quello dell'inserimento. I casi più semplici sono quando il nodo u è una foglia oppure ha un solo figlio (righe 5 e 7): eliminiamo la foglia mettendo a null il riferimento a essa oppure, se il nodo ha un solo figlio, creiamo un "ponte" tra il padre di u e il figlio di u.

Codice 4.6 Algoritmo ricorsivo per la cancellazione dell'elemento con chiave k da un albero di ricerca con radice u.

```
1  Cancella( u, k ):
2    IF (u != null) {
3      IF (u.dato.chiave == k) {
4        IF (u.sx == null) {
5          u = u.dx;
6        } ELSE IF (u.dx == null) {
7          u = u.sx;
8        } ELSE {
9          w = MinimoSottoAlbero( u.dx );
10         u.dato = w.dato;
11         u.dx = Cancella( u.dx, w.dato.chiave );
12       }
13     } ELSE IF (k < u.dato.chiave) {
14       u.sx = Cancella( u.sx, k );
15     } ELSE IF (k > u.dato.chiave) {
16       u.dx = Cancella( u.dx, k );
17     }
18   }
19   RETURN u;
```

```
1  MinimoSottoAlbero( u ):                    ⟨pre: u ≠ null⟩
2    WHILE (u.sx != null)
3      u = u.sx;
4    RETURN u;
```

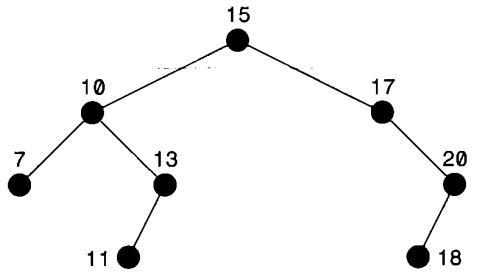

Quando u ha due figli non possiamo cancellarlo fisicamente (righe 9-11), ma dobbiamo individuare il nodo w che contiene il successore di k nell'albero (riga 9), che risulta essere il minimo del sottoalbero destro (u.dx non è null). Sostituiamo quindi l'elemento in u con quello in w per mantenere la proprietà di ricerca dell'albero (riga 10) e, con una chiamata ricorsiva, cancelliamo fisicamente w in quanto contiene la chiave copiata in u (riga 11).

È importante osservare che quest'ultima s'imbatterà in un caso semplice con la cancellazione di w, in quanto w non può avere il figlio sinistro (altrimenti non conterrebbe il minimo del sottoalbero destro). Come osservato in precedenza, propaghiamo l'effetto della cancellazione verso l'alto (riga 19) attraverso le chiamate ricorsive. Per la complessità temporale, osserviamo che il codice percorre il cammino dalla radice al nodo u in cui si trova l'elemento con chiave k e poi percorre due volte il cammino da u al suo discendente w. In totale, il costo rimane tempo O(h) anche in questo caso.

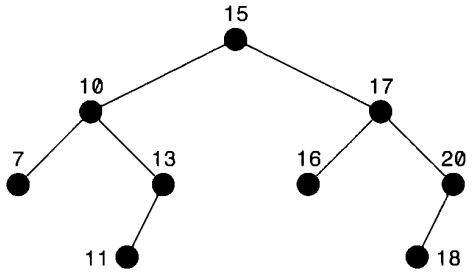
Purtroppo $h = \Theta(n)$ al caso pessimo e un albero non sembra essere vantaggioso rispetto a una lista ordinata. Tuttavia, con elementi inseriti in maniera casuale, l'altezza media è $O(\log n)$ e i metodi discussi finora hanno una complessità $O(\log n)$ al caso medio. Vediamo come ottenere degli alberi binari di ricerca *bilanciati*, che hanno sempre altezza $h = O(\log n)$ dopo qualunque sequenza di inserimenti o cancellazioni. Questo fa sì che la complessità delle operazioni diventi $O(\log n)$ anche al caso pessimo.

ESEMPIO 4.3

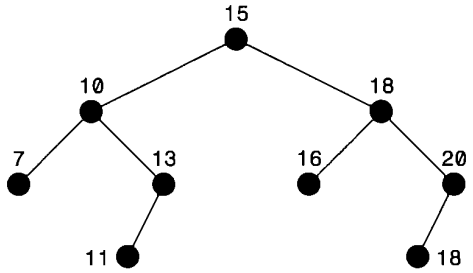
Si vuole inserire un nuovo nodo e avente chiave 16 nel seguente albero binario di ricerca. Supponiamo che la radice dell'albero sia il nodo r e che e sia l'elemento da inserire contenente la chiave 16.



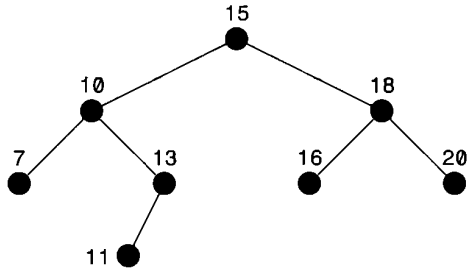
La funzione Inserisci verrà invocata su r (la radice dell'albero), poi sul suo figlio destro (in quanto e.chiave > r.dato.chiave, riga 9) e poi sul figlio sinistro del nodo u contenente la chiave 17 (riga 7). Poiché quest'ultimo nodo è null, viene creato un nuovo nodo con chiave 16 (riga 3) e il suo indirizzo restituito alla funzione chiamante che lo assegnerà a u.sx (riga 7).



Ora cancelliamo il nodo con etichetta 17. La funzione `Cancella(r, 17)` esegue `r.dx = Cancella(r.dx, 17)`. Poiché la chiave in `r.dx` è quella che stiamo cercando e poiché `r.dx` ha entrambi i figli vengono eseguite le righe 9-11. Attraverso la funzione `MinimoSotto-Albero(r.dx.dx)` viene individuato il minimo del sottoalbero destro di `r.dx` che viene copiato in `r.dx`.



Successivamente viene eliminato il nodo con chiave 18 dal sottoalbero con radice `u = r.dx.dx` invocando `Cancella(u, 18)`. Questa a sua volta esegue `u.sx = Cancella(u.sx, 18)` che restituisce il valore del suo sottoalbero destro, ovvero `null` (riga 5).



4.4.2 AVL: alberi binari di ricerca bilanciati

L'**albero AVL** (acronimo derivato dalle iniziali degli autori russi Adel'son-Velsky e Landis che lo inventarono negli anni '60) è un albero binario di ricerca che garantisce avere un'altezza $h = O(\log n)$ per n elementi memorizzati nei suoi nodi. Oltre alla proprietà di ricerca menzionata nel Paragrafo 4.4.1, l'albero AVL soddisfa la proprietà di essere **1-bilanciato** al fine di garantire l'altezza logaritmica.

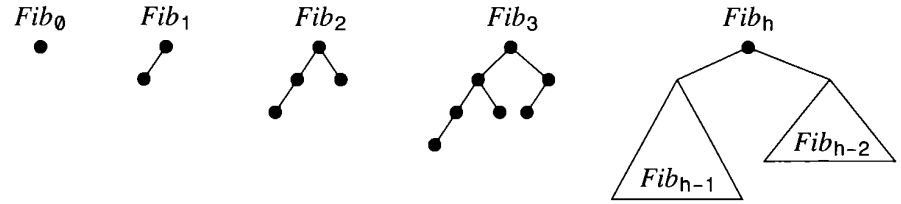


Figura 4.1 Alberi di Fibonacci.

Dato un nodo u, indichiamo con `h(u)` la sua altezza, che identifichiamo con l'altezza del sottoalbero radicato in u, dove `h(null) = -1` (Paragrafo 2.4). Un nodo u è **1-bilanciato** se le altezze dei suoi due figli differiscono per al più di un'unità

$$|h(u.sx) - h(u.dx)| \leq 1 \tag{4.2}$$

Un albero binario è 1-bilanciato se *ogni* suo nodo è 1-bilanciato. Gli alberi dell'Esempio 5.3 sono alberi 1-bilanciati.

La connessione tra l'essere 1-bilanciato e avere altezza logaritmica non è immediata e passa attraverso gli **alberi di Fibonacci**, che sono un sottoinsieme degli alberi 1-bilanciati con il *minor* numero di nodi a *parità* di altezza. In altre parole, indicato con Fib_h un albero di Fibonacci di altezza h e con n_h il suo numero di nodi, eliminando un solo nodo da Fib_h otteniamo che l'altezza diminuisce o che l'albero risultante non è più 1-bilanciato: nessun albero 1-bilanciato con n nodi e altezza h può dunque avere meno di n_h nodi, ossia $n \geq n_h$.

L'albero di Fibonacci Fib_h di altezza h è definito ricorsivamente (Figura 4.1): per $h = 0$, abbiamo un solo nodo, per cui $n_0 = 1$, e, per $h = 1$, abbiamo un albero con $n_1 = 2$ nodi (la radice e un solo figlio, che nella figura è quello sinistro). Per $h > 1$, l'albero Fib_h è costruito prendendo un albero Fib_{h-1} e un albero Fib_{h-2}, le cui radici diventano i figli di una nuova radice (quella di Fib_h). Di conseguenza, abbiamo che $n_h = n_{h-1} + n_{h-2} + 1$, relazione ricorsiva che ricorda quella dei numeri di Fibonacci[5] motivando così il nome di tali alberi.

Teorema 4.3 *Sia* T *un albero AVL di* n *nodi e altezza* h, *allora* $h = O(\log n)$.

Dimostrazione Dimostreremo che Fib_h è l'albero 1-bilanciato di altezza h col minimo numero di nodi. Mostrando che $n_h \geq c^h$ per un'opportuna costante $c > 1$, deriviamo che $n \geq c^h$ e, quindi, che $h = O(\log n)$.

Possiamo osservare nella Figura 4.1 che Fib_0 e Fib_1 sono alberi 1-bilanciati di altezza 0 e 1, rispettivamente, con il *minimo* numero di nodi possibile. Ipotizzando che, per induzione, tale proprietà valga per ogni $\ell < h$ con $h > 1$, mostriamo come ciò sia vero anche per Fib_h ragionando per assurdo. Supponiamo di poter rimuovere un

[5] Ricordiamo che la successione dei numeri di Fibonacci $\{F_i\}_{i \geq 0}$ è definita ricorsivamente nel seguente modo: $F_0 = 0$, $F_1 = 1$ e, per ogni $i \geq 2$, $F_i = F_{i-1} + F_{i-2}$.

nodo da Fib_h mantenendo la sua altezza h e garantendo che rimanga 1-bilanciato: non potendo rimuovere la radice, tale nodo deve appartenere a Fib_{h-1} oppure a Fib_{h-2} per costruzione. Ciò è impossibile in quanto questi ultimi sono minimali per ipotesi induttiva e, se Fib_{h-1} cambiasse altezza, anche Fib_h la cambierebbe, mentre se Fib_{h-2} cambiasse altezza, Fib_h non sarebbe più 1-bilanciato. Quindi, anche Fib_h è minimale e concludiamo che ogni albero 1-bilanciato di altezza h con n nodi soddisfa $n \geq n_h$.

Tabulando i primi 15 valori di n_h e altrettanti numeri di Fibonacci F_h, possiamo verificare per ispezione diretta che vale la relazione $n_h = F_{h+3} - 1$:

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
n_h	1	2	4	7	12	20	33	54	88	143	232	376	609	986	1596
F_h	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Utilizzando la nota forma chiusa dei numeri di Fibonacci

$$F_h = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}, \quad \text{dove } \phi = \frac{1+\sqrt{5}}{2} \approx 1,6180339\cdots$$

possiamo affermare che $F_h > \frac{\phi^h - 1}{\sqrt{5}}$ e che, quindi, esiste una costante $c > 1$ tale che $F_h > c^h$ per $h > 2$. Pertanto, $n_h = F_{h+3} - 1 \geq c^h$ e, poiché $n \geq n_h$, possiamo concludere che $n \geq c^h$: in altre parole, ogni albero 1-bilanciato di n nodi e altezza h verifica $h = O(\log n)$. ❑

Per implementare gli alberi AVL, introduciamo un ulteriore campo `u.altezza` nei suoi nodi u, tale che `u.altezza` = `h(u)`. Notiamo che l'operazione di ricerca negli alberi AVL rimane identica a quella descritta nel Codice 4.5. Mostriamo quindi nel Codice 4.7 come estendere l'inserimento per garantire che l'albero AVL rimanga 1-bilanciato.

Codice 4.7 Algoritmo per l'inserimento di un elemento e in un albero AVL con radice u.

```
1  Inserisci( u, e ):
2    IF (u == null) {
3      RETURN f = NuovaFoglia( e );
4    } ELSE IF (e.chiave < u.dato.chiave) {
5      u.sx = Inserisci( u.sx, e );
6      IF (Altezza(u.sx) - Altezza(u.dx) == 2) {
7        IF (e.chiave>u.sx.dato.chiave) u.sx=RuotaAntiOraria(u.sx);
8        u = RuotaOraria( u );
9      }
10   } ELSE IF (e.chiave > u.dato.chiave) {
11     u.dx = Inserisci( u.dx, e );
12     IF (Altezza(u.dx) - Altezza(u.sx) == 2) {
13       IF (e.chiave < u.dx.dato.chiave) u.dx = RuotaOraria(u.dx);
14       u = RuotaAntiOraria( u );
15     }
16   }
17   u.altezza = max( Altezza(u.sx), Altezza(u.dx) ) + 1;
18   RETURN u;
```

```
1  Altezza( u ):
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      RETURN u.altezza;
6    }
```

```
1  NuovaFoglia( e ):
2    u = NuovoNodo();
3    u.dato = e;
4    u.altezza = 0;
5    u.sx = u.dx = null;
6    RETURN u;
```


Dopo la creazione della foglia `f` contenente l'elemento e (riga 3), aggiorniamo le altezze ricorsivamente nei campi `u.altezza` degli antenati u di `f`, essendo questi ultimi i soli nodi che possono cambiare altezza (riga 17). Allo stesso tempo, controlliamo se il nodo corrente è 1-bilanciato: definiamo **nodo critico** il minimo antenato di `f` che viola tale proprietà.

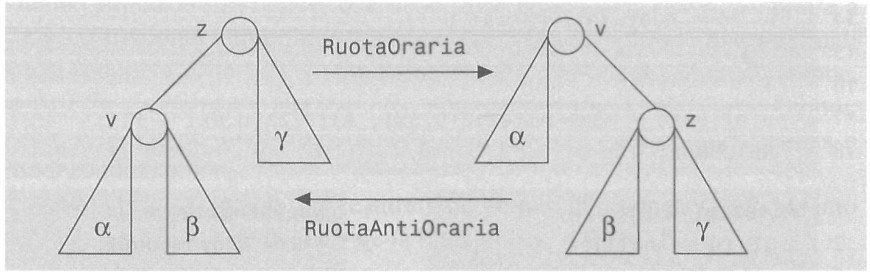
A tal fine, percorriamo in ordine inverso le chiamate ricorsive sugli antenati di `f` (righe 5 e 11), fino a individuare il nodo critico u, se esiste: in tal caso, se `f` discende da `u.sx`, l'altezza del sottoalbero sinistro di u differisce di due rispetto a quella del destro (riga 6), mentre se `f` discende da `u.dx`, abbiamo che è l'altezza del sottoalbero destro di u a differire di due rispetto a quella del sinistro (riga 12). Comunque vada, aggiorniamo l'altezza del nodo prima di terminare la chiamata attuale (riga 17).

Discutiamo ora come ristrutturare l'albero in corrispondenza del nodo critico u, utilizzando le **rotazioni** orarie e antiorarie per rendere nuovamente u un nodo 1-bilanciato (righe 7-8 e 13-14): tali rotazioni sono illustrate e descritte nel Codice 4.8. Notiamo che esse richiedono tempo `O(1)` e preservano la proprietà di ricerca: le chiavi in α sono minori di `v.dato.chiave`; quest'ultima è minore delle chiavi in β, le quali sono minori di `z.dato.chiave`; infine, quest'ultima è minore delle chiavi in γ.

Utilizziamo le rotazioni per trattare i quattro casi che si possono presentare (individuati con un semplice codice mnemonico), in base alla posizione di `f` rispetto ai nipoti del nodo critico u (Figura 4.2):

1. caso SS: la foglia `f` appartiene al sottoalbero α radicato in `u.sx.sx`;
2. caso SD: la foglia `f` appartiene al sottoalbero β radicato in `u.sx.dx`;
3. caso DS: la foglia `f` appartiene al sottoalbero γ radicato in `u.dx.sx`;
4. caso DD: la foglia `f` appartiene al sottoalbero δ radicato in `u.dx.dx`.

Codice 4.8 Rotazioni oraria e antioraria.




```
1  RuotaOraria( z ):
2    v = z.sx;
3    z.sx = v.dx;
4    v.dx = z;
5    z.altezza = max( Altezza(z.sx), Altezza(z.dx) ) + 1;
6    v.altezza = max( Altezza(v.sx), Altezza(v.dx) ) + 1;
7    RETURN v;
```

```
1  RuotaAntiOraria( v ):
2    z = v.dx;
3    v.dx = z.sx;
4    z.sx = v;
5    v.altezza = max( Altezza(v.sx), Altezza(v.dx) ) + 1;
6    z.altezza = max( Altezza(z.sx), Altezza(z.dx) ) + 1;
7    RETURN z;
```

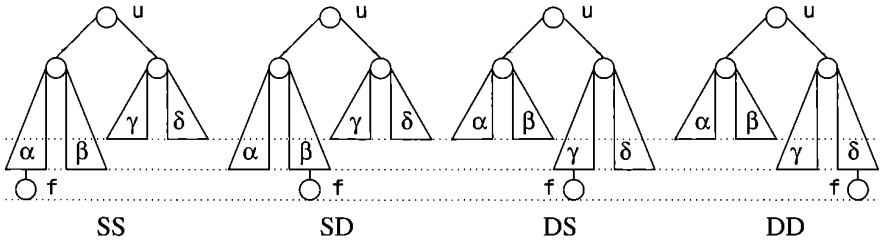


Figura 4.2 I quattro casi possibili di sbilanciamento del nodo critico u a causa della creazione della foglia f (contenente la chiave k).

In tutti e quattro i casi, lo sbilanciamento conduce l'albero AVL in una configurazione in cui due sottoalberi hanno un dislivello pari a due. Con riferimento all'inserimento descritto nel Codice 4.7, trattiamo il caso SS con una rotazione oraria effettuata sul nodo critico u, riportando l'altezza del sottoalbero a quella immediatamente prima che la foglia f venisse creata (riga 8). Se invece incontriamo il caso SD, prima effettuiamo una rotazione antioraria sul figlio sinistro di u (riga 7) e poi una oraria su u stesso (riga 8): anche in tal caso, l'altezza del sottoalbero torna a essere quella immediatamente prima che la foglia f venisse creata (Figura 4.3). I casi DD e DS sono speculari: effettuiamo una rotazione antioraria su u (riga 14) oppure una rotazione oraria sul figlio destro di u seguita da una antioraria su u (righe 13 e 14). Siccome ogni caso richiede una o due rotazioni, il costo è tempo O(1) per eseguire le rotazioni (al più due), a cui va aggiunto il tempo di inserimento O(log n).

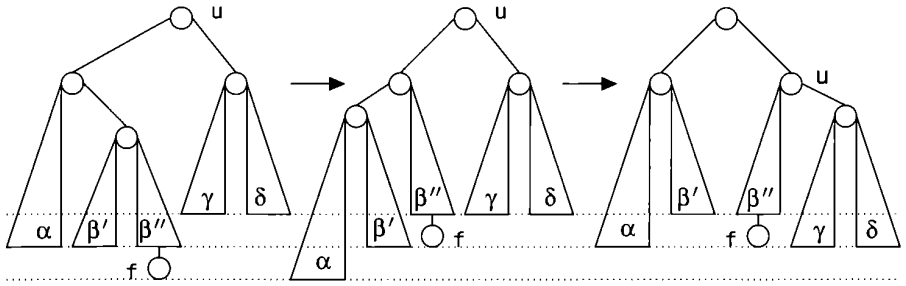
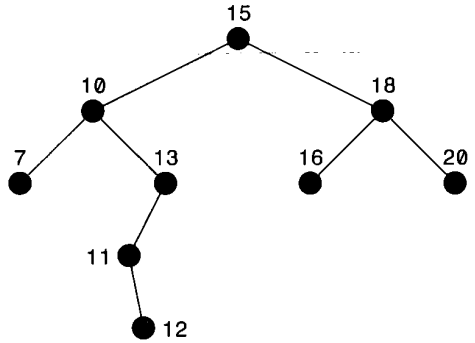


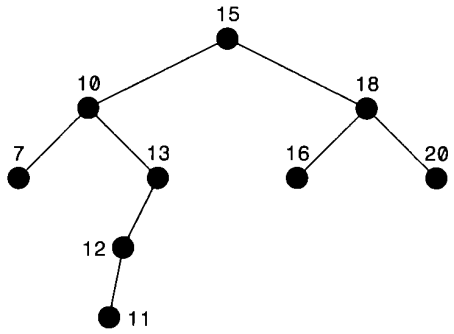
Figura 4.3 Effetto delle rotazioni sul nodo critico u per il caso SD.

ESEMPIO 4.4

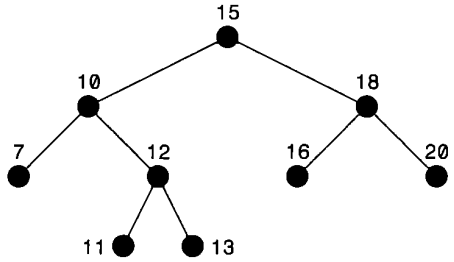
Consideriamo l'inserimento del nodo con chiave 12 nell'ultimo albero mostrato nell'Esempio 5.3 che è un albero AVL. Dopo la prima fase, prima del ribilanciamento, l'albero appare come segue.



I nodi con chiave 12 e 11 risultano essere 1-bilanciati mentre quello con chiave 13 no: quest'ultimo è il nodo critico che chiameremo u. Poiché la chiave inserita è maggiore della chiave del figlio sinistro di u (ovvero 11) viene prima eseguita una rotazione antioraria su figlio sinistro di u (riga 7).



Segue una rotazione oraria su u (riga 8).



Quest'ultima operazione restituisce l'albero 1-bilanciato.

Per la cancellazione, possiamo trattare dei casi simili a quelli dell'inserimento, solo che possono esserci più nodi critici tra gli antenati del nodo cancellato: preferiamo quindi marcare logicamente i nodi cancellati, che vengono ignorati ai fini della ricerca. Quando una frazione costante dei nodi sono marcati come cancellati, ricostruiamo l'albero con le sole chiavi valide ottenendo un costo ammortizzato di `O(log n)`. Possiamo trasformare il costo ammortizzato in un costo al caso pessimo interlacciando la ricostruzione dell'albero, che produce una copia dell'albero attuale, con le operazioni normalmente eseguite su quest'ultimo.

Nonostante siano stati concepiti diverso tempo fa, gli alberi AVL sono tuttora molto competitivi rispetto a strutture di dati più recenti: la maggior parte delle rotazioni avvengono negli ultimi due livelli di nodi, per cui gli alberi AVL risultano molto efficienti se implementati opportunamente (all'atto pratico, la loro forma è molto vicina a quella di un albero completo e quasi perfettamente bilanciato).

4.5 Opus libri: liste invertite e trie

Nei sistemi di recupero dei documenti (**information retrieval**), i dati sono documenti testuali e il loro contenuto è relativamente stabile: pertanto, in tali sistemi lo scopo principale è quello di fornire una risposta veloce alle numerose interrogazioni degli utenti (contrariamente alle basi di dati che sono progettate per garantire

un alto flusso di transazioni che ne modificano i contenuti). In tal caso, la scelta adottata è quella di elaborare preliminarmente l'archivio dei documenti per ottenere un indice in cui le ricerche siano molto veloci, di fatto evitando di scandire l'intera collezione dei documenti a ogni interrogazione (come vedremo, i motori di ricerca sul Web rappresentano un noto esempio di questa strategia). Infatti, il tempo di calcolo aggiuntivo che è richiesto nella costruzione degli indici, viene ampiamente ripagato dal guadagno in velocità di risposta alle innumerevoli richieste che pervengono a tali sistemi.

Le **liste invertite** (chiamate anche file invertiti, file dei *posting* o concordanze) costituiscono uno dei cardini nell'organizzazione di documenti in tale scenario. Le consideriamo un'organizzazione logica dei dati piuttosto che una specifica struttura di dati, in quanto le componenti possono essere realizzate utilizzando strutture di dati differenti. La nozione di liste invertite si basa sulla possibilità di definire in modo preciso la nozione di **termine** (inteso come una parola o una sequenza massimale alfanumerica), in modo da partizionare ciascun documento o **testo** `T` della collezione di documenti `D` in segmenti disgiunti corrispondenti alle occorrenze dei vari termini (quindi le occorrenze di due termini non possono sovrapporsi in `T`). La struttura delle liste invertite è infatti riconducibile alle seguenti due componenti (ipotizzando che il testo sia visto come una sequenza di caratteri):

- il vocabolario o lessico `V`, contenente l'insieme dei termini distinti che appaiono nei testi di `D`;
- la lista invertita L_P (detta dei *posting*) per ciascun termine $P \in V$, contenente un riferimento alla posizione iniziale di ciascuna occorrenza di `P` nei testi $T \in D$: in altre parole, la lista per `P` contiene la coppia $\langle T, i \rangle$ se il segmento `T[i, i + |P| - 1]` del testo è proprio uguale a `P` (ogni documento $T \in D$ ha un suo identificatore numerico che lo contraddistingue dagli altri documenti in `D`).

Notiamo che le liste invertite sono spesso mantenute in forma compressa per ridurre lo spazio a circa il 10-25% del testo e, inoltre, le occorrenze sono spesso riferite al numero di linea, piuttosto che alla posizione precisa nel testo. Solitamente, la lista invertita L_P memorizza anche la sua lunghezza in quanto rappresenta la frequenza del termine `P` nei documenti in `D`. Per completezza, osserviamo che esistono metodi alternativi alle liste invertite come le *bitmap* e i *signature file*, ma osserviamo anche che tali metodi non risultano superiori alle liste invertite come prestazioni.

La realizzazione delle liste invertite prevede l'utilizzo di un dizionario (tabella hash o albero di ricerca) per memorizzare i termini $P \in V$: nello specifico, ciascun elemento `e` memorizzato nel dizionario ha un distinto termine $P \in V$ contenuto nel campo `e.chiave` e ha un'implementazione della corrispondente lista L_P nel campo `e.sat`. Nel seguito, ipotizziamo quindi che `e.sat` sia una lista doppia che fornisce le operazioni descritte nel Paragrafo 4.2; inoltre, ipotizziamo che un termine sia una sequenza massimale alfanumerica nel testo dato in ingresso.

Il Codice 4.9 descrive come costruire le liste invertite usando un dizionario per memorizzare i termini distinti, secondo quanto abbiamo osservato sopra. Lo scopo è quello di identificare una sequenza alfanumerica massima rappresentata dal segmento di testo `T[i, j - 1]` per `i < j` (righe 4-8): tale termine viene cercato nel dizionario (riga 10) e, se appare in esso, la coppia ⟨T, i⟩ che ne denota l'occorrenza in T viene posta in fondo alla corrispondente lista invertita (riga 12); se invece `T[i, j - 1]` non appare nel dizionario, tale lista viene creata e memorizzata nel dizionario (righe 14-17).

Codice 4.9 Costruzione di liste invertite di un testo $T \in D$ (`elemento` indica un nuovo elemento).

```
1  CostruzioneListeInvertite( T ):      ⟨pre: T ∈ D è un testo di lunghezza n⟩
2    i = 0;
3    WHILE ( i < n ) {
4      WHILE (i < n && !AlfaNumerico( T[i] ))
5        i = i+1;
6      j = i;
7      WHILE (j < n && AlfaNumerico( T[j] ))
8        j = j+1;
9        IF ( i < n ) {
10         e = dizionarioListeInvertite.Ricerca( T[i,j-1] );
11         IF (e != null) {
12           e.sat.InserisciFondo( <T,i> );
13         } ELSE {
14           elemento.chiave = T[i,j-1];
15           elemento.sat = NuovaLista( );
16           elemento.sat.InserisciFondo( <T,i> );
17           dizionarioListeInvertite.Inserisci( elemento )
18         }
19         i = j;
20       }
21   }
```

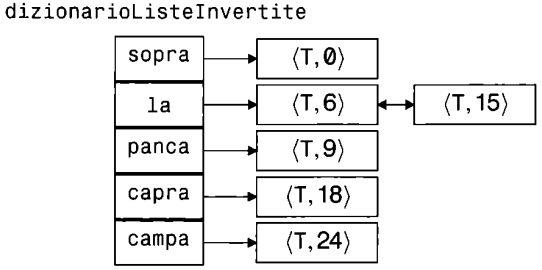
```
1  AlfaNumerico( c ):
2    RETURN ('a' <= c <= 'z' || 'A' <= c <= 'Z' || '0' <= c <= '9');
```

ESEMPIO 4.5

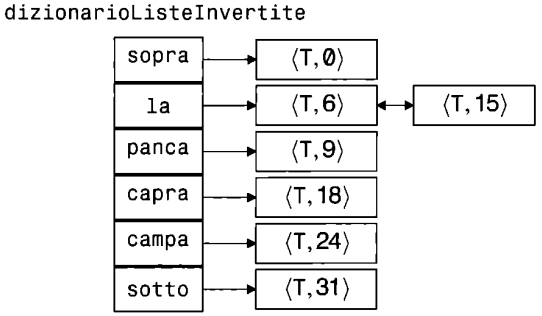
Si consideri il testo T riportato di seguito, i numeri rappresentano la posizione della lettera corrispondente all'interno di T.

```
0     6  9     15 18    24     31    37 40    46 49  55
SOPRA LA PANCA LA CAPRA CAMPA, SOTTO LA PANCA LA CAPRA CREPA.
```

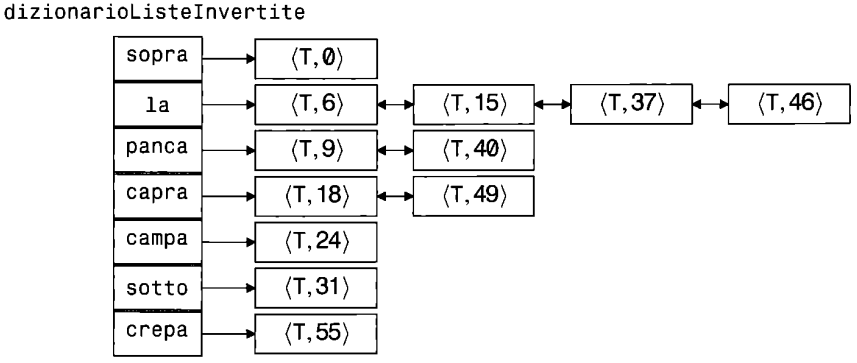

Supponiamo di aver già inserito nel dizionario le prime 6 parole. La situazione di `dizionarioListeInvertite` è rappresentata nella figura che segue.



Dopo aver inserito `campa`, `i = j = 29`. Alla fine del ciclo della riga 4, `i` e `j` puntano al primo carattere alfanumerico di T che segue, ovvero `i = j = 31`. Il ciclo successivo sposta `j` nella posizione 36 e quindi viene considerata la sequenza alfanumerica T[31, 35], vale a dire la parola `sotto`. Essa non è presente nel dizionario (riga 9), quindi vi viene aggiunta (righe 13-16) inserendo nel dizionario una nuova lista composta da un unico elemento contenente il valore di ⟨T, i⟩



I quattro termini che seguono sono già presenti nel dizionario, pertanto si aggiungono in fondo alle liste corrispondenti le informazioni sulle posizioni di queste nuove occorrenze. Infine l'ultimo termine, `crepa`, viene aggiunto al dizionario e viene creata la lista composta da un elemento contenente l'informazione su questa unica occorrenza.



Supponendo che n sia il numero totale di caratteri esaminati, il costo della costruzione descritta nel Codice 4.9 è pari a O(n) al caso medio usando le tabelle hash e O(n log n) usando gli alberi di ricerca bilanciati. Nei casi reali, la costruzione è concettualmente simile a quella descritta nel Codice 4.9 ma notevolmente differente nelle prestazioni. Per esempio, dovremmo aspettarci di applicare tale codice a una vasta collezione di documenti che, per la sua dimensione, viene memorizzata nella memoria secondaria.

Ne risulta che, per l'analisi del costo finale, possiamo utilizzare il modello di memoria a due livelli valutando le varie decisioni progettuali: di solito, il vocabolario V è mantenuto nella memoria principale, mentre le liste e i documenti risiedono nella memoria secondaria; tuttavia, diversi motori di ricerca memorizzano anche le liste invertite (ma non i documenti) nella memoria principale utilizzando decine di migliaia di macchine in rete, ciascuna dotata di ampia memoria principale a basso costo.

Per quanto riguarda le interrogazioni effettuate in diversi motori di ricerca, esse prevedono l'uso di termini collegati tra loro mediante gli operatori booleani: l'operatore di congiunzione (AND) tra due termini prevede che entrambi i termini occorrano nel documento; quello di disgiunzione (OR) prevede che almeno uno dei termini occorra; infine, l'operatore di negazione (NOT) indica che il termine non debba occorrere.

È possibile usufruire anche dell'operatore di vicinanza (NEAR) come estensione dell'operatore di congiunzione, in cui viene espresso che i termini specificati occorrano a poche posizioni l'uno dall'altro. Tali interrogazioni possono essere composte come una qualunque espressione booleana, anche se le statistiche mostrano che la maggior parte delle interrogazioni nei motori di ricerca consiste nella ricerca di due termini connessi dall'operatore di congiunzione.

Le liste invertite aiutano a formulare tali interrogazioni in termini di operazioni elementari su liste, supponendo che ciascuna lista L_P sia ordinata. Per esempio, l'interrogazione (P AND Q) altro non è che l'intersezione tra L_P e L_Q, mentre (P OR Q) ne è l'unione senza ripetizioni: entrambe le operazioni possono essere svolte efficientemente con una variante dell'algoritmo di fusione adoperato per il *mergesort*. L'operazione di negazione è il complemento della lista e, infine, l'interrogazione (P NEAR Q) è anch'essa una variante della fusione delle liste L_P e L_Q, come discutiamo ora in dettaglio a scopo illustrativo (le interrogazioni AND e OR sono trattate in modo analogo): a tale scopo, ipotizziamo che le coppie in L_P e L_Q siano in ordine lessicografico crescente.

Il Codice 4.10 descrive come procedere per trovare le occorrenze di due termini P e Q che distano tra di loro per al più maxPos posizioni, facendo uso del Codice 4.11 per verificare tale condizione evitando di esaminare tutte le $O(|L_P| \times |L_Q|)$ coppie di occorrenze: le occorrenze restituite in uscita sono delle triple composte da T, i e j, dove $0 \le j - i \le$ maxPos, a indicare che $T[i, i+|P|-1] = P$ e $T[j, j+|Q|-1] = Q$ oppure $T[i, i+|Q|-1] = Q$ e $T[j, j+|P|-1] = P$. Notiamo che tali triple sono fornite

in uscita in ordine lessicografico da VerificaNear, considerando nell'ordinamento delle triple prima il valore di T, poi il minimo tra i e j e poi il loro massimo: tale ordine lessicografico permette di esaminare ogni testo in ordine di identificatore e di scorrere le occorrenze al suo interno riportate nell'ordine simulato di scansione del testo, fornendo quindi un utile formato di uscita all'utente dell'interrogazione (in quanto non deve saltare da una parte all'altra del testo).

Codice 4.10 Algoritmo per la risoluzione dell'interrogazione (P NEAR Q), in cui specifichiamo anche il massimo numero di posizioni in cui P e Q possono distare.

```
1  InterrogazioneNear( P, Q, maxPos ):                    ⟨pre: maxPos ≥ 0⟩
2    LP = Ricerca( dizionarioListeInvertite, P );
3    LQ = Ricerca( dizionarioListeInvertite, Q );
4    IF (LP != null && LQ != null) {
5      listaP = LP.sat.inizio;
6      listaQ = LQ.sat.inizio;
7      WHILE (listaP != null && listaQ != null) {
8        <testoP, posP> = listaP.dato;
9        <testoQ, posQ> = listaQ.dato;
10       IF (testoP < testoQ) {
11         listaP = listaP.succ;
12       } ELSE IF (testoP > testoQ) {
13         listaQ = listaQ.succ;
14       } ELSE IF (posP <= posQ) {
15         VerificaNear( listaP, listaQ, maxPos );
16         listaP = listaP.succ;
17       } ELSE {
18         VerificaNear( listaQ, listaP, maxPos );
19         listaQ = listaQ.succ;
20       }
21     }
22   }
```

La funzione InterrogazioneNear nel Codice 4.10 provvede quindi a cercare P e Q nel vocabolario utilizzando il dizionario e supponendo che i testi T nella collezione sia identificati da interi (righe 2 e 3). Nel caso che le liste invertite L_P e L_Q siano entrambe non vuote, il codice provvede a scandirle come se volesse eseguire una fusione di tali liste (righe 4-22): ricordiamo che ciascuna lista è composta da una coppia che memorizza l'identificatore del testo e la posizione all'interno del testo dell'occorrenza del termine corrispettivo (P o Q).

Di conseguenza, se i testi sono diversi nel nodo corrente delle due liste, la funzione avanza di una posizione sulla lista che contiene il testo con identificatore minore (righe 10-13). Altrimenti, i testi sono i medesimi per cui essa deve produrre

tutte le occorrenze vicine usando VerificaNear e distinguendo il caso in cui l'occorrenza di P sia precedente a quella di Q o viceversa (righe 14-19).

Codice 4.11 Algoritmo di verifica di vicinanza per l'interrogazione (P NEAR Q).

```
1  VerificaNear( listaX, listaY, M ):                    ⟨pre: posX ≤ posY⟩
2    <testoX, posX> = listaX.dato;
3    <testoY, posY> = listaY.dato;
4    WHILE (listaY != null && testoX == testoY && posY-posX <= M) {
5      PRINT testoX, posX, posY;
6      listaY = listaY.succ;
7      <testoY, posY> = listaY.dato;
8    }
```

ESEMPIO 4.6

Consideriamo il dizionario delle liste invertite dell'Esempio 5.5 ed eseguiamo InterrogazioneNear cercando i termini la e panca a distanza al più 6. Dopo aver individuato le liste relative ai due termini (righe 5-6), poiché il testo T è lo stesso per tutti i termini del dizionario, viene eseguita la riga 15 in quanto la prima occorrenza del termine la precede la prima occorrenza del termine panca. La funzione VerificaNear cerca nella lista invertita relativa a panca tutte le occorrenze a distanza al più 6 dalla prima occorrenza di la: in questo caso stampa solo la tripla T, 6, 9 ed esce perché la successiva occorrenza del temine panca (posizione 40) è a distanza maggiore di 6 dal termine la. Quando il controllo torna alla funzione InterrogazioneNear, viene fatto avanzare il puntatore sulla lista invertita relativa al primo termine la e viene invocata per la seconda volta la funzione VerificaNear: questa volta, dato che l'occorrenza attuale del termine panca precede l'occorrenza attuale del termine la, si ricercano sulla lista relativa al termine la tutte le occorrenze a distanza al più 6 dall'occorrenza attuale di panca, ovvero verrà stampata la tripla T, 9, 15. Procedendo con l'algoritmo, verranno stampate anche le triple T, 37, 40 e T, 40, 46.

Per la complessità notiamo che, se r indica il numero di triple che soddisfano l'interrogazione di vicinanza e che, quindi, sono fornite in uscita dal Codice 4.10, il tempo totale impiegato da esso è pari a $O(|L_P| + |L_Q| + r)$ e quindi dipende dalla quantità r di risultati forniti in uscita (*output sensitive*). Tale codice può essere modificato in modo da sostituire la verifica di distanza sulle posizioni con quella sulle linee del testo.

Gli altri tipi di interrogazione sono trattati in modo analogo a quella per vicinanza. Per esempio, l'interrogazione con la congiunzione (AND) calcolata come intersezione di L_P e L_Q richiede tempo $O(|L_P| + |L_Q|)$: da notare però, che se memorizziamo tali liste usando degli alberi di ricerca, possiamo effettuare l'intersezione attraverso una ricerca di ogni elemento della lista più corta tra L_P e L_Q nell'albero che memorizza la più lunga. Quindi, se $m = \min\{|L_P|, |L_Q|\}$ e $n = \max\{|L_P|, |L_Q|\}$, il costo è pari a $O(m \log n)$ e che, quando m è molto minore di n,

risulta inferiore al costo $O(m + n)$ stabilito sopra. In generale, il costo ottimo per l'intersezione è pari $O(m \log (n/m))$ che è sempre migliore sia di $O(m \log n)$ che di $O(m + n)$. Possiamo ottenere tale costo utilizzando gli alberi che permettono la *finger search*, in quanto ognuna delle m ricerche richiede $O(\log d)$ tempo invece che $O(\log n)$ tempo, dove $d < n$ è il numero di chiavi che intercorrono, in ordine di visita anticipata, tra il nodo a cui perveniamo con la chiave attualmente cercata e il nodo a cui siamo pervenuti con la precedente chiave cercata: al caso pessimo, abbiamo che le m ricerche conducono a m nodi equidistanti tra loro, per cui $d = \Theta(n/m)$ massimizza tale costo cumulativo fornendo $O(m \log (n/m))$ tempo.

Tale costo motiva la strategia di risoluzione delle interrogazioni che vedono la congiunzione di $t > 2$ termini (invece che di soli due): prima ordiniamo le t liste invertite dei termini in ordine crescente di frequenza (pari alla loro lunghezza); poi, calcoliamo l'intersezione tra le prime due liste e, per $3 \le i \le t$, effettuiamo l'intersezione tra l'i-esima lista e il risultato calcolato fino a quel momento con le prime $i - 1$ liste in ordine non decrescente di frequenza. Nel considerare anche le espressioni in disgiunzione (OR), procediamo come prima per ordine di frequenza delle liste, utilizzando una stima basata sulla somma delle loro lunghezze.

Notiamo, infine, che alternativamente le liste invertite possono essere mantenute ordinate in base a un ordine o rango di importanza delle occorrenze. Se tale ordine è preservato in modo coerente in tutte le liste, possiamo procedere come sopra con il vantaggio di dover esaminare solo i primi elementi di ciascuna lista invertita, in quanto la scansione può terminare non appena viene raggiunto un numero sufficiente di occorrenze. Queste, infatti, avranno necessariamente importanza maggiore di quelle che questo metodo tralascia. In tal modo, possiamo mediamente evitare di esaminare tutti gli elementi delle liste invertite.

4.5.1 Trie o alberi digitali di ricerca

Per realizzare il vocabolario V con delle liste invertite abbiamo utilizzato finora le tabelle hash oppure gli alberi di ricerca. Nel seguito modelliamo i termini da memorizzare in V come stringhe di lunghezza variabile, ossia come sequenze di simboli o caratteri presi da un alfabeto Σ di σ simboli, dove σ è un valore prefissato che non dipende dalla lunghezza e dal numero delle stringhe prese in considerazione (per esempio, $\sigma = 256$ per l'alfabeto ASCII mentre $\sigma = 2^{32}$ per l'alfabeto UNICODE). Ne deriva che ciascuna delle operazioni di un dizionario per una stringa di m caratteri richiede tempo medio $O(m)$ con le tabelle hash oppure tempo $O(m \log n)$ con gli alberi di ricerca, dove $n = |V|$: in quest'ultimo caso, abbiamo $O(\log n)$ confronti da eseguire e ciascuno di essi richiede tempo $O(m)$.

Mostriamo come ottenere un dizionario che garantisce tempo $O(m)$ per operazione utilizzando una struttura di dati denominata trie o albero digitale di ricerca, largamente impiegata per memorizzare un insieme $S = \{a_0, a_1, \ldots, a_{n-1}\}$ di n stringhe. Il termine trie si pronuncia come la parola inglese *try* e deriva dalla

parola inglese *retrieval* utilizzata per descrivere il recupero delle informazioni. I trie hanno innumerevoli applicazioni in quanto permettono di estendere la ricerca in un dizionario ai prefissi della stringa cercata: in altre parole, oltre a verificare se una data stringa `P` appare in `S`, essi permettono di trovare il più lungo **prefisso** di `P` che appare come prefisso di una delle stringhe in `S` (tale operazione non è immediata con le tabelle hash e con gli alberi di ricerca). A tal fine, definiamo il prefisso `i` della stringa `P` di lunghezza `m`, come la sua parte iniziale `P[0, ..., i]`, dove $0 \leq i \leq m - 1$. Per esempio, la ricerca del prefisso `ve` nell'insieme `S` composto dai nomi di alcune province italiane, ovvero `catania`, `catanzaro`, `pisa`, `pistoia`, `verbania`, `vercelli` e `verona`, fornisce come risultato le stringhe `verbania`, `vercelli` e `verona`.

Il **trie** per un insieme $S = \{a_0, a_1, \ldots, a_{n-1}\}$ di `n` stringhe definite su un alfabeto Σ è un albero cardinale σ-ario (Paragrafo 1.4.2), i cui nodi hanno σ figli (vuoti o meno), e la cui seguente definizione è ricorsiva in termini di `S`:

1. se l'insieme `S` delle stringhe è vuoto, il trie corrispondente a `S` è vuoto e viene rappresentato con `null`;
2. se `S` non è vuoto, sia S_c l'insieme delle stringhe in `S` aventi `c` come carattere iniziale, dove $c \in \Sigma$ (ai soli fini della definizione ricorsiva del trie, il carattere iniziale `c` comune alle stringhe in S_c viene ignorato e temporaneamente rimosso): il trie corrispondente a `S` è quindi composto da un nodo `u` chiamato radice tale che `u.figlio[c]` memorizza il trie ricorsivamente definito per S_c (alcuni dei figli possono essere vuoti).

Per poter realizzare un dizionario, ciascun nodo `u` di un trie è dotato di un campo `u.dato` in cui memorizzare un elemento `e`: ricordiamo che `e` ha un campo di ricerca `e.chiave`, che in questo scenario è una stringa, e un campo `e.sat` per i dati satellite, come specificato in precedenza. Mostriamo, nella parte sinistra della Figura 4.4, il trie corrispondente all'insieme `S` di stringhe composto dai sette nomi di provincia menzionati precedentemente.

Ogni stringa in `S` corrisponde a un nodo `u` del trie ed è quindi memorizzata nel campo `u.dato.chiave` (e gli eventuali dati satellite sono in `u.dato.sat`): per esempio, la foglia 5 corrisponde a `vercelli` come mostrato nella Figura 4.4, dove le foglie del trie sono numerate da `0` a `6`, e la foglia numero `i` memorizza il nome della `(i+1)`-esima provincia ($0 \leq i \leq 6$).

In generale, estendendo tutte le stringhe con un simbolo speciale, da appendere in fondo a esse come terminatore di stringa, e memorizzando le stringhe così estese nel trie, otteniamo la proprietà che queste stringhe sono associate in modo univoco alle foglie del trie (questa estensione non è necessaria se nessuna stringa in `S` è a sua volta prefisso di un'altra stringa in `S`).

Notiamo che l'altezza di un trie è data dalla lunghezza massima tra le stringhe. Nel nostro esempio, l'altezza del trie è 9 in quanto la stringa più lunga nell'insieme è `catanzaro`. Potando i sottoalberi che portano a una sola foglia, come mostrato

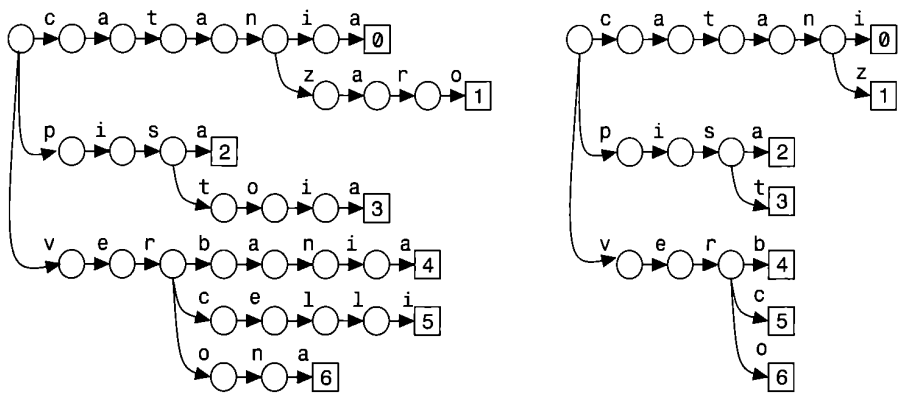


Figura 4.4 Trie per i nomi di alcune province (a sinistra) e sua versione potata (a destra).

nella parte destra della Figura 4.4, l'altezza media non dipende dalla lunghezza delle stringhe, ma è limitata superiormente da $2 \log_\sigma n + O(1)$, dove `n` è il numero di stringhe nell'insieme `S`. Tale potatura presume che le stringhe siano memorizzate altrove, in modo da poter ricostruire il sottoalbero potato in quanto è un filamento di nodi contenente la sequenza di caratteri finali di una delle stringhe (alternativamente, tali caratteri possono essere memorizzati nella foglia ottenuta dalla potatura).

Per quanto riguarda la dimensione del trie, indicando con `N` la lunghezza totale delle `n` stringhe memorizzate in `S`, ovvero la somma delle loro lunghezze, abbiamo che la dimensione di un trie è al più `N + 1`. Tale valore viene raggiunto quando ciascuna stringa in `S` ha il primo carattere differente da quello delle altre, per cui il trie è composto da una radice e poi da $|S|$ filamenti di nodi, ciascun filamento corrispondente a una stringa. Tuttavia, se si adotta la versione potata del trie, la dimensione al caso medio non dipende dalla lunghezza delle stringhe e vale all'incirca `1,44n`.

Come possiamo notare dall'esempio nella Figura 4.4, i nodi del trie rappresentano prefissi delle stringhe memorizzate, e le stringhe che sono memorizzate nei nodi discendenti da un nodo `u` hanno in comune il prefisso associato a `u`. Nel nostro esempio, le foglie discendenti dal nodo che memorizza il prefisso `pi` hanno associate le stringhe `pisa` e `pistoia`.

Sfruttiamo questa proprietà per implementare in modo semplice la ricerca di una stringa di lunghezza `m` in un trie utilizzando la sua definizione ricorsiva, come mostrato nel Codice 4.12: partiamo dalla radice per decidere quale figlio scegliere a livello `i = 0` e, al generico passo in cui dobbiamo scegliere un figlio a livello `i` del nodo corrente `u`, esaminiamo il carattere `P[i]` della stringa da cercare. Osserviamo che, se il corrispondente figlio non è vuoto, continuiamo la ricerca portando il livello a `i + 1`. Se invece tale figlio è vuoto, possiamo concludere che `P` non è

memorizzata nel dizionario. Quando `i = m`, abbiamo esaminato tutta la stringa `P` con successo pervenendo al nodo `u` di cui restituiamo l'elemento contenuto nel campo `u.dato`: in tal caso, osserviamo che `P` è memorizzato nel dizionario se e solo se tale campo è diverso da `null`.

Codice 4.12 Algoritmo di ricerca in un trie.

```
1  Ricerca( radiceTrie, P ):          ⟨pre: P è una stringa di lunghezza m⟩
2    u = radiceTrie;
3    FOR (i = 0; i < m; i = i+1) {
4      IF (u.figlio[ P[i] ] != null) {
5        u = u.figlio[ P[i] ];
6      } ELSE {
7        RETURN null;
8      }
9    }
10   RETURN u.dato;
```

La parte interessante della ricerca mostrata nel Codice 4.12 è che, a differenza della ricerca mostrata per le tabelle hash e per gli alberi di ricerca, essa può facilmente identificare il nodo `u` che corrisponde al *più lungo* prefisso di `P` che appare nel trie: a tal fine, possiamo modificare la riga 7 del codice in modo che restituisca il nodo `u` (al posto di `null`).

Di conseguenza, tutte le stringhe in `S` che hanno tale prefisso di `P` come loro prefisso possono essere recuperate nei nodi che discendono da `u` (incluso) attraverso una semplice visita, ottenendo il Codice 4.13: notiamo che le ricerche di stringhe lunghe, quando queste ultime non compaiono nel dizionario, sono generalmente più veloci delle corrispondenti ricerche nei dizionari implementati con le tabelle hash, poiché la ricerca nei trie si ferma non appena trova un prefisso di `P` che non occorre nel trie, mentre la funzione hash è comunque calcolata sull'intera stringa prima di effettuare la ricerca.

Non è difficile analizzare il costo della ricerca, in quanto vengono visitati `O(m)` nodi: poiché ogni nodo richiede tempo costante, abbiamo `O(m)` tempo di ricerca.

Codice 4.13 Algoritmo di ricerca per prefissi in un trie (`numStringhe` è una variabile globale).

```
1  RicercaPrefissi( radiceTrie, P ):   ⟨pre: P è una stringa di lunghezza m⟩
2    u = radiceTrie;
3    fine = false;
4    FOR (i = 0; !fine && i < m; i = i+1) {
5      IF (u.figlio[ P[i] ] != null) {
6        u = u.figlio[ P[i] ];
7      } ELSE {
8        fine = true;
9      }
10   }
11   numStringhe = 0;
12   Recupera( u, elenco );
13   RETURN elenco;
```

```
1  Recupera( u, elenco ):
2  IF (u != null) {
3    IF (u.dato != null) {
4      elenco[numStringhe]= u.dato;
5      numStringhe = numStringhe + 1;
6    }
7    FOR (c = 0; c < sigma; c = c + 1)
8      Recupera( u.figlio[c], elenco );
9  }
```


ESEMPIO 4.7

Come esempio quantitativo sulla velocità di ricerca dei trie, supponiamo di volere memorizzare i codici fiscali in un trie: ricordiamo che un codice fiscale contiene 9 lettere prese dall'alfabeto A ... Z di 26 caratteri, e 7 cifre prese dall'alfabeto 0 ... 9 di 10 caratteri, per un totale di $26^9 \times 10^7$ possibili codici fiscali. Cercare un codice fiscale in un trie richiede di attraversare al più 16 nodi, *indipendentemente* dal numero di codici fiscali memorizzati nel trie, in quanto l'altezza del trie è 16. In contrasto, la ricerca binaria o quella in un albero AVL, per esempio, avrebbe una dipendenza dal numero di chiavi: nel caso estremo, memorizzando metà dei possibili codici fiscali in un array ordinato, tale ricerca richiederebbe circa $\log(26^9 \times 10^7) - 1 \geq 64$ confronti tra chiavi. Il trie è quindi una struttura di dati molto efficiente per la ricerca di sequenze.

L'inserimento di un nuovo elemento nel dizionario rappresentato con un trie segue nuovamente la sua definizione ricorsiva, come mostrato nel Codice 4.14: se il trie è vuoto viene creata una radice (righe 2-7) e, quindi, dopo aver verificato che la chiave dell'elemento non appaia nel dizionario (riga 8), il trie viene attraversato fino a trovare un figlio vuoto in cui inserire ricorsivamente il trie per il resto dei caratteri (righe 14-18) oppure il nodo esiste già ma l'elemento da inserire deve essergli associato (riga 21).

Codice 4.14 Algoritmo di inserimento di un elemento in un trie.

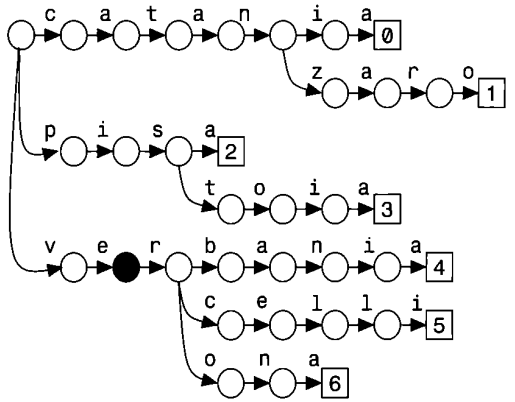
```
1  Inserisci( radiceTrie, e ):          ⟨pre: e.chiave ha lunghezza m⟩
2    IF (radiceTrie == null) {
3      radiceTrie = NuovoNodo( );
4      radiceTrie.dato = null;
5      FOR (c = 0; c < sigma; c = c + 1)
6        radiceTrie.figlio[c] = null;
7    }
8    IF (Ricerca( radiceTrie, e.chiave ) == null) {
9      u = radiceTrie;
10     FOR (i = 0; i < m; i = i+1) {
11       IF (u.figlio[ e.chiave[i] ] != null) {
12         u = u.figlio[ e.chiave[i] ];
13       } ELSE {
14         u.figlio[ e.chiave[i] ] = NuovoNodo( );
15         u = u.figlio[ e.chiave[i] ];
16         u.dato = null;
17         FOR (c = 0; c < sigma; c = c + 1)
18           u.figlio[c] = null;
19       }
20     }
21     u.dato = e;
22   }
23   RETURN radiceTrie;
```

In altre parole, l'inserimento della stringa P cerca prima il suo più lungo prefisso x che occorre in un nodo u del trie, analogamente alla procedura Ricerca, e scompone P come xy: se y non è vuoto sostituisce il sottoalbero vuoto raggiunto con la ricerca di x, mettendo al suo posto il trie per y; altrimenti, semplicemente associa P al nodo u identificato.

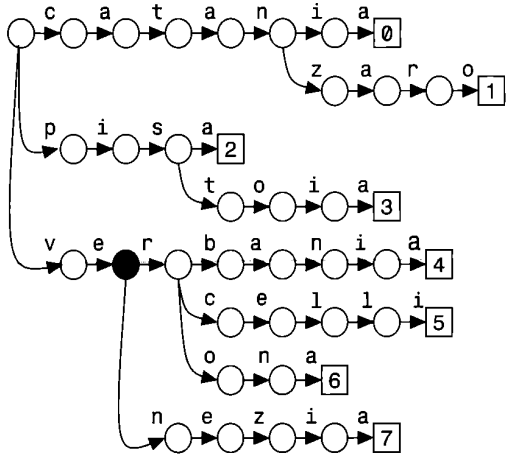
Il costo dell'inserimento è tempo O(m) in accordo a quanto discusso per la ricerca (e la cancellazione può essere trattata in modo analogo): da notare che non occorrono operazioni di ribilanciamento (rotazioni o divisioni) come succede negli alberi di ricerca. Infatti, un'importante proprietà è che la forma del trie è determinata univocamente dalle stringhe in esso contenute e non dal loro ordine di inserimento (contrariamente agli alberi di ricerca).

ESEMPIO 4.8

Consideriamo l'operazione di inserimento del termine venezia nel trie rappresentato nella parte sinistra della Figura 4.4. Dopo aver raggiunto in nodo u corrispondente al prefisso ve (rappresentato da un cerchio pieno nella figura che segue) viene riscontrato che u.figlio[n] è null (riga 11).



Quindi viene creato un nuovo filamento contenente i caratteri di nezia collegato al nodo u (righe 14-18).



Inoltre, poiché i trie preservano l'ordine lessicografico delle stringhe in esso contenute, possiamo fornire un semplice metodo per ordinare un array S di stringhe come mostrato nel Codice 4.15, in cui adoperiamo la funzione Recupera del Codice 4.13 per effettuare una visita anticipata del trie costruito attraverso l'inserimento iterativo delle stringhe contenute nell'array S.

Codice 4.15 Algoritmo di ordinamento di stringhe (fa uso di una variabile globale numStringhe e del Codice 4.13).

```
OrdinaLessicograficamente( S ):          ⟨pre: S è un array di n stringhe⟩
  radiceTrie = null;
  elemento.sat = null;
  FOR (i = 0; i < n; i = i+1) {
    elemento.chiave = S[i];
    radiceTrie = Inserisci( radiceTrie, elemento );
  }
  numStringhe = 0;
  Recupera( radiceTrie, S );
  RETURN S;
```

La complessità dell'ordinamento proposto nel Codice 4.15 è $O(N)$ tempo se la somma delle lunghezze delle n stringhe in S è pari a N. Un algoritmo ottimo per confronti, come lo heapsort, impiegherebbe $O(n \log n)$ confronti, dove però ciascun confronto richiederebbe di accedere mediamente a N/n caratteri di una stringa, per un totale di $O\left(\frac{N}{n} n \log n\right) = O(N \log n)$ tempo: l'ordinamento di stringhe con un trie è quindi più efficiente in tal caso. Analogamente a quanto discusso per la ricerca in tabelle hash, il limite così ottenuto non contraddice il limite inferiore (in questo caso sull'ordinamento) poiché i caratteri negli elementi da ordinare sono utilizzati per altre operazioni oltre ai confronti.

ESEMPIO 4.9

La visita anticipata del trie finale dell'Esempio 5.8 produce la seguente sequenza: catania, catanzaro, pisa, pistoia, venezia, verbania, vercelli e verona. Si osservi che la visita anticipata, una volta giunta sul nodo nero, segue prima l'arco etichettato con n e poi quello etichettato con r.

Nati per ricerche come quelle discusse finora, i trie sono talmente flessibili da risultare utili per altri tipi di ricerche più complesse. Inoltre, sono utilizzati nella compressione dei dati, nei compilatori e negli analizzatori sintattici. Servono a completare termini specificati solo in parte; per esempio, i comandi nella shell, le parole nella composizione dei testi, i numeri telefonici e i messaggi SMS nei telefoni cellulari, gli indirizzi del Web o della posta elettronica. Permettono la realizzazione efficiente di correttori ortografici, di analizzatori di linguaggio naturale e di sistemi per il recupero di informazioni mediante basi di conoscenza. Forniscono operazioni di ricerca più complesse di quella per prefissi, come la ricerca con espressioni regolari e con errori. Permettono di individuare ripetizioni nelle stringhe (utili, per esempio, nell'analisi degli stili di scrittura di vari autori)

e di recuperare le stringhe comprese in un certo intervallo. Le loro prestazioni ne hanno favorito l'impiego anche nel trattamento di dati multidimensionali, nell'elaborazione dei segnali e nelle telecomunicazioni. Per esempio sono utilmente impiegati nella codifica e decodifica dei messaggi, nella risoluzione dei conflitti nell'accesso ai canali e nell'instradamento veloce dei router in Internet. A fronte della loro duttilità, i trie hanno una struttura sorprendentemente semplice.

Purtroppo essi presentano alcuni svantaggi dal punto di vista dello spazio occupato per alfabeti grandi, in quanto ciascuno dei loro nodi richiede l'allocazione di un array di σ elementi: inoltre, le loro prestazioni possono peggiorare se il linguaggio di programmazione adottato non rende disponibile un accesso efficiente ai singoli caratteri delle stringhe. Esistono diverse alternative per l'effettiva rappresentazione in memoria di un trie che sono basate sulla rappresentazioni dei suoi nodi mediante strutture di dati alternative agli array come le liste, le tabelle hash e gli alberi binari.

4.5.2 Trie compatti

I trie discussi finora hanno una certa ridondanza nel numero dei nodi, in quanto quelli con un solo figlio non vuoto rappresentano una scelta obbligata e non raffinano ulteriormente la ricerca nei trie, al contrario dei nodi che hanno due o più figli non vuoti. Tale ridondanza è rimossa nel **trie compatto**, mostrato nella parte sinistra della Figura 4.5, dove i nodi con un solo figlio non vuoto sono altrimenti rappresentati per preservare le funzionalità del trie: a tal fine, gli archi sono etichettati utilizzando le sottostringhe delle chiavi appartenenti agli elementi dell'insieme S, invece che i loro singoli caratteri.

Formalmente, dato il trie per le chiavi contenute negli elementi dell'insieme S, classifichiamo un nodo del trie come **unario** se ha esattamente un figlio non vuoto. Una *catena* di nodi unari è una sequenza massimale $u_0, u_1, \ldots, u_{r-1}$ di $r \geq 2$ nodi nel trie tale che ciascun nodo u_i è unario per $1 \leq i \leq r-2$ (notiamo che u_0 potrebbe essere la radice oppure u_{r-1} potrebbe essere una foglia) e u_{i+1} è figlio di u_i per $0 \leq i \leq r-2$. Sia c_i il carattere per cui $u_i = u_{i-1}.figlio[c_i]$ e $\beta = c_1 \cdots c_{r-1}$ la sottostringa ottenuta dalla concatenazione dei caratteri incontrati percorrendo la catena da u_0 a u_{r-1}: definiamo l'operazione di *compattazione* della catena sostituendo l'intera catena con la coppia di nodi u_0 e u_{r-1} collegati da un *singolo* arco (u_0, u_{r-1}), che è concettualmente etichettato con β.

Notiamo infatti che l'esplicita memorizzazione di β non è necessaria se associamo, a ciascun nodo u, il prefisso ottenuto percorrendo il cammino dalla radice fino a u (la radice ha quindi un prefisso vuoto): in tal modo, indicando con α il prefisso nel padre di u e con γ quello in u, possiamo ricavare β per differenza in quanto $\alpha\beta = \gamma$ e la sottostringa β è data dagli ultimi $r - 1 = |\beta|$ caratteri di γ.

Il trie compatto per l'insieme S è ottenuto dal trie costruito su S applicando l'operazione di compattazione a tutte le catene presenti nel trie stesso. Ne risulta che i nodi del trie compatto sono foglie oppure hanno almeno due figli non vuoti.

Per implementare un trie compatto, estendiamo l'approccio adottato per i trie, utilizzando la rappresentazione degli alberi cardinali σ-ari (Paragrafo 1.4.2): ciascun nodo u di un trie compatto è dotato di un campo `u.dato` in cui memorizzare un elemento `e` $\in$ `S` (quindi i campi di e sono indicati come `u.dato.chiave` e `u.dato.sat`) a cui aggiungiamo un campo `u.prefisso` per memorizzare il prefisso associato a u.

Tale prefisso è memorizzato mediante una coppia $\langle$e, j$\rangle$ per indicare che esso è dato dai primi j caratteri della stringa contenuta nel campo `e.chiave`: il vantaggio è che rappresentiamo ciascun prefisso con soli due interi indipendentemente dalla lunghezza del prefisso stesso, perché lo spazio richiesto da ciascun nodo rimane $O(\sigma)$ (purché i campi chiave degli elementi siano memorizzati a parte). La parte destra della Figura 4.5 mostra un esempio di tale rappresentazione, dove possiamo osservare che un nodo interno u appare nel trie compatto se e solo se, prendendo il prefisso α associato a u, esistono almeno due caratteri $c \neq c'$ dell'alfabeto Σ tali che entrambi αc e $\alpha c'$ sono prefissi delle chiavi di alcuni elementi in S.

La presenza di due chiavi, una prefisso dell'altra, potrebbe introdurre nodi unari che non possiamo rimuovere. Per tale ragione, estendiamo tutte le chiavi degli elementi in S con un ulteriore carattere \$, che è un simbolo speciale da usare come terminatore di stringa (analogamente al carattere '\0' nel linguaggio C). In tal modo, poiché \$ appare solo in fondo alle stringhe, nessuna può essere prefisso dell'altra e, come osservato in precedenza, esiste una corrispondenza biunivoca tra le n chiavi e le n foglie del trie compatto costruito su di esse: quindi, presumiamo che ciascuna delle n foglie contenga un distinto elemento e $\in$ S (in particolare, illustriamo questa corrispondenza nella Figura 4.5 etichettando con `i` la foglia

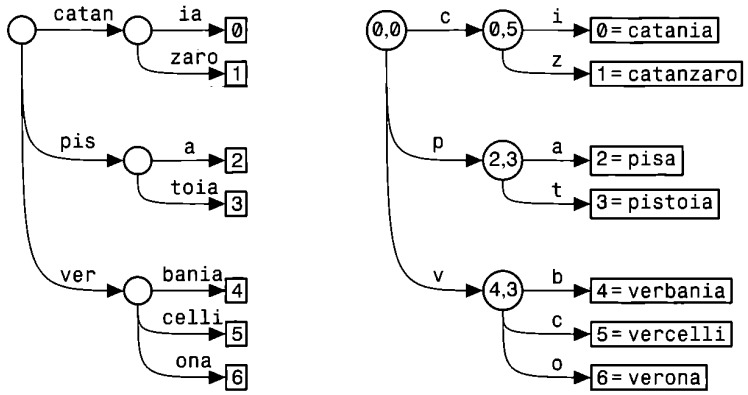


Figura 4.5 A sinistra, il trie compatto per memorizzare i nomi di alcune province; a destra, la versione con le sottostringhe rappresentate mediante coppie. Ogni foglia è associata a un elemento, identificato da un intero da 0 a n = 6, il cui campo chiave è una stringa del dizionario. Ogni nodo interno contiene la coppia $\langle$e, i$\rangle$ che rappresenta i primi i caratteri di `e.chiave`. L'etichetta i sull'arco (u, v) serve a sottolineare che v è il figlio i di u.

contenente e_i). Utilizzando il simbolo speciale e la rappresentazione dei prefissi mediante coppie, il trie compatto ha al più n nodi interni e n foglie, e quindi richiede $O(n\sigma)$ spazio, dove $\sigma = O(1)$ per le nostre ipotesi: lo spazio dipende quindi solo dal numero n delle stringhe nel caso pessimo e non dalla somma N delle loro lunghezze, contrariamente al trie.

La rappresentazione compatta di un trie non ne pregiudica le caratteristiche discusse finora. Per esempio la ricerca per prefissi in un trie compatto simula quella per prefissi in un trie (Codice 4.13) ed è mostrata nel Codice 4.16: la differenza risiede nelle righe 8-11 dove, dopo aver raggiunto il nodo u, ne prendiamo il prefisso a esso associato e ne confrontiamo i caratteri con P, dalla posizione `i` in poi, fino alla fine di una delle due stringhe oppure quando troviamo due caratteri differenti (riga 9). Analogamente alla ricerca nei trie, terminiamo di effettuare confronti quando tutti caratteri di P sono stati esaminati con successo oppure troviamo il suo più lungo prefisso che occorre nel trie compatto, e il costo del Codice 4.16 rimane pari a tempo `O(m)` più il numero di occorrenze riportate. L'operazione `Ricerca` è realizzata in modo analogo a quella per prefissi e mantiene la complessità di tempo `O(m)`.

Codice 4.16 Algoritmo di ricerca per prefissi in un trie compatto (fa uso di una variabile globale `numStringhe` e della funzione `Recupera` del Codice 4.13).

```
1  RicercaPrefissi( radiceTrieCompatto, P ):   ⟨pre: P contiene m caratteri⟩
2    u = radiceTrieCompatto;
3    fine = false;
4    i = 0;
5    WHILE (!fine && i < m) {
6      IF (u.figlio[ P[i] ] != null) {
7        u = u.figlio[ P[i] ];
8        <e, j> = u.prefisso;
9        WHILE ((i < m) && (i < j) && (P[i] == e.chiave[i]))
10         i = i + 1;
11       fine = (i < m) && (i < j);
12     } else {
13       fine = true;
14     }
15   }
16   numStringhe = 0;
17   Recupera( u, elenco );
```

Analogamente alla ricerca, l'inserimento di un nuovo elemento e nel trie compatto richiede tempo `O(m)` come mostrato nel Codice 4.17. Dopo aver creato la radice (righe 2-8), se necessario, a cui associamo il prefisso vuoto (di lunghezza 0), verifichiamo che l'elemento non sia già nel dizionario. A questo punto, procediamo

come nel caso della ricerca per prefissi per identificare il più lungo prefisso x della chiave di e che occorre nel trie compatto, dove P = xy. Sia u il nodo raggiunto e v il nodo calcolato nelle righe 11-23 e sia i = |x|: se x coincide con il prefisso associato a u, allora v = u; se invece, x è più breve del prefisso associato a u, allora v è il padre di u. Invochiamo ora `CreaFoglia`, descritta nel Codice 4.18, che fa la seguente cosa: se u ≠ v, spezza l'arco (u, v) in due creando un nodo intermedio a cui associa x come prefisso (corrispondente ai primi i caratteri di e.chiave) e a cui aggancia la nuova foglia che memorizza l'elemento e, la cui chiave ne diventa il prefisso di lunghezza m (in quanto prendiamo tutti i caratteri di e.chiave); se invece u = v, crea soltanto la nuova foglia come descritto sopra, agganciandola però a u. Ricordiamo che, non essendoci una chiave prefisso di un'altra, ogni inserimento di un nuovo elemento crea sicuramente una foglia.

Codice 4.17 Algoritmo per l'inserimento di un elemento in trie compatto.

```
1  Inserisci( radiceTrieCompatto, e ):     ⟨pre: e.chiave ha lunghezza m⟩
2    IF (radiceTrieCompatto == null) {
3      radiceTrieCompatto = NuovoNodo( );
4      radiceTrieCompatto.prefisso = <e, 0>;
5      radiceTrieCompatto.dato = null;
6      FOR (c = 0; c < sigma; c = c + 1)
7        radiceTrieCompatto.figlio[c] = null;
8    }
9    IF (Ricerca( radiceTrieCompatto, e.chiave ) == null) {
10     u = radiceTrieCompatto; fine = false; i = 0;
11     WHILE (!fine && i < m) {
12       v = u;
13       indice = i;
14       IF (u.figlio[ e.chiave[i] ] != null) {
15         u = u.figlio[ e.chiave[i] ];
16         <p, j> = u.prefisso;
17         WHILE (i < m && i < j && p.chiave[i] == e.chiave[i])
18           i = i + 1;
19         fine = (i < m) && (i < j);
20       } ELSE {
21         fine = true;
22       }
23     }
24     IF (fine) CreaFoglia( v, u, indice, i );
25   }
26   RETURN radiceTrieCompatto;
```


Codice 4.18 Algoritmo per la creazione di una foglia e di un eventuale nodo (suo padre).

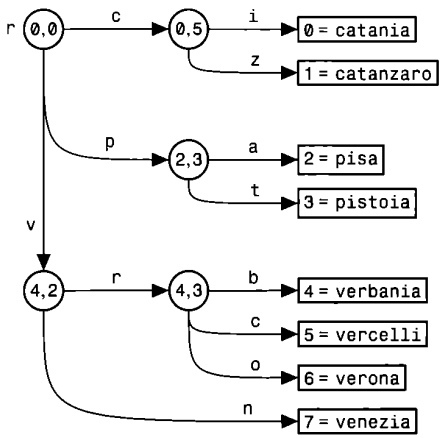
```
1  CreaFoglia ( v, u, indice, i ):          ⟨pre: e.chiave ha lunghezza m⟩
2    IF (v != u) {
3      v.figlio[ e.chiave[indice] ] = NuovoNodo( );
4      v = v.figlio[ e.chiave[indice] ];
5      v.prefisso = <e, i>;
6      v.dato = null;
7      FOR (c = 0; c < sigma; c = c + 1)
8        v.figlio[c] = null;
9      <p, j> = u.prefisso;
10     v.figlio[ p.chiave[i] ] = u;
11     u = v;
12   }
13   u.figlio[ e.chiave[i] ] = NuovoNodo( );
14   u = u.figlio[ e.chiave[i] ];
15   u.prefisso = <e, m>;
16   u.dato = e;
17   FOR (c = 0; c < sigma; c = c + 1)
18     u.figlio[c] = null;
```

Infine, la cancellazione dell'elemento avente chiave uguale a P, specificata in ingresso, richiede la rimozione della foglia raggiunta con la ricerca di P, nonché dell'arco che collega la foglia a suo padre u. Se u diventa unario, allora dobbiamo rimuovere u, il suo arco entrante e il suo arco uscente, sostituendoli con un unico arco la cui etichetta è la concatenazione della sottostringa nell'arco entrante con quella nell'arco uscente.

Tuttavia dobbiamo stare attenti a non usare elementi cancellati per i prefissi associati ai nodi u del trie compatto: se e viene cancellato e un antenato u della corrispondente foglia contiene il prefisso ⟨e, j⟩, allora è sufficiente individuare un altro elemento e′ contenuto in una foglia che discende da u e sostituire quel prefisso con ⟨e′, j⟩. Il costo della cancellazione è O(m) poiché σ = O(1).

ESEMPIO 4.10

Riprendiamo il trie della Figura 4.4 e consideriamo l'operazione di inserimento dell'elemento avente chiave venezia di lunghezza m = 7. Nel ciclo delimitato dalle righe da 11 a 23 u viene spostato su r.figlio['v'] (si veda la Figura 4.4). Ora, la coppia ⟨4, 3⟩ si riferisce ai primi 3 caratteri della stringa identificata dall'intero 4 (ovvero verbania). Il ciclo nella riga 18 calcola la lunghezza del massimo prefisso comune tra la stringa venezia e ver: questa informazione è contenuta nella variabile i. Quindi viene invocata la funzione CreaFoglia con input v = r, u = r.figlio['v'], indice = 0 e i = 2.



Questa crea un nuovo nodo figlio `'v'` della radice con campo `prefisso` dato dalla coppia $\langle 4, 2 \rangle$ (il primo elemento della coppia è l'identificativo di e mentre il secondo è il valore di `i`). Il nodo `u` diventa figlio `'r'` del nodo appena inserito (`r` è il carattere in posizione `i` di `verbania`). Infine viene creata una nuova foglia per il nuovo elemento con chiave `venezia`.

4.6 Esercizi

4.1 Discutere la complessità in tempo delle operazioni dei dizionari quando questi sono realizzati mediante array, array ordinati, liste e liste ordinate.

4.2 Mostrare come estendere i dizionari discussi nel capitolo in modo che possano gestire multi-insiemi con chiavi eventualmente ripetute.

4.3 Inserire la sequenza di chiavi `S = (5, 3, 4, 6, 7, 10)` in una tabella hash inizialmente vuota di dimensione `m = 7`, con indirizzamento aperto e sequenza di probing basata su hash doppio `Hash[i](k) = (Hash(k) + i × Hash'(k)) % m`, specificando quali funzioni `Hash` e `Hash'` si intende utilizzare.

4.4 Descrivere la cancellazione fisica da tabelle `hash` a indirizzamento aperto.

4.5 Si consideri la seguente variante della tabella hash a indirizzamento aperto e scansione lineare che utilizza due funzioni hash $\texttt{Hash}_1()$ e $\texttt{Hash}_2()$ invece che una singola funzione: per l'inserimento di una chiave `k`, se la posizione $\texttt{Hash}_1(k)$ nella tabella è libera, `k` viene inserita in tale posizione; se invece risulta occupata da un'altra chiave `k'`, allora `k` prende il posto di `k'` in tale posizione e l'inserimento continua con l'inserimento di `k'` utilizzando $\texttt{Hash}_2$ (come nella solita scansione lineare). Supponendo di utilizzare una tabella di dimensione `m`:

(a) scrivere lo pseudocodice per l'inserimento descritto sopra;

(b) discutere come cambia l'algoritmo di ricerca.

4.6 Si considerino le operazioni sugli alberi binari di ricerca.

(a) A partire da un albero binario di ricerca vuoto, simulare l'inserimento delle chiavi 15, 72, 80, 63, 54, 56, 55, 66, 64, 65 e la successiva cancellazione della chiave 63, disegnando l'albero prima e dopo tale cancellazione.

(b) Considerare la seguente modifica nell'operazione di ricerca di una chiave `k`: quando la ricerca trova un nodo `u` (diverso dalla radice) contenente la chiave `k`, fa salire `u` di un livello verso la radice usando una rotazione semplice (oraria o antioraria). Mostrare cosa succede nell'esempio precedente con la ricerca della chiave 54, indicando quale rotazione applicare al padre del nodo contenente la chiave 54.

(c) Ipotizzando di poter usare `RuotaOraria` e `RuotaAntiOraria`, scrivere lo pseudocodice per effettuare tale operazione.

4.7 Mostrare come costruire un albero binario bilanciato a partire da un array ordinato, simulando e unificando i cammini di ricerca effettuati dalla ricerca binaria di tutte le chiavi nell'array.

4.8 Estendere la ricerca in un albero binario di ricerca di altezza `h` in modo che permetta di trovare l'elemento di rango `r` in tempo `O(h)` (memorizzare in ogni nodo la dimensione del sottoalbero radicato in esso).

4.9 Discutere come realizzare l'operazione `Intervallo(k, k')` negli alberi binari di ricerca di altezza `h`: se $m \leq n$ indica il numero di chiavi così riportate, analizzare il costo mostrando che è pari a $O(h + m)$ tempo.

4.10 Si consideri un albero binario di ricerca, di altezza `h`, che memorizza un insieme di chiavi (una chiave per nodo), alcune delle quali possono essere uguali.

(a) Scrivere il codice per l'operazione di inserimento di una chiave `k`. Nel caso trovi nel nodo corrente una chiave che è uguale a `k`, l'operazione prosegue nel figlio sinistro del nodo. Il tempo di esecuzione deve essere `O(h)`.

(b) Scrivere il codice per l'operazione `conta(k)`, che restituisce il numero di occorrenze della chiave `k` nell'albero costruito mediante l'inserimento delle chiavi, come descritto nel punto (a). Il tempo di esecuzione deve essere `O(h)`.

4.11 Sia data la sequenza di chiavi `S = {7, 18, 19, 6, 5, 10}`. Inserirle in un albero AVL inizialmente vuoto, indicando a ogni inserimento l'eventuale nodo critico, e l'operazione di ribilanciamento eseguita. Infine indicare come avviene la cancellazione della chiave 7, e se questa sbilancia l'albero.

4.12 Dimostrare per induzione su h che $n_h = F_{h+3} - 1$ negli alberi di Fibonacci.

4.13 Mostrare che la complessità dell'inserimento in un AVL cambia significativamente se sostituiamo la funzione `Altezza` del Codice 4.7 con quella ricorsiva definita nel Capitolo 3.

4.14 Mostrare che, dopo aver ribilanciato tramite le rotazioni un nodo critico a seguito di un inserimento, non ci sono altri nodi critici.

4.15 Mostrare come gestire il campo `u.padre` negli alberi binari di ricerca e negli alberi AVL.

4.16 Discutere come realizzare, per i dizionari ordinati, le operazioni `Successore`, `Predecessore` e `Rango` descritte nel Paragrafo 4.1, utilizzando gli alberi AVL e i trie, analizzandone la complessità.

CAPITOLO

5 Casualità e ammortamento

In questo capitolo descriviamo due tecniche algoritmiche molto diffuse che hanno lo scopo di ottenere costi totali più bassi di quelli forniti dall'analisi al caso pessimo. La prima utilizza la casualità per ottenere strutture di dati e algoritmi randomizzati efficienti. La seconda utilizza un'analisi più raffinata su sequenze di operazioni per strutture di dati ammortizzate.

5.1 Ordinamento randomizzato per distribuzione
5.2 Dizionario basato su liste randomizzate
5.3 Unione e appartenenza a liste disgiunte
5.4 Liste ad auto-organizzazione
5.5 Tecniche di analisi ammortizzata
5.6 Esercizi

5.1 Ordinamento randomizzato per distribuzione

Nel Capitolo 3 abbiamo osservato che l'algoritmo di ordinamento per distribuzione o *quicksort* descritto nel Codice 3.5 ha una complessità che dipende dall'ordine iniziale degli elementi: se la distribuzione iniziale degli elementi è sbilanciata si ha un costo quadratico contro un costo $O(n \log n)$ nel caso di distribuzioni bilanciate.

In questo paragrafo mostreremo un'analisi al caso medio più robusta che risulta essere *indipendente* dall'ordine iniziale degli elementi nell'array e si basa sull'uso della **casualità** per far sì che la distribuzione sbilanciata occorra con una probabilità trascurabile. Concretamente, se gli `n` elementi sono già ordinati, *quicksort* richiede sempre tempo $\Theta(n^2)$ anche se in media il tempo è $O(n \log n)$ se uno considera tutti i possibili array di ingresso. Con la sua versione randomizzata, facciamo in modo che ogni volta che *quicksort* viene eseguito su uno *stesso* array in ingresso, il comportamento non sia deterministico, ma sia piuttosto dettato da scelte casuali (che comunque calcolano correttamente l'ordinamento finale). Ne deriva che nell'analisi di complessità il numero medio di passi non si calcola più su *tutti* i possibili array di ingresso, ma su *tutte* le scelte casuali per un array d'ingresso: è una nozione più forte perché uno stesso array non può essere sempre sfavorevole a *quicksort*, in quanto anche le scelte casuali di quest'ultimo adesso entrano in gioco.

A tal fine, l'unica modifica che occorre apportare all'algoritmo *quicksort* (mostrato nel Codice 5.1) è nella riga 4 e riguarda la scelta del `pivot` che deve avvenire in modo aleatorio, equiprobabile e uniforme nell'intervallo `[sinistra...destra]`: a questo scopo viene utilizzata la primitiva `random()` per generare un valore reale `r` pseudocasuale compreso tra `0` e `1` inclusi, in modo uniforme ed equiprobabile (tale generatore è disponibile in molte librerie per la programmazione e non è semplice ottenerne uno statisticamente significativo, in quanto il programma che lo genera è deterministico). Un tale algoritmo si chiama **casuale** o **randomizzato** perché impiega la casualità per sfuggire a situazioni sfavorevoli, risultando più robusto rispetto a tali eventi (come nel nostro caso, in presenza di un array già in ordine crescente).

Codice 5.1 Ordinamento randomizzato per distribuzione di un array a.

```
1  QuickSort( a, sinistra, destra ):
2                                  ⟨pre: 0 ≤ sinistra, destra ≤ n - 1⟩
3    IF (sinistra < destra) {
4      pivot = sinistra + (destra - sinistra) × random();
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```


Teorema 5.1 *L'algoritmo* quicksort *randomizzato impiega tempo ottimo* $O(n \log n)$ *nel caso medio per ordinare* `n` *elementi.*

Dimostrazione Il risultato della scelta casuale e uniforme del `pivot` è che il valore di `rango` restituito da `Distribuzione` nella riga 5 è anch'esso uniformemente distribuito tra le (equiprobabili) posizioni in `[sinistra ... destra]`. Supponiamo pertanto di dividere tale segmento in quattro parti uguali, chiamate **zone**. In base a quale zona contiene la posizione `rango` restituita nella riga 5, otteniamo i seguenti *due* eventi *equiprobabili*:

- la posizione `rango` ricade nella prima o nell'ultima zona: in tal caso, `rango` è detto essere *esterno*;
- la posizione `rango` ricade nella seconda o nella terza zona: in tal caso, `rango` è detto essere *interno*.

Indichiamo con $T(n)$ il *costo medio* dell'algoritmo *quicksort* eseguito su un array di `n` elementi. Osserviamo che la media $\frac{x+y}{2}$ di due valori `x` e `y` può essere vista come la loro somma pesata con la rispettiva probabilità $\frac{1}{2}$, ovvero $\frac{1}{2}x + \frac{1}{2}y$, considerando i due valori come equiprobabili. Nella nostra analisi, `x` e `y` sono sostituiti da opportuni valori di $T(n)$ corrispondenti ai due eventi equiprobabili sopra introdotti. Più precisamente, quando `rango` è esterno (con probabilità $\frac{1}{2}$), la distribuzione può essere estremamente sbilanciata nella ricorsione e, come abbiamo visto, quest'ultima può richiedere fino a $x = T(n-1) + O(n) \leq T(n) + c'n$ tempo, dove il termine $O(n)$ si riferisce al costo della distribuzione effettuata nel Codice 3.6 e $c' > 0$ è una costante sufficientemente grande. Quando invece `rango` è interno (con probabilità $\frac{1}{2}$), la distribuzione più sbilanciata possibile nella ricorsione avviene se `rango` corrisponde al minimo della seconda zona oppure al massimo della terza. Ne deriva una distribuzione dei dati che porta alla ricorsione su circa $\frac{n}{4}$ elementi in una chiamata di `QuickSort` e $\frac{3}{4}n$ elementi nell'altra (le altre distribuzioni in questo caso non possono andare peggio perché sono meno sbilanciate). In tal caso, la ricorsione richiede al più $y = T\left(\frac{n}{4}\right) + T\left(\frac{3}{4}n\right) + O(n) \leq T(n) + c'n$ tempo, dove la costante c' è scelta sufficientemente grande da coprire entrambi gli eventi. Facendo la media pesata di `x` e `y`, otteniamo

$$T(n) \leq \frac{1}{2}x + \frac{1}{2}y = \frac{1}{2}\left[T(n) + T\left(\frac{n}{4}\right) + T\left(\frac{3}{4}n\right)\right] + c'n \qquad (5.1)$$

Moltiplicando entrambi i termini nella (5.1) per 2, risolvendo rispetto a $T(n)$ e ponendo $c = 2c'$, otteniamo la relazione di ricorrenza

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3}{4}n\right) + cn \tag{5.2}$$

la cui soluzione dimostriamo essere $T(n) = O(n \log n)$ nel Paragrafo 5.1.1. Il tempo medio è ottimo se contiamo il numero di confronti tra elementi. ❑

5.1.1 Alternativa al teorema fondamentale delle ricorrenze

La relazione di ricorrenza (5.2) non è risolvibile con il teorema fondamentale delle ricorrenze (Teorema 3.1), in quanto non è un'istanza della (4.2). In generale, quando una relazione di ricorrenza non ricade nei casi del teorema fondamentale delle ricorrenze, occorre determinare tecniche di risoluzione alternative come più specificatamente vedremo nella dimostrazione del risultato che segue.

Teorema 5.2 *La soluzione della relazione di ricorrenza (5.2) è* $T(n) = O(n \log n)$.

Dimostrazione Notiamo che il valore $T(n)$ nella (5.2) (livello 0 della ricorsione) è ottenuto sommando a cn i valori restituiti dalle due chiamate ricorsive: quest'ultime, che costituiscono il livello 1 della ricorsione, sono invocate l'una con input $\frac{n}{4}$ e l'altra con input $\frac{3}{4}n$ e, in corrispondenza di tale livello, contribuiscono al valore $T(n)$ per un totale di $c\frac{n}{4} + c\frac{3}{4}n = cn$.

Passando al livello 2 della ricorsione, ciascuna delle chiamate del livello 1 ne genera altre due, per un totale di quattro chiamate, rispettivamente con input $\frac{n}{4^2}$, $\frac{3}{4^2}n$, $\frac{3}{4^2}n$ e $\frac{3^2}{4^2}n$, che contribuiscono al valore $T(n)$ per un totale di $c\frac{n}{4^2} + c\frac{3}{4^2}n + c\frac{3}{4^2}n + c\frac{3^2}{4^2}n = cn$ in corrispondenza del livello 2. Non ci dovrebbe sorprendere, a questo punto, che il contributo del livello 3 della ricorsione sia al più cn (in generale qualche chiamata ricorsiva può raggiungere il caso base e terminare).

Per calcolare il valore finale di $T(n)$ in forma chiusa, occorre sommare i contributi di tutti i livelli. Il livello s più profondo si presenta quando seguiamo ripetutamente il "ramo $\frac{3}{4}$", ovvero viene soddisfatta la relazione $\left(\frac{3}{4}\right)^s n = 1$ da cui deriva che $s = \log_{4/3} n = O(\log n)$. Possiamo quindi limitare superiormente $T(n)$ osservando che ciascuno degli $O(\log n)$ livelli di ricorsione contribuisce al suo valore per al più $cn = O(n)$ e, pertanto, $T(n) = O(n \log n)$. ❑

Intuitivamente, dividere n elementi in proporzione a $\frac{1}{4}$ e $\frac{3}{4}$, invece che a $\frac{1}{2}$ e $\frac{1}{2}$ (come accade nel caso dell'algoritmo di ordinamento per fusione), fornisce comunque una partizione bilanciata perché la dimensione di ciascuna parte differisce dall'altra soltanto per un fattore costante. La proprietà che $T(n) = O(n \log n)$ può essere estesa a una partizione di n in proporzione a $\frac{1}{3}$ e $\frac{2}{3}$ e, in generale, in proporzione a δ e $1 - \delta$ per una qualunque costante $0 < \delta < 1$.

Da quanto discusso finora, possiamo dedurre una linea guida per lo sviluppo di una forma chiusa della soluzione $T(n)$ di una relazione di ricorrenza del tipo

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ T(\delta n) + T((1-\delta)n) + cf(n) & \text{altrimenti} \end{cases} \tag{5.3}$$

una qualunque costante $0 < \delta < 1$. Non potendo applicare il teorema fondamentale delle ricorrenze, procediamo per passaggi intermedi con le corrispondenti chiamate ricorsive. La chiamata ricorsiva iniziale (livello 0) con input n contribuisce al valore di $T(n)$ per un totale di $cf(n)$ e dà luogo a due chiamate ricorsive di livello 1, una con input $n' = \delta n$ e l'altra con input $n'' = (1-\delta)n$, dove $n' + n'' = n$. Queste ultime due chiamate contribuiscono per un totale di $cf(n') + cf(n'')$ e inoltre invocano ulteriori due chiamate ricorsive a testa, le quali costituiscono il livello 2 della ricorsione e ricevono in input m_0, m_1, m_2 e m_3 tali che $m_0 + m_1 + m_2 + m_3 = n$. Dovrebbe essere chiaro a questo punto che queste chiamate contribuiscono per un totale di $cf(m_0) + cf(m_1) + cf(m_2) + cf(m_3)$ e inoltre invocano ulteriori due chiamate ricorsive a testa. La forma chiusa per un limite superiore della (5.3) è data dalla somma dei termini noti

$$T(n) \leq cf(n) + [cf(n') + cf(n'')] + [cf(m_0) + cf(m_1) + cf(m_2) + cf(m_3)] + \cdots$$

la cui valutazione dipende dal tipo della funzione $f(n)$: per esempio, abbiamo visto che se $f(n) = n$, allora otteniamo $T(n) = O(n \log n)$. Siccome alcune chiamate potrebbero terminare prima, abbiamo che la somma dei termini noti rappresenta un limite superiore.

Esercizio svolto 5.1 Consideriamo la funzione `QuickSelect` riportata nel Codice 3.7 del Capitolo 3: rendiamola randomizzata operando la scelta del `pivot` come avviene nella riga 4 del Codice 5.1. Mostrare che la media del costo nel caso della `QuickSelect` è $O(n)$.

Soluzione L'equazione di ricorrenza per il costo al caso medio è costruita in modo simile all'equazione (5.1), con la differenza che conteggiamo una sola chiamata ricorsiva (la più sbilanciata) ottenendo $T(n) \leq \frac{1}{2}\left[T(n) + T\left(\frac{3}{4}n\right)\right] + c'n$.

Moltiplicando entrambi i termini per 2, risolvendo rispetto a $T(n)$ e sostituendo $c = 2c'$, otteniamo la relazione di ricorrenza $T(n) \leq T\left(\frac{3}{4}n\right) + cn$. Possiamo applicare il teorema fondamentale delle ricorrenze ponendo $\alpha = 1$, $\beta = \frac{4}{3}$ e $f(n) = n$ nell'equazione (4.2), per cui rientriamo nel primo caso $\left(\gamma = \frac{3}{4}\right)$ ottenendo in media una complessità temporale $O(n)$ per l'algoritmo random di selezione per distribuzione. (Osserviamo che esiste un algoritmo lineare al caso pessimo, ma di interesse più teorico.)

5.2 Dizionario basato su liste randomizzate

Abbiamo discusso nel Paragrafo 5.1 come la casualità possa essere applicata all'algoritmo di ordinamento per distribuzione (*quicksort*) nella scelta del `pivot`. Una configurazione sfavorevole dei dati o della sequenza di operazioni che agisce su di essi può rendere inefficiente diversi algoritmi se analizziamo la loro complessità nel caso pessimo. In generale, la strategia che consente di individuare le configurazioni che peggiorano le prestazioni di un algoritmo è chiamata *avversariale* in quanto suppone che un avversario malizioso generi tali configurazioni sfavorevoli in modo continuo. In tale contesto, la casualità riveste un ruolo rilevante per la sua caratteristica imprevedibilità: vogliamo sfruttare quest'ultima a nostro vantaggio, impedendo a un tale avversario di prevedere le configurazioni sfavorevoli (in senso algoritmico).

Nel seguito descriviamo un algoritmo random per il problema dell'inserimento e della ricerca di una chiave k in una lista e dimostriamo che la strategia da esso adottata è vincente, sotto opportune condizioni. In particolare, usando una lista randomizzata di n elementi ordinati, i tempi *medi* o *attesi* delle operazioni di ricerca e inserimento sono ridotti a $O(\log n)$: anche se, al caso pessimo, tali operazioni possono richiedere tempo $O(n)$, è altamente improbabile che ciò accada.

Descriviamo una particolare realizzazione del dizionario mediante liste randomizzate, chiamate **liste a salti** (*skip list*), la cui idea di base (non random) può essere riassunta nel modo seguente, secondo quanto illustrato nella Figura 5.1. Partiamo da una lista *ordinata* di $n + 2$ elementi, $L_0 = e_0, e_1, \ldots, e_{n+1}$, la quale costituisce il livello 0 della lista a salti: poniamo che il primo e l'ultimo elemento della lista siano i due valori speciali, $-\infty$ e $+\infty$, per cui vale sempre $-\infty < e_i < +\infty$, per $1 \leq i \leq n$. Per ogni elemento e_i della lista L_0 $(1 \leq i \leq n)$ creiamo r_i copie di e_i, dove 2^{r_i} è la massima potenza di 2 che divide i (nel nostro esempio, per $i = 1, 2, 3, 4, 5, 6, 7$, abbiamo $r_i = 0, 1, 0, 2, 0, 1, 0$). Ciascuna copia ha livello crescente $\ell = 1, 2, \ldots, r_i$ e punta alla copia di livello inferiore $\ell - 1$: supponiamo inoltre che $-\infty$ e $+\infty$ abbiano sempre una copia per ogni livello creato. Chiaramente, il massimo livello o *altezza* h della lista a salti è dato dal massimo valore di r_i e, quindi, $h = O(\log n)$.

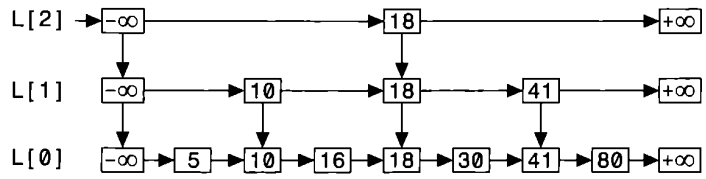


Figura 5.1 Un esempio di lista a salti di altezza $h = 2$.

Passando a una visione orizzontale, tutte le copie dello stesso livello ℓ $(0 \leq \ell \leq h)$ sono collegate e formano una sottolista L_ℓ, tale che $L_h \subseteq L_{h-1} \subseteq \cdots \subseteq L_0$:[1] come possiamo vedere nell'esempio mostrato nella Figura 5.1, le liste dei livelli superiori "saltellano" (*skip* in inglese) su quelle dei livelli inferiori. Osserviamo che, se la lista di partenza, L_0, contiene $n + 2$ elementi ordinati, allora L_1 ne contiene al più $2 + n/2$, L_2 ne contiene al più $2 + n/4$ e, in generale, L_ℓ contiene al più $2 + n/2^\ell$ elementi ordinati. Pertanto, il numero totale di copie presenti nella lista a salti è limitato superiormente dal seguente valore:

$$(2 + n) + (2 + n/2) + \cdots + (2 + n/2^h) = 2(h + 1) + \sum_{\ell=0}^{h} n/2^\ell$$
$$= 2(h + 1) + n\sum_{\ell=0}^{h} 1/2^\ell < 2(h + 1) + 2n$$

In altre parole, il numero totale di copie è $O(n)$.

Per descrivere le operazioni di ricerca e inserimento, necessitiamo della nozione di predecessore. Data una lista $L_\ell = e'_0, e'_1, \ldots, e'_{m-1}$ di elementi ordinati e un elemento x, diciamo che $e'_j \in L_\ell$ (con $0 \leq j < m - 1$) è il **predecessore** di x (in L_ℓ) se e'_j è il massimo tra i minoranti di x, ovvero $e'_j \leq x < e'_{j+1}$: osserviamo che il predecessore è sempre ben definito perché il primo elemento di L_ℓ è $-\infty$ e l'ultimo elemento è $+\infty$.

ESEMPIO 5.1

La ricerca di una chiave k è concettualmente semplice. Per esempio, supponiamo di voler cercare la chiave 80 nella lista mostrata nella Figura 5.1. Partendo da L_2, troviamo che il predecessore di 80 in L_2 è 18: a questo punto, passiamo alla copia di 18 nella lista L_1 e troviamo che il predecessore di 80 in quest'ultima lista è 41. Passando alla copia di 41 in L_0, troviamo il predecessore di 80 in questa lista, ovvero 80 stesso: pertanto, la chiave è stata trovata.

Tale modo di procedere è mostrato nel Codice 5.2, in cui supponiamo che gli elementi in ciascuna lista L_ℓ per $\ell > 0$ abbiamo un riferimento `inf` per raggiungere la corrispondente copia nella lista inferiore $L_{\ell-1}$. Partiamo dalla lista L_h (riga 2) e tro-

[1] Con un piccolo abuso di notazione, scriviamo $L \subseteq L'$ se l'insieme degli elementi di L è un sottoinsieme di quello degli elementi di L'.

viamo il predecessore p_h di k in tale lista (righe 4-6). Poiché $L_h \subseteq L_{h-1}$, possiamo raggiungere la copia di p_h in L_{h-1} (riga 7) e, a partire da questa posizione, scandire quest'ultima lista in avanti per trovare il predecessore p_{h-1} di k in L_{h-1}. Ripetiamo questo procedimento per tutti i livelli ℓ a decrescere: partendo dal predecessore p_ℓ di k in L_ℓ, raggiungiamo la sua copia in $L_{\ell-1}$, e percorriamo quest'ultima lista in avanti per trovare il predecessore $p_{\ell-1}$ di k in $L_{\ell-1}$ (righe 3-8). Quando raggiungiamo L_0 (ovvero, p è uguale a `null` nella riga 3), la variabile `predecessore` memorizza p_0, che è il predecessore che avremmo trovato se avessimo scandito L_0 dall'inizio di tale lista.

Codice 5.2 Scansione di una lista a salti per la ricerca di una chiave k.

```
1  ScansioneSkipList(k):        ⟨pre: gli elementi −∞ e +∞ fungono da sentinelle⟩
2    p = L[h];
3    WHILE (p != null) {
4      WHILE (p.succ.key <= k)
5        p = p.succ;
6      predecessore = p;
7      p = p.inf;
8    }
9    RETURN predecessore;
```

Il lettore attento avrà certamente notato che l'algoritmo di ricerca realizzato dal Codice 5.2 è molto simile alla ricerca binaria descritta nel caso degli array (Paragrafo 3.3): in effetti, ogni movimento seguendo il campo `succ` corrisponde a dimezzare la porzione di sequenza su cui proseguire la ricerca. Per questo motivo, è facile dimostrare che il costo della ricerca effettuata dal Codice 5.2 è $O(\log n)$ tempo, contrariamente al tempo $O(n)$ richiesto dalla scansione sequenziale di L_0.

Il problema sorge con l'operazione di inserimento, la cui realizzazione ricalca l'algoritmo di ricerca. Una volta trovata la posizione in cui inserire la nuova chiave, però, l'inserimento vero e proprio risulterebbe essere troppo costoso se volessimo continuare a mantenere le proprietà della lista a salti descritte in precedenza, in quanto questo potrebbe voler dire modificare le copie di tutti gli elementi che seguono la chiave appena inserita. Per far fronte a questo problema, usiamo la casualità: il risultato sarà un algoritmo random di inserimento nella lista a salti che non garantisce la struttura perfettamente bilanciata della lista stessa, ma che con alta probabilità continua a mantenere un'altezza media logaritmica e un tempo medio di esecuzione di una ricerca anch'esso logaritmico.

Notiamo che, senza perdita di generalità, la casualità può essere vista come l'esito di una sequenza di lanci di una moneta equiprobabile, dove ciascun lancio ha una possibilità su due che esca testa (codificata con 1) e una possibilità su due che esca croce (codificata con 0). Precisamente, diremo che la probabilità

di ottenere 1 è $q = \frac{1}{2}$ e la probabilità di ottenere 0 è $1 - q = \frac{1}{2}$ (in generale, un truffaldino potrebbe darci una moneta per cui $q \neq \frac{1}{2}$).

Attraverso una sequenza di b lanci, possiamo ottenere una **sequenza random** di b bit casuali.[2] Ciascun lancio è nella pratica simulato mediante la chiamata alla primitiva `random()`: il numero r generato pseudo-casualmente fornisce quindi il bit 0 se $0 \leq r < \frac{1}{2}$ e il bit 1 se $\frac{1}{2} \leq r < 1$.

Osserviamo che i lanci di moneta sono eseguiti in modo indipendente, per cui otteniamo una delle quattro possibili sequenze di $b = 2$ bit (00, 01, 10 oppure 11) in modo casuale, con probabilità $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$: in generale, le probabilità dei lanci si moltiplicano in quanto sono eventi indipendenti, ottenendo una sequenza di b bit casuali con probabilità $1/2^b$. Osserviamo inoltre che prima o poi dobbiamo incontrare un 1 nella sequenza se b è sufficientemente grande.

Nel Codice 5.3 utilizziamo tale concetto di casualità per inserire una chiave k in una lista a salti di altezza h. Una volta identificati i suoi predecessori $p_0, p_1, \ldots, p_h$, in maniera analoga a quanto descritto per l'operazione di ricerca, li memorizziamo in un vettore `pred` (riga 2). Eseguiamo quindi una sequenza di $r \geq 1$ lanci di moneta fermandoci se otteniamo 1 oppure se $r = h + 1$ (riga 3). Se $r = h + 1$, dobbiamo incrementare l'altezza (righe 5-9): creiamo una nuova lista L_{h+1} composta dalle chiavi $-\infty$ e $+\infty$, indichiamo il primo elemento di tale lista come predecessore p_{h+1} della chiave k e incrementiamo il valore di h. In ogni caso, creiamo r copie di k e le inseriamo nelle liste $L_0, L_1, L_2, \ldots, L_r$ (righe 10-13): ciascuna inserzione richiede tempo costante in quanto va creato un nodo immediatamente dopo ciascuno dei predecessori $p_0, p_1, \ldots, p_r$. Come vedremo, il costo totale dell'operazione è, in media, $O(\log n)$.

Codice 5.3 Inserimento di una chiave in una lista a salti L, dove nuovo rappresenta un elemento allocato a ogni iterazione.

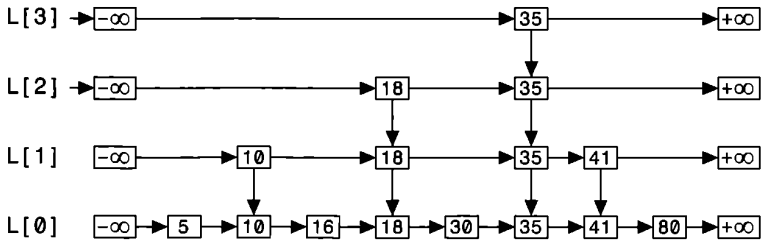
```
1  InserimentoSkipList( k ):
2    pred = [p0,p1,...,ph];
3    FOR (r = 1; r <= h && random() < 0.5; r = r + 1)
4      ;
5    IF (r > h) {
6      piu.chiave = +∞; piu.succ = piu.inf = null;
7      meno.chiave = -∞; meno.succ = piu; meno.inf = L[h];
```

[2] La nozione di sequenza random R è stata formalizzata nella teoria di Kolmogorov in termini di incompressibilità, per cui qualunque programma che generi R non può richiedere significativamente meno bit per la sua descrizione di quanti ne contenga R. Per esempio, R = 010101···01 non è casuale in quanto un programma che scrive per b/2 volte 01 può generarla richiedendo solo $O(\log b)$ bit per la sua descrizione. Purtroppo è indecidibile stabilire se una sequenza è random anche se la stragrande maggioranza delle sequenze binarie lo sono.

```
8     pred[h+1] = L[h+1] = meno; h = h + 1;
9   }
10  FOR (i = 0, ultimo = null; i <= r; i = i + 1) {
11    nuovo.chiave = k; nuovo.succ = pred[i].succ; nuovo.inf = ultimo;
12    ultimo = pred[i].succ = nuovo;
13  }
```

ESEMPIO 5.2

Consideriamo l'inserimento della chiave 35 nella lista a salti illustrata nella Figura 5.1, dove h = 2. Il primo elemento dell'array pred dei predecessori, pred[0], punta all'elemento con chiave 30 di L[0] mentre pred[1] e pred[2] puntano all'elemento contenente 18 di L[1] e L[2], rispettivamente. Supponiamo che dalla riga 3 risulti r = 3 = h + 1: vengono eseguite le righe 5-9 e quindi viene aggiunta una lista L[3] contenente le due chiavi sentinella $-\infty$ e $+\infty$, inizializzando pred[3] al primo elemento della lista ($-\infty$) e aggiornando h = 3. A questo punto, con le righe 10-13 viene aggiunto un elemento con chiave 35 in L[0], L[1], L[2] e L[3] poiché r = 3, utilizzando i puntatori pred[0], pred[1], pred[2] e pred[3]. La lista a salti risultante è mostrata nella figura che segue.



Teorema 5.3 *La complessità media delle operazioni di ricerca e inserimento su una lista a salti (random) è* O(log n).

Dimostrazione La complessità del Codice 5.3 è la stessa della complessità della ricerca, ovvero del Codice 5.2. Pertanto analizziamo la complessità dell'operazione di ricerca.

Iniziamo col valutare un limite superiore per l'altezza media e per il numero medio di copie create con il procedimento appena descritto. La lista di livello più basso, L_0, contiene n + 2 elementi ordinati. Per ciascun inserimento di una chiave k, indipendentemente dagli altri inserimenti abbiamo lanciato una moneta equiprobabile per decidere se L_1 debba contenere una copia di k (bit 0) o meno (bit 1): quindi L_1 contiene circa n/2 + 2 elementi (una frazione costante di n in generale), perché i lanci sono equiprobabili e all'incirca metà degli elementi in L_0 ha ottenuto 0, creando una copia in L_1, e il resto ha ottenuto 1. Ripetendo tale argomento ai livelli successivi, risulta che L_2 contiene circa n/4 + 2 elementi, L_3 ne

contiene circa n/8 + 2 e così via: in generale, L_ℓ contiene circa $n/2^\ell + 2$ elementi ordinati e, quando $\ell = h$, l'ultimo livello ne contiene un numero costante c > 0, ovvero $n/2^h + 2 = c$. Ne deriviamo che l'altezza h è in media O(log n) e, in modo analogo a quanto mostrato in precedenza, che il numero totale medio di copie è O(n) (la dimostrazione formale di tali proprietà sull'altezza e sul numero di copie richiede in realtà strumenti più sofisticati di analisi probabilistica).

Mostriamo ora che la ricerca descritta nel Codice 5.2 richiede tempo medio O(log n). Per un generico livello ℓ nella lista a salti, indichiamo con $T(\ell)$ il numero medio di elementi esaminati dall'algoritmo di scansione, a partire dalla posizione corrente nella lista L_ℓ fino a giungere al predecessore p_0 di k nella lista L_0: il costo della ricerca è quindi proporzionale a O(T(h)).

Per valutare T(h), osserviamo che il cammino di attraversamento della lista a salti segue un profilo a gradino, in cui i tratti orizzontali corrispondono a porzioni della stessa lista mentre quelli verticali al passaggio alla lista del livello inferiore. Percorriamo a ritroso tale cammino attraverso i predecessori p_0, p_1, p_2, ..., p_h, al fine di stabilire induttivamente i valori di T(0), T(1), T(2), ..., T(h) (dove T(0) = O(1), essendo già posizionati su p_0): notiamo che per $T(\ell)$ con $\ell \geq 1$, lungo il percorso (inverso) nel tratto interno a L_ℓ, abbiamo solo due possibilità rispetto all'elemento corrente $e \in L_\ell$.

1. Il percorso inverso proviene dalla copia di e nel livello inferiore (riga 7 del Codice 5.2), nella lista $L_{\ell-1}$, a cui siamo giunti con un costo medio pari a $T(\ell-1)$. Tale evento ha probabilità $\frac{1}{2}$ in quanto la copia è stata creata a seguito di un lancio della moneta che ha fornito 0.
2. Il percorso inverso proviene dall'elemento ê a destra di e in L_ℓ (riga 5 del Codice 5.2), a cui siamo giunti con un costo medio pari a $T(\ell)$: in tal caso, ê non può avere una corrispettiva copia al livello superiore (in $L_{\ell+1}$). Tale evento ha probabilità $\frac{1}{2}$, perché è il complemento dell'evento al punto 1.

Possiamo quindi esprimere il valore medio di $T(\ell)$ attraverso la media pesata (come per il *quicksort*) dei costi espressi nei casi 1 e 2, ottenendo la seguente relazione di ricorrenza per un'opportuna costante $c' > 0$:

$$T(\ell) \leq \frac{1}{2}\left[T(\ell) + T(\ell - 1)\right] + c' \tag{5.4}$$

Moltiplicando i termini della relazione (5.4), risolvendo rispetto a $T(\ell)$ e ponendo $c = 2c'$, otteniamo

$$T(\ell) \leq T(\ell - 1) + c \tag{5.5}$$

Espandendo (5.5), abbiamo $T(\ell) \leq T(\ell-1) + c \leq T(\ell-2) + 2c \leq \cdots \leq T(0) + \ell c = O(\ell)$. Quindi T(h) = O(h) = O(log n) è il costo medio della ricerca. ❑

Esercizio svolto 5.2 Discutere come realizzare l'operazione di cancellazione dalle liste randomizzate, valutandone la complessità in tempo.

Soluzione Sia k la chiave da cancellare e p_0, p_1, p_2, ..., p_h i suoi predecessori *stretti*, identificati mediante una variante della ricerca di k: notare che un predecessore p_i è stretto per k quando $p_i < k$. Dopo averli memorizzati nell'array pred come descritto all'inizio del Codice 5.3, è sufficiente eseguire `pred[i].succ = pred[i].succ.succ` per tutte le liste `L[i]` che contengono k (ossia, per cui `pred[i].succ.chiave = k`, essendo p_i un predecessore stretto). Se `L[h]` non contiene più chiavi (a parte $-\infty$ e $+\infty$), decrementiamo anche h. Il costo è dominato da quello della ricerca, quindi è $O(h) = O(\log n)$. È importante osservare che la cancellazione di k non incide sul numero di copie degli altri elementi. Per convincersene, consideriamo come vengono inseriti gli elementi: quando decidiamo il numero r di copie nel Codice 5.3, questo dipende solo dal lancio delle monete e non dagli elementi inseriti fino a quel momento. Quindi la presenza o meno della chiave k non influenza questo aspetto e le liste risultanti sono ancora randomizzate, ipotizzando che l'algoritmo di cancellazione non conosca quante copie ci sono per ciascun elemento ai fini dell'analisi probabilistica.

In conclusione, i dizionari basati su liste randomizzate sono un esempio concreto di come l'uso accorto della casualità possa portare ad algoritmi semplici che hanno in media (o con alta probabilità) ottime prestazioni in tempo e in spazio.

5.3 Unione e appartenenza a liste disgiunte

Le liste possono essere impiegate per operazioni di tipo insiemistico: avendo già visto come inserire e cancellare un elemento, siamo interessati a gestire una sequenza arbitraria S di operazioni di unione e appartenenza su un insieme di liste contenenti un totale di m elementi. In ogni istante le liste sono *disgiunte*, ossia l'intersezione di due liste qualunque è vuota. Inizialmente, abbiamo m liste, ciascuna formata da un solo elemento. Un'operazione di unione in S prende due delle liste attualmente disponibili e le concatena (non importa l'ordine di concatenazione). Un'operazione di appartenenza in S stabilisce se due elementi appartengono alla stessa lista.

Tale problema viene chiamato di *union-find* e trova applicazione, per esempio, in alcuni algoritmi su grafi che discuteremo in seguito. Mantenendo i riferimenti al primo e all'ultimo elemento di ogni lista, possiamo realizzare l'operazione di unione in tempo costante. Tuttavia, ciascuna operazione di appartenenza può richiedere tempo $O(m)$ al caso pessimo (pari alla lunghezza di una delle liste dopo una serie di unioni), totalizzando $O(nm)$ tempo per una sequenza di n operazioni.

Presentiamo un modo alternativo di implementare tali liste per eseguire un'arbitraria sequenza S di n operazioni delle quali n_1 sono operazioni di unione e

n_2 sono operazioni di appartenenza in $O(n_1 \log n_1 + n_2) = O(n \log n)$ tempo totale, migliorando notevolmente il limite di $O(nm)$ in quanto $n_1 < m$. Rappresentiamo ciascuna lista con un riferimento all'inizio e alla fine della lista stessa nonché con la sua lunghezza. Inoltre, corrediamo ogni elemento z di un riferimento `z.lista` alla propria lista di appartenenza: la regola intuitiva per mantenere tali riferimenti, quando effettuiamo un'unione tra due liste, consiste nel cambiare il riferimento `z.lista` negli elementi z della lista *più corta*. Vediamo come tale intuizione conduce a un'analisi rigorosa.

Il Codice 5.4 realizza tale semplice strategia per risolvere il problema di *union-find*, specificando l'operazione Crea per generare una lista di un solo elemento x, oltre alle funzioni Appartieni e Unisci per eseguire le operazioni di appartenenza e unione per due elementi x e y. In particolare, l'appartenenza è realizzata in tempo costante attraverso la verifica che il riferimento alla propria lista sia il medesimo. L'operazione di unione tra le due liste disgiunte degli elementi x e y determina anzitutto la lista più corta e quella più lunga (righe 2-8): cambia quindi i riferimenti `z.lista` agli elementi z della lista più corta (righe 9-13), concatena la lista lunga con quella corta (righe 14-15) e aggiorna la dimensione della lista risultante (riga 16).

Codice 5.4 Operazioni di creazione, appartenenza e unione nelle liste disgiunte.

```
1  Crea( x ):                                          ⟨pre: x non null⟩
2    lista.inizio = lista.fine = x;
3    lista.lunghezza = 1;
4    x.lista = lista;
5    x.succ = null;

1  Appartieni( x, y ):                                 ⟨pre: x, y non null⟩
2    RETURN (x.lista == y.lista);

1  Unisci( x, y ):                    ⟨pre: x, y non vuoti e x.lista ≠ y.lista⟩
2    IF (x.lista.lunghezza <= y.lista.lunghezza) {
3      corta = x.lista;
4      lunga = y.lista;
5    } ELSE {
6      corta = y.lista;
7      lunga = x.lista;
8    }
9    z = corta.inizio;
10   WHILE (z != null) {
11     z.lista = lunga;
12     z = z.succ;
13   }
```

```
14    lunga.fine.succ = corta.inizio;
15    lunga.fine = corta.fine;
16    lunga.lunghezza = corta.lunghezza + lunga.lunghezza;
```

L'efficacia della modalità di unione può essere mostrata in modo rigoroso facendo uso di un'analisi più approfondita, che prende il nome di **analisi ammortizzata** e che illustreremo in generale nel Paragrafo 5.5. Invece di valutare il costo al caso pessimo di una *singola* operazione, quest'analisi fornisce il costo al caso pessimo di una *sequenza* di operazioni, distribuendo quindi il costo delle poche operazioni costose nella sequenza sulle altre operazioni della sequenza che sono poco costose. La giustificazione di tale approccio è fornita dal fatto che, in tal modo, non ignoriamo gli effetti correlati delle operazioni sulla medesima struttura di dati. In generale, data una sequenza S di operazioni, diremo che il *costo ammortizzato* di un'operazione in S è un limite superiore al costo effettivo (spesso difficile da valutare) totalizzato dalla sequenza S diviso il numero di operazioni contenute in S. Naturalmente, più aderente al costo effettivo è tale limite, migliore è l'analisi ammortizzata fornita.

Teorema 5.4 *Il costo di* $n_1 < m$ *operazioni* `Unisci` *è* $O(n_1 \log m)$*, quindi il costo ammortizzato per operazione è* $O(\log m)$*. Il costo di ciascuna operazione* `Crea` *e* `Appartieni` *è invece* $O(1)$ *al caso pessimo.*

Dimostrazione È facile vedere che ciascuna delle operazioni `Crea` e `Appartieni` richiede tempo costante. Partendo da m elementi, ciascuno dei quali costituisce una lista di un singolo elemento (attraverso l'operazione `Crea`), possiamo concentrarci su un'arbitraria sequenza S di n_1 operazioni `Unisci`. Osserviamo che, al caso pessimo, la complessità in tempo di `Unisci` è proporzionale direttamente al numero di riferimenti `z.lista` che vengono modificati alla riga 11 del Codice 5.4: per calcolare il costo totale delle operazioni in S, è quindi sufficiente valutare un limite superiore al numero totale di riferimenti `z.lista` cambiati.

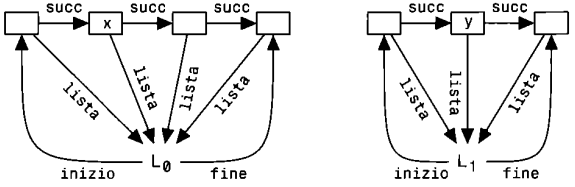
Possiamo conteggiare il numero di volte che la sequenza S può cambiare `z.lista` per un qualunque elemento z nel seguente modo. Inizialmente, l'elemento z appartiene alla lista di un solo elemento (se stesso). In un'operazione `Unisci`, *se* `z.lista` cambia, vuol dire che z va a confluire in una lista che ha una dimensione almeno *doppia* rispetto a quella di partenza. In altre parole, la prima volta che `z.lista` cambia, la dimensione della nuova lista contenente z è almeno 2, la seconda volta è almeno 4 e così via: in generale, l'i-esima volta che `z.lista` cambia, la dimensione della nuova lista contenente z è almeno 2^i. D'altra parte, al termine delle n_1 operazioni `Unisci`, la lunghezza di una qualunque lista è minore oppure uguale a $n_1 + 1 \leq m$: ne deriva che la lista contenente z ha lunghezza compresa tra 2^i e m (ovvero, $2^i \leq m$) e che vale sempre $i = O(\log m)$. Quindi, ogni elemento z vede cambiare il riferimento `z.lista` al più $O(\log m)$ volte. Sommando tale quantità per gli $n_1 + 1$ elementi z coinvolti nelle n_1 operazioni

`Unisci`, otteniamo un limite superiore di $O(n_1 \log m)$ al numero di volte che la riga 11 viene globalmente eseguita: pertanto, la complessità in tempo delle n_1 operazioni `Unisci` è $O(n_1 \log m)$ e, quindi, il costo *ammortizzato* di tale operazione è $O(\log m)$. Al costo di queste operazioni, va aggiunto il costo $O(1)$ per ciascuna delle operazioni `Crea` e `Appartieni`. ❑

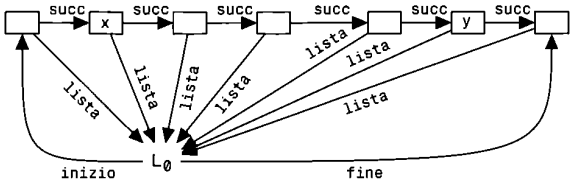
Lo schema adottato per cambiare i riferimenti `z.lista` è piuttosto generale: ipotizzando di avere insiemi disgiunti i cui elementi hanno ciascuno un'etichetta (sia essa `z.lista` o qualunque altra informazione) e applicando la regola che, quando due insiemi vengono uniti, si cambiano solo le etichette agli elementi dell'insieme di cardinalità minore, siamo sicuri che un'etichetta non possa venire cambiata più di $O(\log m)$ volte. L'intuizione di cambiare le etichette agli elementi del più piccolo dei due insiemi da unire viene rigorosamente esplicitata dall'analisi ammortizzata: notiamo che, invece, cambiando le etichette agli elementi del più grande dei due insiemi da unire, un'etichetta potrebbe venire cambiata $\Omega(n_1)$ volte, invalidando l'argomentazione finora svolta.

ESEMPIO 5.3

L'esecuzione dell'operazione `Unisci(x, y)` sulle liste mostrate nella figura che segue per prima cosa confronta la lunghezza della lista L_0 a cui appartiene x con quella della lista L_1 a cui appartiene y. Questa informazione viene reperita utilizzando i puntatori `lista` di x e y.



La lista L_0 è più lunga di L_1 quindi i puntatori `lista` degli elementi di L_1 vanno a puntare L_0. L'ultimo elemento di L_0 punta con `succ` al primo elemento di L_1 e al campo `fine` di L_0 viene assegnato il valore nel campo fine di L_1.



Infine viene aggiornato il campo `lunghezza` di L_0. La lista risultante è mostrata nella figura in alto. Si noti che non è più mostrato L_1 con i puntatori `inizio` e `fine`. Questi tuttavia continueranno a persistere ma d'ora in poi saranno ignorati.

5.4 Liste ad auto-organizzazione

L'auto-organizzazione delle liste è utile quando, per svariati motivi, la lista *non è necessariamente ordinata* in base alle chiavi di ricerca (contrariamente al caso delle liste randomizzate del Paragrafo 5.2). Per semplificare la discussione, consideriamo il solo caso della ricerca di una chiave k in una lista e adottiamo uno schema di scansione sequenziale: percorriamo la lista a partire dall'inizio verificando iterativamente se l'elemento attuale è uguale alla chiave cercata. Estendiamo tale schema per eseguire eventuali operazioni di auto-organizzazione al termine della scansione sequenziale (le operazioni di inserimento e cancellazione possono essere ottenute semplicemente, secondo quanto discusso nel Paragrafo 1.3).

Tale organizzazione sequenziale può trarre beneficio dal **principio di località temporale**, per il quale, se accediamo a un elemento in un dato istante, è molto probabile che accederemo a questo stesso elemento in istanti immediatamente (o quasi) successivi. Seguendo tale principio, sembra naturale che possiamo *riorganizzare* proficuamente gli elementi della lista dopo aver eseguito la loro scansione. Per questo motivo, una lista così gestita viene riferita come struttura di dati ad **auto-organizzazione** (*self-organizing* o *self-adjusting*). Tra le varie strategie di auto-organizzazione, la più diffusa ed efficace viene detta *move-to-front* (MTF), che consideriamo in questo paragrafo: essa consiste nello spostare l'elemento acceduto dalla sua posizione attuale alla cima della lista, senza cambiare l'ordine relativo dei rimanenti elementi, come mostrato nel Codice 5.5. Osserviamo che MTF effettua ogni ricerca senza conoscere le ricerche che dovrà effettuare in seguito: un algoritmo operante in tali condizioni, che deve quindi servire un insieme di richieste man mano che esse pervengono, viene detto **in linea** (*online*).

Codice 5.5 Ricerca di una chiave k in una lista ad auto-organizzazione.

```
1  MoveToFront( a, k ):
2    p = a;
3    IF (p == null || p.dato == k) RETURN p;
4    WHILE (p.succ != null && p.succ.dato != k)
5      p = p.succ;
6    IF (p.succ == null) RETURN null;
7    tmp = p.succ;
8    p.succ = p.succ.succ;
9    tmp.succ = a;
10   a = tmp;
11   RETURN a;
```


Un esempio quotidiano di lista ad auto-organizzazione che utilizza la strategia MTF è costituito dall'elenco delle chiamate effettuate da un telefono cellulare: in effetti, è probabile che un numero di telefono appena chiamato, venga usato nuovamente nel prossimo futuro. Un altro esempio, più informatico, è proprio dei sistemi operativi, dove la strategia MTF viene comunemente denominata *least recently used* (LRU). In questo caso, gli elementi della lista corrispondono alle pagine di memoria, di cui solo le prime r possono essere tenute in una memoria ad accesso veloce. Quando una pagina è richiesta, quest'ultima viene aggiunta alle prime r, mentre quella a cui si è fatto accesso meno recentemente viene rimossa. Quest'operazione equivale a porre la nuova pagina in cima alla lista, per cui quella originariamente in posizione r (acceduta meno recentemente) va in posizione successiva, r + 1, uscendo di fatto dall'insieme delle pagine mantenute nella memoria veloce.

Per valutare le prestazioni della strategia MTF, il termine di paragone utilizzato sarà un algoritmo **non in linea** (*offline*), denominato OPT, che ipotizziamo essere a conoscenza di *tutte* le richieste che perverranno. Le prestazioni dei due algoritmi verranno confrontate rispetto al loro costo, definito come la somma dei costi delle singole operazioni, in accordo a quanto discusso sopra. In particolare, contiamo il numero di elementi della lista attraverso cui si passa prima di raggiungere l'elemento desiderato, a partire dall'inizio della lista: quindi *accedere all'elemento in posizione* i *ha costo* i in quanto dobbiamo attraversare gli i elementi che lo precedono. Lo spostamento in cima alla lista, operato da MTF, non viene conteggiato in quanto richiede sempre un costo costante.

Tale paradigma è ben esemplificato dalla gestione delle chiamate in uscita di un telefono cellulare: l'ultimo numero chiamato è già disponibile in cima alla lista per la prossima chiamata e il costo indica il numero di *clic* sulla tastierina per accedere a ulteriori numeri chiamati precedentemente (occorre un numero di clic pari a i per scandire gli elementi che precedono l'elemento in posizione i nell'ordine inverso di chiamata).

È di fondamentale importanza stabilire le regole di azione di OPT, perché questo può dare luogo a risultati completamente differenti. Nel nostro caso, esaminate tutte le richieste in anticipo, OPT *permuta* gli elementi della lista solo una volta all'inizio, *prima* di servire le richieste. Per semplificare la nostra analisi comparativa, OPT e MTF partono con la stessa lista iniziale: quando arriva una richiesta per l'elemento k in posizione i, OPT restituisce l'elemento scandendo i primi i elementi della lista, senza però muovere k dalla sua posizione, mentre MTF lo pone in cima alla lista. Inoltre, presumiamo che le liste non cambino di lunghezza durante l'elaborazione delle richieste.

Notiamo che OPT permuta gli elementi in un ordine che rende minimo il suo costo futuro. Chiaramente un tale algoritmo dotato di chiaroveggenza non esiste, ma è utile ai fini dell'analisi per valutare le potenzialità di MTF.

A titolo esemplificativo, è utile riportare i costi in termini concreti del numero di clic effettuati sui cellulari. Immaginiamo di essere in possesso, oltre al cellulare di marca MTF, di un futuristico cellulare OPT che conosce in anticipo le n chiamate che saranno effettuate nell'arco di un anno su di esso (l'organizzazione della lista delle chiamate in uscita è mediante le omonime politiche di gestione). Potendo usare entrambi i cellulari con gli stessi m numeri in essi memorizzati, effettuiamo alcune chiamate su tali numeri per un anno: quando effettuiamo una chiamata su di un cellulare, la ripetiamo anche sull'altro (essendo futuristico, OPT si aspetta già la chiamata che intendiamo effettuare). Per la chiamata j, dove $j = 0, 1, \ldots, n-1$, contiamo il numero di clic che siamo costretti a fare per accedere al numero di interesse in MTF e, analogamente, annotiamo il numero di clic per OPT (ricordiamo che MTF pone il numero chiamato in cima alla sua lista, mentre OPT non cambia più l'ordine inizialmente adottato in base alle chiamate future). Allo scadere dell'anno, siamo interessati a stabilire il costo, ovvero il numero totale di clic effettuati su ciascuno dei due cellulari.

Mostriamo che, sotto opportune condizioni, il *costo di* MTF *non supera il doppio del costo di* OPT. In un certo senso, MTF offre una forma limitata di chiaroveggenza delle richieste rispetto a OPT, motivando il suo impiego in vari contesti con successo.

Formalmente, consideriamo una sequenza arbitraria di n operazioni di ricerca su una lista di m elementi, dove le operazioni sono numerate da 0 a $n-1$ in base al loro ordine di esecuzione. Per $0 \le j \le n-1$, l'operazione j accede a un elemento k nelle lista come nel Codice 5.5: sia c_j la posizione di k nella lista di MTF e c'_j la posizione di k nella lista di OPT. Poiché vengono scanditi c_j elementi prima di k nella lista di MTF, e c'_j elementi prima di k nella lista di OPT, definiamo il costo delle n operazioni, rispettivamente,

$$\texttt{costo(MTF)} = \sum_{j=0}^{n-1} c_j \quad \text{e} \quad \texttt{costo(OPT)} = \sum_{j=0}^{n-1} c'_j. \tag{5.6}$$

Teorema 5.5 *Partendo da liste uguali, vale* $\texttt{costo(MTF)} \le 2 \times \texttt{costo(OPT)}$.

Dimostrazione Vogliamo mostrare che

$$\sum_{j=0}^{n-1} c_j \le 2 \sum_{j=0}^{n-1} c'_j \tag{5.7}$$

quando le liste di partenza sono uguali. Da tale diseguaglianza segue che MTF scandisce asintoticamente non più del doppio degli elementi scanditi da OPT. Nel seguito proviamo una condizione più forte di quella espressa nella diseguaglianza (5.7) da cui possiamo facilmente derivare quest'ultima: a tal fine, introduciamo la nozione di **inversione**. Supponiamo di aver appena eseguito l'operazione j che accede all'elemento k, e consideriamo le risultanti liste di MTF e OPT: un esempio di configurazione delle due liste in un certo istante è quello riportato nella Figura 5.2.

Lista MTF	=	e_4	e_2	e_6	e_0	e_1	e_3	e_7	e_5
Lista OPT	=	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7

Figura 5.2 Un'istantanea delle liste manipolate da MTF e OPT.

Presi due elementi distinti x e y in una delle due liste, questi devono occorrere anche nell'altra: diciamo che l'insieme {x, y} è un'inversione quando l'ordine relativo di occorrenza è diverso nelle due liste, ovvero quando x occorre prima di y (non necessariamente in posizioni adiacenti) in una lista mentre y occorre prima di x nell'altra lista. Nel nostro esempio, $\{e_0, e_2\}$ è un'inversione, mentre $\{e_1, e_7\}$ non lo è. Definiamo con Φ_j il numero di inversioni tra le due liste dopo che è stata eseguita l'operazione j: vale $0 \le \Phi_j \le \frac{m(m-1)}{2}$, per $0 \le j \le n-1$, in quanto $\Phi_j = 0$ se le due liste sono uguali mentre, se sono una in ordine inverso rispetto all'altra, ognuno degli $\binom{m}{2}$ insiemi di due elementi è un'inversione. Per dimostrare la (5.7), non possiamo utilizzare direttamente la proprietà che $c_j \le 2c'_j + O(1)$, in quanto questa proprietà in generale non è vera. Invece, ammortizziamo il costo usando il numero di inversioni Φ_j, in modo da dimostrare la seguente relazione (introducendo un valore fittizio $\Phi_{-1} = 0$ con l'ipotesi che MTF e OPT partano da liste uguali):

$$c_j + \Phi_j - \Phi_{j-1} \le 2c'_j \tag{5.8}$$

Possiamo derivare la (5.7) dalla (5.8) in quanto quest'ultima implica che

$$\sum_{j=0}^{n-1} (c_j + \Phi_j - \Phi_{j-1}) \le 2 \sum_{j=0}^{n-1} c'_j$$

I termini Φ nella sommatoria alla sinistra della precedente diseguaglianza formano una cosiddetta **somma telescopica**, $(\Phi_0 - \Phi_{-1}) + (\Phi_1 - \Phi_0) + (\Phi_2 - \Phi_1) + \cdots + (\Phi_{n-1} - \Phi_{n-2}) = \Phi_{n-1} - \Phi_{-1} = \Phi_{n-1} \ge 0$, nella quale le coppie di termini di segno opposto si elidono algebricamente: da questa osservazione segue immediatamente che

$$\sum_{j=0}^{n-1} c_j \le 2 \sum_{j=0}^{n-1} c'_j - \Phi_{n-1} \le 2 \sum_{j=0}^{n-1} c'_j \tag{5.9}$$

ottenendo così la disuguaglianza (5.7).

Possiamo quindi concentrarci sulla dimostrazione dell'equazione (5.8), dove il caso $j = 0$ vale per sostituzione del valore fissato per $\Phi_{-1} = 0$, in quanto $\Phi_0 = c_0 = c'_0$ visto che inizialmente le due liste sono uguali e quindi dopo la prima operazione vengono create c_0 inversioni perché la chiave cercata viene spostata solo da MTF.

Ipotizziamo quindi che l'operazione $j > 0$ sia stata eseguita: a tale scopo, sia k l'elemento acceduto in seguito a tale operazione e ipotizziamo che k occupi la posizione i nella lista di MTF (per cui $c_j = i$): notiamo che la (5.8) è banalmente soddisfatta quando $i = 0$ perché la lista di MTF non cambia e, quindi, $\Phi_j = \Phi_{j-1}$.

Prendiamo l'elemento k' che appare in una generica posizione $i' < i$. Ci sono solo due possibilità se esaminiamo l'insieme $\{k', k\}$: è un'inversione oppure non lo è. Quando MTF pone k in cima alla lista, tale insieme diventa un'inversione se e solo se non lo era prima: nel nostro esempio, se $k = e_3$ (per cui $i = 5$), possiamo riscontrare che, considerando gli elementi k' nelle posizioni da 0 a 4, due di essi, e_4 ed e_6, formano (assieme a e_3) un'inversione mentre i rimanenti tre elementi non danno luogo a inversioni. Quando e_3 viene posto in cima alla lista di MTF, abbiamo che gli insiemi $\{e_3, e_4\}$ ed $\{e_3, e_6\}$ non sono più inversioni, mentre lo diventano gli insiemi $\{e_2, e_3\}$, $\{e_0, e_3\}$ e $\{e_1, e_3\}$.

In generale, gli i elementi che precedono k nella lista di MTF sono composti da f elementi che (assieme a k) *danno* luogo a inversioni e da g elementi che *non danno* luogo a inversioni, dove $f + g = i$. Dopo che MTF pone k in cima alla sua lista, il numero di inversioni che cambiano sono esclusivamente quelle che coinvolgono k. In particolare, le f inversioni *non* sono più tali mentre appaiono g *nuove* inversioni, come illustrato nel nostro esempio. Di conseguenza, pur non sapendo stimare individualmente il numero di inversioni Φ_j e Φ_{j-1}, possiamo inferire che la loro differenza dopo l'operazione j è $\Phi_j - \Phi_{j-1} = -f + g$. Ne deriva che $c_j + \Phi_j - \Phi_{j-1} = i - f + g = (f + g) - f + g = 2g$.

Consideriamo ora la posizione c'_j dell'elemento k nella lista di OPT: sappiamo certamente che $c'_j \geq g$ perché ci sono almeno g elementi che precedono k, in quanto appaiono prima di k anche nella lista di MTF e non formano inversioni con k prima dell'operazione j. A questo punto, otteniamo l'equazione (5.8), in quanto $c_j + \Phi_j - \Phi_{j-1} = 2g \leq 2c'_j$, concludendo di fatto l'analisi ammortizzata. ❑

Esercizio svolto 5.3 Siano $s_0, s_1, \ldots, s_{m-1}$ gli m elementi della lista nell'ordine iniziale stabilito dall'algoritmo OPT. Indicando con f_i il numero di volte in cui s_i viene richiesto dalla sequenza di n accessi, dove $\sum_{i=0}^{m-1} f_i = n$, mostrare che $f_0 \geq f_1 \geq \cdots \geq f_{m-1}$: cioè, mostrare che OPT organizza gli elementi della sua lista in ordine non crescente di frequenza.

Soluzione Poiché OPT paga un costo i per accedere a s_i (senza cambiare la lista) e questo accade f_i volte, possiamo derivare che $\texttt{costo(OPT)} = \sum_{i=0}^{m-1} i \times f_i$. Per assurdo, ipotizziamo che esistano $i' < i$ tali che $f_{i'} < f_i$, ossia la lista di OPT non è in ordine (non crescente di frequenza). Scambiando di posto $s_{i'}$ e s_i nella lista prima dell'esecuzione di OPT, otteniamo un costo strettamente inferiore, producendo una contraddizione sul fatto che OPT è ottimo: infatti, $i' \times f_i + i \times f_{i'} < i' \times f_{i'} + i \times f_i$.

Osserviamo che tale analisi della strategia MTF sfrutta la condizione che l'algoritmo OPT non può manipolare la lista una volta che abbia iniziato a gestire le richieste. È possibile estendere la dimostrazione del Teorema 5.5 al caso in cui *anche* OPT possa portare un elemento in cima alla lista.

In generale, il Teorema 5.5 non è più valido se permettiamo a OPT di manipolare la sua lista in altre maniere. Accedendo all'elemento in posizione i, l'algoritmo può per esempio riorganizzare la lista in tempo $O(i + 1)$: pensiamo a un impiegato con la sua pila disordinata di pratiche dove, pescata la pratica in posizione i, può metterla in cima alla pila ribaltando l'ordine delle prime i nel contempo. In tal caso, è possibile dimostrare che un algoritmo che adotta una tale strategia, denominato REV, ha un costo pari a $O(n \log n)$ mentre il costo di MTF risulta essere $\Theta(n^2)$, invalidando l'equazione (5.7) per n sufficientemente grande.

Tuttavia, MTF rimane una strategia vincente per organizzare le informazioni in base alla frequenza di accesso. L'economista giapponese Noguchi Yukio ha scritto diversi libri di successo sull'organizzazione aziendale e, tra i metodi per l'archiviazione cartacea, ne suggerisce uno particolarmente efficace. Il metodo si basa su MTF e consiste nel mettere l'oggetto dell'archiviazione (un articolo, il passaporto, le schede telefoniche e così via) in una busta di carta etichettata. Le buste sono mantenute in un ripiano lungo lo scaffale e le nuove buste vengono aggiunte in cima. Quando una busta viene presa in una qualche posizione del ripiano, identificata scandendolo dalla cima, viene successivamente riposta in cima dopo l'uso. Nel momento in cui il ripiano è pieno, un certo quantitativo di buste nel fondo viene trasferito in un'opportuna sede, per esempio una scatola di cartone etichettata in modo da identificarne il contenuto. L'economista sostiene che è più facile ricordare l'ordine temporale dell'uso degli oggetti archiviati piuttosto che la loro classificazione in base al contenuto, per cui il metodo proposto permette di recuperare velocemente tali oggetti dallo scaffale.

5.5 Tecniche di analisi ammortizzata

Le operazioni di unione e appartenenza su liste disgiunte e quelle di ricerca in liste ad auto-organizzazione non sono i primi due esempi di algoritmi in cui abbiamo applicato l'analisi ammortizzata. Abbiamo già incontrato un terzo esempio di tale analisi per valutare il costo delle operazioni di ridimensionamento di un array di lunghezza variabile nel Teorema 1.1.[3] Questi tre esempi illustrano tre diffuse modalità di analisi ammortizzata di cui diamo una descrizione utilizzando come motivo conduttore il problema dell'incremento di un contatore.

In tale problema abbiamo un contatore binario di k cifre binarie, memorizzate in un array `contatore` di dimensione k i cui elementi valgono `0` oppure `1`. In particolare, il valore del contatore è dato da $\sum_{i=0}^{k-1} (\texttt{contatore[i]} \times 2^i)$ e supponiamo che inizialmente esso contenga tutti `0`.

Come mostrato nel Codice 5.6, l'operazione di incremento richiede un costo in tempo pari al *numero di elementi cambiati* in `contatore` (righe 4 e 7), e quindi

[3] Nel Capitolo 2 abbiamo utilizzato questo risultato nell'analisi della complessità delle operazioni di inserimento e cancellazione nelle pile, nelle code e negli heap.

$O(k)$ tempo al caso pessimo: discutiamo tre modi di analisi per dimostrare che il costo ammortizzato di una sequenza di $n = 2^k$ incrementi è soltanto $O(1)$ per incremento.

Codice 5.6 Incremento di un contatore binario.

```
1  Incrementa( contatore ):          ⟨pre: k è la dimensione di contatore⟩
2    i = 0;
3    WHILE ((i < k) && (contatore[i] == 1) ) {
4      contatore[i] = 0;
5      i = i+1;
6    }
7    IF (i < k) contatore[i] = 1;
```

Il primo metodo è quello di **aggregazione**: conteggiamo il numero totale $T(n)$ di passi elementari eseguiti e lo dividiamo per il numero n di operazioni effettuate. Nel nostro caso, conteggiamo il numero di elementi cambiati in `contatore` (righe 4 e 7), supponendo che quest'ultimo assuma come valore iniziale zero. Effettuando n incrementi, osserviamo che l'elemento `contatore[0]` cambia (da 0 a 1 o viceversa) a ogni incremento, quindi n volte; il valore di `contatore[1]` cambia ogni due incrementi, quindi $n/2$ volte; in generale, il valore di `contatore[i]` cambia ogni 2^i incrementi e quindi $n/2^i$ volte. In totale, il numero di passi è $T(n) = \sum_{i=0}^{k-1} n/2^i = \left(\sum_{i=0}^{k=1} 1/2^i\right) n < 2n$. Quindi il costo ammortizzato per incremento è $O(1)$ poiché $T(n)/n < 2$. Osserviamo che abbiamo impiegato il metodo di aggregazione per analizzare il costo dell'operazione di unione di liste disgiunte.

Il secondo metodo è basato sul concetto di **credito** (con relativa metafora bancaria): utilizziamo un fondo comune, in cui depositiamo crediti o li preleviamo, con il vincolo che il fondo non deve andare mai in rosso (prelevando più crediti di quanti siano effettivamente disponibili). Le operazioni possono sia depositare crediti nel fondo che prelevarne senza mai andare in rosso per coprire il proprio costo computazionale: il costo ammortizzato per ciascuna operazione è il numero di crediti depositati da essa. Osserviamo che tali operazioni di deposito e prelievo di crediti sono introdotte solo ai fini dell'analisi, senza effettivamente essere realizzate nel codice dell'algoritmo così analizzato. Nel nostro esempio del contatore, partiamo da un contatore nullo e utilizziamo un fondo comune pari a zero. Con riferimento al Codice 5.6, per ogni incremento eseguito associamo i seguenti movimenti sui crediti:

1. preleviamo un credito per ogni valore di `contatore[i]` cambiato da 1 a 0 nella riga 4;
2. depositiamo un credito quando `contatore[i]` cambia da 0 a 1 nella riga 7.

Da notare che la situazione al punto 1 può occorrere un numero variabile di volte durante un singolo incremento (dipende da quanti valori pari a 1 sono esaminati dal ciclo); invece, la situazione al punto 2 occorre al più una volta, lasciando un credito per quando quel valore da 1 tornerà a essere 0: in altre parole, ogni volta che necessitiamo di un credito nel punto 1, possiamo prelevare dal fondo in quanto tale credito è stato sicuramente depositato da un *precedente* incremento nel punto 2. Ogni operazione può essere dotata di $O(1)$ crediti e quindi il costo ammortizzato per incremento è $O(1)$. Possiamo applicare il metodo dei crediti per l'analisi ammortizzata del ridimensionamento di un array a lunghezza variabile: ogni qualvolta che estendiamo l'array di un elemento in fondo, depositiamo c crediti per una certa costante $c > 0$ (di cui uno è utilizzato subito); ogni volta che raddoppiamo la dimensione dell'array, ricopiando gli elementi, utilizziamo i crediti accumulati fino a quel momento.

Infine, il terzo metodo è basato sul concetto di **potenziale** (con relativa metafora fisica). Numerando le n operazioni da 0 a $n-1$, indichiamo con Φ_{-1} il potenziale iniziale e con $\Phi_j \geq 0$ quello raggiunto dopo l'operazione j, dove $0 \leq j \leq n-1$. La difficoltà consiste nello scegliere l'opportuna funzione come potenziale Φ, in modo che la risultante analisi sia la migliore possibile. Indicando con c_j il costo richiesto dall'operazione j, il costo ammortizzato di quest'ultima è definito in termini della differenza di potenziale, nel modo seguente:

$$\hat{c}_j = c_j + \Phi_j - \Phi_{j-1} \tag{5.10}$$

Quindi, il costo totale che ne deriva è dato da $\sum_{j=0}^{n-1} \hat{c}_j = \sum_{j=0}^{n-1} (c_j + \Phi_j - \Phi_{j-1}) = \sum_{j=0}^{n-1} c_j + (\Phi_{n-1} - \Phi_{-1})$: utilizzando il fatto che otteniamo una somma telescopica per le differenze di potenziale, deriviamo che il costo totale per la sequenza di n operazioni può essere espresso in termini del costo ammortizzato nel modo seguente:

$$\sum_{j=0}^{n-1} c_j = \sum_{j=0}^{n-1} \hat{c}_j + (\Phi_{-1} - \Phi_{n-1}) \tag{5.11}$$

Nell'esempio del contatore binario, poniamo Φ_j uguale al numero di valori pari a 1 in `contatore` dopo il $(j+1)$-esimo incremento, dove $0 \leq j \leq n-1$: quindi, $\Phi_{-1} = 0$ in quanto il contatore è inizialmente pari a tutti 0. Per semplicità, ipotizziamo che il contatore contenga sempre uno 0 in testa e, fissato il $(j+1)$-esimo incremento, indichiamo con ℓ il numero di volte che viene eseguita la riga 4 nel ciclo `WHILE` del Codice 5.6: il costo è quindi $c_j = \ell + 1$ in quanto ℓ valori pari a 1 diventano 0 e un valore pari a 0 diventa 1. Inoltre, la differenza di potenziale $\Phi_j - \Phi_{j-1}$ misura quanti 1 sono cambiati: ne abbiamo ℓ in meno e 1 in più, per cui $\Phi_j - \Phi_{j-1} = -\ell + 1$. Utilizzando la formula (5.10), otteniamo un costo ammortizzato pari a $\hat{c}_j = (\ell + 1) + (-\ell + 1) = 2$. Poiché $\Phi_{n-1} \geq 0$ e $\Phi_{-1} = 0$, in base all'equazione (5.11) abbiamo che $\sum_{j=0}^{n-1} c_j \leq \sum_{j=0}^{n-1} \hat{c}_j \leq 2n$. Osserviamo che abbiamo utilizzato il metodo del potenziale per l'analisi della strategia MTF scegliendo come potenziale Φ_j il numero di inversioni rispetto alla lista gestita da OPT.

5.6 Esercizi

5.1 Mostrare che l'analisi al caso medio del *quicksort* randomizzato è $O(n \log n)$ anche dividendo il segmento `[sinistra ... destra]` in tre parti (invece che in quattro).

5.2 Estendere la `QuickSelect` randomizzata in modo da trovare gli elementi di rango compreso tra r_1 e r_2, dove $r_1 < r_2$. Studiare la complessità al caso medio dell'algoritmo proposto.

5.3 Modificare il Codice 5.2 in modo da restituire in un array tutti i predecessori sulle liste L_i della chiave data.

5.4 Dimostrare che la complessità dell'operazione di ricerca in una lista a salti non casuale è logaritmica nel numero degli elementi.

5.5 Scrivere lo pseudocodice che, prese due liste a salti, produce l'intersezione degli elementi in esse contenuti. Discutere la complessità dell'algoritmo proposto.

5.6 Descrivere una rappresentazione degli insiemi per il problema dell'unione di liste disgiunte che, per ogni insieme, utilizzi un albero in cui i soli puntatori siano quelli al padre.

5.7 Mostrare che, nonostante sia $\texttt{costo(MTF)} \leq 2 \times \texttt{costo(OPT)}$, alcune configurazioni hanno $\texttt{costo(MTF)} < \texttt{costo(OPT)}$ (prendere una lista di $m = 2$ elementi e accedere a ciascuno $n/2$ volte).

5.8 Consideriamo un algoritmo non in linea REV, il quale applica la seguente strategia ad auto-organizzazione per la gestione di una lista. Quando REV accede all'elemento in posizione i, va avanti fino alla prima posizione $i' \geq i$ che è una potenza del 2, prende quindi i primi i' elementi e li dispone in ordine di accesso futuro (ovvero il successivo elemento a cui accedere va in prima posizione, l'ulteriore successivo va in seconda posizione e così via). Ipotizziamo che $n = m = 2^k + 1$ per qualche $k \geq 0$, che inizialmente la lista contenga gli elementi $e_0, e_1, \ldots, e_{m-1}$ e che la sequenza di richieste sia $e_0, e_1, \ldots, e_{m-1}$, in questo ordine (vengono cioè richiesti gli elementi nell'ordine in cui appaiono nella lista iniziale). Dimostrate che il costo di MTF risulta essere $\Theta(n^2)$ mentre quello di REV è $O(n \log n)$. Estendete la dimostrazione al caso $n > m$.

5.9 Calcolare un valore della costante c adoperata nell'analisi ammortizzata con i crediti per il ridimensionamento di un array di lunghezza variabile, dettagliando come gestire i crediti.

5.10 Fornire un'analisi ammortizzata basata sul potenziale per il problema del ridimensionamento di un array di lunghezza variabile.

CAPITOLO

6 Programmazione dinamica

La programmazione dinamica è una tecnica fondamentale per trovare soluzioni ottime – di minimo costo oppure di massimo rendimento – per certi problemi di ottimizzazione che possono essere risolti con una regola ricorsiva e con la tabulazione delle soluzioni intermedie via via trovate.

6.1 Il paradigma della programmazione dinamica
6.2 Problema del resto
6.3 Opus libri: sotto-sequenza comune più lunga
6.4 Partizione di un insieme di interi
6.5 Problema della bisaccia
6.6 Massimo insieme indipendente in un albero
6.7 Alberi di ricerca ottimi
6.8 Pseudo-polinomialità e programmazione dinamica
6.9 Esercizi

6.1 Il paradigma della programmazione dinamica

Sarà capitato anche a voi di calcolare la sequenza di numeri di Fibonacci, definita ricorsivamente come $F_0 = 0$, $F_1 = 1$ e $F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$: quindi 0, 1, 1, 2, 3, 5, 8, 13 e così via. L'algoritmo immediato ricorsivo per calcolare F_n richiede tempo esponenziale in n perché ricalcola molte volte gli stessi valori F_k ($k < n$) già calcolati in precedenza: per calcolare $F_6 = F_5 + F_4$, occorre calcolare $F_5 = F_4 + F_3$, ma per calcolare $F_4 = F_3 + F_2$ dobbiamo calcolare ulteriori valori; quando finalmente abbiamo terminato il calcolo ricorsivo di F_5, scopriamo che $F_6 = F_5 + F_4$ richiede nuovamente di calcolare F_4 ricorsivamente. Il numero di chiamate ricorsive esplode esponenzialmente in questo modo. Quasi sicuramente avrete evitato questo approccio ricorsivo in favore di uno iterativo che richiede tempo lineare in n e utilizza una tabella di appoggio fib in cui scrivere `fib[0]` = F_0, `fib[1]` = F_1 e i valori intermedi `fib[k] = fib[k-1] + fib[k-2]` ottenuti attraverso un ciclo per k = 2, 3, ..., n, evitando così di ricalcolare più volte gli stessi numeri di Fibonacci (in realtà è possibile far di meglio, calcolando F_n in tempo logaritmico).

Quello appena descritto è un esempio che illustra l'idea sottostante alla **programmazione dinamica**. Anche se il calcolo dei numeri di Fibonacci non rientra propriamente nella famiglia dei problemi di programmazione dinamica, è un modo semplice per vedere come implementare una *regola ricorsiva di calcolo* (di F_n) mediante un *algoritmo iterativo* che dinamicamente riempie gli elementi di un'opportuna *tabella di programmazione* (fib) di cui restituiamo il valore nell'ultimo elemento (corrispondente a F_n). In generale, la programmazione dinamica è un paradigma basato sullo stesso principio utilizzato per il paradigma del divide et impera, in quanto il problema viene decomposto in sotto-problemi e la soluzione del problema è derivata da quelle di tali sotto-problemi. In effetti, entrambi i paradigmi vengono applicati a partire da una definizione ricorsiva che correla la soluzione di un problema a quella di un insieme di sotto-problemi.

La differenza fondamentale tra i due paradigmi è dovuta al fatto che la programmazione dinamica è nata principalmente per risolvere **problemi di ottimizzazione**, ossia situazioni in cui ogni soluzione ammissibile per un dato problema ha un costo associato e vogliamo trovare quella di **costo ottimo** (sia esso il minimo o il massimo a seconda della natura del problema), mentre il paradigma del divide et impera è più mirato alla risoluzione di problemi che non sono necessariamente di ottimizzazione, come l'ordinamento.

Un'altra differenza è che, mentre la formulazione ricorsiva del divide et impera comporta che un qualunque sotto-problema venga considerato una sola volta e, quindi, non si ponga il problema di evitare calcoli ripetuti di una medesima soluzione, la formulazione ricorsiva della programmazione dinamica comporta che uno stesso sotto-problema rientri come componente nella definizione di più problemi. Tale differenza può essere schematizzata dalla Figura 6.1, dove vengono mostrati degli schemi di decomposizione ricorsiva di un problema

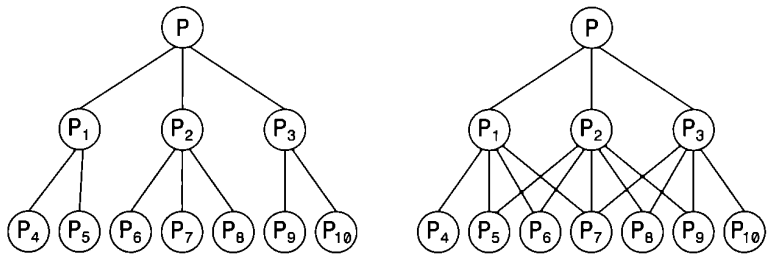


Figura 6.1 Decomposizione in sotto-problemi mediante il paradigma del divide et impera e della programmazione dinamica.

in sotto-problemi, mediante il paradigma del divide et impera e quello della programmazione dinamica.

Come possiamo vedere, nel caso del paradigma del divide et impera il problema P viene decomposto in tre sotto-problemi P_1, P_2 e P_3, ognuno dei quali a sua volta è decomposto in sotto-problemi elementari (risolubili in modo immediato senza decomposizioni ulteriori) in modo tale che uno stesso sotto-problema compare soltanto in una decomposizione: per esempio, P_6 compare soltanto nella decomposizione di P_2, e la sua soluzione dovrà quindi essere calcolata una sola volta, nell'ambito del calcolo della soluzione di P_2 (cui contribuisce insieme al calcolo della soluzione di P_7 e di P_8).

Al contrario, nel caso del paradigma della programmazione dinamica possiamo vedere che uno stesso sotto-problema compare in più decomposizioni di problemi diversi, e quindi il calcolo della sua soluzione viene a costituire parte del calcolo delle soluzioni di più problemi. Per esempio, P_6 compare ora nella decomposizione sia di P_1 che in quella di P_2, con l'effetto che, se i calcoli delle soluzioni di P_1 e di P_2 vengono effettuati senza tener conto di tale situazione, P_6 deve essere risolto due volte, una volta per contribuire alla soluzione di P_1 e l'altra per contribuire alla soluzione di P_2.

In generale, la risoluzione mediante programmazione dinamica di un problema è caratterizzata dalle seguenti due proprietà della decomposizione, la prima delle quali è condivisa con il divide et impera.

Ottimalità dei sotto-problemi. La soluzione ottima di un problema deriva dalle soluzioni ottime dei sotto-problemi in cui esso è stato decomposto.

Sovrapposizione dei sotto-problemi. Uno stesso sotto-problema può essere usato nella soluzione di due (o più) sotto-problemi diversi.

La definizione di un algoritmo di programmazione dinamica è quindi basata su quattro aspetti:

1. caratterizzazione della struttura generale di cui sono istanze sia il problema in questione che tutti i sotto-problemi introdotti in seguito;

2. identificazione dei sotto-problemi elementari e individuazione della relativa modalità di determinazione della soluzione;
3. definizione di una regola ricorsiva per la soluzione in termini di composizione delle soluzioni di un insieme di sotto-problemi;
4. derivazione di un ordinamento di tutti i sotto-problemi così definiti, per il calcolo efficiente e la memorizzazione delle loro soluzioni in una tabella.

Il numero di passi richiesto dall'algoritmo è quindi limitato superiormente dal prodotto tra il numero di sotto-problemi e il costo di ricombinazione delle loro soluzioni ottime. Il resto del capitolo illustra i concetti esposti sopra con una serie di problemi.

6.2 Problema del resto

Consideriamo il problema di restituire il resto con il minor numero possibile di monete. Più precisamente, dato un intero R e un insieme M = {0, ..., m-1} di m tipi di monete aventi valore $v_0, v_1, \ldots, v_{m-1}$, ipotizziamo di disporre di un numero illimitato di monete di tipo i per $0 \le i \le m-1$: vogliamo progettare un algoritmo che, per ogni moneta i, ne restituisce la quantità $n_i \ge 0$ in modo da creare il resto R con il minor numero possibile di monete, quindi

$$\sum_{i=0}^{m-1} n_i v_i = R \quad \text{con} \sum_{i=0}^{m-1} n_i \text{ minima.}$$

Volendo scrivere una regola ricorsiva, indichiamo con r(R) il minor numero di monete che occorre per ottenere il resto R. Il caso base è quando R = 0 e non occorre restituire alcuna moneta, per cui r(0) = 0. Se R > 0, osserviamo che se è possibile scegliere una moneta i per il resto R (cioè $v_i \le R$) allora rimane da creare il resto di $R - v_i$ col minor numero possibile di monete nel seguente modo:

$$r(R) = \begin{cases} 0 & \text{se } R = 0 \\ +\infty & \text{se } v_i > R \text{ per ogni } 0 \le i \le m-1 \\ 1 + \min_{i:v_i \le R} r(R - v_i) & \text{altrimenti.} \end{cases} \quad (6.1)$$

La relazione (6.1) induce un semplice algoritmo per il calcolo del resto, basato sulla tabulazione mediante una tabella resto[R] che memorizza i valori intermedi di r(R). Risolviamo un problema più generale, dove il valore intermedio resto[c] indica il minimo numero di monete per fare il resto c. Quando c=R, abbiamo la nostra soluzione. La tabella viene inizializzata con il caso base resto[0]=0. Viene quindi riempita da sinistra a destra, secondo la relazione(6.1), come mostrato nel Codice 6.1, ponendo inizialmente ciascun valore intermedio resto[c] al valore +∞ che verrà cambiato solo se esiste un modo di fare il resto c. L'algoritmo prende in input il resto R e l'array v di m interi tale che v[i] = v_i ovvero il valore della moneta i. L'algoritmo calcola in resto[c] tutti i valori di r(c) con c = 1, ..., R come descritto dalla (6.1). Siccome richiede O(m) per decidere il valore di resto[c] e ci sono R tali valori da calcolare, la complessità dell'algoritmo è O(mR).

Codice 6.1 Algoritmo per il calcolo del minimo numero di monete per il resto.

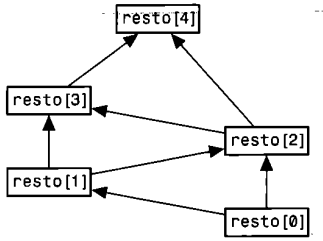
```
1  Resto( R, v ):
2    resto[0] = 0;
3    FOR (c = 1; c <= R; c = c + 1) {
4      resto[c] = +∞;
5      FOR (i = 0; i < m; i = i + 1) {
6        IF ( v[i] <= c && resto[c] > 1 + resto[ c-v[i] ]) {
7          resto[c] = 1 + resto[ c-v[i] ];
8        }
9      }
10   }
11   RETURN resto[R];
```

ESEMPIO 6.1

Consideriamo il caso R = 4 con monete di valore {1, 2}. L'algoritmo costruisce il seguente array resto:

	0	1	2	3	4
resto =	0	1	1	2	2

Infatti, per esempio, resto[3] = 1 + min{resto[2], resto[1]} = 2, invece resto[4] = 1 + min{resto[3], resto[2]} = 2. Come risulta, il sotto-problema resto[2] dipende dai sotto-problemi resto[1] e resto[0] così come il sotto-problema resto[3] dipende dai sotto-problemi resto[2] e resto[1] e così via. Tali dipendenze sono schematizzate graficamente nella figura che segue.



Gli aspetti della programmazione dinamica per il problema del resto corrispondono: all'introduzione del sotto-problema per il costo ottimo per un resto c ≤ R; all'identificazione dei casi quando R < v_i per ogni moneta i oppure R = 0 come sotto-problemi elementari avente costo ottimo pari a 0 e +∞, rispettivamente; infine, al riempimento della tabella dei costi da sinistra a destra come indicato nel Codice 6.1.

Esercizio svolto 6.1 La soluzione riportata nel Codice 6.1 fornisce il numero minimo totale $\sum_{i=0}^{m-1} n_i$ monete per comporre il resto R, ma non ci dice quante volte n_i viene scelta ogni moneta i. Completarlo in modo da ottenere tale quantità nell'array n tale che n[i] = n_i.

Soluzione Quando viene scelto il minimo per resto[c] nella riga 7, occorre memorizzare in un opportuno elemento moneta[c] che la moneta i ha dato luogo al minimo. Alla fine, prima di restituire resto[R], osserviamo che la moneta usata è i = moneta[R] e quindi possiamo ricostruire all'indietro la soluzione: basta considerare il resto intermedo c = R - v[i] e porre i = moneta[c] come moneta successiva e così via. Il Codice 6.2 estende il Codice 6.1 in tal senso e memorizza in n[i] quante volte viene usata la moneta i. Il costo risultante rimane tempo O(mR).

Codice 6.2 Algoritmo per il calcolo al dettaglio del minimo numero di monete per il resto.

```
1  RestoMonete( R, v ):                          ⟨pre: resto[R] ≠ +∞⟩
2    resto[0] = 0;
3    FOR (c = 1; c <= R; c = c + 1) {
4      resto[c] = +∞;
5      FOR (i = 0; i < m; i = i + 1) {
6        IF ( v[i] <= c && resto[c] > 1 + resto[ c-v[i] ]) {
7          resto[c] = 1 + resto[ c-v[i] ];
8          moneta[c] = i;
9        }
10     }
11   }
12   FOR (i = 0; i < m; i = i + 1)
13     n[i] = 0
14   c = R;
15   WHILE (c > 0) {
16     i = moneta[c];
17     n[i] = n[i] + 1;
18     c = c - v[i];
19   }
20   RETURN n;
```

L'algoritmo che probabilmente un commerciante utilizza per il resto è molto più semplice della programmazione dinamica mostrata nel Codice 6.1 e nel Codice 6.2: sceglie la moneta più alta a disposizione e ne fornisce il numero massimo senza oltrepassare R; ripete quindi l'operazione con le altre monete fino a ottenere il resto R.

Ci troviamo quindi di fronte alla tipica situazione che spinge un algoritmo di risoluzione a comportarsi in modo "goloso" nel senso che le sue scelte si basano principalmente sulla base della configurazione attuale, senza cercare di valutarne

le conseguenze sulle configurazioni successive ("pochi, maledetti e subito" come spesso ci si riferisce ai soldi): questa è una caratteristica tipica della tecnica *greedy*. Il paradigma dell'**algoritmo goloso**, che difficilmente può essere formalizzato in modo preciso, risulta talvolta, ma non così spesso, vincente: il suo successo, in verità, dipende quasi sempre da proprietà strutturali del problema che non sempre sono evidenti.

L'algoritmo goloso può essere visto come un caso speciale di programmazione dinamica in cui, una volta ordinati in modo opportuno gli elementi di un'istanza, si decompone il problema da risolvere in un unico sotto-problema definito eliminando l'ultimo elemento nell'ordine specificato: la fase di ricombinazione consiste nel decidere in modo goloso se, e in che modo, aggiungere tale elemento alla soluzione del sotto-problema.

Purtroppo il paradigma dell'algoritmo goloso non è così generale come quello della programmazione dinamica e raramente consente di risolvere in modo esatto un problema computazionale, anche se esso è stato applicato con un certo successo nel campo delle euristiche di ottimizzazione combinatoria e degli algoritmi di approssimazione (vedi Capitolo 7).

Per illustrare ciò, formalizziamo l'algoritmo goloso del commerciante e studiamone il comportamento. Supponendo che le monete siano ordinate in modo decrescente $v_0 > v_1 > \cdots > v_{m-1}$, questo algoritmo si può descrivere ricorsivamente nel seguente modo più rigoroso: sia r(R, i) il numero di monete da restituire come resto R utilizzando i valori $v_i > v_{i+1} > \cdots > v_{m-1}$, ponendo questo valore a $+\infty$ nel caso in cui non esista una soluzione. La risposta che cerchiamo viene quindi fornita da r(R, 0) e il caso base occorre quando R = 0, per cui non occorrono monete. Nel caso induttivo, possiamo dedurre che occorrono $\lfloor R/v_i \rfloor$ monete di tipo i: tale formula tiene conto anche della situazione in cui $v_i > R$, per cui il risultato è zero. In ogni caso, dobbiamo ancora fornire resto $R - \lfloor R/v_i \rfloor$ con le rimanenti monete di valore $v_{i+1} > \cdots > v_{m-1}$. Infine, se $i \geq m$, vuol dire che non siamo in grado di fornire una soluzione con il metodo proposto, per cui il risultato è $+\infty$. Possiamo quindi esprimere con una regola ricorsiva quanto esposto sopra, la cui implementazione è riportata nel Codice 6.3:

$$r(R, i) = \begin{cases} 0 & \text{se } R = 0 \\ +\infty & \text{se } i \geq m \\ \lfloor R/v_i \rfloor + r\left(R - \lfloor R/v_i \rfloor, i + 1\right) & \text{altrimenti.} \end{cases} \tag{6.2}$$

Codice 6.3 Soluzione golosa (non sempre corretta) del commerciante per il resto.

```
1  Resto( R, i ):                          ⟨pre: v[0] > v[1] > ··· > v[m-1]⟩
2    IF (R == 0) RETURN 0;
3    IF (i >= m) RETURN +∞;
4    RETURN R/v[i] + Resto( R-R/v[i], i+1 );
```

Teorema 6.1 *L'algoritmo mostrato nel Codice 6.3 ha complessità* $O(m)$.

Dimostrazione Sia $T(R, m)$ il numero di operazioni eseguite nel caso pessimo dall'algoritmo con resto R e m tipi di monete a disposizione. Se $R = 0$ oppure $m = 0$, vale chiaramente $T(R, m) \leq c_0$ dove c_0 è una costante positiva. Altrimenti per una opportuna costante $c > 0$ e un valore $0 < R' \leq R$, vale la relazione di ricorrenza

$$T(R, m) \leq c + T(R', m - 1).$$

Mostriamo per induzione che $T(R, m) \leq cm + c_0 = O(m)$. Il caso base $R = 0$ o $m = 0$ induce un costo $T(0, m) \leq c_0$ o $T(R, 0) \leq c_0$. Nel passo induttivo, $T(R, m) \leq c + T(R', m - 1) \leq c + c(m - 1) + c_0 = cm + c_0$. ❑

ESEMPIO 6.2

Supponiamo di avere 3 tipi di monete da $v_0 = 5$, $v_1 = 2$ e $v_2 = 1$ e che il resto sia $R = 9$ (quindi $v[0] = 5$, $v[1] = 2$ e $v[2] = 1$). Allora, invocando $\texttt{Resto}(9, 0)$ otteniamo 3 monete come soluzione, ossia una moneta da 5 e due monete da 2:

$$\begin{aligned}\texttt{Resto}(9, 0) &= 1 + \texttt{Resto}(4, 1)\\ &= 3 + \texttt{Resto}(0, 2)\\ &= 3.\end{aligned}$$

La soluzione trovata nell'Esempio 6.2 è una soluzione ottima per il problema del resto. Se l'algoritmo funzionasse sempre, vista la sua complessità polinomiale $O(m)$, sarebbe preferibile rispetto a quella pseudo-polinomiale $O(mR)$ ottenuta mediante la programmazione dinamica. Tuttavia il Codice 6.3 potrebbe restituire soluzioni non ottime o, addirittura, non trovare nessuna soluzione, come mostrato nell'esempio seguente.

ESEMPIO 6.3

Prendiamo le monete di valore 6, 4, 1 e sia $R = 9$. L'algoritmo descritto nel Codice 6.3 trova una soluzione composta da quattro monete: una moneta da 6 e tre monete da 1. Mentre la combinazione ottima è composta da 3 monete, ossia due da 4 e una da 1. Se invece abbiamo solo monete di valore 6, 4 e $R = 8$, l'algoritmo non trova soluzioni quando invece bastano due monete da 4:

$$\begin{aligned}\texttt{Resto}(8, 0) &= 1 + \texttt{Resto}(2, 1)\\ &= 1 + \texttt{Resto}(2, 2)\\ &= +\infty\end{aligned}$$

Tuttavia questo algoritmo è da tener presente in quanto in alcuni casi, non tanto rari, restituisce la soluzione ottima.

Esercizio svolto 6.2 Dimostrare che, nel caso che le monete abbiano i valori $50, 20, 10, 5, 2, 1$, il Codice 6.3 trova sempre la soluzione ottima per qualunque resto R.

Soluzione Sia R il resto da comporre e $v_0 = 50$, $v_1 = 20$, $v_2 = 10$, $v_3 = 5$, $v_4 = 2$ e $v_5 = 1$. Indichiamo con $S = (t_0, t_1, t_2, t_3, t_4, t_5)$ la soluzione ottima: ovvero $R = \sum_{i=0}^{5} t_i v_i$ e $\sum_{i=0}^{5} t_i$ è minimo. Definiamo $R_k = \sum_{i=k}^{5} t_i v_i$, ovvero R_k è la parte di R composta con le monete di valore al più v_k. Quindi $R_0 = R$ e $R_i = t_i v_i + R_{i+1}$. Se per $i = 0, \ldots, 4$, $R_{i+1} < v_i$ allora $R_{i+1} = R_i \bmod v_i$. Ovvero $t_i = \lfloor R_i / v_i \rfloor$, che è la scelta che esegue l'algoritmo nel Codice 6.3. Quindi, per dimostrare il risultato è sufficiente far vedere che per $i = 0, \ldots, 4$, $R_{i+1} < v_i$.

- Nella soluzione ottima c'è al più una moneta da 1, quindi $R_5 = t_5 < 2 = v_4$.
- Consideriamo $R_4 = 2t_4 + t_5$. Se $t_5 = 0$ allora $t_4 \leq 2$ in quanto se fosse $t_4 = 3$ potremmo sostituire le 3 monete da $2 = v_4$ con una da $5 = v_3$ e una da $1 = v_5$ migliorando la soluzione ottima. Quindi $R_4 \leq 2 \times 2 < 5 = v_3$. Se invece $t_5 = 1$ allora $t_4 \leq 1$ altrimenti 2 monete da 2 e quella da 1 potrebbero essere sostituite da una da 5. Anche in questo caso $R_4 \leq 2 \times 1 + 1 < 5 = v_3$.
- Considerando che nella soluzione ottima non possono esserci né 2 monete da $5 = v_3$, né 2 monete da $10 = v_2$ allora $R_3 = 5t_3 + R_4 < 5t_3 + 5 \leq 10 = v_2$ e $R_2 = 10t_2 + R_3 < 10t_2 + 10 \leq 20 = v_1$.
- Osserviamo che $t_1 \leq 2$: infatti se fosse $t_1 = 3$ potremmo raggiungere la cifra di 60 con 2 monete $(v_0 + v_1)$. Se $t_1 = 1$, $R_1 = v_1 + R_2 < 2v_1 = 40 < v_0$. Se $t_1 = 2$ allora nella soluzione ottima non possono esserci monete da $10 = v_2$ quindi $t_1 = 0$ e $R_2 = v_2 t_2 + R_3 = R_3 < v_3 = 5$. Da cui segue che $R_1 = 20t_1 + R_2 = 20t_1 + R_3 < 40 + 5 < v_0 = 50$.

Questo conclude la dimostrazione.

6.3 Opus libri: sotto-sequenza comune più lunga

I sistemi operativi mantengono traccia dei comandi invocati dagli utenti, memorizzandoli in opportune sequenze chiamate *log* (i file nella directory `/var/log` dei sistemi Unix/Linux sono un esempio di tali sequenze). I sistemisti usano i log per verificare eventuali intrusioni (*intrusion detection*) che possano minare la sicurezza e l'integrità del sistema: quest'esigenza è molto diffusa a causa del collegamento dei calcolatori a Internet, che può permettere l'accesso remoto da qualunque parte del mondo. Uno dei metodi usati consiste nell'individuare particolari sotto-sequenze che appaiono nelle sequenze di log e che sono caratteristiche degli attacchi alla sicurezza del sistema. Sia F la sequenza dei comandi di cui il log tiene traccia: quando avviene un attacco, i comandi lanciati durante l'intrusione formano una sequenza S ma, purtroppo, appaiono in F mescolati ai comandi legalmente

invocati sul sistema. Per poter distinguere S all'interno di F, i comandi vengono etichettati in base alla loro tipologia (useremo semplici etichette come A, B, C e così via), per cui S e F sono entrambe rappresentate come sequenze di etichette: i singoli comandi di S sono probabilmente legali se presi individualmente mentre è la loro sequenza a essere dannosa.

Dobbiamo quindi individuare S quando appare come **sotto-sequenza** di F: in altre parole, indicata con k la lunghezza di S e con n quella di F, dove $k \leq n$, vogliamo verificare se esistono k posizioni crescenti in F che contengono ordinatamente gli elementi di S (ossia se esistono k interi $i_0, i_1, \ldots, i_{k-1}$ tali che $0 \leq i_0 < i_1 < \ldots < i_{k-1} \leq n-1$ e $S[j] = F[i_j]$ per $0 \leq j \leq k-1$). Per esempio, S = A, D, C, A, A, B appare come sotto-sequenza di F = B, A, A, B, D, C, D, C, A, A, C, A, C, B, A (dove le lettere sottolineate contrassegnano una delle possibili occorrenze, in quanto S può essere alternativamente vista come il risultato della cancellazione di zero o più caratteri da F).

Il problema che in realtà intendiamo risolvere con la programmazione dinamica è quello di individuare la lunghezza delle sotto-sequenze comuni più lunghe per due sequenze date a e b. Diciamo che x è una **sotto-sequenza comune** ad a e b se appare come sotto-sequenza in entrambe: x è una **sotto-sequenza comune più lunga** (LCS o *longest common subsequence*) se non ne esistono altre di lunghezza maggiore che siano comuni alle due sequenze a e b (ne possono ovviamente esistere altre di lunghezza pari a quella di x ma non più lunghe). Indicando con LCS(a, b) la lunghezza delle sotto-sequenze comuni più lunghe di a e b, notiamo che questa formulazione generale del problema permette di scoprire se una sequenza di comandi S appare come sotto-sequenza di un log F: basta infatti verificare che sia k = LCS(S, F) (ponendo quindi a = S e b = F).

Le possibili sotto-sequenze comuni di due sequenze a e b, di lunghezza m e n rispettivamente, possono essere in numero esponenziale in m e n ed è quindi inefficiente generarle tutte per poi scegliere quella di lunghezza massima. Applichiamo quindi il paradigma della programmazione dinamica a tale problema per risolverlo in tempo polinomiale O(mn), seguendo la falsariga delineata dai quattro aspetti del paradigma discussi in precedenza. In prima istanza, definiamo

$$L(i, j) = LCS(a[0, i-1], b[0, j-1])$$

come la lunghezza massima delle sotto-sequenze comuni alle due sequenze rispettivamente formate dai primi i elementi di a e dai primi j elementi di b, dove $0 \leq i \leq m$ e $0 \leq j \leq n$ (adottiamo la convenzione che a[0, -1] sia vuota e che b[0, -1] sia vuota).

In seconda istanza, osserviamo che L(m, n) fornisce la soluzione LCS(a, b) al nostro problema e definiamo i sotto-problemi elementari come L(i, 0) = L(0, j) = 0, in quanto se (almeno) una delle sequenze è vuota, allora l'unica sotto-sequenza comune è necessariamente la sequenza vuota (quindi di lunghezza pari a 0).

In terza istanza, forniamo la definizione ricorsiva in termini dei sotto-problemi L(i, j) per $i > 0$ e $j > 0$, prendendo in considerazione le sotto-sequenze comuni di lunghezza massima per a[0, i-1] e b[0, j-1] secondo la seguente regola:

$$L(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ L(i-1, j-1) + 1 & \text{se } i, j > 0 \text{ e } a[i-1] = b[j-1] \\ \max\{L(i, j-1), L(i-1, j)\} & \text{se } i, j > 0 \text{ e } a[i-1] \neq b[j-1] \end{cases} \quad (6.3)$$

La prima riga della regola nella (6.3) riporta i valori per i sotto-problemi elementari (i = 0 o j = 0). Le successive due righe nella (6.3) descrivono come ricombinare le soluzioni dei sotto-problemi (i, j > 0):

- a[i-1] = b[j-1]: se k = L(i-1, j-1) è la lunghezza massima delle sotto-sequenze comuni ad a[0, i-2] e b[0, j-2], allora k + 1 lo è per a[0, i-1] e b[0, j-1], in quanto il loro ultimo elemento è uguale (altrimenti ne esisterebbe una di lunghezza maggiore di k in a[0, i-2] e b[0, j-2], che è assurdo);
- a[i-1] ≠ b[j-1]: se k = L(i, j-1) è la lunghezza massima delle sotto-sequenze comuni ad a[0, i-1] e b[0, j-2] e k′ = L(i-1, j) è la lunghezza massima delle sotto-sequenze comuni ad a[0, i-2] e b[0, j-1], allora tali sotto-sequenze appariranno inalterate come sotto-sequenze comuni in a[0, i-1] e b[0, j-1], poiché non possono essere estese ulteriormente: pertanto, L(i, j) è pari a max{k, k′}.

In quarta istanza, utilizziamo una tabella lunghezza di taglia (m + 1) × (n + 1), tale che lunghezza[i][j] = L(i, j). Dopo aver inizializzato la prima colonna e la prima riga con i valori pari a 0, riempiamo tale tabella in ordine di riga secondo quanto riportato nel Codice 6.4. Essendoci O(mn) sotto-problemi, ciascuno risolvibile in tempo costante (righe 8-14) in base alla regola della (6.3), otteniamo una complessità di O(mn) tempo e spazio.

Codice 6.4 Algoritmo per il calcolo della lunghezza della sotto-sequenza comune più lunga.

```
1  LCS( a, b ):                                  ⟨pre: a e b sono di lunghezza m e n⟩
2    FOR (i = 0; i <= m; i = i+1)
3      lunghezza[i][0] = 0;
4    FOR (j = 0; j <= n; j = j+1)
5      lunghezza[0][j] = 0;
6    FOR (i = 1; i <= m; i = i+1)
7      FOR (j = 1; j <= n; j = j+1) {
8        IF (a[i-1] == b[j-1]) {
9          lunghezza[i][j] = lunghezza[i-1][j-1] + 1;
10       } ELSE IF (lunghezza[i][j-1] > lunghezza[i-1][j]) {
11         lunghezza[i][j] = lunghezza[i][j-1];
12       } ELSE {
13         lunghezza[i][j] = lunghezza[i-1][j];
```

```
14        }
15      }
16    RETURN lunghezza[m][n];
```

Possiamo individuare una delle sotto-sequenze comuni più lunghe, utilizzando un ulteriore array `indice` di taglia `(m + 1) × (n + 1)` nel Codice 6.4, per memorizzare quale delle sue istruzioni determina il valore dell'elemento corrente di `lunghezza`. Realizziamo ciò assegnando a `indice[i][j]` una delle seguenti tre coppie di indici: ⟨`i - 1, j - 1`⟩ nella riga 9, ⟨`i, j - 1`⟩ nella riga 11 e ⟨`i - 1, j`⟩ nella riga 13. Il seguente algoritmo ricorsivo (che deve essere invocato con input `i = m` e `j = n`) usa `indice` per ricavare una sotto-sequenza comune più lunga (righe 3 e 5):

```
1  StampaLCS( i, j ):                          ⟨pre: 0 ≤ i ≤ m e 0 ≤ j ≤ n⟩
2    IF ((i > 0) && (j > 0)) {
3      <i', j'> = indice[i][j];
4      StampaLCS( i', j' );
5      IF ((i' == i-1) && (j' == j-1))) PRINT a[i-1];
6    }
```

Notiamo che possiamo ridurre a `O(n)` spazio la complessità dell'algoritmo descritto nel Codice 6.4, in quanto utilizza soltanto due righe consecutive di `lunghezza` alla volta (in alternativa, possiamo modificare il codice in modo che riempia la tabella `lunghezza` per colonne e ne utilizzi solamente due colonne alla volta). Tuttavia, tale riduzione in spazio non permette di eseguire l'algoritmo `StampaLCS` perché non abbiamo più a disposizione l'array `indice`. Esiste un modo più sofisticato, implementato in alcuni sistemi operativi, che richiede spazio lineare $O(n + m)$ e tempo $O((r + m + n)\log(n + m))$, dove r è il numero di possibili coppie di elementi uguali nelle sequenze a e b. Pur essendo $r = O(mn)$ al caso pessimo, in molte situazioni pratiche vale $r = O(m + n)$: in tal caso, trovare una sotto-sequenza comune più lunga in spazio lineare richiede $O((n + m)\log n)$ tempo.

ESEMPIO 6.4

La figura che segue mostra le tabelle `lunghezza` e `indice` relative alle sequenze a = A, D, C, A, A, B e b = B, A, A, B, D, C, D, C, A, A, C, A, C, B, A. Quindi n = 15 e m = 6.

		B	A	A	B	D	C	D	C	A	A	C	A	C	B	A
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A 1	0	0→	1→	1→	1→	1→	1	1	1	1	1	1	1	1	1	1
D 2	0	0	1	1	1	2	2	2↘	2	2	2	2	2	2	2	2
C 3	0	0	1	1	1	2	3	3	3→	3↘	3	3	3	3	3	3
A 4	0	0	1	2	2	2	3	3	3	4	4→	4↘	4	4	4	4
A 5	0	0	1	2	2	2	3	3	3	4	5	5	5→	5↘	5	5
B 6	0	1	1	2	3	3	3	3	3	4	5	5	5	5	6→	6

In particolare è rappresentata solo una parte della tabella `indice`, quella che serve per ricostruire la soluzione ottima. Il valore di `indice[i][j]` è rappresentato con una freccia da `indice[i][j]` a ⟨`i, j`⟩. Questo valore è dato dalla posizione da cui parte la freccia. Per esempio `indice[4][10]` = ⟨3, 9⟩. La ricostruzione della soluzione ottima procede partendo dalla posizione ⟨`i, j`⟩ = ⟨6, 15⟩ di `indice`. Il suo contenuto è ⟨6, 14⟩ = ⟨`i, j`⟩, quindi iteriamo su `indice[i][j]` = ⟨5, 13⟩ che è uguale a ⟨`i - 1, j - 1`⟩. In questo caso si era avuta una corrispondenza tra il carattere nella posizione `i - 1` di a e quello nella posizione `j - 1` di b quindi viene dato in output il carattere `a[i - 1, j - 1]`.

Esercizio svolto 6.3 Utilizzando la programmazione dinamica, risolviamo il problema di determinare la sotto-sequenza *crescente* più lunga di una sequenza $a = a_0, a_1, \ldots, a_{n-1}$ di n interi. Per esempio la sotto-sequenza crescente più lunga di a = (9, 3, 4, 1, 6, 8, 0, 9) è (3, 4, 6, 8, 9).

Soluzione Consideriamo una sotto-sequenza crescente più lunga per la sequenza `a[0, i]` che termina con a_i e indichiamo con l_i la sua lunghezza. Indichiamo con L la lunghezza di una sotto-sequenza ottima. Deve valere:

$$L = \max_{i=0,\ldots,n-1} l_i.$$

Ora vediamo come calcolare tutti i valori l_i.

Osserviamo che $l_0 = 1$. Per quanto riguarda l_i, con $i > 0$, una sotto-sequenza ottima che termina con a_i deve essere preceduta da a_j, per un opportuno $j < i$, e quindi $a_j < a_i$. Questa osservazione induce immediatamente la formulazione ricorsiva che ci permette di esprimere la soluzione del problema mediante composizione di soluzioni dei sotto-problemi che lo compongono:

$$l_i = \begin{cases} 1 & \text{se } i = 0 \\ 1 & \text{se } i > 0 \text{ e } a_j \geq a_i \text{ per ogni } j < i \\ 1 + \max_{j<i:a_j<a_i} l_j & \text{altrimenti} \end{cases} \qquad (6.4)$$

Dalla (6.4) ricaviamo l'algoritmo per il calcolo della sotto-sequenza crescente più lunga illustrata nel Codice 6.5, dove la tabella `lsc[i]` memorizza il valore di l_i. L'algoritmo prende in input l'array `a` di `n` interi che memorizza la sequenza. L'output è rappresentato da un array `seq` di `n` interi che ha il seguente significato: `seq[i]` = $j \geq 0$ se e solo se a_j è l'elemento che precede a_i in una sotto-sequenza crescente più lunga di `a[0, i]` che termina con a_i; poniamo `seq[i]` = -1 se e solo se a_i è l'unico elemento di questa sotto-sequenza.

L'algoritmo restituisce l'array `seq` e l'intero `k` che rappresenta l'indice dell'ultimo elemento della soluzione ottima. Attraverso `seq` e `k` è possibile ricostruire tutta la sequenza ottima a ritroso: l'ultimo elemento è a_k, il penultimo è $a_{seq[k]}$, quello prima è $a_{seq[seq[k]]}$ e così via fino a che `seq[seq[...seq[k]...]] = -1`. Il ciclo tra le righe 4 e 8 cerca la sotto-sequenza più lunga da completare con `a[i]`: se questa viene trovata `k` è l'indice dell'ultimo elemento di questa e viene assegnato a `seq[i]`; altrimenti inizia una nuova sequenza con `a[i]` (righe 9-13). L'ultimo ciclo cerca la sequenza più lunga e memorizza in `k` l'indice dell'ultimo elemento che la compone. L'algoritmo richiede tempo $O(n^2)$.

Codice 6.5 L'algoritmo per il calcolo della sotto-sequenza crescente più lunga.

```
1  SottosequenzaCrescente( a ):                    ⟨pre: a array di n interi⟩
2    FOR (i = 0; i < n; i = i+1) {
3      k = 0; trovato = FALSE;
4      FOR (j = 0; j < i; j = j+1) {
5        IF (a[j] < a[i] && lsc[j] > lsc[k]) {
6          k = j; trovato = TRUE;
7        }
8      }
9      IF (trovato) {
10       lsc[i] = 1 + lsc[k]; seq[i] = k;
11     } else {
12       lsc[i] = 1; seq[i] = -1;
13     }
14   }
15   k = 0;
16   FOR (i = 0; i < n; i = i+1) {
17     IF (lsc[i] > lsc[k]) {
18       k = i;
19     }
20   }
21   RETURN <seq, k>;
```

ESEMPIO 6.5

Consideriamo la sequenza a = (9, 3, 4, 1, 6, 8, 0, 9) dell'inizio paragrafo; mostriamo i vettori `lsc` e `seq` che costruisce l'algoritmo.

	0	1	2	3	4	5	6	7
a =	9	3	4	1	6	8	0	9
lsc =	1	1	2	1	3	4	1	5
seq =	-1	-1	1	-1	2	4	-1	5

Per esempio, per `i = 4` usciamo dal ciclo nelle righe 5-9 con `k = 2` e `trovato` uguale a TRUE; nella riga 11 viene assegnato `1 + lsc[k] = 3` a `lsc[4]` e `k` a `seq[i]`. L'algoritmo restituisce

7 (`lsc[7]` è massimo) e, ovviamente, l'array `seq`. Per costruire la sequenza cercata andiamo a ritroso: l'ultimo elemento è $a_k = 9$, `seq[k] = 5` quindi l'elemento che precede 9 è $a_5 = 8$; `seq[5] = 4` quindi $a_4 = 6$ viene prima di 8; `seq[4] = 2` quindi $a_2 = 4$ precede 6; `seq[2] = 1` quindi $a_1 = 3$ precede 4; `seq[1] = -1` quindi non ci sono altri elementi. Ricapitolando, la sequenza cercata è (3, 4, 6, 8, 9).

6.4 Partizione di un insieme di interi

Consideriamo la situazione in cui abbiamo due supporti esterni, ciascuno avente capacità di `s` byte, sui quali vogliamo memorizzare `n` file (di *backup*) che occupano `2s` byte in totale, seguendo la regola che ciascun file può andare in uno qualunque dei due supporti purché il file non venga spezzato in due o più parti. Il paradigma della programmazione dinamica può aiutarci in tale situazione, permettendo di verificare se è possibile dividere i file in due gruppi in modo che ciascun gruppo occupi esattamente `s` byte. Tale problema viene detto della **partizione** (*partition*) ed è definito nel modo seguente.

Supponiamo di avere un insieme[1] di interi positivi $A = \{a_0, a_1, ..., a_{n-1}\}$ aventi somma totale (pari) $\sum_{i=0}^{n-1} a_i = 2s$: vogliamo determinare se esiste un suo sottoinsieme $A' = \{a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}\} \subseteq A$ tale che $\sum_{j=0}^{k-1} a_{i_j} = s$, vale a dire tale che la somma degli interi in A' è pari alla metà della somma di tutti gli interi in A. È chiaro che vogliamo evitare di generare esplicitamente tutti i sottoinsiemi perché un tale metodo richiede tempo esponenziale.

Definiamo il sotto-problema generale consistente nel determinare il valore booleano `T(i, j)`, per $0 \le i \le n$ e $0 \le j \le s$, che poniamo pari a TRUE se e solo se esiste un sottoinsieme di $a_0, a_1, ..., a_{i-1}$ avente somma pari a `j`: chiaramente, `T(n, s)` fornisce la soluzione del problema originario.

ESEMPIO 6.6

Se consideriamo l'istanza rappresentata dall'insieme A = {9, 7, 5, 4, 1, 2, 3, 8, 4, 3} abbiamo che `T(i, j)` sarà definito per $0 \le i \le 10$ e $0 \le j \le 23$, e che `T(3, 12)` = TRUE in quanto l'insieme A = {9, 7, 5} contiene il sottoinsieme {7, 5} la cui somma è pari a 12.

Codice 6.6 Algoritmo iterativo per il problema della partizione.

```
1  Partizione( a ):      ⟨pre: a è un array di n interi positivi la cui somma è 2s⟩
2    FOR (i = 0; i <= n; i = i+1)
3      FOR (j = 0; j <= s; j = j+1) {
4        parti[i][j] = FALSE;
5      }
```

[1] In questo paragrafo e nel successivo useremo impropriamente il termine insieme per indicare anche multi-insiemi, in quanto questi possono contenere elementi ripetuti.

```
6   parti[0][0] = TRUE;
7   FOR (i = 1; i <= n; i = i+1)
8     FOR (j = 0; j <= s; j = j+1) {
9       IF (parti[i-1][j]) {
10        parti[i][j] = TRUE;
11      }
12      IF (j >= a[i-1] && parti[i-1][ j-a[i-1] ]) {
13        parti[i][j] = TRUE;
14      }
15    }
16  RETURN parti[n][s];
```

I valori T(0, j), per $0 \le j \le s$, costituiscono l'insieme degli s + 1 sotto-problemi elementari, la cui soluzione deriva immediatamente osservando che T(0, 0) = TRUE e T(0, j) = FALSE per j > 0 poiché il sottoinsieme in questione è l'insieme vuoto con somma pari a 0. Nel caso generale, T(i, j) soddisfa la seguente regola ricorsiva:

$$T(i,j) = \begin{cases} \text{TRUE} & \text{se } i = 0 \text{ e } j = 0 \\ \text{TRUE} & \text{se } i > 0 \text{ e } T(i-1, j) = \text{TRUE} \\ \text{TRUE} & \text{se } i > 0 \text{ e } j \ge a_{i-1} \text{ e } T(i-1, j-a_{i-1}) = \text{TRUE} \\ \text{FALSE} & \text{altrimenti} \end{cases} \quad (6.5)$$

Per quanto riguarda la definizione ricorsiva dei sotto-problemi T(i,j) con $1 \le i \le n$ nella regola della (6.5), osserviamo che se T(i, j) = TRUE, allora ci sono due soli casi possibili:

- il sottoinsieme di $\{a_0, ..., a_{i-1}\}$ la cui somma è pari a j non comprende a_{i-1}: tale insieme è dunque sottoinsieme anche di $\{a_0, ..., a_{i-2}\}$ e vale T(i - 1, j) = TRUE;
- il sottoinsieme di $\{a_0, ..., a_{i-1}\}$ la cui somma è pari a j include a_{i-1}: esiste quindi in $\{a_0, ..., a_{i-2}\}$ un sottoinsieme di somma pari a $j - a_{i-1}$ per cui, se $j - a_{i-1} \ge 0$, vale T(i - 1, j - a_{i-1}) = TRUE.

Combinando i due casi sopra descritti (i soli possibili), abbiamo che T(i, j) = TRUE se e solo se T(i - 1, j) = TRUE oppure ($j - a_{i-1} \ge 0$ e T(i - 1, j - a_{i-1}) = TRUE, ottenendo così il corrispondente Codice 6.6, in cui la tabella booleana parti di taglia (n + 1) × (s + 1) viene riempita per righe in modo da soddisfare la relazione parti[i][j] = T(i, j) secondo la regola della (6.5).

Osserviamo che abbiamo introdotto (n + 1) × (s + 1) sotto-problemi e che la soluzione di un sotto-problema richiede tempo costante per la regola della (6.5): pertanto, abbiamo che l'algoritmo iterativo descritto nel Codice 6.6 richiede tempo O(ns), il quale dipende dal valore di s piuttosto che dalla sua dimensione, come discuteremo nel Paragrafo 6.8.

ESEMPIO 6.7

Consideriamo l'insieme a = {9, 7, 5, 4, 1, 2}, n = 6 e s = 14. La tabella parti è riportata di seguito.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	F	F	F	F	F	T	F	F	F	F	F
2	T	F	F	F	F	F	F	T	F	T	F	F	F	F	F
3	T	F	F	F	F	T	F	T	F	T	F	F	T	F	T
4	T	F	F	F	T	T	F	T	F	T	F	T	T	T	T
5	T	T	F	F	T	T	T	T	T	T	T	T	T	T	T
6	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Per esempio, il valore parti[4][11] = parti[3][11 - a[3]] = parti[3][7] = TRUE. L'algoritmo restituisce parti[6][14] = TRUE.

Come abbiamo discusso in precedenza, usare un algoritmo puramente ricorsivo per un problema di programmazione dinamica è inefficiente. In effetti, esiste un modo di evitare tale perdita di efficienza mantenendo la struttura ricorsiva dell'algoritmo, *prendendo nota* (tale accorgimento viene denominato *memoization* in inglese) delle soluzioni già calcolate e prevedendo, in corrispondenza a ogni chiamata ricorsiva, di verificare anzitutto se la soluzione del sotto-problema in questione è già stata calcolata e annotata, così da poterla restituire immediatamente. Ciò comporta l'uso di un array di dimensione opportuna per mantenere le soluzioni dei vari sotto-problemi considerati, prevedendo inoltre che gli elementi di tale array possano assumere un valore "indefinito", che indichiamo con il simbolo ϕ, per segnalare il caso in cui il corrispondente sotto-problema non sia ancora stato risolto.

A titolo di esempio, possiamo sviluppare un algoritmo ricorsivo per il problema della partizione che utilizza il suddetto meccanismo (ricordando comunque che è consigliabile usare la programmazione dinamica). Tale algoritmo inizializza la prima riga dell'array parti come nel Codice 6.6 e riempie le righe successive con il valore indefinito ϕ. Successivamente, l'algoritmo invoca la funzione ricorsiva che segue la regola della (6.5).

```
1  PartizioneMemoization( ):
2    FOR (j = 0; j <= s; j = j+1)
3      parti[0][j] = FALSE;
4    parti[0][0] = true;
5    FOR (i = 1; i <= n; i = i+1)
6      FOR (j = 0; j <= s; j = j+1) {
7        parti[i][j] = ϕ;
8      }
9    RETURN PartizioneRicNota( n, s );
```

```
1  PartizioneRicNota( i, j ):                    ⟨pre: 0 ≤ i ≤ n, 0 ≤ j ≤ s⟩
2    IF (parti[i][j] == ϕ) {
3      parti[i][j] = PartizioneRicNota( i-1, j );
4      IF (!parti[i][j] && (j >= a[i-1])) {
5        parti[i][j] = PartizioneRicNota( i-1, j-a[i-1] );
6      }
7    }
8    RETURN parti[i][j];
```

6.5 Problema della bisaccia

Mostriamo ora una generalizzazione del problema della partizione a un famoso problema di ottimizzazione: tale problema, denominato problema della **bisaccia** (*zaino* o *knapsack*), può essere definito, in modo pittoresco, come segue.

Supponiamo che un ladro riesca a introdursi, nottetempo, in un museo dove sono esposti una quantità di oggetti preziosi, più o meno di valore e più o meno pesanti. Tenuto conto che il ladro può trasportare una quantità massima di peso, il suo problema è selezionare un insieme di oggetti da trafugare il cui peso complessivo non superi il peso (o *possanza*) che il ladro è in grado di sopportare, massimizzando al tempo stesso il valore complessivo degli oggetti rubati.

Abbiamo quindi un multi-insieme di elementi $A = \{a_0, a_1, ..., a_{n-1}\}$ su cui sono definite le due funzioni `valore` e `peso`, le quali associano il valore e il peso a ogni elemento di A (supponiamo che sia il valore che il peso siano numeri interi positivi). Conosciamo inoltre un intero positivo `possanza`, che indica il massimo peso totale che il ladro può portare. Vogliamo determinare un sottoinsieme $A' = \{a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}\} \subseteq A$ tale che il peso totale dei suoi elementi rientri nella possanza, ovvero $\sum_{j=0}^{k-1} \texttt{peso}(a_{i_j}) \leq \texttt{possanza}$, e tale che il valore degli oggetti selezionati, ovvero $\sum_{j=0}^{k-1} \texttt{valore}(a_{i_j})$, sia il massimo possibile.

Per applicare il paradigma della programmazione dinamica possiamo definire, come sotto-problema generico, la ricerca della soluzione ottima supponendo una minore possanza e un più ristretto insieme di elementi. In altri termini, per $0 \leq i \leq n$ e $0 \leq j \leq \texttt{possanza}$, denotiamo con $P_{i,j}$ il sotto-problema relativo al caso in cui possiamo utilizzare i soli elementi $A = \{a_0, a_1, ..., a_{i-1}\}$ con il vincolo di non superare un peso pari a `j` (dove A è vuoto se `i=0`), e indichiamo con `V(i,j)` il massimo valore ottenibile in tale situazione.

Per caratterizzare i sotto-problemi elementari, osserviamo che `V(i,0) = 0`, per $0 \leq i \leq n$, in quanto il peso trasportabile è nullo e, quindi, il valore complessivo deve essere pari a 0, mentre `V(0,j) = 0`, per $0 \leq j \leq \texttt{possanza}$, poiché non ci sono elementi disponibili e il valore complessivo è necessariamente 0. Possiamo definire la decomposizione ricorsiva osservando che la soluzione di valore

massimo `V(i,j)` per il sotto-problema $P_{i,j}$ può essere ottenuta a partire da tutte le soluzioni ottime che utilizzano i soli elementi $a_0, a_1, ..., a_{i-2}$, in due soli possibili modi:

- il primo modo è che la soluzione ottima di $P_{i,j}$ non includa a_{i-1} e che, in tal caso, abbia lo stesso valore della soluzione ottima del sotto-problema $P_{i-1,j}$, ossia `V(i,j) = V(i-1,j)`;
- il secondo modo è che la soluzione ottima di $P_{i,j}$ includa a_{i-1} e che, pertanto, il suo valore sia dato dalla somma di $\texttt{valore}(a_{i-1})$ con il valore della soluzione ottima di $P_{i-1,m}$, dove $m = j - \texttt{peso}(a_{i-1})$ se $j \geq \texttt{peso}(a_{i-1})$: in tal caso, quindi, $\texttt{V}(i,j) = \texttt{V}(i-1, j - \texttt{peso}(a_{i-1})) + \texttt{valore}(a_{i-1})$.

La soluzione ottima di $P_{i,j}$ sarà quella corrispondente alla migliore delle due (sole) possibilità, ovvero $\texttt{V}(i,j) = \max\{\texttt{V}(i-1,j), \texttt{V}(i-1, j - \texttt{peso}(a_{i-1})) + \texttt{valore}(a_{i-1})\}$. Per tornare al nostro esempio figurato, supponiamo che il ladro abbia a disposizione gli elementi $a_0, a_1, ..., a_{i-1}$ e la possibilità di trasportare un peso massimo `j`: se egli decide di non prendere l'elemento a_{i-1}, allora il meglio che può ottenere è la scelta ottima tra $a_0, a_1, ..., a_{i-2}$, sempre con peso massimo `j`; se invece decide di prendere a_{i-1}, allora il meglio lo ottiene trovando la scelta migliore tra $a_0, a_1, ..., a_{i-2}$ tenendo presente che, dato che dovrà trasportare anche a_{i-1} in aggiunta agli elementi scelti, si deve limitare a un peso massimo pari a `j` decrementato del peso di a_{i-1}. La scelta migliore sarà quella che massimizza il valore.

Codice 6.7 Algoritmo iterativo per il problema della bisaccia.

```
1  Bisaccia( peso, valore, possanza ):
2  ⟨pre: peso e valore sono array di n interi positivi, possanza è un intero positivo⟩
3    FOR (i = 0; i <= n; i = i+1) {
4      FOR (j = 0; j <= possanza; j = j+1) {
5        V[i][j] = 0;
6      }
7    }
8    FOR (i = 1; i <= n; i = i+1) {
9      FOR (j = 1; j <= possanza; j = j+1) {
10       V[i][j] = V[i-1][j];
11       IF (j >= peso[i-1])) {
12         m = V[i-1][j-peso[i-1]] + valore[i-1];
13         IF (m > V[i][j]) V[i][j] = m;
14       }
15     }
16   }
17   RETURN V[n][possanza];
```

L'algoritmo iterativo descritto nel Codice 6.7 realizza tale strategia facendo uso di un array bidimensionale V di taglia (n + 1) × (possanza + 1), in cui l'elemento V[i][j] contiene il costo V(i, j) della soluzione ottima di $P_{i,j}$. Dato che il numero di sotto-problemi è O(n × possanza) e che derivare il costo di soluzione di un sotto-problema comporta il calcolo del massimo tra i costi di due altri sotto-problemi in tempo costante, ne consegue che il problema della bisaccia può essere risolto mediante il paradigma della programmazione dinamica in tempo O(n × possanza).

ESEMPIO 6.8

Consideriamo l'istanza del problema della bisaccia con i seguenti parametri: n = 5, peso = (9, 3, 7, 5, 4), valore = (7, 5, 8, 4, 8) e possanza = 12. La tabella V è riportata in seguito.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	7	7	7	7
2	0	0	0	5	5	5	5	5	5	7	7	7	12
3	0	0	0	5	5	5	5	8	8	8	13	13	13
4	0	0	0	5	5	5	5	8	9	9	13	13	13
5	0	0	0	5	8	8	8	13	13	13	13	16	17

L'algoritmo restituisce il valore 17 che risulta da V[4][8] + 8, ovvero nella soluzione ottima compare l'oggetto 4. A sua volta V[4][8] = 9 è dato da V[3][4] + 4 quindi anche l'oggetto 3 fa parte della soluzione. Infine V[3][4] = 5 = V[2][4] = valore[1] = 5, ovvero l'oggetto 1 è l'ultimo elemento della soluzione.

Un caso speciale interessante del problema della bisaccia occorre quando gli elementi sono *frazionabili*, ovvero è possibile prendere una parte di un determinato elemento (si pensi per esempio ai liquidi o elementi non numerabili tipo zucchero, farina, sabbia ...). In tale scenario, vogliamo adesso determinare, per ogni elemento a_i, la frazione dose(i) di a_i da inserire nella bisaccia in modo tale che $\sum_{i=0}^{n-1}$ peso(a_i) × dose(a_i) ≤ possanza e $\sum_{i=0}^{n-1}$ valore(a_i) × dose(a_i), sia il massimo possibile. Ovviamente, per ogni a_i in A, dose(a_i) non può essere negativo e non può essere maggiore di uno. Questa formulazione viene detta **rilassamento** del problema della bisaccia, in quanto quest'ultimo si ottiene imponendo l'ulteriore condizione dose(a_i) = 1 per 0 ≤ i ≤ n − 1.

Per il rilassamento del problema della bisaccia, non abbiamo bisogno della programmazione dinamica ma possiamo utilizzare un algoritmo goloso corretto (vedi la discussione nella seconda metà del Paragrafo 6.2). Intuitivamente conviene inserire nella bisaccia quanto più possibile l'elemento con il più alto valore per unità di peso. Se dopo questa prima operazione nella bisaccia resta altro spazio proseguiamo passando al secondo elemento col più alto valore per unità di peso. Proseguiamo in questo modo fintanto che resta spazio nella bisaccia. Se riempiamo perfettamente lo zaino, possiamo terminare. Altrimenti, troviamo un elemento che non riesce a entrare interamente nello zaino: basta prendere allora la sua frazione che entra nello zaino e terminiamo. Notare che frazioniamo un solo elemento in questo modo.

Nel Codice 6.8 viene formalizzato questo algoritmo, il quale restituisce l'array di n elementi dose. Per ogni i, dose[i] indica la frazione di a_i nella soluzione costruita. Questo array viene inizializzato a zero nelle righe 2-5 insieme con l'array valoreUnitaPeso che memorizza i rapporti valore[i]/peso[i]. Quest'ultimo array viene ordinato in modo non crescente rispetto al primo elemento delle coppie che lo compongono (riga 6). La variabile pesoRaggiunto indica il peso corrente degli elementi attualmente inseriti nella bisaccia. All'interno del ciclo (righe 8-17) si considera l'elemento e che fra gli elementi rimasti ha il maggior rapporto valore/peso: se il peso di e sommato al pesoRaggiunto è inferiore a possanza (riga 10) l'elemento e viene aggiunto per intero alla soluzione (riga 11); altrimenti, si sceglie la frazione dell'elemento e sufficiente a riempire la bisaccia (riga 14).

Codice 6.8 Soluzione golosa (corretta) per la bisaccia con elementi frazionabili.

```
1  BisacciaFrazionabili( valore, peso, possanza ):
2    FOR (i = 0; i < n; i = i+1) {
3      valoreUnitaPeso[i] = <valore[i]/peso[i], i>;
4      dose[i] = 0;
5    }
6    OrdinaDescescente( valoreUnitaPeso );
7    pesoRaggiunto = 0;
8    FOR (i = 0; i < n; i = i+1) {
9      <x, e> = valoreUnitaPeso[i];
10     IF (pesoRaggiunto + peso[e] < possanza) {
11       dose[e] = 1;
12       pesoRaggiunto = pesoRaggiunto + peso[e];
13     } ELSE {
14       dose[e] = (possanza - pesoRaggiunto)/peso[e];
15       pesoRaggiunto = possanza;
16     }
17   }
18   RETURN dose;
```

Prima di dimostrare la correttezza dell'algoritmo osserviamo che il costo computazionale è dominato da quello dell'ordinamento, per cui la complessità totale è O(n log n) tempo.

Teorema 6.2 *L'algoritmo mostrato nel Codice 6.8 restituisce la soluzione ottima.*

Dimostrazione Innanzi tutto osserviamo che se $\sum_{i=0}^{n-1} \texttt{peso}(a_i) \leq \texttt{possanza}$ tutti gli elementi verrebbero presi per intero ottenendo la soluzione ottima. Quindi assumiamo che $\sum_{i=0}^{n-1} \texttt{peso}(a_i) > \texttt{possanza}$, in questo caso osserviamo che la bisaccia viene riempita per intero in quanto esisterà un j tale che $\texttt{pesoRaggiunto} + \texttt{peso}(a_j) \geq \texttt{possanza}$ e quindi $\texttt{dose}(a_j) = (\texttt{possanza} - \texttt{pesoRaggiunto}) / \texttt{peso}(a_j)$. Pertanto, $\texttt{pesoRaggiunto} + \texttt{peso}(aj) \times \texttt{dose}(a_j) = \texttt{possanza}$.

Supponiamo per assurdo che la soluzione trovata dall'algoritmo non sia ottima. Allora esiste una funzione ottima dose′ tale che

$$\sum_{i=0}^{n-1} \texttt{dose}'(a_i) \times \texttt{valore}(a_i) > \sum_{i=0}^{n-1} \texttt{dose}(a_i) \times \texttt{valore}(a_i).$$

Anche per dose′ deve risultare $\sum_{i=0}^{n-a} \texttt{peso}(a_i) \times \texttt{dose}'(a_i) = \texttt{possanza}$. Visto che gli elementi sono ordinati per rapporto valore/peso non crescente, siano $j < k$ i più piccoli indici per cui $\texttt{dose}'(a_j) < 1$ e $\texttt{dose}'(a_k) > 0$. Se j e k non esistessero, $j = k$ e quindi dose′ = dose. Esiste $\varepsilon > 0$ tale che $\texttt{dose}'(a_j) + \varepsilon \leq 1$ e $\texttt{dose}'(a_k) - \varepsilon \geq 0$. La soluzione dose″ così definita

$$\texttt{dose}''(a_i) = \begin{cases} \texttt{dose}'(a_i) + \varepsilon & \text{se } i = j; \\ \texttt{dose}'(a_i) - \varepsilon & \text{se } i = k; \\ \texttt{dose}'(a_i) & \text{altrimenti.} \end{cases}$$

vale $\varepsilon(\texttt{valore}(a_j) - (\texttt{valore}(a_k))$ più di dose′, quindi quest'ultima non può essere ottima. ❑

ESEMPIO 6.9

Consideriamo la stessa istanza dell'Esempio 6.8: n = 5, peso = (9, 3, 7, 5, 4), valore = (7,5, 8, 4, 8) e possanza = 12. Dopo aver ordinato gli elementi in modo non decrescente rispetto al rapporto valore/peso otteniamo il seguente array valoreUnitaPeso.

$$(\langle 2, 4\rangle, \langle 5/3, 1\rangle, \langle 8/7, 2\rangle, \langle 4/5, 3\rangle, \langle 7/9, 0\rangle).$$

L'elemento 4 viene aggiunto alla soluzione per intero: quindi dose[4] = 1 e pesoRaggiunto = peso[4] = 4. Segue l'elemento 1, il suo peso sommato a pesoRaggiunto è ancora inferiore a possanza, quindi dose[1] = 1 e pesoRaggiunto = 7. Infine peso[2] + pesoRaggiunto = 14 > possanza allora dose[2] = (possanza – pesoRaggiunto)/peso[2] = 5/7. Tutti gli altri elementi dell'array dose sono uguali a 0. Il valore della soluzione ottenuta è 8 + 5 + 8 × 5/7 = 131/7 ≈ 18.714.

6.6 Massimo insieme indipendente in un albero

Passiamo ora a illustrare un esempio di programmazione dinamica che non si basa su sequenze o insiemi ma bensì su strutture non-lineari. Dato un albero T di n nodi che compongono l'insieme $V = \{u_0, \ldots, u_{n-1}\}$, sia $r \in V$ la sua radice: un **insieme indipendente** di T è un sottoinsieme I di V tale che per ogni coppia u e v in I né u è padre di v né v è padre di u. Ovvero per ogni coppia di nodi in I non vi è una relazione diretta padre-figlio. Il problema del **massimo insieme indipendente** consiste nel trovare un insieme indipendente di cardinalità massima.

Limitiamoci a calcolare la dimensione del massimo insieme indipendente di T e indichiamo con T_u il sotto-albero di T che ha per radice il nodo u. Il sottoproblema generale che risolveremo è il seguente: per ogni nodo u dell'albero trovare la dimensione del massimo insieme indipendente di T_u che contiene u e la dimensione del massimo insieme indipendente di T_u che *non* contiene u. Chiamando $\texttt{size}_T(u)$ e $\texttt{size}_F(u)$ queste due quantità, ovviamente il massimo insieme indipendente dell'albero ha cardinalità $\max\{\texttt{size}_T(r), \texttt{size}_F(r)\}$.

Se u è una foglia $\texttt{size}_T(u) = 1$ e $\texttt{size}_F(u) = 0$. Altrimenti siano $v_0, v_2, \ldots, v_{k-1}$ i figli del nodo u. Se u appartiene al massimo insieme indipendente nessuno dei suoi figli vi può far parte quindi

$$\texttt{size}_T(u) = 1 + \sum_{i=0}^{k-1} \texttt{size}_F(v_i).$$

Viceversa se u non appartiene al massimo insieme indipendente di T allora i suoi figli vi possono far parte oppure no, pertanto

$$\texttt{size}_F(u) = \sum_{i=0}^{k-1} \max\{\texttt{size}_T(v_i), \texttt{size}_F(v_i)\}.$$

Osserviamo che il calcolo ricorsivo della coppia $\langle \texttt{size}_T(u), \texttt{size}_F(u)\rangle$ per ogni nodo u dell'albero induce una visita posticipata dell'albero. Il Codice 6.9 mostra l'algoritmo risultante.

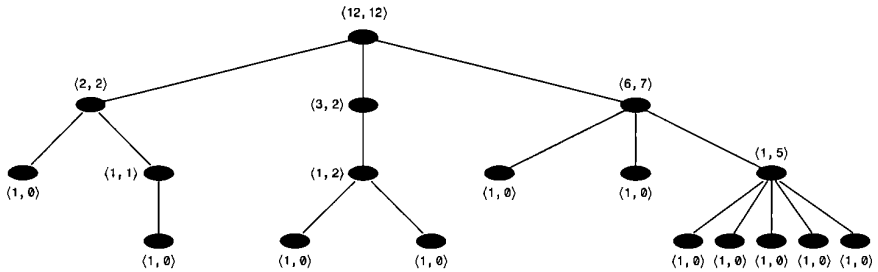
Nel Codice 6.9 ipotizziamo che T sia un albero ordinale (Paragrafo 1.4.2). Per ogni nodo u dell'albero u.primo indica la lista contenente i figli di u e, quindi, possiamo capire se u è una foglia valutando se la sua lista è vuota. Se v è figlio di u allora v.fratello è il successivo figlio di u. L'algoritmo MassimoInsiemeIndipendente restituisce $\max\{\texttt{size}_T(r), \texttt{size}_F(r)\}$ utilizzando la funzione Size che non è altro che una visita posticipata opportunamente modificata. Pertanto la complessità dell'algoritmo è data dalla complessità della procedura di visita, ossia O(n).

Codice 6.9 L'algoritmo per il calcolo del massimo insieme indipendente di un albero utilizza la funzione `Size` che restituisce la coppia $\langle \texttt{size}_T(u), \texttt{size}_F(u) \rangle$.

```
Size( u ):
  IF (u.primo == null) {
    RETURN <1, 0>;
  } ELSE {
    sizeT = 1; sizeF = 0;
    FOR (v = u.primo; v != NULL; v = v.fratello) {
      <vsizeT, vsizeF> = Size( v );
      sizeT = sizeT + vsizeF;
      sizeF = sizeF + max( vsizeT, vsizeF );
    }
    RETURN <sizeT, sizeF>;
  }

MassimoInsiemeIndipendente( T ):          ⟨pre: l'albero T con radice r⟩
  <sizeT,sizeF> = Size( r );
  RETURN max( sizeT, sizeF );
```

ESEMPIO 6.10

Di seguito è mostrato un albero in cui per ogni nodo u è indicata la coppia $\langle \texttt{size}_T(u), \texttt{size}_F(u) \rangle$. I valori sulla radice ci dicono che il massimo insieme indipendente di T ha cardinalità 12.



Osserviamo che $\texttt{size}_T(r) = \texttt{size}_F(r)$ quindi la radice può o non può far parte della soluzione ottima. Ovvero esiste almeno una soluzione ottima che la contiene ed almeno una che non la contiene.

Il paradigma della programmazione dinamica non appare immediatamente a un esame del Codice 6.9. Per renderci conto, dovremmo utilizzare due tabelle `sizeT` e `sizeF` di programmazione dinamica e numerare i nodi dell'albero in ordine posticipato. A questo punto, dovremmo calcolare `sizeT[u]` e `sizeF[u]` utilizzando i valori che appaiono nelle posizioni precedenti: tale schema è realizzato indirettamente nel Codice 6.9 evitando due chiamate ricorsive per u, ma facendo restituire all'unica chiamata ricorsiva su u sia `sizeT[u]` che `sizeF[u]`.

Esercizio svolto 6.4 Dato un albero binario pesato T di n nodi, dove r indica la sua radice, indichiamo il campo peso di un nodo u con `u.peso`: un **ricoprimento** di T è un sottoinsieme R di vertici tale che per ogni collegamento diretto (padre-figlio) tra due nodi u e v in T vale che $u \in R$ oppure $v \in R$. Ovvero per ogni coppia di nodi collegati da una relazione diretta padre-figlio, almeno uno dei due nodi è in R. Il costo di un ricoprimento R è dato dalla somma dei pesi dei nodi in esso contenuti, $\texttt{costo}(R) = \sum_{u \in R} \texttt{u.peso}$. Risolvere il problema del **minimo ricoprimento** per un albero T nel quale vogliamo trovare il costo minimo di un ricoprimento di T.

Soluzione Ricalchiamo la soluzione proposta nel Codice 6.9 con le dovute differenze: il problema è di minimizzazione invece che di massimizzazione, l'albero è binario e vogliamo ottimizzare il costo piuttosto che la cardinalità dell'insieme dei nodi scelti.

Per ogni nodo u dell'albero troviamo il costo del ricoprimento minimo di T_u che contiene u e il costo del ricoprimento minino di T_u che non contiene u. Chiamiamo $\texttt{costo}_T(u)$ e $\texttt{costo}_F(u)$ queste due quantità e poniamole a zero quando u è vuoto. Il minimo ricoprimento ha costo $\texttt{min}\{\texttt{costo}_T(r), \texttt{costo}_F(r)\}$. Il caso base, quando u è una foglia, ha $\texttt{costo}_T(u) = \texttt{u.peso}$ e $\texttt{costo}_F(u) = 0$. Per un nodo interno u, se u non appartiene al ricoprimento minimo, i suoi figli devono farne parte

$$\texttt{costo}_F(u) = \texttt{costo}_T(\texttt{u.sx}) + \texttt{costo}_T(\texttt{u.dx}).$$

Viceversa se u appartiene al ricoprimento minimo di T allora i suoi figli vi possono far parte oppure no, pertanto

$$\texttt{costo}_T(u) = \texttt{u.peso} + \texttt{min}\{\texttt{costo}_T(\texttt{u.sx}), \texttt{costo}_F(\texttt{u.sx})\} + \texttt{min}\{\texttt{costo}_T(\texttt{u.dx}), \texttt{costo}_F(\texttt{u.dx})\}.$$

A questo punto il calcolo procede in maniera analoga al Codice 6.9, come mostrato nel Codice 6.10.

Codice 6.10 L'algoritmo per il calcolo del costo del minimo ricoprimento di un albero utilizza la funzione `Costo` che restituisce la coppia $\langle \texttt{costo}_T(u), \texttt{costo}_F(u) \rangle$.

```
Costo( u ):
  IF (u == null) {
    RETURN <0, 0>;
  } ELSE IF (u.sx == u.dx == null) {
    RETURN <u.peso, 0>;
  } ELSE {
    <sxcostoT, sxcostoF> = Costo( u.sx );
    <dxcostoT, dxcostoF> = Costo( u.dx );
    costoF = sxcostoT + dxcostoT;
```

```
10      costoT = u.peso + min( sxcostoT, sxcostoF ) +
                  min( dxcostoT, dxcostoF );
11      RETURN <costoT, costoF>;
12    }
```

```
1   MinimoRicoprimento( T ): ⟨pre: l'albero T con radice r⟩
2     <costoT, costoF> = Costo( r );
3     RETURN min( costoT, costoF );
```

6.7 Alberi di ricerca ottimi

Gli alberi binari di ricerca bilanciati (Paragrafo 4.4) garantiscono che ogni operazione di ricerca ha tempo di esecuzione logaritmico nella dimensione dell'albero. In alcuni casi questo potrebbe non bastare. Supponiamo che alcune chiavi siano ricercate più frequentemente di altre, ovviamente converrebbe che queste siano più vicine possibile alla radice dell'albero.

ESEMPIO 6.11

Consideriamo un insieme di 4 elementi aventi le chiavi {3, 5, 7, 9}. Nella parte sinistra della seguente figura queste sono organizzate in un albero AVL mentre a destra lo sono in un albero binario di ricerca molto sbilanciato.

Tuttavia osserviamo che la sequenza di ricerche (5, 3, 3, 9, 3, 7, 5, 3) induce $2 \times 4 + 3 \times 2 + 1 + 2 = 17$ confronti sull'albero AVL e $1 \times 4 + 2 \times 2 + 4 + 3 = 15$ sull'albero molto sbilanciato. In media ogni ricerca richiede $17/8 = 2.125$ confronti nell'albero AVL e $15/8 = 1.875$ confronti sull'albero molto sbilanciato.

Quindi se associato a ogni elemento dell'insieme abbiamo anche l'informazione sulla probabilità che questo elemento venga ricercato tramite la sua chiave, possiamo costruire alberi di ricerca nei quali il tempo medio per la ricerca è inferiore a quello che si avrebbe sugli alberi di ricerca bilanciati.

Supponiamo di avere n elementi; associate a ogni elemento i abbiamo una chiave k_i intera e una probabilità p_i che l'elemento venga ricercato. Assumiamo senza perdita di generalità che $k_0 \le k_1 \le \ldots \le k_{n-1}$. Se gli elementi sono in un albero binario di ricerca per il quale risulta che, per ogni elemento i, la distanza di i dalla radice è d_i allora il numero medio di confronti per la ricerca di una chiave è

$$\sum_{i=0}^{n-1} p_i \, (d_i + 1) \tag{6.6}$$

poiché $d_i + 1$ nodi sono attraversati nel cammino dalla radice al nodo contenente k_i e questo accade con probabilità p_i.

Vogliamo organizzare gli n elementi in un albero binario di ricerca sul quale il *tempo di ricerca medio* (ovvero il numero medio di confronti) delle chiavi espresso sia asintoticamente minimo, ossia sia proporzionale alla formula (6.6). Tali alberi sono chiamati **alberi di ricerca ottimi**.

Utilizzeremo il paradigma della programmazione dinamica per risolvere questo problema. Per $0 \le u \le v \le n-1$ identifichiamo con $t(u,v)$ il tempo di ricerca medio nell'albero di ricerca ottimo contenente gli elementi $\{u, u+1, \ldots, v\}$ in base alle loro probabilità $p_u, p_{u+1}, \ldots, p_v$.[2] Ovviamente il valore della soluzione ottima è $t(0, n-1)$. Il caso base occorre quando $v < u$, cioè l'albero è vuoto e pertanto $t(u, v) = 0$. Nel caso generale, $u \le v$, possiamo scrivere la regola ricorsiva per il nostro schema di programmazione dinamica:

Lemma 6.1 *Per* $0 \le u \le v \le n-1$,

$$t(u, v) = \sum_{i=u}^{v} p_i + \min_{w=u,\ldots,v} \{t(u, w-1) + t(w+1, v)\}. \tag{6.7}$$

Dimostrazione Sia T_{uv} l'albero di ricerca ottimo contenente gli elementi $\{u, u+1, \ldots, v\}$ in base alle loro probabilità $p_u, p_{u+1}, \ldots, p_v$. Indicando con d'_i la distanza in T_{uv} del nodo con chiave k_i, per $u \le i \le v$, abbiamo che il tempo di ricerca medio $t(u, v)$ può essere espresso come

$$t(u, v) = \sum_{i=u}^{v} p_i \, (d'_i + 1). \tag{6.8}$$

Supponiamo di scegliere w come radice di T_{uv} e supponiamo di aver già calcolato $t(u, w-1)$ e $t(w+1, v)$. Il nostro scopo è di esprimere la relazione (6.8) in funzione di $t(u, w-1)$ e $t(w+1, v)$. A tale proposito osserviamo che il tempo di ricerca medio per w è $1 \times p_w$ in quanto è la radice di T_{uv} e quindi $d_w = 0$. Nei sotto-alberi sinistro e destro di w abbiamo che i loro costi ottimi sono ottenuti prendendo $d'_i - 1$ come distanza dalla loro radice, in quanto quest'ultima è una delle figlie di w.

$$t(u, w-1) = \sum_{i=u}^{w-1} p_i d'_i \quad \text{e} \quad t(w+1, v) = \sum_{i=w+1}^{v} p_i d'_i$$

[2] Notare che la somma di tali probabilità può essere inferiore a 1, in quanto a noi interessa considerare la somma pesata espressa nella (6.6) dove le distanze sono relative all'albero corrente, piuttosto che lo spazio delle probabilità indotte dalle corrispettive chiavi di ricerca.

Possiamo ora riscrivere la (6.8) come

$$t(u, v) = p_w + \sum_{i=u}^{w-1} p_i \left(d'_i + 1\right) + \sum_{i=w+1}^{v} p_i \left(d'_i + 1\right) = \sum_{i=u}^{v} p_i + t(u, w-1) + t(w+1, v).$$

Non conoscendo a priori quale sia la scelta ottima per w, dobbiamo considerare il minimo tra le scelte di w = u, ..., v, dimostrando così la relazione (6.7). ❑

Dalle considerazioni appena fatte scaturisce l'algoritmo illustrato nel Codice 6.11 che costruisce la tabella `tempo` per contenere i valori `t(u, v)`. Tuttavia, per simulare la condizione al contorno `t(u, v) = 0` quando `u > v`, utilizziamo una tabella di dimensione `(n+1)×(n+1)` per `tempo`: infatti per calcolare `t(0,1)` abbiamo bisogno di `t(0,-1) = 0` e per calcolare `t(n-1, n-1)` abbiamo bisogno di `t(n, n-1) = 0`. Quindi ricorriamo all'artificio di usare una riga e una colonna in più memorizzando in `tempo[u][v+1]` l'effettivo valore di `t(u, v)`.

Codice 6.11 Algoritmo che restituisce il costo dell'albero di ricerca ottimo.

```
1  Tempo( p ):                          ⟨pre: array p delle probabilità di n elementi⟩
2    FOR (u = 0; u <= n; u = u+1) {
3      FOR (v = 0; v <= n; v = v+1)
4        tempo[u][v] = 0;
5    }
6    FOR (diagonale = 1; diagonale < n; diagonale = diagonale+1) {
7      FOR (u = 0; u < n-diagonale; u = u+1) {
8        v = u + diagonale;
9        minimo = +∞;
10       FOR (w = u; w < v; w = w+1) {
11         IF (tempo[u][w-1]+tempo[w+1][v] < minimo) {
12           minimo = tempo[u][w-1]+tempo[w+1][v];
13         }
14       }
15       tempo[u][v] = sommap( u, v-1 ) + minimo;
16     }
17   }
18   RETURN tempo[0][n]
```

```
1  sommap( x, y ):
2    s = 0;
3    FOR (i = x; i <= y; i = i+1)
4      s = s + p[i];
5    RETURN s;
```


Osserviamo che `tempo` è una tabella triangolare superiore poiché la diagonale principale e la parte triangolare inferiore è composta da tutti zeri (vedi Esempio 6.12). Ricordiamo che una diagonale `diagonale` è composta da tutte le posizioni u e v tali che `v - u = diagonale`. Il codice quindi procede a riempire `tempo` per `diagonale` = 1, ..., n - 1. Prese le posizioni u e v su tale `diagonale`, possiamo osservare che sono garantite le condizioni al contorno `tempo[u][u - 1] = 0` e `tempo[v][v] = 0` perché sono nella parte triangolare inferiore o sulla diagonale principale. Possiamo quindi calcolare il minimo secondo la relazione (6.7) scegliendo per w = u, ..., v in O(n) tempo. Rimane quindi da calcolare `sommap(u, v-1)` = $\sum_{i=u}^{v-1} p_i$ tenendo presente che abbiamo traslato `tempo[u][v + 1] = t(u, v)` di una posizione a destra: tale somma richiede O(n) tempo e, anche se possiamo calcolarla in tempo costante (vedi esercizio 6.15), questo non migliora il costo finale. La complessità dell'algoritmo è quindi $O(n^3)$ tempo, poiché ci sono $O(n^2)$ entrate nella tabella `tempo` e il calcolo di ogni elemento richiede O(n) tempo.

ESEMPIO 6.12

Consideriamo i dati dell'Esempio 6.11. Abbiamo 4 elementi con chiavi (3, 5, 7, 9) e con probabilità, nell'ordine, (1/2, 1/4, 1/8, 1/8). Dopo la fase di inizializzazione e l'esecuzione del ciclo per `diagonale` = 1, la tabella `tempo` appare come segue.

	0	1	2	3	4
0	0	1/2	0	0	0
1	0	0	1/4	0	0
2	0	0	0	1/8	0
3	0	0	0	0	1/8
4	0	0	0	0	0

Per `diagonale` = 2 il primo elemento della tabella da considerare è `tempo[0][2]` = t(0, 1). Il valore di `minimo` viene ottenuto come il minimo tra `tempo[0][0] + tempo[1][2]` = 1/4 e `tempo[0][1] + tempo[2][2]` = 1/2, a cui va aggiunto `sommap(0,1)` = $p_0 + p_1$ = 3/4, quindi `tempo[0][2]` = 1. Continuando in questo modo costruiamo la tabella `tempo` come mostrato nella figura seguente

	0	1	2	3	4
0	0	1/2	1	11/8	15/8
1	0	0	1/4	1/2	7/8
2	0	0	0	1/8	3/8
3	0	0	0	0	1/8
4	0	0	0	0	0

Il numero di confronti medi lo leggiamo in `tempo[0][4]` = 15/8.

6.8 Pseudo-polinomialità e programmazione dinamica

Concludiamo il capitolo sulla programmazione dinamica commentando la complessità computazionale in tempo derivata con tale paradigma. Ricordiamo che gli algoritmi presentati per le sottosequenze nel Paragrafo 6.3 e per gli alberi nei Paragrafi 6.6 e 6.7 hanno complessità polinomiale nel numero di elementi in ingresso.

Gli altri problemi hanno invece algoritmi il cui costo dipende anche da alcuni dei *valori* degli elementi, non solo dal loro *numero*. Per il problema del resto e per il problema della partizione di n interi di somma totale $2s$, abbiamo un costo pari a $O(mR)$ per il primo e $O(ns)$ per il secondo, che sono polinomiali in m, n, s e R ma *non* lo sono necessariamente nella dimensione dei dati di ingresso. Ciò deriva dal fatto che s e R sono valori che possono essere esponenzialmente più grandi del numero di elementi coinvolti.

Prendendo come riferimento il problema della partizione, pur avendo n interi da partizionare, ciascuno di essi richiede $k = O(\log s)$ bit di rappresentazione. Quindi la dimensione dei dati è nk mentre il costo dell'algoritmo è $O(ns) = O(n2^k)$: tale costo non è polinomiale rispetto alla dimensione dei dati e, per questo motivo, l'algoritmo viene detto **pseudo-polinomiale**, in quanto il suo costo è polinomiale solo se si usano interi piccoli rispetto a n (per esempio, quando $s = O(n^c)$ per una costante $c > 0$). Anche l'algoritmo discusso per il problema della bisaccia è pseudo-polinomiale in quanto richiede $O(n \times \text{possanza}) = O(n2^k)$ tempo, mentre la dimensione dei dati in ingresso richiede $O(nk)$ bit dove $k = O(\log(\text{possanza}))$. Anche in questo caso, il costo dell'algoritmo non è polinomiale, ma lo diviene nel momento in cui il valore della possanza è polinomiale rispetto al numero di oggetti. Notare che anche la soluzione immediata di costo $O(2^n nk)$ è pseudo-polinomiale, visto che k può essere molto grande rispetto a n, mentre il rilassamento della bisaccia con elementi frazionabili è polinomiale. In generale, per tutti i problemi che coinvolgono quantità numeriche che possono crescere velocemente rispetto al numero n dei dati in ingresso, è opportuno adottare il costo non-uniforme per il modello RAM, in cui ciascuna operazione su interi di k bit richiede un tempo di esecuzione che aumenta al crescere di k.

Da ciò consegue che, mentre gli algoritmi sulle sotto-sequenze e sugli alberi visti in questo capitolo sono polinomiali a tutti gli effetti, gli altri algoritmi sono solo apparentemente polinomiali: l'anomalia deriva dal fatto che i rispettivi problemi sono NP-completi, come vedremo nel seguito del libro (da notare, però, che non è vero che ogni problema NP-completo ammetta un algoritmo pseudo-polinomiale).

6.9 Esercizi

6.1 Mostrare come calcolare il numero di Fibonacci F_n in tempo $O(\log n)$ utilizzando il calcolo veloce di A^n e scegliendo un'opportuna matrice binaria A di taglia 2×2.

6.2 Modificare il Codice 6.1 con la condizione aggiuntiva di avere a disposizione e_i monete di valore v_i (e ogni soluzione deve quindi rispettare $0 \leq n_i \leq e_i$).

6.3 Dimostrare che il Codice 6.3 per il calcolo del resto restituisce la soluzione ottima quando i possibili valori delle monete sono 25, 10, 5, 1.

6.4 Il problema della distanza di edit tra due stringhe x e y chiede di calcolare il minimo numero di operazioni su singoli caratteri (inserimento, cancellazione e sostituzione) per trasformare x in y (o viceversa). Fornire un algoritmo quadratico di programmazione dinamica per calcolare la distanza di edit tra x e y.

6.5 Fornire un algoritmo per il problema della sotto-sequenza crescente ottima che richieda tempo provatamente più piccolo di $O(n^2)$.

6.6 Progettare gli algoritmi per stampare uno dei due insiemi ottenuti nel problema della partizione e il contenuto ottimo della bisaccia degli elementi, analogamente a quanto visto per il problema del resto e per le sotto-sequenze comuni più lunghe.

6.7 Formulare il problema della partizione come un'istanza del problema della bisaccia. Progettare un algoritmo ricorsivo con presa di nota (*memoization*) per il problema della bisaccia.

6.8 Adattare l'algoritmo utilizzato per il problema della bisaccia per risolvere il problema del resto. Come cambia la complessità?

6.9 Riconsiderate la strategia SJF (Shortest Job First) per lo scheduling di programmi descritta nel Capitolo 1 alla luce degli algoritmi golosi. Questa ordina i programmi da eseguire sulla CPU in base al loro tempo di esecuzione e, quindi, li assegna a essa in base a tale ordinamento. Dimostrare l'ottimalità di SJF con una tecnica simile a quella utilizzata per la bisaccia con elementi frazionari nel Teorema 6.2.

6.10 Il codice di Huffman per una sequenza S di n caratteri codifica ciascun carattere di S con un codice a lunghezza variabile al fine di utilizzare, se possibile, meno bit per rappresentare S. Prendiamo come esempio S = abracadabra di $n = 11$ caratteri. Con il codice ASCII normalmente usato, abbiamo 88 bit (8 bit per carattere). Vi sono tuttavia $n_a = 5$ caratteri a, $n_b = n_r = 2$ caratteri b e r e, infine, $n_c = n_d = 1$ carattere c e d. Associamo codici più lunghi ai caratteri meno frequenti. Prendiamo i due meno frequenti, c e d, e gli assegniamo 0 e 1 rispettivamente. Adesso li consideriamo un unico carattere

cd di frequenza $n_{cd} = n_c + n_d = 2$. Ripetiamo, estraendo, per esempio, b e r e assegnando loro 0 e 1, considerandoli un unico carattere br di frequenza $n_{br} = n_b + n_r = 4$. Estraiamo cd e br aventi frequenza minima, assegniamo loro 0 e 1, considerandoli un unico carattere cdbr di frequenza $n_{cdbr} = 6$. Infine estraiamo a e cdbr, assegnando loro 0 e 1. Se ricostruiamo a ritroso i bit b_c via via assegnati a ciascun $c \in \Sigma$, otteniamo i seguenti codici, $b_a = 0$, $b_b = 100$, $b_r = 101$, $b_c = 110$ e $b_b = 111$, per un totale di $\sum_c |b_c| \times n_c = 23$ bit invece che 88.

(a) Dimostrare che i codici b_c ottenuti sono liberi da prefissi, ossia nessuno è prefisso dell'altro, costruendo il loro trie binario e osservando che le foglie sono in corrispondenza biunivoca con i caratteri di Σ e i nodi interni corrispondono ai raggruppamenti di caratteri.

(b) Mostrare che i codici di Huffman ottengono il minimo numero totale di bit, $\sum_c |b_c| \times n_c$, tra i codici b_c liberi da prefissi, con una tecnica simile a quella utilizzata per la bisaccia con elementi frazionari nel Teorema 6.2.

(c) Descrivere l'algoritmo di costruzione dei codici b_c in modo goloso e discuterne la correttezza e la complessità.

6.11 Progettare l'algoritmo per stampare il massimo insieme indipendente di un albero.

6.12 Modificare il Codice 6.9 in modo da calcolare il numero di massimi insiemi indipendenti di un albero.

6.13 Progettare l'algoritmo per stampare il ricoprimento minimo di un albero.

6.14 Dimostrare che un albero di ricerca ottimo per un insieme di chiavi equiprobabili è bilanciato.

6.15 Progettare un algoritmo che, memorizzando le somme prefisse delle probabilità $s_i = \sum_{j=0}^{i} p_j$ per $0 \le i \le n-1$ in $O(n)$ tempo totale, possa poi rendere possibile il calcolo veloce di sommap in $O(1)$ tempo nel Codice 6.11.

6.16 Modificare il Codice 6.11 per calcolare la tabella radice, in cui radice[u][v] indica la radice w dell'albero di ricerca ottimo T_{uv} per u, u + 1, ..., v. Progettare un algoritmo che, a partire dalla tabella radice, costruisca l'albero di ricerca ottimo: se questo ha radice w = radice[0][n - 1] allora il figlio sinistro di w è radice[0][w - 1] e quello destro radice[w + 1][n - 1] e così via.

6.17 Progettare un algoritmo di programmazione dinamica per trovare la sequenza di moltiplicazioni che minimizzi il costo complessivo del prodotto $A^* = A_0 \times A_1 \times \cdots \times A_{n-1}$ di n matrici, dove la loro taglia è specificata mediante una sequenza di $n+1$ interi positivi $d_0, d_1, \ldots, d_n$: la matrice A_i ha taglia $d_i \times d_{i+1}$ per $0 \le i \le n-1$. Ipotizzare che il costo della moltiplicazione di due matrici di taglia $r \times s$ e $s \times t$ sia proporzionale a $r \times s \times t$.

CAPITOLO

7 Grafi

In questo capitolo esaminiamo le caratteristiche principali dei grafi, fornendo le definizioni relative a tali strutture. Mostriamo come attraverso di essi sia possibile modellare una quantità di situazioni e come molti problemi possano essere interpretati come problemi su grafi, descrivendo alcuni algoritmi di base per operare su di essi.

7.1 Grafi
7.2 Opus libri: Web crawler e visite di grafi
7.3 Applicazioni delle visite di grafi
7.4 Opus libri: routing su Internet e cammini minimi
7.5 Opus libri: data mining e minimi alberi ricoprenti
7.6 Esercizi

7.1 Grafi

I grafi rappresentano una generalizzazione della relazione espressa da liste e alberi: il collegamento tra due nodi nelle liste rappresenta la relazione tra predecessore e successore, mentre il collegamento negli alberi rappresenta la relazione tra figlio e padre; nel caso di grafi, il collegamento tra due nodi rappresenta una relazione binaria, espressa come adiacenza o vicinanza tra tali nodi. L'importanza dei grafi deriva dal fatto che una grande quantità di situazioni può essere modellata e rappresentata mediante essi, e quindi una grande quantità di problemi può essere espressa per mezzo di problemi su grafi: gli algoritmi efficienti su grafi rappresentano alcuni strumenti generali per la risoluzione di numerosi problemi di rilevanza pratica e teorica.

Un grafo `G` è definito come una coppia di insiemi finiti `G = (V, E)`, dove `V` rappresenta l'insieme dei **nodi** o **vertici** e le coppie di nodi in $E \subseteq V \times V$ sono chiamate **archi** o **lati**. Il numero $n = |V|$ di nodi è detto **ordine** del grafo, mentre $m = |E|$ indica il numero di archi (i due numeri possono essere estremamente variabili, l'uno rispetto all'altro).

La **dimensione** del grafo è data dal numero $n + m$ totale di nodi e di archi, per cui la dimensione dei dati in ingresso è espressa usando due parametri nel caso dei grafi, contrariamente al singolo parametro adottato per la dimensione di array, liste e alberi. Infatti, n e m vengono considerati parametri indipendenti, per cui nel caso di grafi la complessità lineare viene riferita al costo $O(n + m)$ di *entrambi* i parametri. In generale, poiché l'arco `(u, v)` è considerato uguale all'arco `(v, u)`, vale $0 \leq m \leq \binom{n}{2}$ poiché il numero massimo di archi è dato dal numero $\binom{n}{2} = O(n^2)$ di tutte le possibili coppie di nodi: il grafo è **sparso** se $m = O(n)$ e **denso** se $m = \Theta(n^2)$.

Nella trattazione di grafi un arco viene inteso come un collegamento tra due nodi `u` e `v` e viene rappresentato con la notazione `(u, v)` (che, come vedremo, è un piccolo abuso per semplificare la notazione del libro). Ciò è motivato dalla descrizione grafica utilizzata per rappresentare grafi, in cui i nodi sono elementi grafici (punti o cerchi) e gli archi sono linee colleganti le relative coppie di nodi, questi ultimi detti **terminali** o **estremi** degli archi.

Consideriamo l'esempio mostrato nella Figura 7.1, che riporta le rotte di una nota compagnia aerea relativamente all'insieme `V` degli aeroporti dislocati presso alcune capitali europee, identificate mediante il codice internazionale del corrispondente aeroporto: `BVA` (Parigi), `CIA` (Roma), `CRL` (Bruxelles), `DUB` (Dublino), `MAD` (Madrid), `NYO` (Stoccolma), `STN` (Londra), `SXF` (Berlino), `TRF` (Oslo).

Possiamo rappresentare le rotte usando una forma tabellare come quella riportata nella Figura 7.2, in cui la casella all'incrocio tra la riga `x` e la colonna `y` contiene il tempo di volo (in minuti) per la rotta che collega gli aeroporti `x` e `y`. La casella è vuota se non esiste una rotta aerea.

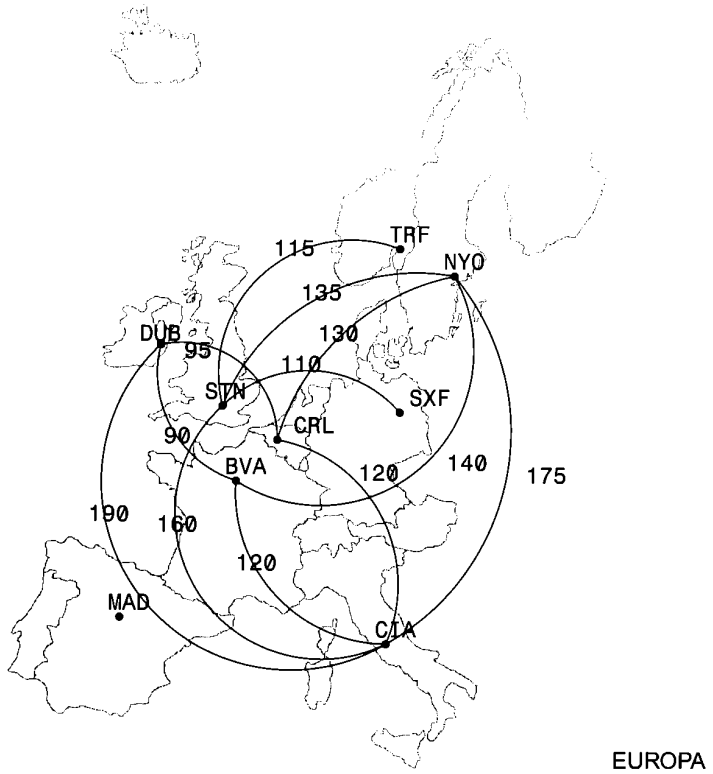
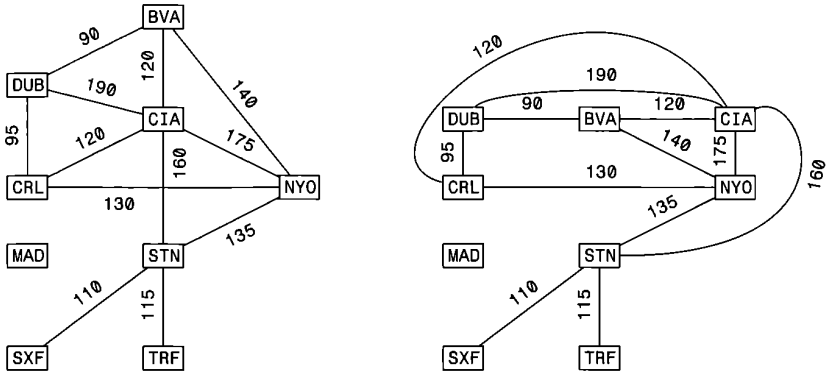


Figura 7.1 Rotte aeree di collegamento tra alcune capitali europee.

Tale rappresentazione è mostrata graficamente mediante uno dei due grafi in alto nella Figura 7.2, dove ciascun arco $(x, y) \in E$ rappresenta la rotta tra `x` e `y` etichettata con il tempo di volo corrispondente (osserviamo che una lista lineare o un albero non riescono a modellare l'insieme delle rotte).

Un grafo `G = (V, E)` è detto **pesato** o **etichettato** sugli archi se è definita una funzione $W : E \mapsto \mathbb{R}$ che assegna un valore (reale) a ogni arco del grafo. Nell'esempio della Figura 7.2, i pesi sono dati dai tempi di volo. Nel seguito, con il termine **grafo pesato** `G = (V, E, W)` indicheremo un grafo pesato sugli archi.

L'esempio mostrato nelle Figure 7.1 e 7.2 illustra una serie di nozioni sulla percorribilità e raggiungibilità dei nodi di un grafo. Dato un arco `(u, v)`, diremo che i nodi `u` e `v` sono **adiacenti** e che l'arco `(u, v)` è **incidente** a ciascuno di essi: in altri termini, un arco `(u, v)` è incidente al nodo `x` se e solo se `x = u` oppure `x = v`. Il numero di archi incidenti a un nodo è detto **grado** del nodo e un nodo di grado `0` è detto **isolato**. Facendo riferimento al grafo nell'esempio, il nodo `CIA`, ha grado pari a `5` mentre il nodo `MAD` è isolato.



	SXF	CRL	DUB	STN	MAD	TRF	BVA	CIA	NYO
SXF	–	–	–	110	–	–	–	–	–
CRL	–	–	95	–	–	–	–	120	130
DUB	–	95	–	–	–	–	90	190	–
STN	110	–	–	–	–	115	–	160	135
MAD	–	–	–	–	–	–	–	–	–
TRF	–	–	–	115	–	–	–	–	–
BVA	–	–	90	–	–	–	–	120	140
CIA	–	120	190	160	–	–	120	–	175
NYO	–	130	–	135	–	–	140	175	–

Figura 7.2 Rappresentazione a grafo (con n = 9 nodi e m = 12 archi) e tabellare delle rotte mostrate nella Figura 7.1.

Una proprietà che viene spesso utilizzata è che la somma dei gradi dei nodi è pari a 2m (il doppio del numero di archi). Per mostrare ciò, dobbiamo vedere ciascun arco come incidente a due nodi, per cui la presenza di un arco fa aumentare di 1 il grado di entrambi i suoi due estremi: il contributo di ogni arco alla somma dei gradi è pari a 2. Invece di sommare i gradi di tutti i nodi, possiamo calcolare tale valore moltiplicando per 2 il numero m di archi. Nell'esempio, m = 12 e la somma dei gradi è 24.

È naturale chiederci se, a partire da un nodo, è possibile raggiungere altri nodi attraversando gli archi: relativamente al nostro esempio, vogliamo sapere se è possibile andare da una città a un'altra prendendo uno o più voli. Tale percorso viene modellato nei grafi attraverso un **cammino** da un nodo u a un nodo z, definito come una sequenza di nodi $x_0, x_1, x_2, \ldots, x_k$ tale che $x_0 = u$, $x_k = z$ e $(x_i, x_{i+1}) \in E$ per ogni $0 \leq i < k$: l'intero $k \geq 0$ è detto **lunghezza** del cammino. Un **ciclo** è un cammino per cui vale $x_0 = x_k$, ovvero un cammino che ritorna nel nodo di partenza. Un cammino (o un ciclo) è **semplice** se non attraversa alcun nodo più di una volta, tranne eventualmente il caso in cui il primo e l'ultimo nodo siano uguali, ossia se non esiste alcun ciclo annidato al suo interno.

Nella Figura 7.2 esiste un cammino semplice di lunghezza k = 3 da u = BVA a z = SXF, dato da BVA, NYO, STN, SXF. Il cammino BVA, CIA, DUB, CRL, CIA, NYO, STN, SXF non è semplice a causa del ciclo CIA, DUB, CRL, CIA. Invece, STN, TRF, SXF non è un cammino in quanto TRF e SXF non sono collegati da un arco.

Un grafo viene detto **ciclico** se contiene almeno un ciclo, mentre è **aciclico** se non contiene cicli.

Un **cammino minimo** da u a z è caratterizzato dall'avere lunghezza minima tra tutti i possibili cammini da u a z: in altre parole, vogliamo sapere qual è il modo di andare dalla città u alla città z usando il minor numero di voli.

Nell'esempio, sia BVA, CIA, STN, SXF che BVA, NYO, STN, SXF sono cammini minimi. La **distanza** tra due nodi u e z è pari alla lunghezza di un cammino minimo che li congiunge e, se tale cammino *non* esiste, è pari a $+\infty$: la distanza tra BVA e SXF è 3, mentre tra BVA e MAD è $+\infty$.

Nel caso di grafi pesati ha senso definire il **peso di un cammino** come la somma dei pesi degli archi attraversati (che sono quindi intesi come "lunghezze" degli archi), ovvero come $\sum_{i=0}^{k-1} W(x_i, x_{i+1})$: nel nostro esempio, ipotizzando che il tempo di commutazione tra un volo e il successivo sia nullo, vogliamo sapere qual è il modo più veloce per andare da una città a un'altra (a differenza del cammino minimo).

Il **cammino minimo pesato** è il cammino di peso minimo tra due nodi e la **distanza pesata** è il suo peso (oppure $+\infty$ se non esiste alcun cammino tra i due nodi). Nel nostro esempio, il cammino minimo pesato è BVA, NYO, STN, SXF (e quindi la distanza pesata è 385) perché il cammino BVA, CIA, STN, SXF ha peso pari a 390 (non è detto che un cammino minimo pesato debba essere anche un cammino minimo).

I cammini permettono di stabilire se i nodi del grafo sono raggiungibili: due nodi u e z sono detti **connessi** se esiste un cammino tra di essi. Nell'esempio i nodi BVA e SXF sono connessi, in quanto esiste il cammino BVA, NYO, STN, SXF che li congiunge, mentre i nodi BVA e MAD non lo sono. Un grafo in cui ogni coppia di nodi è connessa è detto a sua volta **connesso**.

Dato un grafo G = (V, E), un **sottografo** di G è un grafo G′ = (V′, E′) composto da un sottoinsieme dei nodi e degli archi presenti in G: ossia, $V' \subseteq V$ ed $E' \subseteq V' \times V'$ e, inoltre, vale $E' \subseteq E$. Nella Figura 7.3 è mostrato a sinistra un sottografo del grafo presentato nella Figura 7.2. Se vale la condizione aggiuntiva che in E′ appaiono *tutti* gli archi di E che connettono nodi di V′, allora G′ viene denominato **sottografo indotto** da V′. È sufficiente specificare solo V′ in tal caso poiché $E' = E \cap (V' \times V')$: il grafo mostrato a destra nella Figura 7.3 è il sottografo indotto dall'insieme di nodi V′ = {BVA, CIA, DUB, NYO, MAD, TRF}.

Possiamo quindi definire una **componente connessa** di un grafo G come un sottografo G′ connesso e **massimale** di G, vale a dire un sottografo di G avente tutti nodi connessi tra loro e che non può essere esteso, in quanto non esistono

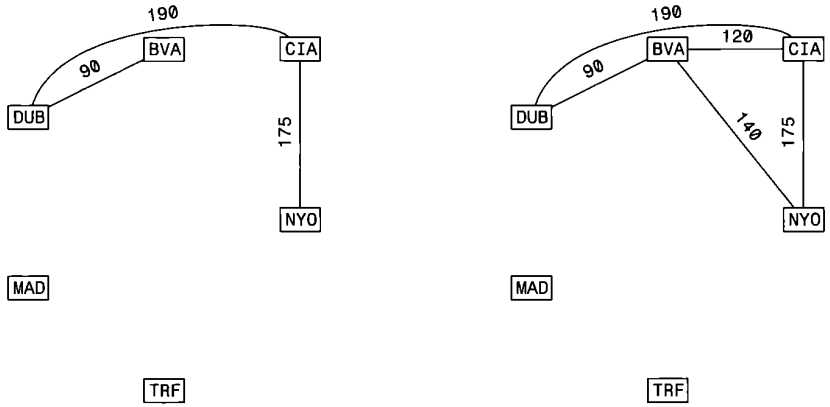


Figura 7.3 Un sottografo e un sottografo indotto del grafo nella Figura 7.2.

ulteriori nodi in G che siano connessi ai nodi di G′. All'interno di una componente connessa possiamo raggiungere qualunque nodo della componente stessa, mentre non possiamo passare da una componente all'altra percorrendo gli archi del grafo: la richiesta di massimalità nelle componenti connesse è motivata dall'esigenza di determinare con precisione tutti i nodi raggiungibili. Facendo riferimento al grafo nella Figura 7.2, il sottografo indotto dai nodi STN, SXF, TRF è connesso, ma non è una componente connessa del grafo, in quanto può essere esteso, ad esempio, aggiungendo il nodo CIA. In effetti, il grafo in questione risulta composto da due componenti connesse: la prima indotta dal nodo isolato MAD e la seconda indotta dai restanti nodi. Come possiamo osservare, un grafo connesso è composto da una sola componente connessa.[1]

Un **grafo completo** o **cricca** (*clique*) è caratterizzato dall'avere tutti i suoi nodi a due a due *adiacenti*. Facendo riferimento al grafo nella Figura 7.2, il sottografo indotto dai nodi CIA, CRL, DUB, è una cricca (anche se non è una componente connessa in quanto esistono altri nodi, come BVA, che sono collegati alla cricca). La notazione K_r è usata per indicare una cricca di r vertici e, nel nostro grafo, compaiono diversi sottografi che sono K_3 (ma nessun K_4 vi appare).

I grafi discussi finora sono detti **non orientati** in quanto un arco (u, v) non è distinguibile da un arco (v, u), per cui (u, v) = (v, u): entrambi rappresentano simmetricamente un collegamento tra u e v. In diverse situazioni, tale simmetria è volutamente evitata, come nel grafo illustrato nella Figura 7.4, che rappresenta la viabilità stradale di alcuni punti nella città di Pisa, con i sensi unici indicati da singoli archi orientati e le strade a doppio senso di circolazione indicate da coppie

[1] È interessante notare che un albero può essere equivalentemente visto come un grafo che è connesso e non contiene cicli, in cui un nodo viene designato come radice.

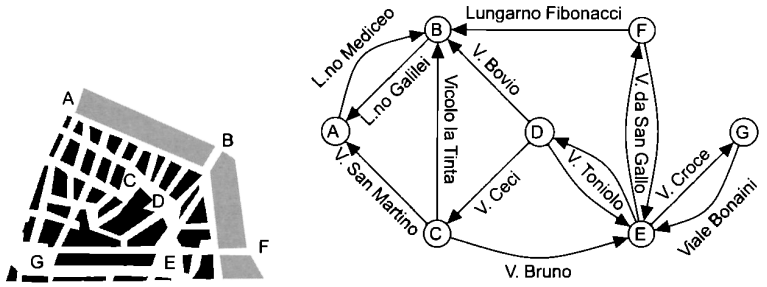


Figura 7.4 Parte della rete stradale della città di Pisa, dove le strade a doppio senso di circolazione sono rappresentate mediante una coppia di archi aventi etichetta comune.

di archi. Gli archi hanno un senso di percorrenza e la notazione (u, v) indica che l'arco va percorso dal nodo u verso il nodo v, e quindi (u, v) ≠ (v, u), in quanto il grafo è **orientato** o **diretto**.[2]

L'arco (u, v) viene detto **diretto** da u a v, quindi **uscente** da u ed **entrante** in v. Il nodo u è denominato nodo **iniziale** o **di partenza** mentre v è denominato nodo **finale**, **di arrivo** o **di destinazione**. Il **grado in uscita** di un nodo è pari al numero di archi uscenti da esso, mentre il **grado in ingresso** è dato dal numero di archi entranti. Facendo riferimento al grafo nella Figura 7.4, il nodo B ha grado in ingresso pari a 4 e grado in uscita pari a 1, mentre il nodo C ha grado in ingresso pari a 1 e grado in uscita pari a 3. Il **grado** è la somma del grado d'ingresso e di quello d'uscita: in un grafo orientato la somma dei gradi dei nodi in uscita è uguale a m, poiché ciascun arco fornisce un contributo pari a 1 nella somma dei gradi in uscita. Inoltre, il numero di archi è $0 \leq m \leq 2 \times \binom{n}{2}$ poiché otteniamo il massimo numero di archi quando ci sono due archi diretti per ciascuna coppia di nodi. Nel seguito, sarà sempre chiaro dal contesto se il grafo sarà orientato o meno.

Le definizioni viste finora per i grafi non orientati si adattano ai grafi orientati. Un **cammino** (**orientato**) da un nodo u a un nodo z soddisfa la condizione che tutti gli archi percorsi nella sequenza di nodi sono orientati da u a z. In questo caso, diciamo che il nodo u è connesso al nodo z (e questo non implica che z sia connesso a u perché la direzione è opposta). Allo stesso modo, possiamo definire un **ciclo orientato** come un cammino orientato da un nodo verso se stesso. Usando i pesi degli archi, la definizione di cammino minimo (pesato o non) e la nozione

[2] Nella teoria dei grafi, un arco che collega due nodi u e v di un grafo non orientato viene rappresentato come un insieme di due nodi {u, v}, spesso abbreviato come uv, mentre se il grafo è orientato l'arco viene rappresentato con la coppia (u, v) (infatti, {u, v} = {v, u} mentre, se u ≠ v, (u, v) ≠ (v, u)). Con un piccolo abuso di notazione, nel libro useremo (u, v) anche per gli archi non orientati, e in tal caso varrà (u, v) = (v, u), in quanto sarà sempre chiaro dal contesto se il grafo è orientato o meno.

di distanza rimangono inalterate. La stessa cosa avviene per la definizione di sottografo. È importante evidenziare il concetto di **grafo fortemente connesso**, quando ogni coppia di nodi è connessa, e di **componente fortemente connessa**: quest'ultima va intesa come un sottografo massimale tale che, per ogni coppia di nodi u e z, in esso esistono due cammini orientati all'interno del sottografo, uno da u a z e l'altro da z a u. Nel grafo nella Figura 7.4, le due componenti fortemente connesse risultano dai sottografi indotti rispettivamente dai nodi A, B e da C, D, E, F, G. Nel presente e nei seguenti capitoli, discuteremo alcuni algoritmi che usano le nozioni introdotte finora, talvolta ipotizzando che i nodi siano numerati, cioè che $V = \{0, 1, ..., n-1\}$ oppure $V = \{v_0, v_1, ..., v_{n-1}\}$.

7.1.1 Alcuni problemi su grafi

La versatilità dei grafi nel modellare molte situazioni, e i relativi problemi computazionali che ne derivano, sono ben illustrati da un esempio "giocattolo" in cui vogliamo organizzare una gita in montagna per n persone.

Per il viaggio, le poltrone nel pullman sono disposte a coppie e si vogliono assegnare le poltrone ai partecipanti in modo tale che due persone siano assegnate a una coppia di poltrone soltanto se si conoscono già (supponiamo che n sia un numero pari).

Una tale situazione può essere modellata mediante un grafo $G = (V, E)$ delle "conoscenze", in cui V corrisponde all'insieme degli n partecipanti ed E contiene l'arco (x, y) se e solo se le persone x e y si conoscono: nella Figura 7.5 è fornito un esempio di grafo di tale tipo.

In questo modello, un assegnamento dei posti che soddisfi le condizioni richieste corrisponde a un sottoinsieme di archi $E' \subseteq E$ tale che tutti i nodi in V siano incidenti agli archi di E' (quindi tutti i partecipanti abbiano un compagno di viaggio) e ogni nodo in V compaia soltanto in un arco di E' (quindi ciascun partecipante abbia esattamente un compagno): tale sottoinsieme viene denominato **abbinamento** o **accoppiamento perfetto** (*perfect matching*) dei nodi di G.

Nel caso del grafo mostrato nella Figura 7.5 esistono due abbinamenti diversi: il primo è $\{(v_0, v_1), (v_2, v_4), (v_3, v_5)\}$, mentre il secondo è $\{(v_0, v_4), (v_1, v_2), (v_3, v_5)\}$.

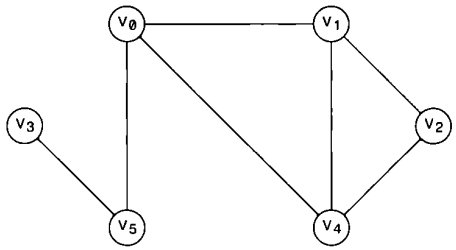


Figura 7.5 Esempio di grafo delle conoscenze.

Nel caso in cui si voglia assegnare una coppia di poltrone vicine a componenti di sessi diversi il problema può essere modellato come un abbinamento su un **grafo bipartito** $G = (V_0, V_1, E)$, caratterizzato dal fatto di avere due insiemi di vertici V_0 e V_1 (nel nostro caso viaggiatori uomini e viaggiatori donne) tali che ogni arco $(x, y) \in E$ ha gli estremi in insiemi diversi, quindi $x \in V_0$, $y \in V_1$ oppure $x \in V_1$, $y \in V_0$.

Tornando alla gita, supponiamo che sia prevista un'escursione in quota per cui i partecipanti devono procedere in fila indiana lungo vari tratti del percorso. Ancora una volta, i partecipanti preferiscono che ognuno conosca sia chi lo precede che chi lo segue.

In tal caso, si cerca un **cammino hamiltoniano** (dal nome del matematico del XIX secolo William Rowan Hamilton) ovvero un cammino che passi attraverso tutti i nodi una e una sola volta: si tratta quindi di trovare una permutazione $\langle \pi_0, \pi_1, \ldots, \pi_{n-1} \rangle$ dei nodi che sia un cammino, ovvero $(\pi_i, \pi_{i+1}) \in E$ per ogni $0 \le i \le n-2$. Nel grafo considerato esistono quattro cammini hamiltoniani diversi, dati dalle sequenze $\langle v_3, v_5, v_0, v_1, v_2, v_4 \rangle$, $\langle v_3, v_5, v_0, v_1, v_4, v_2 \rangle$, $\langle v_3, v_5, v_0, v_4, v_2, v_1 \rangle$ e $\langle v_3, v_5, v_0, v_4, v_1, v_2 \rangle$.[3]

Consideriamo quindi il caso in cui, giunti al ristorante del rifugio montano, vogliamo disporre i partecipanti intorno a un tavolo in modo tale che ognuno conosca i suoi vicini, di destra e di sinistra. Quel che vogliamo, ora, è un cammino hamiltoniano nel grafo delle conoscenze in cui valga l'ulteriore condizione $(\pi_{n-1}, \pi_0) \in E$: tale ordinamento prende il nome di **ciclo hamiltoniano**. A differenza del caso precedente, possiamo notare come una permutazione di tale tipo non esista per il grafo considerato nell'esempio.

Infine, tornati a valle, i partecipanti visitano un parco naturale ricco di torrenti che formano una serie di isole collegate da ponti di legno. I partecipanti vogliono sapere se è possibile effettuare un giro del parco attraversando tutti i ponti una e una sola volta, tornando al punto di partenza della gita. Il problema non è nuovo e, infatti, il principale matematico del XVIII secolo, Leonhard Euler, lo studiò relativamente ai ponti della città di Königsberg mostrati nella Figura 7.6, per cui l'origine della teoria dei grafi viene fatta risalire a Euler. Le zone delimitate dai fiumi sono i vertici di un grafo e gli archi sono i ponti da attraversare: nel caso che più ponti colleghino due stesse zone, ne risulta un **multigrafo** ovvero un grafo in cui la stessa coppia di vertici è collegata da archi multipli, come nel caso della Figura 7.6. In tal caso, sostituiamo ciascun arco multiplo (x, y) da una coppia di archi (x, w) e (w, z), dove w è un nuovo vertice usato soltanto per (x, y). In termini moderni, ne risulta un grafo G in cui vogliamo trovare un **ciclo euleriano**, ovvero un ciclo (non necessariamente semplice) che attraversa *tutti gli archi* una e una sola volta (mentre un ciclo hamiltoniano attraversa *tutti i nodi* una e una sola volta). È possibile attraversare tutti i ponti come richiesto se e solo se G ammette un

[3] In realtà, i cammini sarebbero otto, considerando quelli risultanti da un percorso "al contrario" dei quattro elencati.

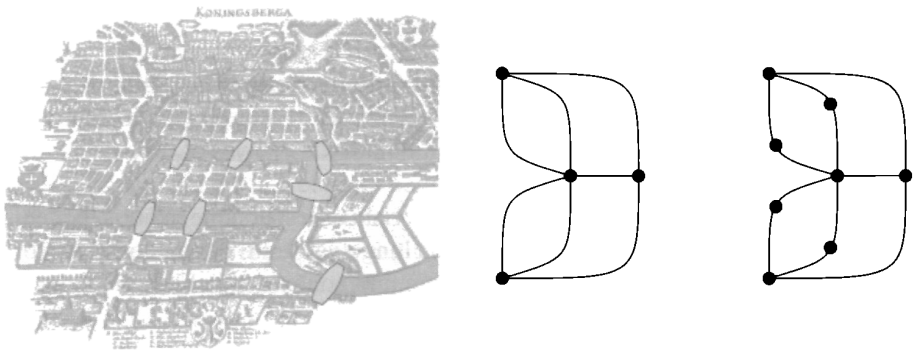


Figura 7.6 La città di Königsberg (immagine proveniente da www.wikipedia.org) con evidenziati i ponti rappresentati da Euler come archi di un multigrafo che viene trasformato in un grafo non orientato.

ciclo euleriano: Euler dimostrò che la condizione necessaria e sufficiente perché ciò avvenga è che G sia connesso e i suoi nodi abbiano tutti grado pari, pertanto il grafo nella parte destra della Figura 7.6 non contiene un ciclo euleriano, in quanto presenta 4 nodi di grado dispari, mentre è facile verificare che K_5 ammette un ciclo euleriano in quanto tutti i suoi nodi hanno grado 4. Euler dimostrò anche che se esattamente due nodi hanno grado dispari allora il grafo contiene un cammino euleriano, che comprende quindi ogni arco una e una sola volta: il grafo nella Figura 7.6 non contiene neanche un cammino euleriano. Come vedremo, la verifica che G sia connesso richiede tempo lineare, quindi il problema di Euler richiede O(n + m) tempo e spazio.

I problemi esaminati sinora derivano dalla modellazione di numerosi problemi reali e sono stati studiati nell'ambito della teoria degli algoritmi. È interessante a tale proposito osservare che, pur avendo tali problemi una descrizione molto semplice, l'efficienza della loro soluzione è molto diversa: mentre per il problema dell'abbinamento e del ciclo euleriano sono noti algoritmi operanti in tempo polinomiale nella dimensione del grafo, per i problemi del cammino e del ciclo hamiltoniano non sono noti algoritmi polinomiali. Approfondiremo questo aspetto nel Capitolo 8.

7.1.2 Rappresentazione di grafi

La rappresentazione utilizzata per un grafo è un aspetto rilevante per la gestione efficiente del grafo stesso, e viene realizzata secondo due modalità principali (di cui esistono varianti): le matrici di adiacenza e le liste di adiacenza.

Dato un grafo G = (V,E), la **matrice di adiacenza** A di G è un array bidimensionale di n × n elementi in {0, 1} tale che A[i][j] = 1 se e solo se (i, j) ∈ E per 0 ≤ i, j ≤ n - 1. In altre parole, A[i][j] = 1 se esiste un arco tra il nodo i e il nodo

	SXF	CRL	DUB	STN	MAD	TRF	BVA	CIA	NYO
SXF	0	0	0	1	0	0	0	0	0
CRL	0	0	1	0	0	0	0	1	1
DUB	0	1	0	0	0	0	1	1	0
STN	1	0	0	0	0	1	0	1	1
MAD	0	0	0	0	0	0	0	0	0
TRF	0	0	0	1	0	0	0	0	0
BVA	0	0	1	0	0	0	0	1	1
CIA	0	1	1	1	0	0	1	0	1
NYO	0	1	0	1	0	0	1	1	0

Figura 7.7 Matrice di adiacenza per il grafo nella Figura 7.2.

j, mentre A[i][j] = 0 altrimenti. Nella Figura 7.7 è fornita, a titolo di esempio, la matrice di adiacenza del grafo non orientato mostrato nella Figura 7.2 in cui il nodo i è associato alla riga i e alla colonna i, dove 0 ≤ i < n, così fornendo, in corrispondenza degli elementi con valore 1, l'elenco dei nodi adiacenti a tale nodo. Se consideriamo grafi non orientati, vale A[i][j]=A[j][i] per ogni 0≤i, j≤n-1, e quindi A è una matrice simmetrica. Volendo rappresentare un grafo pesato, possiamo associare alla matrice di adiacenza una matrice P dei pesi che rappresenta in forma tabellare la funzione W, come mostrato nella tabella in basso nella Figura 7.2 (talvolta le matrici A e P vengono combinate in un'unica matrice per occupare meno spazio).

La rappresentazione di un grafo mediante matrice di adiacenza consente di verificare in tempo O(1) l'esistenza di un arco tra due nodi ma, dato un nodo i, richiede tempo O(n) per scandire l'insieme dei nodi adiacenti, anche se tale insieme include un numero di nodi molto inferiore a n, come mostrato dalle seguenti istruzioni:

```
FOR (j = 0; j < n; j = j+1) {
  IF (A[i][j] != 0) {
    PRINT arco (i,j);
    PRINT peso P[i][j] dell'arco, se previsto;
  }
}
```

Per scandire efficientemente i vertici adiacenti, conviene utilizzare un array contenente n liste di adiacenza. In questa rappresentazione, a ogni nodo i del grafo è associata la lista dei nodi adiacenti, detta **lista di adiacenza** e indicata con listaAdiacenza[i], che implementiamo come un dizionario a lista (Paragrafo 4.2) di lunghezza pari al grado del nodo. Se il nodo ha grado zero, la lista è vuota. Altrimenti, listaAdiacenza[i] è una lista doppia con un riferimento sia all'elemento iniziale che a quello finale della lista di adiacenza per il nodo i: ogni elemento x di tale lista corrisponde a un arco (i, j) incidente a i e il corrispettivo campo x.dato contiene l'altro estremo j. Per esempio, il grafo mostrato nella

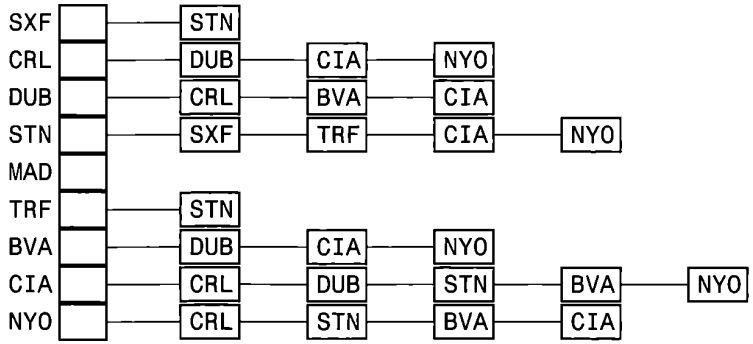


Figura 7.8 Lista di adiacenza per il grafo nella Figura 7.2, in cui SXF, CRL, DUB, STN, MAD, TRF, BVA, CIA, NYO sono implicitamente enumerati 0, 1, 2, ..., 8, in quest'ordine.

Figura 7.2 viene rappresentato mediante liste di adiacenza come illustrato nella Figura 7.8. Volendo rappresentare un grafo pesato, è sufficiente aggiungere un campo `x.peso` contenente il peso `W(i, j)` dell'arco `(i, j)`.

La scansione degli archi incidenti a un dato nodo `i` può essere effettuata mediante la scansione della corrispondente lista di adiacenza, in tempo pari al grado di `i`, come illustrato nelle istruzioni del seguente codice.

```
x = listaAdiacenza[i].inizio;
WHILE (x != null) {
  j = x.dato;
  PRINT (i,j);
  PRINT x.peso (se previsto);
  x = x.succ;
}
```

Tuttavia, la verifica della presenza di un arco tra una generica coppia di nodi `i` e `j` richiede la scansione della lista di adiacenza di `i` oppure di `j`, mentre tale verifica richiede tempo costante nelle matrici di adiacenza. Non esiste una rappresentazione preferibile all'altra, in quanto ciascuna delle due rappresentazioni presenta quindi vantaggi e svantaggi che vanno ponderati al momento della loro applicazione, valutandone la convenienza in termini di complessità in tempo e spazio che ne derivano.

Per quanto riguarda lo spazio utilizzato, la rappresentazione del grafo mediante matrice di adiacenza richiede spazio $\Theta(n^2)$, indipendentemente dal numero m di archi presenti: ciò risulta ottimo per grafi densi ma poco conveniente nel caso in cui trattiamo grafi sparsi in quanto hanno $m = O(n)$ oppure, in generale, per grafi in cui il numero di archi sia $m = o(n^2)$. Per tali grafi, la rappresentazione mediante matrice di adiacenza risulta poco efficiente dal punto di vista dello spazio utilizzato. Invece, lo spazio è pari a $O(n + m)$ celle di memoria nella rappresentazione mediante liste di adiacenza: infatti la lista per ciascun nodo è

di lunghezza pari al grado del nodo stesso e, come abbiamo visto, la somma dei gradi di tutti i nodi risulta essere $O(m)$; inoltre, usiamo spazio $O(n)$ per l'array dei riferimenti.

La rappresentazione con liste di adiacenza può essere vista anche come una rappresentazione compatta della matrice di adiacenza, in cui ogni lista corrisponde a una riga della matrice e include i soli indici delle colonne corrispondenti a valori pari a 1. Per avere un metro di paragone per la complessità in spazio, occorre confrontare lo spazio per memorizzare $O(n + m)$ interi e riferimenti, come richiesto dalle liste di adiacenza, con lo spazio per memorizzare un array bidimensionale di $O(n^2)$ bit, come richiesto dalla matrice di adiacenza.

Per grafi particolari, possiamo usare rappresentazioni più succinte. Appartengono a questo tipo di grafi sia gli alberi (che hanno $n - 1$ archi) che i **grafi planari**, che possono essere sempre disegnati sul piano senza intersezioni degli archi: Euler dimostrò infatti che un grafo planare di n vertici contiene $m = O(n)$ archi e quindi è sparso. Un esempio di grafo planare è mostrato nella Figura 7.2, dove sono riportate due disposizioni nel piano senza intersezioni degli archi (*embedding planare*). Mentre il grafo completo K_4 è planare, come mostrato dal suo embedding planare nella Figura 7.9, non lo sono K_5 e il grafo bipartito completo $K_{3,3}$ in quanto è possibile dimostrare che non hanno un embedding planare.

Da quanto detto deriva che un grafo planare non può contenere né K_5 né $K_{3,3}$ come sottografo: sorprendentemente, questi due grafi completi consentono di caratterizzare i grafi planari. Preso un grafo `G`, definiamo la sua **contrazione** `G'` come il grafo ottenuto collassando i vertici `w` di grado 2, per cui le coppie di archi `(u, w)` e `(w, v)` a essi incidenti diventano un unico arco `(u, v)`: nella Figura 7.6, il grafo `G` a destra ha il grafo `G'` al centro come contrazione. Il **teorema di Kuratowski-Pontryagin-Wagner** afferma che `G` non è planare se e solo se esiste un suo sottografo `G'` la cui contrazione fornisce K_5 oppure $K_{3,3}$. Tale proprietà può essere impiegata per *certificare* che un grafo non è planare, esibendo `G'` come prova che non è possibile trovare un embedding planare di `G`.

Tornando alla rappresentazione nel calcolatore dei grafi, consideriamo il caso di quella per grafi orientati: notiamo che essa non presenta differenze sostanziali rispetto alla rappresentazione discussa finora per i grafi non orientati. La matrice di adiacenza non è più simmetrica come nel caso di grafi non orientati e, inoltre, vengono solitamente rappresentati gli *archi uscenti* nelle liste di adiacenza. L'arco

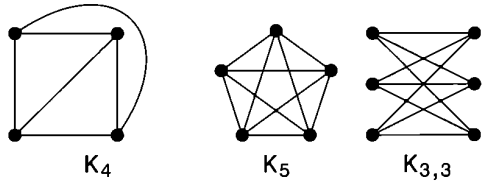
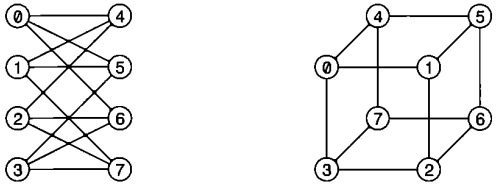


Figura 7.9 I grafi completi K_4 e K_5 e il grafo bipartito completo $K_{3,3}$.

orientato (i, j) viene memorizzato solo nella lista listaAdiacenza[i] (come elemento x contenente j nel campo x.dato) in quanto è differente dall'arco orientato (j, i) (che, se esiste, va memorizzato nella lista listaAdiacenza[j] come elemento contenente i). Osserviamo che nei grafi non orientati l'arco (i, j) è invece memorizzato sia in listaAdiacenza[i] che in listaAdiacenza[j]. Nel seguito, ciascuna lista di adiacenza è ordinata in ordine crescente di numerazione dei vertici in essa contenuti, se non specificato diversamente, e fornisce tutte le operazioni su liste doppie indicate nel Paragrafo 4.2.

Infine notiamo che, mentre la rappresentazione di un grafo lo identifica in modo univoco, non è vero il contrario. Infatti, esistono n! modi per enumerare i vertici con valori distinti in V = {0, 1, ..., n - 1} e, quindi, altrettanti modi per rappresentare lo stesso grafo. Per esempio, i due grafi di seguito sono apparentemente distinti:



La distinzione nasce dall'artificio di enumerare arbitrariamente gli *stessi vertici*, ma è chiaro che la relazione tra i vertici è la medesima se ignoriamo la numerazione (ebbene sì, il cubo a destra è un grafo bipartito). Tali grafi sono detti **isomorfi** in quanto una semplice rinumerazione dei vertici li rende uguali e, nel nostro esempio, i vertici del grafo a sinistra vanno rinumerati come

$$0 \to 4,\quad 1 \to 1,\quad 2 \to 6,\quad 3 \to 3,\quad 4 \to 5,\quad 5 \to 0,\quad 6 \to 7,\quad 7 \to 2,$$

per ottenere il grafo a destra: il problema di decidere in tempo polinomiale se due grafi *arbitrari* di n vertici sono isomorfi equivale a trovare tale rinumerazione, se esiste, in tempo polinomiale in n ed è uno dei problemi algoritmici fondamentali tuttora irrisolti, con molte implicazioni (per esempio, stabilire se due grafi arbitrari *non* sono isomorfi ha delle importanti implicazioni nella crittografia).

7.1.3 Cammini minimi, chiusura transitiva e prodotto di matrici

La rappresentazione di un grafo G = (V, E) mediante matrice di adiacenza fornisce un semplice metodo per il calcolo della chiusura transitiva del grafo stesso: un grafo G* = (V, E*) è la **chiusura transitiva** di G se, per ogni coppia di vertici i e j in V, vale (i, j) ∈ E* se e solo se esiste un *cammino* in G da i a j.

Sia A la matrice di adiacenza del grafo G, *modificata* in modo che gli elementi della diagonale principale hanno tutti valori pari a 1 (come mostrato nella Figura 7.10). Calcolando il prodotto booleano $A^2 = A \times A$ dove l'operazione di somma tra elementi è l'OR e la moltiplicazione è l'AND, possiamo notare che un elemento $A^2[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot A[k][j]$ di tale matrice è pari a 1 se e solo se esiste almeno un indice $0 \le t \le n-1$ tale che A[i][t] = A[t][j] = 1 (nella Figura 7.11 a sinistra è mostrata la matrice di adiacenza A^2 corrispondente a quella della Figura 7.10).

	0	1	2	3	4	5	6	7	8	9	10
0	1	1	0	0	1	1	0	0	0	0	0
1	1	1	1	1	1	0	0	1	0	0	0
2	0	1	1	0	1	0	0	0	0	0	0
3	0	1	0	1	0	1	1	1	0	0	0
4	1	1	1	0	1	0	0	0	0	0	0
5	1	0	0	1	0	1	1	1	0	0	0
6	0	0	0	1	0	1	1	1	0	0	0
7	0	1	0	1	0	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	1	1

Figura 7.10 Matrice di adiacenza modificata per il calcolo della chiusura transitiva.

Tenendo conto dell'interpretazione dei valori degli elementi della matrice A, ne deriva che, dati due nodi i e j, vale A^2[i][j] = 1 se e solo se esiste un nodo t, dove $0 \le t \le n-1$, adiacente sia a i che a j, e quindi se e solo se esiste un cammino di lunghezza al più 2 tra i e j. L'eventualità che il cammino abbia lunghezza inferiore a 2 deriva dal caso in cui t = i oppure t = j (motivando così la nostra scelta di porre a 1 tutti gli elementi della diagonale principale di A).

Moltiplicando la matrice A^2 per A otteniamo, in base alle considerazioni precedenti, la matrice A^3 tale che A^3[i][j] = 1 se e solo se i nodi i e j sono collegati da un cammino di lunghezza al più 3: nella Figura 7.11 a destra è mostrata la matrice di adiacenza A^3 corrispondente alla matrice di adiacenza della Figura 7.10.

Possiamo verificare che, moltiplicando A^3 per A, la matrice risultante è uguale a A^3: da ciò deriva che $A^i = A^3$ per ogni $i \ge 3$, e che quindi A^3 rappresenta la relazione di connessione tra i nodi per cammini di lunghezza qualunque. Indicheremo in generale tale matrice come A^*, osservando che essa rappresenta il grafo G* di chiusura transitiva del grafo G.

	0	1	2	3	4	5	6	7	8	9	10
0	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0
2	1	1	1	1	1	0	0	1	0	0	0
3	1	1	1	1	1	1	1	1	0	0	0
4	1	1	1	1	1	1	0	1	0	0	0
5	1	1	0	1	1	1	1	1	0	0	0
6	1	1	0	1	0	1	1	1	0	0	0
7	1	1	1	1	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	1	1

	0	1	2	3	4	5	6	7	8	9	10
0	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0
2	1	1	1	1	1	1	1	1	0	0	0
3	1	1	1	1	1	1	1	1	0	0	0
4	1	1	1	1	1	1	1	1	0	0	0
5	1	1	1	1	1	1	1	1	0	0	0
6	1	1	1	1	1	1	1	1	0	0	0
7	1	1	1	1	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	1	1

Figura 7.11 Matrici di adiacenza A^2 (a sinistra) e $A^3 = A^*$ (a destra).

Per la corrispondenza tra nodi adiacenti in G^* e nodi connessi in G, la matrice A^* consente di verificare in tempo costante la presenza di un cammino in G tra due nodi (non consente però di ottenere il cammino, se esiste), oltre che di ottenere in tempo $O(n)$, dato un nodo, l'insieme dei nodi nella stessa componente connessa in G.

Poniamoci ora il problema del calcolo efficiente di A^* a partire da A, esemplificato dal codice seguente.

```
1  A* = A;
2  DO {
3    B = A*;
4    A* = B × B;
5  } WHILE (A* != B);
6  RETURN A*;
```

Il codice, a partire da A, moltiplica la matrice per se stessa, ottenendo in questo modo la sequenza A^2, A^4, A^8, ..., fino a quando la matrice risultante non viene più modificata da tale moltiplicazione, ottenendo così A^*. Valutiamo il numero di passi eseguiti da tale codice: l'istruzione nella riga 1 viene eseguita una sola volta e ha costo $\Theta(n^2)$, richiedendo la copia degli n^2 elementi di A, mentre l'istruzione nella riga 3 e il controllo nella riga 5 sono eseguiti un numero di volte pari al numero di iterazioni del ciclo, richiedendo un costo $\Theta(n^2)$ a ogni iterazione. Per quanto riguarda l'istruzione nella riga 4, anch'essa viene eseguita a ogni iterazione e ha un costo pari a quello della moltiplicazione di una matrice $n \times n$ per se stessa, costo che indichiamo per ora con $C_M(n)$.

Per contare il numero massimo di iterazioni, consideriamo che la matrice A^i rappresenta come adiacenti elementi a distanza al più i e che il diametro (la massima distanza tra due nodi) di un grafo con n vertici è al più $n-1$. Dato che a ogni iterazione la potenza i della matrice A^i calcolata raddoppia, saranno necessarie al più $\log(n-1) = \Theta(\log n)$ iterazioni per ottenere la matrice A^*. Da ciò consegue che il costo computazionale del codice precedente sarà pari a $\Theta((n^2 + C_M(n)) \log n)$. Come discusso nel Paragrafo 3.6.1, abbiamo che $C_M(n) = \Omega(n^2)$ e che $C_M(n) = \Theta(n^3)$ con il metodo classico di moltiplicazione di matrici: quindi il costo totale di calcolo di A^* è $\Theta(n^3 \log n)$. Una riduzione del costo $C_M(n)$ può essere ottenuta utilizzando algoritmi più efficienti per la moltiplicazione di matrici, come ad esempio l'*algoritmo di Strassen* introdotto nel Paragrafo 3.6.1.

7.2 Opus libri: Web crawler e visite di grafi

Uno degli esempi più noti di grafo orientato $G = (V, E)$ è fornito dal World Wide Web, in cui l'insieme dei vertici in V è costituito dalle pagine web e l'insieme degli archi orientati in E sono i collegamenti tra le pagine. Ogni pagina web è identificata da un indirizzo URL (*Uniform Resource Locator*). Per esempio, l'URL della pagina web `//www.w3.org/Addressing/Addressing.html` descrive la specifica tecnica delle URL e la loro sintassi: la parte racchiusa tra `//` e il primo `/` successivo (`www.w3.org`) è un riferimento simbolico a un indirizzo di un calcolatore ospite (*host*) connesso a Internet. Tale indirizzo è una sequenza di 32 bit se viene utilizzato il protocollo IPv4 (oppure di 128 bit nel caso di IPv6). Infine, la parte rimanente dell'URL, ovvero `/Addressing/Addressing.html`, indica un percorso interno al *file system* dell'host, che identifica il file corrispondente alla pagina web. Un collegamento da una pagina a un'altra è specificato, all'interno della prima, utilizzando la seguente sintassi definita nel linguaggio HTML (*HyperText Markup Language*), che permette di associare, tra l'altro, un testo a ogni link con la seguente sintassi `<a href="http:URL">testo</a>`.

Queste informazioni sono utilizzate da specifici programmi dei motori di ricerca, detti *crawler* o *spider*, per attraversare il grafo del web in maniera sistematica ed efficiente raccogliendo informazioni sulle pagine visitate. Infatti, vista la dimensione del grafo del Web, è improponibile generare tutti gli indirizzi a 32 o 128 bit dei possibili host di siti web, per accedere alle loro pagine. I crawler effettuano invece la **visita** del grafo del Web partendo da un insieme S di pagine selezionate: in pratica, S viene formato con gli indirizzi disponibili in alcune collezioni, come *Open Directory Project*, che contengono un insieme di pagine web raccolte e classificate da editori "umani". Quando un crawler scopre una nuova pagina, la lista dei link in uscita da essa permette di estendere la visita a ulteriori pagine (in realtà la situazione è più complessa per la presenza di pagine dinamiche e di formati diversi e, inoltre, per altre problematiche come la ripartizione del carico di lavoro tra i vari crawler).

Possiamo quindi modellare il comportamento dei crawler come una visita di tutti i nodi e gli archi di un grafo raggiungibili da un nodo di partenza, ipotizzando che i vertici siano identificati con numeri compresi tra 0 e $n-1$ (quindi $V = \{0, 1, ..., n-1\}$) e il numero di archi sia indicato con $m = |E|$. Osserviamo che gli algoritmi di visita discussi nel seguito utilizzano le pile e le code discusse nel Capitolo 2 e funzionano sia per grafi orientati che per grafi non orientati.

7.2.1 Visita in ampiezza di un grafo

La caratteristica della visita in ampiezza o BFS (*Breadth-First Search*) è che essa esplora i nodi in ordine crescente di distanza da un nodo iniziale tenendo presente l'esigenza di evitare che la presenza di cicli possa portare a esaminare ripetutamente gli stessi cammini. Nell'esporre la visita in ampiezza su un grafo, faremo inizialmente l'ipotesi che l'insieme dei nodi sia conosciuto: successivamente considereremo il caso più generale in cui tale insieme non sia preventivamente noto.

Al fine di esaminare ogni arco un numero limitato di volte nel corso della visita, usiamo un array booleano di appoggio `raggiunto`, tale che `raggiunto[u]` vale `TRUE` se e solo se il nodo `u` è stato scoperto nel corso della visita effettuata fino a ora. Ogni volta che viene raggiunto un nuovo nodo, tutti i suoi vicini vengono

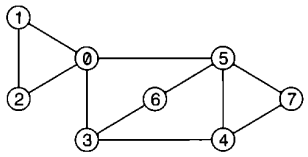
inseriti in una coda Q dalla quale viene prelevato il prossimo nodo da visitare. Come vedremo in seguito, l'utilizzo della coda ci garantirà l'ordine di visita in ampiezza. La lista di adiacenza del vertice corrente fornisce, come al solito, i riferimenti ai suoi vicini. Il Codice 7.1 riporta lo schema di visita a partire da un vertice prescelto s, dove `listaAdiacenza[u].inizio` indica il riferimento all'inizio della lista di adiacenza per il vertice u. Dopo aver inizializzato `raggiunto` e la coda Q (righe 2-4), inizia il ciclo di visita. Il vertice u in testa alla coda viene estratto (riga 6) e, se non è stato ancora raggiunto, viene marcato come tale e la sua lista di adiacenza viene scandita a partire dal primo elemento (righe 7-13). Poiché per ogni vertice u i suoi vicini vengono inseriti nella coda soltanto se u non è ancora marcato, ciascuna delle liste di adiacenza esaminate viene anch'essa considerata una sola volta: in conseguenza di ciò, il costo totale della visita è dato dalla somma delle lunghezze delle liste di adiacenza esaminate, ovvero al più dalla somma dei gradi di tutti i vertici del grafo, ottenendo un tempo totale $O(n + m)$ e $O(n + m)$ celle di memoria aggiuntive.

Codice 7.1 Visita in ampiezza di un grafo con n vertici a partire dal vertice s, utilizzando una coda Q inizialmente vuota e un array `raggiunto` per marcare i vertici visitati.

```
1  BreadthFirstSearch( s ):
2    FOR (u = 0 ; u < n; u = u+1)
3      raggiunto[u] = FALSE;
4    Q.Enqueue( s );
5    WHILE (!Q.Empty( )) {
6      u = Q.Dequeue( );
7      IF (!raggiunto[u]) {
8        raggiunto[u] = TRUE;
         FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
10         v = x.dato;
11         Q.Enqueue( v );
         }
       }
     }
```

ESEMPIO 7.1

Prendiamo come riferimento il grafo mostrato nella figura che segue.



L'ordine dei vertici in ciascuna lista di adiacenza determina l'ordine di visita dei vertici stessi nel grafo. Supponendo che i vertici in ciascuna lista di adiacenza siano mantenuti in ordine crescente e che il nodo di partenza sia s = 0, la coda Q e l'array `raggiunto` evolveranno nel modo indicato di seguito.

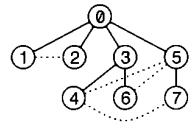
		0	1	2	3	4	5	6	7
Q = 0	raggiunto =	F	F	F	F	F	F	F	F
Q = 1,2,3,5	raggiunto =	T	F	F	F	F	F	F	F
Q = 2,3,5,0,2	raggiunto =	T	T	F	F	F	F	F	F
Q = 3,5,0,2,0,1	raggiunto =	T	T	T	F	F	F	F	F
Q = 5,0,2,0,1,0,4,6	raggiunto =	T	T	T	T	F	F	F	F
Q = 0,2,0,1,0,4,6,0,4,6,7	raggiunto =	T	T	T	T	F	T	F	F

A ogni passo viene estratto l'elemento in testa alla coda (quello più a sinistra) e, non essendo visitato, viene marcato e tutti i suoi vicini vengono messi in coda. I nodi che seguono in coda sono stati già marcati quindi vengono solo tolti dalla coda.

		0	1	2	3	4	5	6	7
Q = 4,6,0,4,6,7	raggiunto =	T	T	T	T	F	T	F	F
Q = 6,0,4,6,7,3,5,7	raggiunto =	T	T	T	T	T	T	F	F
Q = 0,4,6,7,3,5,7,3,5	raggiunto =	T	T	T	T	T	T	T	F
Q = 7,3,5,7,3,5	raggiunto =	T	T	T	T	T	T	T	F
Q = 3,5,7,3,5,4,5	raggiunto =	T	T	T	T	T	T	T	T
Q = null	raggiunto =	T	T	T	T	T	T	T	T

Con la coda vuota la procedura ha termine.

L'ordine con cui i vertici vengono raggiunti dalla visita del grafo, illustrato nella figura che segue, è dato da 0, 1, 2, 3, 5, 4, 6, 7 (il loro ordine di estrazione dalla coda e la conseguente marcatura mediante `raggiunto`).



In particolare, dal vertice 0 raggiungiamo i vertici 1, 2, 3 e 5, dal vertice 3 raggiungiamo 4 e 6 e dal vertice 5 raggiungiamo 7 (mentre i rimanenti vertici non permettono di raggiungerne altri).

Esercizio svolto 7.1 Modificare la visita BFS in modo che metta in coda soltanto i vicini non ancora visitati del vertice corrente. Valutare la dimensione massima della coda in tal caso.

Soluzione Prima di inserire un vertice in coda, verifichiamo attraverso un array `inCoda` che questo non sia già presente nella coda `Q`. L'invariante è che il vertice `u` estratto da `Q` sicuramente soddisfa la condizione che `raggiunto[u]` è falso. Per cui possiamo tranquillamente porlo a vero e mettere in coda i suoi vicini per cui `inCoda` è falso, ponendolo poi a vero, come illustrato nel codice riportato di seguito. La dimensione massima della coda diventa `n - 1` in quanto contiene soltanto vertici ancora da raggiungere.

```
1  BreadthFirstSearchSemplificata( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      raggiunto[u] = FALSE;
4      inCoda[u] = FALSE;
5    }
6    Q.Enqueue( s );
7    inCoda[s] = TRUE;
8    WHILE (!Q.Empty( )) {
9      u = Q.Dequeue( );
10     raggiunto[u] = TRUE;
11     FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
12       v = x.dato;
13       IF (!inCoda[v]) {
14         Q.Enqueue( v );
15         inCoda[v] = TRUE;
16       }
17     }
18   }
```

Elenchiamo alcune proprietà interessanti della visita. Gli archi che conducono a vertici ancora non visitati, permettendone la scoperta, formano un albero detto **albero BFS**, la cui struttura dipende dall'ordine di visita (si veda l'ultima figura dell'Esempio 7.1). Per poter costruire tale albero, modifichiamo lo schema di visita illustrato nel Codice 7.1: invece di usare una coda `Q` in cui sono inseriti i *vertici*, usiamo `Q` come coda in cui gli *archi* sono inseriti ed estratti una sola volta. Il Codice 7.2 riporta tale modifica della visita in ampiezza: dopo aver estratto l'arco (u′, u) dalla coda (riga 6), scandiamo la lista di adiacenza di u solo se quest'ultimo non è stato scoperto (riga 7). La visita richiede $O(n+m)$ tempo, in quanto ogni arco è inserito ed estratto una sola volta e le liste di adiacenza sono scandite solo quando i corrispondenti vertici sono visitati la prima volta.

Codice 7.2 Visita in ampiezza di un grafo in cui la coda `Q` contiene archi anziché vertici.

```
1  BreadthFirstSearch( s ):
2  FOR (u = 0; u < n; u = u + 1)
3    raggiunto[u] = FALSE;
4  Q.Enqueue( (null, s) );
5  WHILE (!Q.Empty( )) {
6    (u', u) = Q.Dequeue( );
7    IF (!raggiunto[u]) {
8      raggiunto[u] = TRUE;
9      FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
10       v = x.dato;
11       Q.Enqueue( (u, v) );
12     }
13   }
14 }
```

Utilizzando il Codice 7.2, non è difficile individuare gli archi dell'albero BFS: basta memorizzare l'arco (u′, u) quando il nodo u viene marcato come raggiunto nella riga 8 e, inoltre, u′ diventa il padre di u nell'albero BFS. In generale, gli archi individuati in tal modo formano un sottografo aciclico e, quando gli archi di tale sottografo sono incidenti a tutti i vertici, l'albero BFS ottenuto è un **albero di ricoprimento** (*spanning tree*) del grafo, vale a dire un albero i cui nodi coincidono con quelli del grafo. Cambiando l'ordine relativo dei vertici all'interno delle liste di adiacenza, possiamo ottenere alberi diversi. Notiamo infine che tali alberi, avendo grado variabile, possono essere rappresentati come alberi ordinali (Paragrafo 1.4.2).

ESEMPIO 7.2

La sequenza degli archi visitati dal Codice 7.2 sul grafo dell'Esempio 7.1 è (`null`, `0`), (`0`, 1), (`0`, 2), (`0`, 3), (`0`, 5), (3, 4), (3, 6) e (5, 7). Infatti, per esempio, i nodi 1, 2, 3 e 5 sono scoperti la prima volta dopo la visita del nodo `0` che sarà il loro padre. Analogamente il nodo 6 viene scoperto la prima volta dopo la visita del nodo 3 e quindi quest'ultimo sarà il suo padre.

L'albero BFS è utile per rappresentare i cammini minimi dal vertice di partenza s verso tutti gli altri vertici in un grafo non pesato: tale proprietà è vera in quanto gli archi *non* sono pesati (altrimenti non è detto che valga, e vedremo successivamente come gestire il caso in cui gli archi sono pesati). Per verificare tale proprietà, basta osservare che l'algoritmo visita prima i vertici a distanza 1 (ovvero i vertici adiacenti a s), poi quelli a distanza 2 e così via, come un semplice ragionamento per induzione può stabilire. In altre parole vale il seguente risultato.

Teorema 7.1 *La distanza minima di un vertice* v *da* s *nel grafo equivale alla profondità di* v *nell'albero BFS.*

Dimostrazione Per verificare quanto affermato sopra ragioniamo in modo induttivo rispetto alla distanza dei nodi da s: il caso base dell'induzione è banalmente verificato in quanto l'unico nodo a profondità 0 è s che evidentemente ha distanza 0 da se stesso.
Per mostrare il passo induttivo, supponiamo che per ogni nodo u a distanza $p' < p$ da s la profondità di u sia pari a p' e ipotizziamo che esista, per assurdo, un nodo v a distanza δ da s la cui profondità nell'albero BFS sia $p \neq \delta$, e quindi tale che il cammino minimo da s a v sia di lunghezza δ. Consideriamo allora sia il vertice v' (a distanza $\delta - 1$ da s) che precede v in tale cammino, che il padre u di v a profondità $p - 1$ nell'albero BFS. Evidentemente, non può essere $p < \delta$ in quanto, in tal caso, il cammino da s a v che attraversa u avrebbe lunghezza minore di δ, il che contraddice l'ipotesi che δ sia la distanza tra s e v. Al tempo stesso, se $\delta < p$ l'algoritmo di visita avrebbe raggiunto v da v' e quindi v avrebbe profondità δ (infatti v' sarebbe stato visitato prima di u). Deve quindi essere $p = \delta$. ❑

La proprietà appena discussa ha due conseguenze rilevanti.

1. Gli archi del grafo che non sono nell'albero BFS sono chiamati **all'indietro** (*back*): possono collegare solo due vertici alla stessa profondità nell'albero BFS oppure a profondità consecutive p e $p+1$.
2. Il *diametro* del grafo può essere calcolato come la massima tra le altezze degli alberi BFS radicati nei diversi vertici del grafo (in generale, ne possono esistere n diversi). Il tempo richiesto per calcolare il diametro è $O(n(n+m))$.

Osserviamo che i nodi irraggiungibili da s sono a distanza infinita e non vengono inclusi nell'albero BFS.

Un'altra applicazione interessante è che la visita BFS ci permette di stabilire se un grafo non orientato G è connesso. Infatti, G è connesso se e solo se `raggiunto[u]` vale TRUE per ogni $0 \leq u \leq n-1$, ovvero tutti i vertici sono stati raggiunti e quindi abbiamo che l'albero BFS è un albero di ricoprimento. Il costo computazionale è lo stesso della visita BFS, quindi $O(n+m)$ tempo e $O(n+m)$ celle di memoria.

Nel caso in cui l'insieme dei nodi del grafo non sia preventivamente noto, come succede nell'esplorazione della struttura a grafo del Web, non è possibile utilizzare un array per distinguere i nodi raggiunti da quelli non ancora trovati. Per ottenere tale funzionalità è necessario fare uso di una struttura di dati che consenta di rappresentare un insieme, nello specifico l'insieme dei vertici raggiunti, dando la possibilità di aggiungere nuovi elementi all'insieme e di verificare l'appartenenza di un elemento all'insieme stesso. Inoltre, l'array `listaAdiacenza[u]` viene sostituito da una funzione `listaAdiacenza(u)` che estrae dal contenuto di u le connessioni ai suoi vertici adiacenti (per esempio, u è una pagina web che al suo interno contiene delle URL ai vicini).

Tali funzionalità sono offerte da un dizionario (Capitolo 4), che quindi impieghiamo nel Codice 7.3, che è una semplice riscrittura del Codice 7.1. Nel codice in questione, il dizionario D, inizialmente vuoto, viene utilizzato in sostituzione dell'array `raggiunto` per rappresentare, a ogni istante, l'insieme dei nodi già raggiunti dalla visita.

Codice 7.3 Esplorazione mediante visita in ampiezza di un grafo, utilizzando una coda Q e un dizionario D per memorizzare i vertici visitati.

```
1  BreadthFirstSearchExplore( s ):
2    Q.Enqueue( s );
3    WHILE (!Q.Empty( )) {
4      u = Q.Dequeue( );
5      IF (!D.Appartiene(u)) {
6        D.Inserisci(u);
7        FOR (x = listaAdiacenza(u).inizio; x != null; x = x.succ) {
8          v = x.dato;
9          Q.Enqueue( v );
10       }
11     }
12   }
```

Dal punto di vista del costo computazionale, la sostituzione dell'array con un dizionario fa sì che tale costo dipenda dal costo delle operazioni definite sul dizionario, dipendente a sua volta dall'implementazione adottata per tale struttura di dati.

In particolare, esaminando il Codice 7.3 risulta che l'operazione `Inserisci` è eseguita $O(n)$ volte, mentre l'operazione `Appartiene` è invocata $O(m)$ volte: per esempio, utilizzando una tabella hash, per la quale le due operazioni richiedono tempo medio $O(1)$, il costo complessivo della visita è $O(n+m)$ nel caso medio. Utilizzando invece un albero di ricerca bilanciato, i costi delle due operazioni sono $O(\log n)$ nel caso peggiore, e quindi il costo conseguente dell'algoritmo è $O((n+m)\log n)$.

7.2.2 Visita in profondità di un grafo

Se nello schema di visita illustrato nei Codici 7.1 e 7.2 sostituiamo la coda Q con una pila P, otteniamo un altro algoritmo di visita di grafi, noto come algoritmo di visita in profondità, o DFS (*Depth-First Search*), realizzato dai Codici 7.4 e 7.5. Dato che in una pila un insieme di elementi viene estratto in ordine opposto a quello di inserimento, volendo estrarre dalla pila gli archi incidenti a un nodo u nello stesso ordine con cui li incontriamo scandendo la lista di adiacenza di u, dobbiamo inserire nella pila tali archi in ordine inverso rispetto a quello della lista. Analogamente alla visita in ampiezza, anche nella visita in profondità viene costruito un albero, detto **albero DFS**, i cui archi vengono individuati in corrispondenza alla scoperta di nuovi vertici.

Codice 7.4 Visita in profondità di un grafo utilizzando una pila P di archi, inizialmente vuota. Ciascuna lista di adiacenza viene scandita all'indietro.

```
1  DepthFirstSearch( s ):
   FOR (u = 0; u < n; u = u + 1)
3    raggiunto[u] = FALSE;
4  P.Push( s );
5  WHILE (!P.Empty( )) {
6    u = P.Pop( );
7    IF (!raggiunto[u]) {
8      raggiunto[u] = TRUE;
9      FOR (x = listaAdiacenza[u].fine; x != null; x = x.pred) {
10       v = x.dato;
11       P.Push( v );
12     }
13   }
14 }
```

Codice 7.5 Visita in profondità di un grafo in cui la pila P contiene archi anziché vertici.

```
1  DepthFirstSearch( s ):
2  FOR (u = 0; u < n; u = u + 1)
3    raggiunto[u] = false;
4  P.Push( (null, s) );
5  WHILE (!P.Empty( )) {
6    (u', u) = P.Pop( );
7    IF (!raggiunto[u]) {
8      raggiunto[u] = TRUE;
9      FOR (x = listaAdiacenza[u].fine; x != null; x = x.pred) {
10       v = x.dato;
11       P.Push( (u, v) );
12     }
13   }
14 }
```

La visita in profondità, per la natura stessa della politica LIFO che adotta la pila, si presta in modo naturale a un'implementazione ricorsiva, riportata nel Codice 7.6. Tale implementazione è ampiamente usata in varie applicazioni discusse in seguito, in alternativa a quella iterativa; notate che in questo caso il fatto che l'utilizzo della ricorsione non richieda una gestione esplicita di una pila fa sì che non sia più necessario effettuare una scansione al contrario delle liste di adiacenza.

Codice 7.6 Visita in profondità di un grafo utilizzando la ricorsione.

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

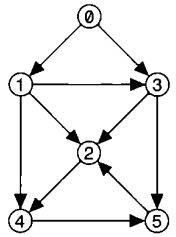
```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
```

Unitamente alla visita ricorsiva, il codice mostra la funzione `Scansione`, che esamina tutti i vertici del grafo alla ricerca di quelli non ancora scoperti, invocando la ricorsione su ciascun nodo s di questo tipo, utilizzandolo come vertice di partenza di una nuova visita. Osserviamo che l'esame di tutti i vertici del grafo in effetti non richiede necessariamente l'adozione di una visita in profondità per ogni nodo non ancora raggiunto: in linea di principio, anzi, potremmo utilizzare tipi di visita diversi.

Il costo computazionale delle visite in profondità discusse sopra è analogo a quello della visita in ampiezza, ovvero $O(n + m)$ tempo sia per grafi orientati che per grafi non orientati. Il numero di celle di memoria richieste per la visita iterativa è $O(m)$ mentre per quella ricorsiva è $O(n)$.

ESEMPIO 7.3

Riportiamo un esempio di visita del grafo orientato mostrato nella figura, a partire dal vertice s = 0.



Ecco come evolvono la pila P e l'array `raggiunto` utilizzando il Codice 7.5.

P = (null, 0)

	0	1	2	3	4	5
raggiunto =	F	F	F	F	F	F

Viene estratto dalla pila l'arco in testa (null, 0), il nodo 0 viene marcato e tutti gli archi uscenti da 0 vengono aggiunti in pila in ordine inverso rispetto a quello della lista di adiacenza di 0.

		0	1	2	3	4	5
P = (0, 1), (0, 3)	raggiunto =	T	F	F	F	F	F
P = (1, 2), (1, 3), (1, 4), (0, 3)	raggiunto =	T	T	F	F	F	F
P = (2, 4), (1, 3), (1, 4), (0, 3)	raggiunto =	T	T	T	F	F	F
P = (4, 5), (1, 3), (1, 4), (0, 3)	raggiunto =	T	T	T	F	T	F
P = (5, 2), (1, 3), (1, 4), (0, 3)	raggiunto =	T	T	T	F	T	T

Poiché il nodo 2 dell'arco in testa alla pila è già stato visitato, l'arco viene rimosso dalla pila senza dare seguito ad altro.

		0	1	2	3	4	5
P = (1, 3), (1, 4), (0, 3)	raggiunto =	T	T	T	T	T	T
P = null	raggiunto =	T	T	T	T	T	T

Ora che la pila è vuota l'algoritmo ha termine.

Nella Figura 7.12 sono mostrati sia l'albero BFS che albero DFS per il grafo orientato dell'Esempio 7.3. Una caratteristica importante della visita descritta nel codice di `DepthFirstSearchRicorsiva` è che mantiene implicitamente, e in ordine inverso, il cammino π nell'albero DFS dal nodo di partenza s al vertice u attualmente considerato nella visita: i vertici lungo π sono quelli in cui la visita ricorsiva è iniziata ma non ancora terminata. In altre parole, eseguendo tale codice, otteniamo implicitamente la visita anticipata dell'albero DFS. Per esempio, quando la chiamata di `DepthFirstSearchRicorsiva` esamina il vertice u = 3 nella Figura 7.12, i vertici corrispondenti al cammino π nell'albero DFS sono 3, 1, 0.

Per quanto riguarda invece gli archi non appartenenti all'albero DFS, questi possono essere classificati ulteriormente nel caso di grafi orientati. In particolare, un arco (u, v) non appartenente all'albero DFS, può essere catalogato come segue:

- **all'indietro** (*back*): se v è antenato di u nell'albero DFS;
- **in avanti** (*forward*): se v è discendente di u nell'albero DFS (nipote, pronipote e così via, ma non figlio perché altrimenti l'arco apparterrebbe all'albero);
- **trasversale** (*cross*): se v e u non sono uno antenato dell'altro.

Nei grafi non orientati possono esserci solo archi in avanti o all'indietro, che sono gli unici a condurre a vertici già visitati durante la visita in profondità.

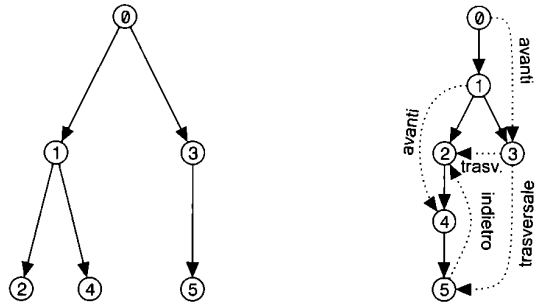


Figura 7.12 Per il grafo orientato dell'Esempio 7.3 viene mostrato il relativo albero BFS a partire dal vertice s, dove s = 0, e il relativo albero DFS a partire da s con la classificazione degli archi in avanti, all'indietro e trasversali.

Dall'esempio nella Figura 7.12 emerge anche la differenza tra le due visite. Nella visita in ampiezza, i vertici sono esaminati in ordine crescente di distanza dal nodo di partenza s, per cui la visita risulta adatta in problemi che richiedono la conoscenza della distanza e dei cammini minimi (non pesati): successivamente, vedremo come, in effetti, un limitato adattamento della visita in ampiezza consenta di individuare i cammini minimi anche in grafi con pesi sugli archi.

Nella visita in profondità, l'algoritmo raggiunge rapidamente vertici lontani dal vertice di partenza s, e quindi la visita è adatta per problemi collegati alla percorribilità, alla connessione e alla ciclicità dei cammini.

Anche per la visita in profondità, l'esplorazione di un grafo di cui non sono preventivamente noti i vertici richiede la sostituzione dell'array `raggiunto` con un dizionario: valgono rispetto a ciò le considerazioni effettuate per la visita in ampiezza nel Paragrafo 7.2.1. Nel caso specifico del grafo del Web, possiamo sostituire la coda o la pila con una coda che estrae gli elementi in base a un loro valore di rilevanza (il *rank* delle pagine web), garantendo in questo modo la priorità di caricamento, durante la visita, alle pagine classificate come più interessanti.

7.3 Applicazioni delle visite di grafi

7.3.1 Grafi diretti aciclici e ordinamento topologico

La visita in profondità trova applicazione, tra l'altro, nell'identificazione dei cicli in un grafo: vale infatti la seguente proprietà.

Teorema 7.2 *Un grafo* G *è ciclico se e solo se contiene almeno un arco all'indietro (definito per le visite BFS e DFS).*

Dimostrazione Esaminiamo prima il caso di un grafo non diretto. Consideriamo un grafo con un arco all'indietro (u, v) che, ricordiamo, non appartiene all'albero di ricoprimento (BFS o DFS) di G: esiste un cammino da v a u nell'albero in quanto, per definizione di arco all'indietro, v è antenato di u e da ciò deriva che,

estendendo questo cammino con (u, v), ritorniamo in v, ottenendo quindi un ciclo. Viceversa, consideriamo un grafo contenente un ciclo: la visita costruisce un albero DFS che non può contenere tutti gli archi del ciclo, in quanto un albero è aciclico, e quindi almeno un arco è all'indietro.

Nel caso di grafi orientati, se un ciclo contiene un arco in avanti (u, v) allora possiamo sostituire tale arco con il cammino da u a v nell'albero e ottenere comunque un ciclo (ricordiamo che, nel caso di grafi orientati, gli archi nell'albero sono diretti dai padri ai figli). Quindi, se il grafo è ciclico, esiste necessariamente un ciclo che non include archi in avanti. Infine, un arco trasversale (u, v) non può appartenere a un ciclo. Se così fosse, nell'albero il nodo u sarebbe antenato di v o viceversa, pervenendo a una contraddizione. ❑

Un grafo orientato aciclico è chiamato **DAG** (*Directed Acyclic Graph*) e viene utilizzato in quei contesti in cui esiste una dipendenza tra oggetti espressa da una relazione d'ordine: per esempio gli esami propedeutici in un corso di studi, la relazione di ereditarietà tra classi nella programmazione a oggetti, la relazione tra ingressi e uscite delle porte in un circuito logico, oppure l'ordine di valutazione delle formule in un foglio elettronico. In generale, supponiamo di avere decomposto un'attività complessa in un insieme di attività elementari e di avere individuato le relativa dipendenze. Un esempio concreto potrebbe essere la costruzione di un'automobile: volendo eseguire sequenzialmente le attività elementari (per esempio, in una catena di montaggio), bisogna soddisfare il vincolo che, se l'attività B dipende dall'attività A, allora A va eseguita prima di B.

Usando i DAG per modellare tale situazione, poniamo i vertici in corrispondenza biunivoca con le attività elementari, e introduciamo un arco (A, B) per indicare che l'attività A va eseguita prima dell'attività B. Un'esecuzione in sequenza delle attività che soddisfi i vincoli di precedenza tra esse, corrisponde a un ordinamento topologico del DAG costruito su tali attività, come illustrato nella Figura 7.13.

Dato un DAG G = (V, E), un **ordinamento topologico** di G è una numerazione $\eta : V \mapsto \{0, 1, ..., n-1\}$ dei suoi vertici tale che per ogni arco (u, v) ∈ E vale $\eta(u) < \eta(v)$. In altre parole, se disponiamo i vertici lungo una linea orizzontale in base alla loro numerazione η, in ordine crescente, otteniamo che gli archi risultano tutti orientati da sinistra verso destra. L'ordinamento topologico può anche essere visto come un ordinamento totale compatibile con l'ordinamento parziale rappresentato dal DAG. Osserviamo che i grafi ciclici non hanno un ordinamento topologico.

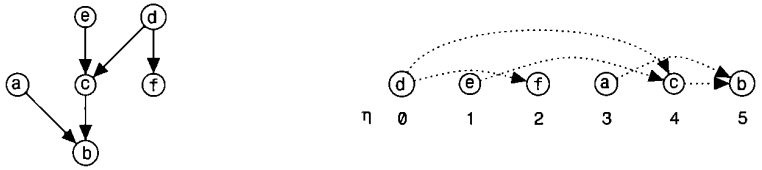


Figura 7.13 Un esempio di DAG e di un suo ordinamento topologico.

Concettualmente, l'ordinamento topologico di un DAG G può essere trovato come segue: prendiamo un vertice z avente grado di uscita nullo, ovvero tale che la sua lista di adiacenza è vuota (tale vertice deve necessariamente esistere, altrimenti G sarebbe ciclico). Assegniamo il valore η(z) = n - 1 a tale vertice, rimuovendolo quindi da G insieme a tutti i suoi archi entranti, ottenendo così un grafo residuo G′ che sarà ancora un DAG. Identifichiamo ora un vertice z′ di grado di uscita nullo in G′, a cui assegniamo η(z′) = n - 2, rimuovendolo come descritto sopra.

Iterando questo procedimento, otteniamo alla fine un grafo residuo con un solo vertice s, a cui assegniamo numerazione η(s) = 0. Ogni arco (u, v) è evidentemente orientato da sinistra a destra nell'ordinamento indotto da η, semplicemente perché v viene rimosso prima di u per cui η(v) > η(u). Da tale procedimento deduciamo, inoltre, che non è detto che esista un unico ordinamento topologico per un dato DAG, in quanto, in generale, in un dato istante possono esistere più nodi aventi grado di uscita nullo, e quindi più possibilità (tutte corrette) di scelta del nodo da rimuovere.

L'algoritmo per trovare un ordinamento topologico, in realtà, può essere realizzato in modo semplice, come mostrato nel Codice 7.7. Esso si basa sulla visita ricorsiva in profondità discussa precedentemente nel Codice 7.6: tale visita viene estesa realizzando la funzione η(u) mediante un array eta[u] e un contatore globale, inizializzato a n - 1 (riga 4 dell'ordinamento topologico). Dopo che le visite lungo gli archi uscenti da u sono terminate, il contatore viene assegnato a eta[u] e decrementato di uno (righe 7-8 della visita ricorsiva). Per garantire di assegnare una numerazione a tutti i vertici, scandiamo l'insieme dei vertici stessi, invocando la visita ricorsiva su quelli non ancora raggiunti dalle visite precedenti.

Codice 7.7 Ordinamento topologico di un grafo orientato aciclico. Per ogni vertice, l'array eta indica l'ordine inverso di terminazione della visita in profondità ricorsiva.

```
1  OrdinamentoTopologico( ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    contatore = n - 1;
5    FOR (s = 0; s < n; s = s + 1) {
6      IF (!raggiunto[s]) DepthFirstSearchRicorsivaOrdina( s );
7    }
```

```
1  DepthFirstSearchRicorsivaOrdina( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsivaOrdina( v );
6    }
7    eta[u] = contatore;
8    contatore = contatore - 1;
```

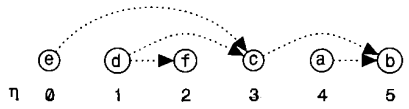
Teorema 7.3 *L'algoritmo* `OrdinamentoTopologico` *descritto nel Codice 7.7 è corretto.*

Dimostrazione Fissato un nodo `u` dobbiamo dimostrare che `eta[u] < eta[v]` per tutti gli archi `(u, v)`. Osserviamo che `eta[u]` è definito solo dopo che tutti i suoi vicini sono stati raggiunti e a questi è stato assegnato un valore nell'ordinamento. Infatti l'assegnamento `eta[u] = contatore` (riga 7) avviene dopo aver accertato che tutti i vicini di `u` siano stati visitati e quindi dopo l'invocazione di `DepthFirstSearchRicorsivaOrdina` su di essi (riga 5). Infine ogni volta che `contatore` viene assegnato questo viene decrementato (riga 8). Pertanto il valore di `contatore` assegnato a `eta[u]` è inferiore a quello assegnato ai suoi vicini. ❑

Il costo computazionale rimane $O(n + m)$ tempo e lo spazio occupato è $O(n)$ celle di memoria, richieste dalla pila implicitamente gestita dalla ricorsione.

ESEMPIO 7.4

Consideriamo il grafo della Figura 7.13 e supponiamo che il nodo a sia identificato dall'intero `0`, b da 1 e così via. La procedura parte dal nodo a e procede sul nodo b; quest'ultimo non ha vicini, quindi `eta[b] = 5` e `contatore = 4`. Segue l'esecuzione delle righe 7 e 8 di `DepthFirstSearchRicorsivaOrdina` su a, ovvero `eta[a] = 4` e `contatore = 3`. Il controllo torna alla funzione `OrdinamentoTopologico` che esegue `DepthFirstSearchRicorsivaOrdina` sul nodo c in quanto il nodo b risulta raggiunto. Tutti i vicini di c sono stati già raggiunti, quindi `eta[c] = 3` e `contatore = 2`. L'esecuzione di `DepthFirstSearchRicorsivaOrdina` su d raggiunge f, pertanto `eta[f] = 2`, `eta[d] = 1` e `contatore = 0`. Infine con la chiamata di `DepthFirstSearchRicorsivaOrdina` su e, `eta[e] = 0`. L'ordinamento ottenuto è mostrato nella figura.



Si osservi come l'ordinamento topologico prodotto differisca sensibilmente da quello mostrato nella Figura 7.13, entrambi validi per il DAG.

7.3.2 Componenti (fortemente) connesse

Le visite di grafi sono utili anche per individuare le componenti connesse di un grafo non orientato. Riconsiderando la scansione effettuata nel Codice 7.6 da questo punto di vista, notiamo che tutti i vertici sono connessi se e solo se, al termine della visita, tutti i vertici risultano raggiunti. Ne deriva che, in tal caso, tutti i valori dell'array `raggiunto` sono `TRUE` dopo la terminazione di `Scansione`. Se, al contrario, qualche vertice non risulta raggiunto, esso verrà esaminato successivamente nel corso della visita ricorsiva invocata su un qualche nodo $s \neq 0$: ogni

qualvolta abbiamo che `raggiunto[s]` vale `FALSE`, viene individuata una nuova componente connessa, che include il nodo `s`. L'individuazione delle componenti connesse in un grafo non orientato richiede pertanto $O(n + m)$ tempo e $O(n)$ celle di memoria.

Per individuare le componenti fortemente connesse in un grafo orientato $G = (V, E)$ applichiamo a `G` una visita in profondità ottenendo, come vedremo, una partizione dell'insieme dei vertici `V` in sottoinsiemi $V_0, V_1, \ldots, V_{s-1}$ massimali e disgiunti, e tale che ciascun sottoinsieme V_i soddisfa la proprietà che due qualunque vertici `u`, $v \in V_i$ sono collegati da un cammino orientato sia da `u` a `v` che da `v` a `u` (mentre questa proprietà non vale se $u \in V_i$ e $v \in V_j$ per $i \neq j$).

Un semplice esempio di grafo composto da una singola componente fortemente connessa è dato da un ciclo orientato di vertici, oppure da un grafo contenente un ciclo Euleriano (che, ricordiamo, attraversa tutti gli archi una e una sola volta).

Un esempio di grafo composto da più componenti è invece mostrato nella Figura 7.14, dove le componenti (massimali) sono racchiuse in cerchi a scopo illustrativo. Le componenti possono essere anche viste come *macro-vertici*, collegati da archi multipli. Un'importante proprietà è che, non considerando la molteplicità di tali archi, i macro-vertici formano un DAG: se così non fosse, infatti, un ciclo di macro-vertici formerebbe una componente fortemente connessa più grande, in quanto due vertici arbitrari all'interno di due macro-vertici distinti sarebbero comunque collegati da cammini in entrambe le direzioni, ma questo non è possibile per la massimalità delle componenti. Il DAG ottenuto in questo modo a partire dall'esempio mostrato nella Figura 7.14 è rappresentato nella Figura 7.13.

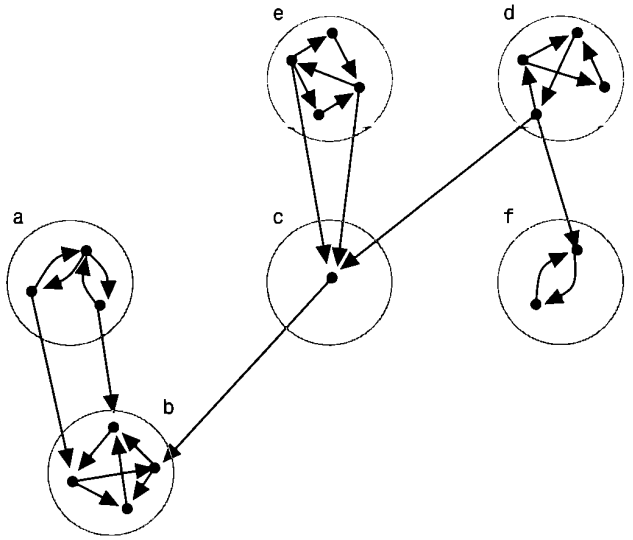


Figura 7.14 Un esempio di grafo orientato con le sue componenti fortemente connesse.

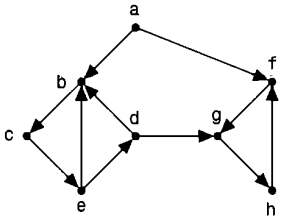
La strutturazione di un grafo orientato in un DAG di macro-vertici (corrispondenti a componenti fortemente connesse) è fondamentale per la comprensione dell'algoritmo che stiamo per discutere. In particolare ci permette di formulare una descrizione ad alto livello dell'algoritmo nel seguente modo, dove concettualmente contraiamo i vertici del grafo man mano che li scopriamo appartenere alla stessa componente fortemente connessa. Sia P il cammino parziale prodotto da una visita DFS a partire da un vertice stabilito e sia u l'ultimo vertice di P e (u, v) il prossimo arco analizzato dalla visita.

- Se v è in P sia P_v la parte del cammino P che va da v a u. Contraiamo il ciclo composto da P_v e dall'arco (u, v) in un macro-vertice il cui rappresentante è v e continuiamo la visita col prossimo arco uscente da u.
- Se v non è in P, aggiungiamo v a P e proseguiamo la visita analizzando gli archi uscenti da v.

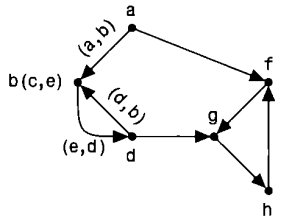
Nel caso in cui u non abbia archi uscenti eliminiamo u (e tutti gli eventuali vertici che fanno parte del macro-vertice che u rappresenta) dal grafo, diamo in output la componente fortemente connessa costituita dai nodi del macro-vertice rappresentato da u e proseguiamo l'algoritmo con la visita del prossimo nodo seguendo l'ordine della visita in profondità. Come vedremo in seguito, un cammino π nel grafo di partenza verrà implicitamente mappato nel cammino P nel grafo parzialmente contratto, secondo quanto descritto sopra. Alla fine della computazione, il grafo contratto corrisponderà al DAG menzionato sopra, avendo individuato tutte le componenti fortemente connesse.

ESEMPIO 7.5

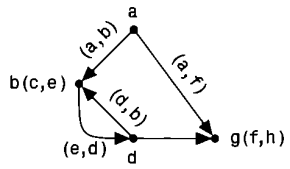
Applichiamo l'algoritmo descritto al grafo che segue iniziando la visita dal nodo a.



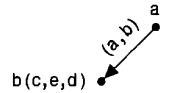
Supponiamo che vengano visitati, nell'ordine, i nodi a, b, c ed e e che il prossimo arco a essere preso in considerazione sia (e, b). Questo chiude il ciclo b, c, e, b che viene contratto in un macro-vertice rappresentato proprio da b.



Nella figura, in corrispondenza del nodo b, sono indicati tra parentesi anche i nodi del macro-vertice che esso rappresenta. Inoltre in corrispondenza degli archi che escono o entrano nel macro-vertice sono indicati gli estremi originali. Il cammino attuale P è dato dalla sequenza dai vertici a e b. Questo viene esteso con gli archi (e, d), (d, g), (g, h) e (h, f). Quando viene analizzato (f, g) si riscontra un ciclo che viene contratto in g.



Il vertice g non ha archi uscenti quindi diamo in output la prima componente fortemente connessa che è costituita dai vertici {g, f, h}, eliminiamo g e proseguiamo la visita da d. Consideriamo l'arco (d, b) che chiude il ciclo col vertice b. Dopo la contrazione in b abbiamo questa situazione.



Dal nodo b non possiamo più proseguire quindi diamo in output la seconda componente fortemente connessa {b, c, d, e}. Dopo aver rimosso anche il vertice b si prosegue da a che non ha più archi uscenti, quindi la terza componente fortemente connessa è composta dal solo nodo a. Il grafo restante dall'eliminazione di a è vuoto e l'algoritmo termina.

Per renderci conto della correttezza dell'approccio delineato sopra, consideriamo il primo nodo u eliminato dall'algoritmo e vediamo come i nodi che esso rappresenta, indicati con l'insieme C_u, costituiscano una componente fortemente connessa. L'insieme C_u è ottenuto per contrazione di cicli, pertanto ogni due nodi al suo interno risultano connessi. Dimostriamo che C_u è massimale. Se non lo fosse dovrebbe esistere un cammino P' che parte da un nodo di C_u, passa per un nodo v non in C_u e termina in un nodo in C_u. Al momento in cui tutti i nodi di C_u sono stati contratti in u il cammino P' è diventato un cammino che forma un ciclo con u. Questo ciclo a sua volta verrebbe contratto e tutti i vertici che contiene, tra cui v, verrebbero assegnati alla stessa componente fortemente connessa. La correttezza dell'approccio segue osservando che le componenti fortemente connesse del grafo originale $G = (V, E)$ sono date da C_u più le componenti fortemente connesse del sottografo di G indotto da $V - C_u$.

Nel resto del paragrafo formalizziamo l'approccio appena descritto, utilizzando una visita DFS in tempo lineare $O(n+m)$. A tal proposito ricordiamo che i vertici lungo il cammino π di una visita DFS nel grafo dato sono quelli in cui la visita è iniziata ma non ancora terminata. I rimanenti vertici ricadono in due tipologie, ovvero quelli la cui visita è stata già completata (quindi la chiamata ricorsiva per loro è terminata) oppure quelli che non sono stati ancora raggiunti.

I vertici sui quali la visita è stata già completata corrispondono a quelli che sono stati contratti in un unico nodo che, non avendo altri archi uscenti, è già stato eliminato dal grafo. Quindi per distinguere gli archi che formano un ciclo e che inducono una contrazione dagli archi trasversali utilizziamo l'array `completo` tale che, se `v` è un nodo del grafo, `completo[v] = TRUE` se e solo se `v` e tutti i suoi vicini sono stati visitati: in tal caso il nodo `v` viene chiamato *completo*.

Come abbiamo visto l'algoritmo identifica dei rappresentanti durante la visita. Il rappresentante della componente fortemente connessa è quello che viene eliminato dal grafo perché non ha più archi uscenti. Osserviamo che tale nodo è anche il primo a essere visitato rispetto gli altri nodi della componente fortemente connessa alla quale appartiene ed è l'ultimo a diventare completo. Quando il rappresentante diventa completo perde il suo *status* di rappresentante, di conseguenza, i rappresentanti sono nel cammino π. Per esempio, si consideri il momento in cui il cammino `P` dell'Esempio 7.5 raggiunge `g`: in π abbiamo i nodi `a`, `b`, `d` e `g` e tra questi ci sono i rappresentanti delle tre componenti fortemente connesse a cui essi appartengono (`a`, `b` e `g`).

Quest'ultima osservazione ci suggerisce un'ulteriore classificazione dei vertici del grafo rispetto all'esecuzione dell'algoritmo: un vertice `u` è *parziale* se è in π e quindi fa parte di una componente fortemente connessa non ancora esplorata completamente. Per esempio, al momento della visita del nodo `g` nell'Esempio 7.5 i nodi `a`, `b` e `d` sono parziali. Inoltre una componente fortemente connessa è detta parziale se contiene alcuni nodi parziali. Per completare diciamo che un vertice è *ignoto* se non è stato ancora raggiunto dall'algoritmo, mentre una componente è ignota se contiene solo vertici ignoti.

Durante la visita occorre conoscere i nodi rappresentanti e quelli che essi rappresentano (parziali). Quando viene scoperto un nuovo nodo esso è un rappresentante e rappresenta se stesso. Quando viene scoperto un ciclo tutti i vertici o macro-vertici del ciclo sono rappresentati dal primo vertice o macro-vertice del ciclo incontrato nella visita. I due insiemi vengono gestiti da due pile:

- `parziali`: contiene tutti i vertici parziali inseriti in ordine di visita;
- `rappresentanti`: contiene i vertici rappresentanti inseriti in ordine di visita.

Numeriamo tutti i vertici nell'ordine di scoperta da parte della visita usando un array `dfsNumero` a tal fine: nel momento in cui viene preso in considerazione un arco `(u, v)` che chiude un ciclo (in particolare vale `dfsNumero[v] < dfsNumero[u]`), notiamo che `v` diventa un macro-vertice rappresentante di tutti i nodi del ciclo. Questi compaiono in ordine crescente di numerazione lungo il cammino π. Nella pila `rappresentanti` sopra `v` compariranno i macro-vertici che compongono il ciclo che da qui devono essere eliminati in quanto ora tutti rappresentati da `v`.

La prima componente fortemente connessa è completa se e solo se tutti i cicli di macro-vertici che la compongono sono stati contratti in un unico macro-vertice rappresentato da un nodo `u` che non ha altri archi uscenti. Questo è vero se e solo se al termine della visita dei nodi adiacenti a `u` questo si trova a essere in cima alla pila `rappresentanti`. Poiché ogni nuovo nodo scoperto è stato inserito in cima alla pila `parziali`, in testa a questa fino al vertice `u` troviamo tutti i nodi che fanno parte della componente fortemente connessa di `u` che possono essere dati in output, eliminati da `parziali` e marcati completi.

Il Codice 7.8 segue l'idea appena descritta. Inizialmente, nessun vertice è ancora esaminato (e quindi neanche `completo`) e le pile sono vuote. Inoltre, usiamo l'array `dfsNumero` sia per numerare i vertici in ordine di scoperta (attraverso `contatore`) sia per stabilire se un vertice è stato visitato o meno: inizializzando gli elementi di tale array al valore `-1`, il successivo assegnamento di un valore maggiore oppure uguale a `0` (a seguito della visita) ci permette di stabilire se il corrispondente vertice sia stato raggiunto o meno. Osserviamo che, a differenza della visita in profondità, non occorre utilizzare esplicitamente l'array `raggiunto`.

Codice 7.8 Stampa delle componenti fortemente connesse in un grafo orientato.

```
1  ComponentiFortementeConnesse( ):
2    FOR (s = 0; s < n; s = s + 1) {
3      dfsNumero[s] = -1;
4      completo[s] = FALSE;
5    }
6    contatore = 0;
7    FOR (s = 0; s < n; s = s + 1) {
8      IF (dfsNumero[s] == -1) DepthFirstSearchRicorsivaEstesa( s );
9    }
```

```
1  DepthFirstSearchRicorsivaEstesa( u ):
2    dfsNumero[u] = contatore;
3    contatore = contatore + 1;
4    parziali.Push( u );
5    rappresentanti.Push( u );
6    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
7      v = x.dato;
8      IF (dfsNumero[v] == -1) {
9        DepthFirstSearchRicorsivaEstesa( v );
10     } ELSE IF (!completo[v]) {
11       WHILE (dfsNumero[rappresentanti.Top()] > dfsNumero[v])
12         rappresentanti.Pop();
13     }
14   }
15   IF (u == rappresentanti.Top()) {
16     PRINT 'Nuova componente fortemente connessa:'
```

```
17      DO {
18        PRINT z = parziali.Pop();
19        completo[z] = TRUE;
20      } WHILE (z != u);
21      rappresentanti.Pop();
22    }
```

Le righe 2-3 assegnano la numerazione di visita a u, e le successive righe 4-5 inseriscono u in cima a entrambe le pile (in quanto potrebbe iniziare una nuova componente parziale che prima era ignota). A questo punto, esaminiamo la lista dei vertici adiacenti di u e invochiamo ricorsivamente la visita su quei vertici non ancora raggiunti (righe 8-9). Al contrario delle visite discusse in precedenza, se un vertice v adiacente a u è stato raggiunto precedentemente (righe 10-12), occorre verificare se l'arco (u, v) contribuisce alla creazione di un ciclo. Questo è possibile solo se v non è completo (riga 10): altrimenti (u, v) è un arco del DAG di macro-vertici di cui abbiamo discusso prima e non può creare un ciclo.

Ipotizzando quindi che v non sia completo, siamo nella situazione mostrata nell'Esempio 7.5 al momento in cui viene analizzato l'arco (d, b), dove l'arco (d, b) chiude un ciclo di componenti parziali, che devono essere unite in una singola componente parziale. Poiché i rispettivi vertici parziali occupano posizioni contigue nella pila `parziali`, è sufficiente rimuovere i soli rappresentanti di tali componenti dalla pila `rappresentanti`: ne sopravvive solo uno, ovvero quello con numerazione di visita minore, che diventa il rappresentante della nuova componente parziale così creata implicitamente (righe 10-12, dove la numerazione di v è usata per eliminare i rappresentanti che non sopravvivono).

Terminata la scansione della lista di adiacenza di u, e le relative chiamate ricorsive, abbiamo che u diventa completo quando il suo antenato più vecchio raggiungibile conclude la visita. Se u è anche rappresentante della propria componente (riga 15), vuol dire che u e tutti i vertici parziali che si trovano sopra di esso in `parziali` formano una nuova componente completa. È sufficiente, quindi, estrarre ciascuno di tali vertici dalla pila `parziali`, marcarlo come `completo` (righe 18-19) ed eliminare u dalla pila `rappresentanti` (riga 21): notiamo che il corpo del ciclo alle righe 17-20 viene eseguito prima della guardia che, se *non* verificata, fa uscire dal ciclo.

Teorema 7.4 *Il Codice 7.8 calcola correttamente le componenti fortemente connesse del grafo in tempo* $O(n + m)$.

Dimostrazione Facciamo vedere che l'algoritmo descritto dal Codice 7.8 implementa l'algoritmo per il calcolo delle componenti fortemente descritto precedentemente.

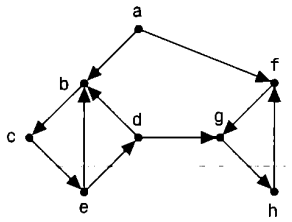
Mettiamoci nel caso in cui nessun vertice è completo. Se (u, v) raggiunge un nodo già visitato allora l'arco chiude un ciclo di macro-vertici. I macro-vertici che lo compongono sono nel cammino π da v a u e quindi in `rappresentanti` e, poiché raggiunti dopo v, hanno `dfsNumero` maggiore di quello di v. Questi possono essere tolti dalla pila `rappresentanti` lasciando in testa il nodo v che diventa rappresentante del ciclo appena trattato.

Sia u il primo nodo a essere stato visitato completamente e ad essere presente in `rappresentanti.Top()`. Il nodo u rappresenta un ciclo composto da macro-vertici, ovvero u rappresenta una componente fortemente connessa in quanto non ci sono altri archi da esplorare. I nodi che fanno parte di questa componente (l'insieme C_u) sono tutti i nodi in `parziali` che si trovano sopra u. Questo perché in `parziali` sono stati aggiunti tutti i nodi incontrati da quando è iniziata la visita di u in poi e non sono più stati estratti. Quindi tutti questi nodi possono correttamente essere dati in output e marcati completi. Da questo punto in poi tali nodi non verranno più presi in considerazione dall'algoritmo, ovvero è come se fossero eliminati dal grafo in modo tale che l'algoritmo proceda sul sottografo indotto da $V - C_u$.

Infine, ogni vertice viene inserito ed estratto una sola volta al più in ciascuna pila, per cui il costo asintotico rimane quello della visita ricorsiva, ovvero $O(n+m)$ tempo e $O(n)$ celle di memoria. ❑

ESEMPIO 7.6

Applichiamo l'algoritmo descritto dal Codice 7.8 sul grafo dell'Esempio 7.5 partendo dal nodo a. Di seguito per comodità riproduciamo il grafo.



L'algoritmo visita, nell'ordine i nodi a, b, c ed e e attribuisce a questi `dfsNumero` 0, 1, 2 e 3 rispettivamente. Le due pile appariranno come segue:

`rappresentanti` = [e|c|b|a] `parziali` = [e|c|b|a]

Viene ora preso in considerazione l'arco (e, b): poiché `dfsNumero[b]` è definito, b è stato già visitato e pertanto (e, b) crea un ciclo dato che b non è completo. Tutti i nodi con `dfsNumero` maggiore di b in testa a `rappresentanti` vengono eliminati dalla pila.

`rappresentanti` = [b|a] `parziali` = [e|c|b|a]

L'algoritmo prosegue visitando d, g, h e f ai quali assegna `dfsNumero` 4, 5, 6 e 7. Quindi viene analizzato l'arco (f, g) e il ciclo che questo chiude è contratto nel nodo g. La situazione delle pile è la seguente.

rappresentanti = [g][d][b][a] parziali = [f][h][g][d][e][c][b][a]

I nodi f e h non hanno altri archi uscenti quindi l'algoritmo torna sul nodo g. Nemmeno quest'ultimo ha altri archi uscenti; inoltre questo è in cima alla pila rappresentanti, quindi vengono eseguite le righe 17-21. Ovvero vengono dati in output e marcati completi i nodi f, h e g, quest'ultimo viene rimosso da rappresetanti.

rappresentanti = [d][b][a] parziali = [d][e][c][b][a]

La visita torna su d e viene analizzato l'arco (d, b). Questo crea un ciclo con b che viene quindi contratto.

rappresentanti = [b][a] parziali = [d][e][c][b][a]

Dopo aver constatato che d, e, c e b non hanno altri archi uscenti e che quest'ultimo è in cima a rappresentanti, viene data in output la seconda componente fortemente connessa formata dai nodi in cima alla pila parziali che precedono b, ovvero d, e, c e lo stesso d che viene anche tolto da rappresentanti.

rappresentanti = [a] parziali = [a]

Il nodo a ha un altro arco uscente che però raggiunge il nodo f già completo. Quindi a viene estratto da entrambe le pile e dato in output come terza componente fortemente connessa.

7.4 Opus libri: routing su Internet e cammini minimi

Le reti di computer, di cui Internet è il più importante esempio, forniscono canali di comunicazione tra i singoli nodi, che rendono possibile lo scambio di informazioni tra i nodi stessi, e con esso l'interazione e la cooperazione tra computer situati a grande distanza l'uno dall'altro. Il Web e la posta elettronica sono, in questo senso, due applicazioni di grandissima diffusione che si avvalgono proprio di questa possibilità di comunicazione tra computer diversi.

L'interazione di computer attraverso Internet avviene mediante scambio di informazioni suddivise in *pacchetti*: anche se la quantità di informazione da passare è molto elevata, come per esempio nel caso di trasmissione di documenti video, tale informazione viene suddivisa in pacchetti di dimensione fissa, che sono poi inviati in sequenza dal computer mittente al destinatario. Dato che i computer mittente e destinatario non sono direttamente collegati, tale trasmissione non coinvolge soltanto loro, ma anche un ulteriore insieme di nodi della rete, che vengono attraversati dai pacchetti nel loro percorso verso la destinazione.

Il protocollo IP (*Internet Protocol*), che è il responsabile del recapito dei pacchetti al destinatario, opera secondo un meccanismo di *packet switching*, in accordo al quale ogni pacchetto viene trasmesso in modo indipendente da tutti

gli altri nella sequenza. In questo senso, lo stesso pacchetto può attraversare percorsi diversi in dipendenza di mutate condizioni della rete, quali per esempio sopravvenuti malfunzionamenti o sovraccarico di traffico in determinati nodi.

In quest'ambito, è necessario che nella rete siano presenti alcuni meccanismi che consentano, a ogni nodo a cui è pervenuto un pacchetto diretto a qualche altro nodo, di individuare una "direzione" verso cui indirizzare il pacchetto per avvicinarlo al relativo destinatario: nello specifico, se un nodo ha d altri nodi a esso collegati direttamente, il nodo invia il pacchetto a quello adiacente più vicino al destinatario. Questo problema, noto come instradamento (*routing*) dei messaggi viene risolto su Internet nel modo seguente:[4] nella rete è presente un'infrastruttura composta da una quantità di computer (nodi) specializzati per svolgere la funzione di instradamento; ognuno di tali nodi, detti *router*, è collegato direttamente a un insieme di altri, oltre che a numerosi nodi "semplici" che fanno riferimento a esso e che svolgono il ruolo di mittenti e destinatari finali delle comunicazioni.

Ogni router mantiene in memoria una struttura di dati, che di solito è una tabella, detta *tabella di routing*, rappresentata in una forma compressa per limitarne la dimensione, che permette di associare a ogni nodo sulla rete, identificato in modo univoco dal corrispondente indirizzo IP, a 32 o 128 bit (a seconda che sia utilizzato IPv4 o IPv6), uno dei router a esso direttamente collegati.

L'instradamento di un pacchetto p da un nodo u a un nodo v di Internet viene allora eseguito nel modo seguente: p contiene, oltre all'informazione da inviare a v (il *carico* del pacchetto), un'intestazione (*header*) che contiene informazioni utili per il suo recapito; la più importante di tali informazioni è l'indirizzo IP di v. Il mittente u invia p al proprio router di riferimento r_1, il quale, esaminando lo header di p, determina se v sia un nodo con cui ha un collegamento diretto: se è così, r_1 recapita il pacchetto al destinatario, mentre, in caso contrario, determina, esaminando la propria tabella di routing, quale sia il router r_2 a esso collegato cui passare il pacchetto. Questa medesima operazione viene svolta da r_2, e così via, fino a quando non viene raggiunto il router r_t direttamente connesso a v, che trasmette il pacchetto al destinatario.

Il percorso seguito da un pacchetto è quindi determinato dalle tabelle di routing dei router nella rete, e in particolare dai router attraversati dal pacchetto stesso. Per rendere più efficiente la trasmissione del messaggio, e in generale, l'utilizzo complessivo della rete, tali tabelle devono instradare il messaggio lungo il percorso più efficiente (o, in altri termini, meno costoso) che collega u a v. Le caratteristiche di un collegamento diretto tra due router (come la velocità di trasmissione), così come la quantità di traffico (ad esempio, pacchetti per secondo) che viaggia su di esso, consentono, a ogni istante, di assegnare un costo alla trasmissione di un pacchetto sul collegamento.

[4] Al fine di rendere più agevole la comprensione degli aspetti rilevanti per le finalità di questo libro, stiamo volutamente semplificando l'esposizione dei meccanismi di routing su Internet.

Un'assegnazione di costi a tutti i collegamenti tra router consente di modellare l'infrastruttura dei router come un grafo orientato pesato sugli archi, i cui nodi rappresentano i router, gli archi i collegamenti diretti tra router e il peso associato a un arco il costo di trasmissione di un pacchetto sul relativo collegamento. In tal modo, l'obiettivo di effettuare il routing di un pacchetto nel modo più efficiente si riduce nel cercare, dato il grafo pesato che modella la rete, il cammino di costo minimo dal router associato a u a quello associato a v.

I router utilizzano dei protocolli appositi per raccogliere le informazioni sui costi di tutti i collegamenti, e per costruire quindi le tabelle di routing che instradano i messaggi lungo i percorsi più efficienti. Tali protocolli rientrano in due tipologie: *link state* e *distance vector*.

I protocolli *link state*, come ad esempio OSPF (*Open Shortest Path First*), operano in modo tale che tutti i router, scambiandosi opportuni messaggi, acquisiscono l'intero stato della rete, e quindi tutto il grafo pesato che modella la rete stessa. A questo punto ogni router r_i applica su tale grafo un algoritmo di ricerca, come l'algoritmo di Dijkstra che esamineremo nel paragrafo successivo, per determinare l'insieme dei cammini minimi da r a qualunque altro router: se $r_i, r_j, ..., r_s$ è il cammino minimo individuato da r_i che passa attraverso il router r_j a lui vicino, la tabella di routing di r_i associa il router r_j a tutti gli indirizzi verso r_s.

I protocolli *distance vector*, come per esempio RIP (*Routing Information Protocol*) effettuano la determinazione dei cammini minimi senza scambiare l'intero grafo tra i router. Essi invece applicano un diverso algoritmo per la ricerca dei cammini minimi da un nodo a tutti gli altri, l'algoritmo di Bellman-Ford, che esamineremo anch'esso nel seguito: tale algoritmo, come vedremo, ha la peculiarità di operare mediante aggiornamenti continui operati in corrispondenza agli archi del grafo e, per tale caratteristica, si presta a un'elaborazione "collettiva" dei cammini minimi da parte di tutti i router nella rete.

7.4.1 Problema della ricerca di cammini minimi su grafi

La ricerca del cammino più corto (*shortest path*) tra due nodi in un grafo rappresenta un'operazione fondamentale su questo tipo di struttura, con importanti applicazioni. In generale, come osservato sopra, la conoscenza dei cammini minimi tra i nodi è un importante elemento in tutti i metodi di routing su rete, vale a dire in tutti i metodi che, dati un'origine e una destinazione di un messaggio, determinano il percorso più conveniente da seguire per il messaggio stesso. Tra questi, particolare importanza riveste l'instradamento di pacchetti su Internet, come visto sopra, ma anche altre applicazioni di larga diffusione, quali per esempio la ricerca del miglior percorso stradale verso una destinazione effettuata da un navigatore satellitare o da sistemi disponibili su Internet e largamente utilizzati

quali MapQuest, GoogleMaps, Bing Maps o YahooMaps che operano sulla base di algoritmi per la ricerca di cammini minimi.

Come già visto nel Paragrafo 7.2, in un grafo non pesato il cammino minimo tra due nodi u e v può essere trovato mediante una visita in ampiezza a partire da u, utilizzando una coda in cui memorizzare i nodi man mano che vengono raggiunti e da cui estrarli, secondo una politica FIFO, per procedere nella visita.

Consideriamo ora il caso generale in cui il grafo $G = (V, E)$ sia pesato con valori reali sugli archi attraverso una funzione $W : E \mapsto \mathbb{R}$: come già notato nel Paragrafo 7.1, un cammino $v_0, v_1, v_2, ..., v_k$ ha associato un peso (che interpretiamo come lunghezza) pari alla somma $W(v_0, v_1) + W(v_1, v_2) + ... + W(v_{k-1}, v_k)$ dei pesi degli archi che lo compongono. Nel seguito, la lunghezza sarà intesa come peso totale del cammino.

Dati due nodi $u, v \in V$, esistono in generale più cammini che collegano u a v, ognuno con una propria lunghezza: ricordiamo che la distanza pesata $\delta(u, v)$ da u a v è definita come la lunghezza di un cammino di peso minimo da u a v. Notiamo che, se il grafo è non orientato, vale $\delta(u, v) = \delta(v, u)$, in quanto a ogni cammino da u a v corrisponde un cammino (il medesimo percorso al contrario) da v a u della stessa lunghezza: ciò non è vero, in generale, se il grafo è orientato. Per il grafo nella Figura 7.15, possiamo osservare per esempio che $\delta(v_1, v_6) = 35$, corrispondente al cammino orientato v_1, v_3, v_7, v_6, mentre $\delta(v_6, v_1) = 57$, corrispondente al cammino orientato v_6, v_2, v_5, v_1. In effetti, se il grafo non è fortemente connesso può avvenire che per due nodi u, v la distanza $\delta(u, v)$ sia finita, mentre $\delta(v, u)$ è infinita: questo è il caso per esempio dei nodi v_{11} e v_{12} nella Figura 7.15, per i quali $\delta(v_{11}, v_{12}) = 12$ mentre $\delta(v_{12}, v_{11}) = +\infty$, in quanto non è possibile raggiungere v_{11} da v_{12}.

Il problema che consideriamo è quello che, dato un grafo pesato $G = (V, E, W)$ (orientato o meno) con funzione di peso $W : E \mapsto \mathbb{R}$, richiede di individuare i cammini di lunghezza minima tra i nodi del grafo stesso. Tale problema assume caratteristiche diverse, in termini di miglior modo di risolverlo, in dipendenza del numero di cammini minimi che ci interessa individuare nel grafo: in particolare, se siamo interessati a individuare gli $n(n-1)$ cammini minimi tra tutte le coppie di nodi (*all pairs shortest path*), avremo che i metodi più efficienti di soluzione

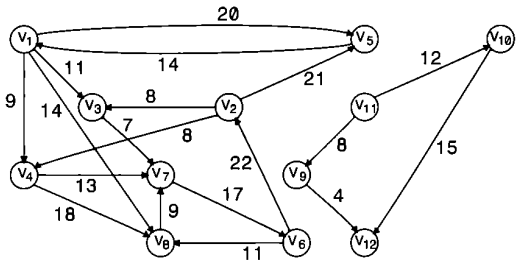


Figura 7.15 Esempio di grafo orientato con pesi sugli archi.

possono essere diversi rispetto al caso in cui ci interessano soltanto gli $n-1$ cammini minimi da un nodo a tutti gli altri (*single source shortest path*).

Come vedremo, la complessità di risoluzione di questo problema dipende, tra l'altro, dalle caratteristiche della funzione W: in particolare, considereremo dapprima il caso in cui $W : E \mapsto \mathbb{R}^+$, in cui cioè i pesi degli archi sono positivi. Sotto questa ipotesi introdurremo il classico algoritmo di Dijkstra (definito per il caso *single source*, ma utilizzabile anche per quello *all pairs*), e vedremo che questo algoritmo è una riproposizione degli algoritmi di visita discussi nel Paragrafo 7.2, in cui viene utilizzata però una coda con priorità come struttura di dati, al posto della coda e della pila.

Nel caso generale in cui i pesi degli archi possono essere anche nulli o negativi, non possiamo usare l'algoritmo di Dijkstra. Vedremo allora come risolvere il problema diversamente, per quanto riguarda sia la ricerca *all pairs* che quella *single source*, anche se meno efficientemente del caso in cui i pesi sono positivi.

7.4.2 Cammini minimi in grafi con pesi positivi

Consideriamo inizialmente il problema di tipo *single source*: in tal caso, dato un grafo $G = (V, E, W)$ con $W : E \mapsto \mathbb{R}^+$ e dato un nodo $s \in V$, vogliamo derivare la distanza $\delta(s, v)$ da s a v, per ogni nodo $v \in V$. Se per esempio consideriamo ancora il grafo nella Figura 7.15, allora per $s = v_1$ vogliamo ottenere l'insieme di coppie $(v_1,0)$, $(v_2,57)$, $(v_3,11)$, $(v_4,9)$, $(v_5,20)$, $(v_6,35)$, $(v_7,18)$, $(v_8,14)$, $(v_9,+\infty)$, $(v_{10},+\infty)$, $(v_{11},+\infty)$, $(v_{12},+\infty)$, associando a ogni nodo la relativa distanza da v_1.

Inoltre, vogliamo ottenere anche, per ogni nodo, il cammino minimo stesso: a tal fine, è sufficiente ottenere, per ogni nodo v, l'indicazione del nodo u che precede v nel cammino minimo da s a v stesso. Ciò è sufficiente in quanto vale la seguente proprietà.

Lemma 7.1 *Se* $s, u_0, u_1, \ldots, u_r, u, v$ *è il cammino minimo da* s *a* v, *allora la sequenza* $s, u_0, u_1, \ldots, u_k, u$ *è il cammino minimo da* s *a* u.

Dimostrazione Per assurdo, se il risultato non valesse dovrebbe esistere un altro cammino $s, w_0, w_1, \ldots, w_t, u$ più corto, allora anche il cammino $s, w_0, w_1, \ldots, w_t, u, v$ avrebbe lunghezza inferiore a $s, u_0, u_1, \ldots, u_r, u, v$, contraddicendo l'ipotesi fatta che tale cammino sia minimo. ❑

Come vedremo ora, questo problema può essere risolto mediante un algoritmo di visita del grafo che fa uso di una coda con priorità per determinare l'ordine di visita dei nodi e che viene indicato come algoritmo di Dijkstra.

Gli elementi nella coda con priorità sono coppie (v, p), con $v \in V$ e $p \in \mathbb{R}^+$, ordinate rispetto ai rispettivi pesi p: come vedremo, l'algoritmo mantiene l'invariante che, per ogni coppia (v, p) nella coda con priorità, abbiamo $p \geq \delta(s,v)$ e, nel momento in cui (v,p) viene estratta dalla coda con priorità, abbiamo $p = \delta(s, v)$.

In particolare, a ogni istante il peso p associato al nodo v nella coda con priorità indica la lunghezza del cammino più corto trovato finora nel grafo: tale peso viene aggiornato ogni qual volta viene individuato un cammino più breve da s a v di quello considerato finora.

L'algoritmo, mostrato nel Codice 7.9, determina la distanza di ogni nodo in V da s, oltre al corrispondente cammino minimo, utilizzando due array: `dist` associa a ogni nodo v la lunghezza del più breve cammino da s a v individuato finora e `pred` rappresenta il nodo che precede v in tale cammino.

Codice 7.9 Algoritmo di Dijkstra per la ricerca dei cammini minimi *single source*, dove la variabile `elemento` indica un nuovo elemento allocato.

```
1  Dijkstra( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      pred[u] = -1;
4      dist[u] = +∞;
5    }
6    pred[s] = s;
7    dist[s] = 0;
8    FOR (u = 0; u < n; u = u + 1) {
9      elemento.peso = dist[u];
10     elemento.dato = u;
11     PQ.Enqueue( elemento );
12   }
13   WHILE (!PQ.Empty( )) {
14     e = PQ.Dequeue( );
15     v = e.dato;
16     FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
17       u = x.dato;
18       IF (dist[u] > dist[v] + x.peso) {
19         dist[u] = dist[v] + x.peso;
20         pred[u] = v;
21         PQ.DecreaseKey( u, dist[u] );
22       }
23     }
24   }
```

Per effetto dell'esecuzione del ciclo iniziale (righe 2-12), per ogni nodo $v \in V - \{s\}$, `dist[v]` viene posto pari a $+\infty$ in quanto al momento non conosciamo alcun cammino da s a v; inoltre viene posto `dist[s]` pari a `0` in quanto s dista `0` da se stesso.

I valori `pred[v]` vengono posti, per tutti i nodi eccetto s, pari a -1, valore utilizzato per codificare il fatto che non è noto alcun cammino da s a v, mentre per

s adottiamo la convenzione che `pred[s] = s`. Tutti i nodi vengono inoltre inseriti nella coda con priorità, con associati i rispettivi pesi.

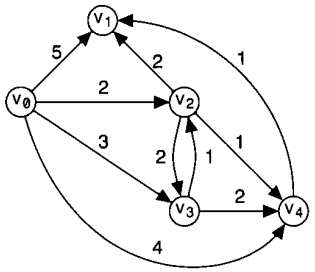
Nel ciclo successivo (righe 13-24), l'algoritmo procede a estrarre dalla coda con priorità il nodo v con peso minimo (riga 14): come vedremo sotto, il peso associato al nodo è pari alla distanza $\delta(s, v)$.

In corrispondenza di questa estrazione, vengono esaminati tutti gli archi uscenti da v (righe 16-23): per ogni arco `x = (v, u)` considerato, ci chiediamo se il cammino s, ..., v, u, di lunghezza $\delta(s, v) + W(v, u) = $ `dist[v] + x.peso`, è più corto della distanza del cammino minimo da s a u trovato fino a ora, dove tale distanza è memorizzata in `dist[u]`.

In tal caso (o nel caso in cui s, ..., v, u sia il primo cammino trovato da s a u) il peso associato a u viene decrementato al valore $\delta(s, v) + W(v, u)$ (riga 21) e v diventa il predecessore di u nel cammino minimo.

ESEMPIO 7.7

Applichiamo l'algoritmo di Dijkstra al grafo riportato nella seguente figura utilizzando come nodo radice v_0.



Dopo la fase di inizializzazione ecco come appaiono gli array `pred` e `dist` e lo heap PQ. Si osservi che negli array i nodi vengono riferiti utilizzando i loro indici.

	0	1	2	3	4
pred =	0	-1	-1	-1	-1
dist =	0	+∞	+∞	+∞	+∞

PQ = $(0, v_0)$, $(+\infty, v_1)$, $(+\infty, v_2)$, $(+\infty, v_3)$, $(+\infty, v_4)$

Nel primo passo del ciclo delle righe 13-24 viene estratto da PQ la coppia $(0, v_0)$ e vengono presi in considerazione tutti i nodi raggiungibili con un arco da v_0. Per questi vale sempre che la loro distanza attuale è maggiore della distanza da v_0 quindi il loro valore sull'array `dist` viene aggiornato alla lunghezza dell'arco da v_0 mentre il predecessore sull'array `pred` diventa 0. Infine viene decrementato il peso su PQ in base alla nuova distanza calcolata.

	0	1	2	3	4
pred =	0	5	2	3	4
dist =	0	0	0	0	0

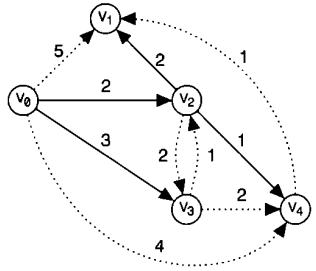
PQ = $(2, v_2)$, $(4, v_4)$, $(3, v_3)$, $(5, v_1)$

Viene estratta dalla pila la coppia $(2, v_2)$ e presi in considerazione i vicini di v_2. Per il nodo v_1 vale `dist[1] = 5 > dist[2] + W(`v_2`, `v_1`) = 4` quindi vengono aggiornati sia `dist[1]` che `pred[1]` oltre, ovviamente, il peso di v_1 in PQ. In maniera analoga la stessa cosa si applica a v_4. Mentre per l'altro vicino, v_3, il vecchio cammino è ancora il migliore.

	0	1	2	3	4
pred =	0	4	2	3	3
dist =	0	2	0	0	2

PQ = $(3, v_4)$, $(4, v_1)$, $(3, v_3)$

Si estrae da PQ la coppia $(3, v_4)$. Il nodo v_4 ha un solo arco uscente verso v_1 che però non accorcia la distanza di v_0 da v_1. La stessa cosa vale per la coppia estratta successivamente ovvero $(3, v_3)$ mentre l'ultimo nodo in PQ, v_1, non ha archi uscenti. Essendo PQ vuoto l'algoritmo ha termine. Nella figura che segue sono evidenziati gli archi che fanno parte dei cammini minimi da v_0.



Gli archi evidenziati costituiscono l'albero dei cammini minimi del grafo G a partire dal nodo v_0.

Il decremento del peso di un elemento in una coda con priorità viene eseguito mediante l'operazione `DecreaseKey`. Questa funzione deve prima cercare l'elemento nella coda avente la chiave specificata e poi muovere l'elemento nella posizione corretta (si veda il Paragrafo 2.4). Mentre siamo in grado di eseguire la seconda parte in modo efficiente, per la prima parte abbiamo bisogno di modificare le funzioni di gestione della coda con priorità onde evitare il ricorso a una ricerca lineare. Se la coda con priorità viene implementata con uno heap implicito è sufficiente tener traccia della posizione di ogni elemento: se le chiavi sono di tipo intero, come nel nostro caso, basta introdurre un array `posizioneHeapArray` di n interi, tanti quanti sono gli elementi nello heap, che rappresenta, per ogni chiave k, la posizione che occupa k nell'array che definisce lo heap. In questo modo si accede a una determinata chiave dello heap in tempo costante. Infine è facile modificare le funzioni di gestione dello heap (Codice 2.4 e 2.5), in modo da tener aggiornato l'array `posizioneHeapArray`.

Teorema 7.5 *L'algoritmo di Dijkstra è corretto, ovvero calcola i cammini minimi da* s *verso tutti i nodi del grafo.*

Dimostrazione Per prima cosa notiamo che il peso associato a un nodo $v \in V - \{s\}$ non è mai inferiore alla distanza $\delta(s, v)$, in quanto all'inizio tale peso è pari a $+\infty$ e ogni volta che viene aggiornato viene posto pari alla lunghezza di un qualche cammino esistente da s a v, di lunghezza quindi non inferiore a $\delta(s, v)$. Per quanto riguarda s, il suo peso è posto pari alla distanza da se stesso e mai aggiornato, in quanto s è necessariamente il primo nodo estratto dalla coda con priorità. Inoltre, per la struttura dell'algoritmo, la sequenza dei pesi associati nel tempo a ogni nodo v è monotona decrescente.

Ciò che rimane da dimostrare è che il peso dist[v] di un nodo v al momento della sua estrazione dalla coda con priorità è pari a $\delta(s, v)$. A tal fine, dato un intero $i > 0$ e un nodo $v \in V$, indichiamo con S_i l'insieme dei primi i nodi estratti dalla coda e con $\delta_i(s, v)$ la lunghezza del cammino minimo da s a v passante per i soli nodi in S_i.

Mostriamo ora che, dopo che l'algoritmo ha considerato i primi i nodi estratti, se un nodo v è ancora nella coda, il peso a esso associato è pari alla lunghezza del cammino minimo da s a v passante per i soli nodi in S_i, e quindi che dist[v] = $\delta_i(s, v)$.

Questo è vero all'inizio dell'algoritmo quando, dopo la prima iterazione nel corso della quale è stato estratto s, per ogni nodo v adiacente a s abbiamo dist[v] = W(s, v). Dato che W(s, v) è anche la lunghezza dell'unico cammino che collega s a v passando per i soli nodi in $S_1 = \{s\}$, e quindi abbiamo in effetti dist[v] = $\delta_1(s, v)$.

Ragionando per induzione, facciamo ora l'ipotesi che la proprietà sia vera dopo k - 1 nodi estratti dalla coda con priorità, e mostriamo che rimane vera anche dopo k estrazioni. Supponiamo che u sia il k-esimo nodo estratto: per l'ipotesi induttiva, il peso di u nella coda al momento dell'estrazione è pari alla lunghezza del più corto cammino da s a u passante per i soli nodi in S_{k-1}, e quindi dist[u] = $\delta_{k-1}(s, u)$.

Consideriamo ora un qualunque nodo v adiacente a u e non ancora estratto e un cammino minimo da s a v passante per i soli nodi in $S_k = S_{k-1} \cup \{u\}$. Possiamo avere due possibilità:

1. tale cammino non passa per u, nel qual caso corrisponde al cammino minimo passante soltanto per nodi in S_{k-1}, e quindi $\delta_k(s, v) = \delta_{k-1}(s, v)$;
2. tale cammino passa anche per u, nel qual caso corrisponde al cammino minimo da s a u seguito dall'arco (u, v), e $\delta_k(s, v) = \delta_{k-1}(s, u) + W(u, v)$.

In entrambi i casi possiamo verificare come l'algoritmo del Codice 7.9 operi in modo da assegnare a dist[v] il valore $\delta_k(s, v)$, e quindi da mantenere vera la proprietà che stiamo mostrando.

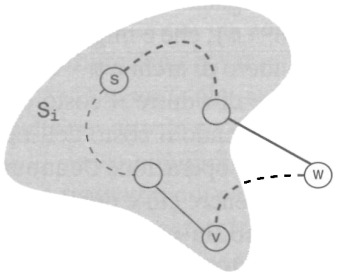


Figura 7.16 Dimostrazione di correttezza dell'algoritmo di Dijkstra.

Mostriamo ora che, se $v \in V$ è l'i-esimo nodo estratto, il cammino minimo da s a v passa necessariamente per i soli nodi in S_i. Infatti, supponiamo per assurdo che tale cammino passi per alcuni nodi che si trovano ancora nella coda al momento in cui v viene estratto, e indichiamo con w il primo di tali nodi che compare nel cammino minimo (Figura 7.16).

Dato che w precede v in tale cammino minimo, avremmo che $\delta(s,w) < \delta(s,v)$; inoltre, dato che stiamo ipotizzando che il cammino minimo non passi tutto per nodi in S_i, abbiamo che $\delta(s, v) < \delta_i(s, v)$; infine, dato che v viene estratto dalla coda con priorità prima di w, abbiamo che $\delta_i(s, v) < \delta_i(s, w)$. Ma ciò porta a una contraddizione, se consideriamo che, dato che w è per ipotesi il primo nodo non appartenente a S_i nel cammino minimo, abbiamo anche $\delta(s, w) = \delta_i(s, w)$: infatti, otteniamo allora che $\delta(s, w) < \delta(s, v) < \delta_i(s, v) < \delta_i(s, w) = \delta(s, w)$, il che è impossibile. ❑

Per valutare il costo computazionale dell'algoritmo del Codice 7.9, osserviamo che esso dipende dal costo delle operazioni eseguite sulla coda di priorità. Indichiamo allora con t_c il costo di costruzione della coda, con t_e il costo di estrazione del minimo con l'operazione Dequeue e con t_d il costo dell'operazione DecreaseKey.

Come possiamo osservare, su un grafo di n nodi e m archi, l'algoritmo esegue n estrazioni del minimo e al più m DecreaseKey, da cui consegue che il suo tempo di esecuzione è $O(t_c + nt_e + mt_d)$. Se utilizziamo l'implementazione della coda con priorità mediante uno heap, descritta nel Paragrafo 2.4, abbiamo che $t_c = O(n)$, $t_e = O(\log n)$ e $t_d = O(\log n)$, per cui l'algoritmo ha costo complessivo pari a $O((n+m)\log n)$ tempo, che risulta $O(m \log n)$ se supponiamo che il grafo sia connesso, e quindi $m = \Omega(n)$.

Possiamo ottenere un miglioramento del costo dell'algoritmo utilizzando la semplice implementazione a lista della coda con priorità: infatti, se tale implementazione viene effettuata per mezzo di una lista non ordinata, avremo che $t_c = O(n)$, $t_e = O(n)$ e $t_d = O(1)$, in quanto il decremento del peso di

un nodo non comporta alcuna riorganizzazione della struttura. In questo caso, il costo dell'algoritmo risulta $O(n^2+m)$, che è migliore del caso precedente per grafi densi, in cui in particolare il numero di archi è $m = \Omega(n^2/\log n)$.

In generale, possiamo pensare di ridurre il costo complessivo dell'algoritmo, almeno su grafi non sparsi, diminuendo il costo dell'operazione `DecreaseKey` a fronte di un aumento del costo dell'operazione `Dequeue`.

Un modo, più bilanciato, di ottenere ciò è quello di utilizzare heap non binari, ma di grado $d \geq 2$: in questo caso, la profondità dello heap, e quindi il costo della `DecreaseKey` diviene $t_d = O(\log_d n)$, mentre la `Dequeue` richiede tempo $t_e = O(d \log_d n)$ in quanto deve esaminare, a ogni livello, tutti i d figli del nodo attuale.

Teorema 7.6 *L'algoritmo di Dijkstra ha costo* $O(m \log_d n)$ *se la coda con priorità è implementata con heap di grado* $d = \max\{2, m/n\}$.

Dimostrazione Se $d = 2$ allora $m = O(n)$ quindi $O(t_c + nt_e + mt_d) = O(n \log n) = O(m \log_d n)$. Altrimenti $d = m/n$ e $O(t_c + nt_e + mt_d) = O(nd \log_d n + m \log_d n) = O(m \log_d n)$. ❑

Quindi se $m = \Theta(n^2)$ il costo risulta $O(m)$ altrimenti se $m = \Theta(n)$ il costo diviene $O(n \log n)$.

Osserviamo infine che esiste un'implementazione della coda con priorità, lo **heap di Fibonacci**, che fornisce le operazioni `Enqueue`, `Dequeue` e `DecreaseKey` con un costo ammortizzato rispettivamente pari a $O(1)$, $O(\log n)$ e $O(1)$. Ciò comporta che ogni sequenza di p operazioni `Enqueue`, di $q \leq p$ operazioni `Dequeue` e di r operazioni `DecreaseKey` ha un costo complessivo di $O(p + r + q \log n)$ tempo. L'utilizzo di questa implementazione permette allora di eseguire l'algoritmo del Codice 7.9 in tempo $O(n \log n + m)$, che rappresenta il miglior costo possibile nel caso di grafi con $m = \Omega(n \log n)$.

La ricerca dei cammini minimi tra tutte le coppie di nodi (vale a dire nel caso *all pairs*) può essere effettuata applicando n volte, una per ogni possibile nodo sorgente, l'algoritmo di Dijkstra illustrato sopra: ciò fornisce un metodo di soluzione di tale problema avente costo $O(n^2 \log n + nm)$.

7.4.3 Cammini minimi in grafi pesati generali

L'algoritmo di Dijkstra non è applicabile al problema dei cammini minimi quando gli archi del grafo possono avere pesi negativi, perché un arco potrebbe diminuire la lunghezza del cammino di un nodo già estratto dalla coda con priorità. Per trattare la versione generale del problema dobbiamo comunque ipotizzare che nel grafo, anche in presenza di archi a peso negativo, non esistano *cicli* aventi peso complessivo negativo (per cui è negativa la somma dei pesi degli archi corrispondenti): se $v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}$ fosse un tale ciclo, avente peso $-D$, la distanza tra due qualunque di tali nodi sarebbe $-\infty$. Notiamo infatti che attraversare il ciclo

comporta che la distanza complessivamente percorsa viene decrementata di D, e quindi, dato che esso può essere percorso un numero arbitrariamente grande di volte, la distanza può divenire arbitrariamente grande in valore assoluto e negativa.

Notiamo inoltre che ciò avviene non solo per quanto riguarda la distanza tra due nodi del ciclo, ma in realtà per tutte le coppie di nodi nella stessa componente connessa che include il ciclo stesso e quindi, se il grafo è connesso per tutte le coppie di nodi, la cui distanza risulta quindi pari a $-\infty$.

Il metodo più diffuso per la ricerca *single source* dei cammini minimi è il cosiddetto algoritmo di Bellman-Ford, che opera come mostrato nel Codice 7.10. L'algoritmo ha una struttura molto semplice e simile a quello di Dijkstra. Come possiamo vedere, non necessita di strutture di dati particolari: anche in questo caso per ogni nodo `v` viene mantenuta in `dist[v]` la lunghezza del cammino più breve da `s` a `v` finora trovato dall'algoritmo e in `pred[v]` il nodo che precede immediatamente `v` lungo tale cammino. Facciamo l'ipotesi che il valore `-1` per `pred[v]` stia a rappresentare il fatto che non viene rappresentato alcun cammino da `s` a `v`; inoltre, poniamo anche `pred[s] = s`.

Codice 7.10 Algoritmo di Bellman-Ford per la ricerca dei cammini minimi *single source*.

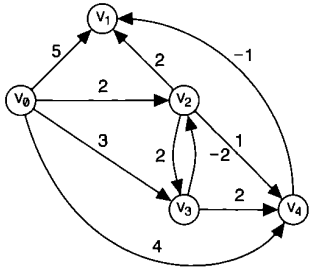
```
1  Bellman-Ford( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      pred[u] = -1;
4      dist[u] = +∞;
5    }
6    pred[s] = s;
7    dist[s] = 0;
8    FOR (i = 0; i < n; i = i + 1)
9      FOR (v = 0; v < n; v = v + 1) {
10       FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
11         u = x.dato;
12         IF (dist[u] > dist[v] + x.peso) {
13           dist[u] = dist[v] + x.peso;
14           pred[u] = v;
15         }
16       }
17     }
```

Dopo l'inizializzazione (righe 2-7) come nell'algoritmo di Dijkstra, l'algoritmo opera per $n = |V|$ iterazioni la scansione di tutti gli archi di `G` (righe 8-17): per ogni arco `(u, v)` esaminato, verifichiamo se i valori `dist[u]` e `dist[v]` soddisfano la condizione `dist[u] > dist[v] + W(u, v)` (riga 12). Se è così, questo vuol dire che esiste un cammino `s, ..., v, u` più breve del miglior cammino da `s` a `u` trovato

finora: le informazioni `dist[u]` e `dist[v]` vengono allora aggiornate per rappresentare questa nuova situazione.

ESEMPIO 7.8

Prendiamo in considerazione il grafo mostrato nell'Esempio 7.7 cambiando soltanto i pesi degli archi (v_3, v_2) e (v_4, v_1). Il grafo risultante, ancora privo di cicli negativi, è mostrato nella figura seguente.



Anche in questo caso calcoliamo i cammini minimi a partire dal nodo v_0. Ecco lo stato degli array `pred` e `dist` dopo la fase di inizializzazione.

	0	1	2	3	4
pred =	0	-1	-1	-1	-1
dist =	0	+∞	+∞	+∞	+∞

Di seguito indichiamo lo stato degli array per i valori di `i` e `v`.

`i=0, v=0,1`

	0	1	2	3	4
pred =	0	0	0	0	0
dist =	0	5	2	3	4

Il nodo v_1 non ha archi uscenti quindi si passa a `v = 2`.

`i=0, v=2`

	0	1	2	3	4
pred =	0	2	0	0	2
dist =	0	4	2	3	3

`i=0, v=3,4`

	0	1	2	3	4
pred =	0	2	3	0	2
dist =	0	4	1	3	3

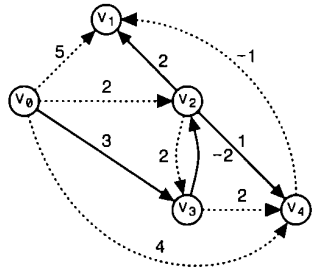
I passi `v = 3` e `v = 4` sono stati accorpati perché l'ultimo non ha apportato modifiche agli array. Osserviamo che solo per il nodo v_3 il cammino minimo da `s` è composto da un unico arco. Effettivamente all'inizio del secondo ciclo più esterno (`i = 1`) per questo nodo l'algoritmo ha trovato il cammino minimo. In realtà ha trovato anche il cammino minimo verso il nodo v_2 composto dagli archi (v_0, v_3) e (v_3, v_2) ma questo dipende solo dall'ordine in cui vengono visitati gli archi, infatti l'arco (v_3, v_2) viene visitato dopo che è stato trovato il cammino per v_3.

`i=1, v=0,1,2,3,4`

	0	1	2	3	4
pred =	0	2	3	0	2
dist =	0	3	1	3	2

Per `i = 1` l'unico cambiamento si riscontra per `v = 2` in quanto il cammino trovato verso v_2 ha peso 1, questo comporta che per v_1 abbiamo un cammino di peso 3 e per v_4 un cammino di peso 2.

Gli array non subiranno più modifiche nei restanti cicli dell'algoritmo, la soluzione trovata è rappresentata nella seguente figura in cui i nodi non tratteggiati sono quelli che fanno parte della soluzione.



Teorema 7.7 *L'algoritmo di Bellman-Ford è corretto.*

Dimostrazione Dimostriamo che al momento della terminazione dell'algoritmo, abbiamo `dist[v]` = δ`(s, v)` per ogni nodo `v`, ragionando per induzione: verifichiamo che, all'inizio dell'iterazione `i` nel ciclo più esterno (righe 8-17), abbiamo che `dist[v]` = δ`(s, v)` per ogni nodo `v` per il quale il cammino minimo da `s` a `v` è composto da al più `i` archi.

Questa proprietà è vera per `i = 0`, in quanto `s` è il solo nodo tale che il cammino minimo da `s` a `s` è composto da 0 archi, e in effetti `dist[s] = 0` = δ`(s, s)`.

Per il passo induttivo, ipotizziamo ora che la proprietà sia vera all'inizio dell'iterazione `i`: se consideriamo un qualunque nodo `v` tale che il cammino minimo `s, ..., u, v` comprende `i + 1` archi, ne consegue che per il nodo `u` il cammino minimo `s, ..., u` comprende `i` primi `i` archi del cammino precedente, e quindi abbiamo `dist[u]` = δ`(s, u)`. Ma allora, nel corso dell'iterazione `i`, abbiamo che `dist[v]` è aggiornato in modo che `dist[v] = dist[u] + W(u, v)` = δ`(s, u) + W(u, v)`, che è uguale a δ`(s, v)` per l'ipotesi che il cammino `s, ..., u, v` sia minimo. Quindi la proprietà risulta soddisfatta quando inizia l'iterazione `i + 1`.

Se osserviamo che, dato che non esistono per ipotesi cicli di lunghezza negativa in `G`, un cammino minimo comprende al più `n - 1` archi, possiamo concludere che quando `i = n` tutti i cammini minimi sono stati individuati. ❑

Per quanto riguarda il costo di esecuzione, possiamo notare che l'algoritmo esegue `O(n)` iterazioni, in ognuna delle quali esamina ogni arco due volte, e quindi esegue `O(m)` operazioni: ne consegue che il tempo di esecuzione è `O(nm)`. Ab-

biamo quindi che la più vasta applicabilità (anche a grafi con pesi negativi, se non abbiamo cicli negativi) rispetto all'algoritmo di Dijkstra viene pagata da una minore efficienza.

Osserviamo infine che, per quanto detto, se l'algoritmo effettuasse più di n iterazioni, l'ultima di esse non vedrebbe alcun aggiornamento dei valori nell'array `dist`, in quanto i cammini minimi sono stati già tutti trovati. Invece, se il grafo avesse cicli di lunghezza negativa, ci sarebbero valori in `dist[v]` che continuerebbero a decrementarsi indefinitamente.

Da questa osservazione deriva che l'algoritmo di Bellman-Ford può essere utilizzato anche per verificare se un grafo G presenta cicli di lunghezza negativa. A tal fine, se n è il numero di nodi, basta eseguire n + 1 iterazioni e verificare se nel corso dell'ultima vengono modificati valori nell'array `dist`: se così è, possiamo concludere che il grafo presenta in effetti dei cicli negativi.

Se consideriamo ora il problema della ricerca dei cammini minimi tra tutti i nodi, possiamo dire che una semplice soluzione è data, come nel caso dei grafi a pesi positivi, dall'iterazione su tutti i nodi dell'algoritmo per la ricerca *single source*: nel nostro caso, questo risulterebbe in un tempo di esecuzione $O(n^2m)$. Possiamo tuttavia ottenere di meglio utilizzando il paradigma della programmazione dinamica, introdotto nel Capitolo 6.

Per seguire questo approccio, utilizziamo la rappresentazione dei grafi pesati mediante la matrice di adiacenza A a cui è associata quella dei pesi P, tale che $P[u,v] = W(u,v)$ se e solo se $A[u,v] = 1$ (vale a dire $(u,v) \in E$), dove richiediamo che $P[u,u] = 0$ per $0 \le u, v < n$: gli altri valori di P sono considerati pari a $+\infty$ in quanto non esiste un arco di collegamento. Dato un cammino da u a v, definiamo come *nodo interno* del cammino un nodo w, diverso sia da u che da v, che compare nel cammino stesso. Per ogni $k \le n$, indichiamo con V_k l'insieme dei primi k nodi di V, abbiamo cioè per definizione che $V_k = \{0, 1, ..., k-1\}$.

Possiamo allora considerare, per ogni coppia di nodi u e v e per ogni k, il cammino minimo $\pi_k(u, v)$ da u a v passante soltanto per nodi in V_k (eccetto per quanto riguarda u e v stessi): se $k = n$ tale cammino è il cammino minimo $\pi(u,v)$ da u a v nell'intero grafo. Inoltre, dato che $V_0 = \phi$, il cammino minimo $\pi_0(u,v)$ da u a v in V_0 è dato dall'arco (u, v), se tale arco esiste. Se indichiamo con $\delta_k(u,v)$ la lunghezza del cammino $\pi_k(u, v)$, avremo allora che $\delta_0(u,v) = W(u,v)$ se $(u,v) \in E$, mentre $\delta_0(u, v) = +\infty$ altrimenti.

Se soffermiamo la nostra attenzione sul cammino minimo $\pi_k(u,v)$ per $0 < k \le n$, possiamo notare che possono verificarsi due eventualità:

1. il nodo $k-1$ non appare in $\pi_k(u, v)$, ma allora $\pi_k(u, v) = \pi_{k-1}(u, v)$ (il cammino minimo è lo stesso che nel caso in cui utilizzavamo soltanto i nodi {0, 1, ..., k-2} come nodi interni);
2. al contrario, il nodo $k-1$ appare in $\pi_k(u, v)$, ma allora, dato che possiamo assumere che tale cammino non contenga cicli in quanto ogni ciclo (che non può avere lunghezza negativa) non lo potrebbe rendere più corto, $\pi_k(u, v)$ può essere suddiviso in due parti: un primo cammino da u a $k-1$ passante soltanto per nodi in V_{k-1} e un secondo cammino da $k-1$ a v passante anch'esso per soli nodi in V_{k-1}.

Possiamo quindi enunciare il seguente teorema che ci fornisce la regola di programmazione dinamica per calcolare la lunghezza $\delta_k(u, v)$ del cammino minimo da u a v.

Teorema 7.8 *Per ogni coppia di nodi* u *e* v *e per ogni intero* $0 < k \le n$ *si ha*

$$\delta_k(u, v) = \begin{cases} P[u, v] & \text{se } k = 0 \\ \min\{\delta_{k-1}(u, v), \delta_{k-1}(u, k-1) + \delta_{k-1}(k-1, v)\} & \text{se } k \ge 1. \end{cases}$$

Seguendo l'approccio della programmazione dinamica, abbiamo quindi un meccanismo di decomposizione del problema in sottoproblemi più semplici, con una soluzione definita per i sottoproblemi elementari, corrispondenti al caso $k = 0$. A questo punto, l'algoritmo risultante, detto algoritmo di Floyd-Warshall, presenta l'usuale struttura di un algoritmo di programmazione dinamica. In particolare, esso opera (almeno concettualmente) su una coppia di tabelle tridimensionali `dist` e `pred`, ciascuna di n "strati", n righe e n colonne: per il cammino $\pi_k(u,v)$, l'algoritmo utilizza queste tabelle per memorizzare $\delta_k(u, v)$ in `dist[k][u][v]` secondo la relazione del Teorema 7.8 e il predecessore di v in tale cammino in `pred[k][u][v]`.

In realtà, un più attento esame dell'equazione del Teorema 7.8 mostra che possiamo ignorare l'indice $k-1$: infatti $\delta_{k-1}(u, k-1) = \delta_k(u, k-1)$ e $\delta_{k-1}(k-1, v) = \delta_k(k-1, v)$ in quanto il nodo di arrivo (nel primo caso) e di partenza (nel secondo caso) è $k-1$, il quale non può essere un nodo interno in tali cammini quando passiamo da V_{k-1} a V_k. Tale osservazione ci permette di utilizzare le tabelle bidimensionali $n \times n$ per rappresentare `dist` e `pred`, come possiamo vedere nel Codice 7.11.

Codice 7.11 Algoritmo di Floyd-Warshall per la ricerca dei cammini minimi *all pairs*.

```
1  Floyd-Warshall( ):
2    FOR (u = 0; u < n; u = u + 1)
3      FOR (v = 0; v < n; v = v + 1) {
4        dist[u][v] = P[u][v]; pred[u][v] = u;
5      }
6    FOR (k = 1; k <= n; k = k + 1)
7      FOR (u = 0; u < n; u = u + 1) {
8        FOR (v = 0; v < n; v = v + 1) {
9          IF (dist[u][v] > dist[u][k-1] + dist[k-1][v]) {
```

```
10          dist[u][v] = dist[u][k-1] + dist[k-1][v];
11          pred[u][v] = pred[k-1][v];
12        }
13      }
14    }
```

L'algoritmo, dopo una prima fase di inizializzazione degli elementi `dist[u][v]` (righe 2-5), passa a derivare iterativamente tutti i cammini minimi in V_k al crescere di k (righe 6-14). A tal fine, utilizza la relazione di programmazione dinamica riportata nel Teorema 7.8 per inferire, nella riga 9, quale sia il cammino minimo $\pi_k(u, v)$ e quindi quali valori memorizzare per rappresentare tale cammino e la relativa lunghezza (righe 10-12).

Il costo dell'algoritmo è determinato dai tre cicli nidificati, da cui deriva un tempo di computazione $O(n^3)$; per quanto riguarda lo spazio utilizzato, il Codice 7.11 fa uso, come detto sopra, di due array di n × n elementi, totalizzando $O(n^2)$ spazio.

ESEMPIO 7.9

Applichiamo l'algoritmo di Floyd-Warshall descritto nel Codice 7.11 al grafo dell'Esempio 7.8 mostrando come evolvono le tabelle `dist` e `pred`. Dopo l'inizializzazione abbiamo la situazione seguente.

dist

	0	1	2	3	4
0	0	5	2	3	4
1	+∞	0	+∞	+∞	+∞
2	+∞	2	0	2	1
3	+∞	+∞	-2	0	2
4	+∞	1	+∞	+∞	0

pred

	0	1	2	3	4
0	0	0	0	0	0
1	-1	1	-1	-1	-1
2	-1	2	2	2	2
3	-1	-1	3	3	3
4	-1	4	-1	-1	4

Al termine del ciclo più esterno per k = 1, 2 le tabelle non cambiano in quanto `dist[u][0]` = +∞ e `dist[1][u]` = +∞ per tutti i v. Alla fine del ciclo per k = 3 le tabelle cambiano nel modo che segue.

dist

	0	1	2	3	4
0	0	4	2	3	3
1	+∞	0	+∞	+∞	+∞
2	+∞	2	0	2	0
3	+∞	0	-2	0	-1
4	+∞	1	+∞	+∞	0

pred

	0	1	2	3	4
0	0	2	0	0	2
1	-1	1	-1	-1	-1
2	-1	2	2	2	2
3	-1	2	3	3	2
4	-1	4	-1	-1	4

In particolare abbiamo: 5 = `dist[0][1]` > `dist[0][2]` + `dist[2][1]` = 4, quindi `dist[0][1]` = 4 e `pred[0][1]` = `pred[2][1]` = 2. Considerazioni analoghe valgono per le coppie (0, 4), (3, 1) e (3, 4). Per k = 4 si ottengono le seguenti tabelle.

dist

	0	1	2	3	4
0	0	4	1	3	2
1	+∞	0	+∞	+∞	+∞
2	+∞	2	0	2	1
3	+∞	0	-2	0	-1
4	+∞	1	+∞	+∞	0

pred

	0	1	2	3	4
0	0	2	3	0	2
1	-1	1	-1	-1	-1
2	-1	2	2	2	2
3	-1	2	3	3	2
4	-1	4	-1	-1	4

Sono cambiati i valori delle coppie (0, 2) e (0, 4). Infine per k = 5 abbiamo

dist

	0	1	2	3	4
0	0	3	1	3	2
1	+∞	0	+∞	+∞	+∞
2	+∞	2	0	2	1
3	+∞	0	-2	0	-1
4	+∞	1	+∞	+∞	0

pred

	0	1	2	3	4
0	0	4	3	0	2
1	-1	1	-1	-1	-1
2	-1	2	2	2	2
3	-1	2	3	3	2
4	-1	4	-1	-1	4

In questo ultimo passaggio viene scoperto il cammino da v_0 a v_1 passante per v_4. Questo cammino ha peso 3 e per ricostruirlo utilizziamo la tabella `pred` nel seguente modo: `pred[0][1]` = 4 ovvero v_4 precede v_1 nel cammino minimo da v_0 a v_1; `pred[0][4]` = 2 quindi v_2 precede v_4; `pred[0][2]` = 3 quindi v_3 precede v_2; infine `pred[0][3]` = 0. Ricapitolando, la sequenza di nodi v_0, v_3, v_2, v_4 e v_1 costituisce il cammino minimo da v_0 a v_1.

7.5 Opus libri: data mining e minimi alberi ricoprenti

In applicazioni di **data mining** è necessario operare su insiemi di dati di grandi dimensioni, per esempio di carattere sperimentale, per individuare delle regolarità nei dati trattati o similitudini tra i loro sottoinsiemi: il tutto nel tentativo di derivare un'ipotesi di legge, un qualche tipo di regola associativa soggiacente ai dati. Tipici casi di tali applicazioni sono per esempio l'analisi di una rete sociale con l'obiettivo di individuare le comunità al suo interno, o il partizionamento di una collezione di documenti su un insieme di tematiche.

In questo contesto, uno dei metodi applicati in modo estensivo è l'analisi di cluster (*cluster analysis*), vale a dire il partizionamento di dati osservati in sottoinsiemi, detti cluster, in modo tale che i dati in ciascun cluster condividano qualche proprietà comune, non posseduta dai dati esterni al cluster.

In genere, l'individuazione dei cluster viene effettuata in termini di prossimità dei dati rispetto a una qualche metrica definita su di essi che ne misura la distanza: a tal fine, i dati possono essere proiettati, eventualmente per mezzo di una qualche funzione predefinita, su uno spazio k-dimensionale, utilizzando la normale distanza euclidea come metrica di riferimento.

Sono stati definiti numerosi metodi per l'individuazione di una partizione in cluster: la scelta del più efficace da utilizzare rispetto a un determinato insieme di dati è spesso molto ardua. La maggior parte dei metodi richiede inoltre

l'assegnazione di valori a una serie di parametri o addirittura la predeterminazione del numero di cluster da ottenere.

In tale contesto, una tecnica piuttosto semplice, diffusa ed efficace in molte situazioni, derivata dalla teoria dei grafi, è basata sull'uso di alberi minimi di ricoprimento. Il problema del **minimo albero di ricoprimento** o minimo albero ricoprente (*minimum spanning tree*) è uno dei problemi su grafi più semplici da definire e più studiati. Ricordiamo che, dato un grafo non orientato e connesso $G=(V,E)$, un albero di ricoprimento di G è un albero T i cui archi sono anche archi del grafo e collegano tutti i nodi in V (Paragrafo 7.2), ossia un albero $T=(V,E')$, dove $E' \subseteq E$.

Un tale albero è minimo se la somma dei pesi nei suoi archi è la minima tra quelle di tutti i possibili alberi di ricoprimento (notiamo come nel caso in cui G non sia connesso non esiste alcun albero che lo ricopre: possiamo però definire una **foresta di ricoprimento** di G, vale a dire un insieme di alberi, ciascuno ricoprente la propria componente connessa). Nel seguito faremo sempre l'ipotesi che G sia connesso. Nella Figura 7.17 è illustrato un grafo non orientato pesato e connesso con evidenziati gli archi di un suo minimo albero di ricoprimento.

Nel data mining basato sull'analisi dei cluster, i dati sono rappresentati da punti in uno spazio k-dimensionale. Un esempio è nell'analisi del genoma, in cui è possibile monitorare simultaneamente k parametri numerici per decine di migliaia di geni (che reagiscono a cambiamenti imposti al loro ambiente) attraverso dei dispositivi speciali chiamati **microarray**, i quali forniscono un insieme di punti k-dimensionali in uscita. In tal caso, il clustering basato sull'albero minimo di ricoprimento opera a partire dal grafo $G=(V,E,W)$ in cui V è l'insieme dei punti, E è l'insieme di tutte le possibili coppie di punti (quindi G è un grafo completo di tutti gli archi) e W è la distanza euclidea tra i punti: in tale contesto l'albero minimo di ricoprimento viene detto **euclideo**.

L'albero risultante può essere utilizzato come base per il partizionamento in cluster: come vedremo, infatti, ogni arco e dell'albero è l'arco di peso inferiore tra quelli in grado di collegare le due porzioni di albero ottenute dall'eventuale rimozione di e.

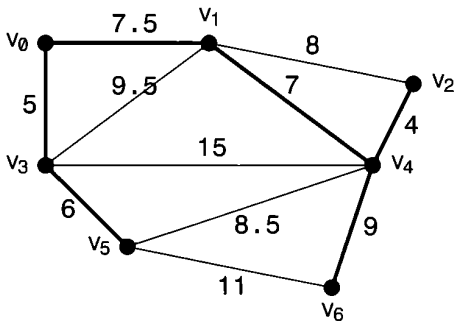


Figura 7.17 Un esempio di grafo pesato con il suo minimo albero di ricoprimento.

In termini di data mining, la rimozione di e dà luogo a una separazione tra due cluster in cui la distanza tra due qualsiasi punti, uno per cluster, non può essere inferiore al peso di e. Scegliendo di rimuovere gli archi più lunghi (di peso maggiore) presenti nell'albero minimo di ricoprimento, iniziamo a separare i cluster più distanti tra di loro: osserviamo infatti che punti appartenenti alla stessa porzione di albero formano un cluster e che cluster diversi tendono a essere collegati da archi più lunghi. La rimozione di ogni successivo arco lungo, separa ulteriormente i cluster. In generale, dopo k rimozioni di archi dall'albero, otteniamo $k+1$ cluster, tendendo così a separare insiemi di nodi "lontani" tra loro, al fine di ottenere cluster il più possibile significativi.

Comunque, non è detto che la sola eliminazione degli archi più lunghi nell'albero fornisca una partizione in cluster accettabile: per esempio, alcuni cluster ottenuti possono essere eccessivamente piccoli per le finalità dell'analisi dei dati effettuata. Spesso, può essere necessario aggiungere ulteriori criteri di scelta degli archi da eliminare, in modo da migliorare la significatività della partizione ottenuta.

7.5.1 Problema della ricerca del minimo albero di ricoprimento

Come già visto nel Paragrafo 7.2, un qualunque albero di ricoprimento di un grafo può essere trovato in tempo $O(n+m)$ mediante una visita (sia in ampiezza che in profondità) del grafo stesso: gli alberi BFS e DFS risultanti da tali visite non sono altro che particolari alberi di ricoprimento.

Consideriamo ora, come nel paragrafo precedente, il caso in cui l'insieme degli archi sia pesato per mezzo di una funzione $W: E \mapsto \mathbb{R}^+$ e che il grafo sia connesso e non orientato. In questa ipotesi ogni albero ricoprente $T=(V,E')$ di G ha associato un peso $\sum_{e \in E'} W(e)$ pari cioè alla somma dei pesi dei suoi archi. Quel che vogliamo è allora trovare, fra tutti gli alberi ricoprenti di G, uno avente peso minimo (laddove gli alberi BFS e DFS non sono necessariamente di peso minimo). Osserviamo che possiamo assumere, senza perdita di generalità, che i pesi degli archi del grafo $G=(V,E,W)$ siano tutti distinti. Infatti se così non fosse possiamo sempre imporre un ordinamento stretto degli archi anche quando questi hanno lo stesso peso: è sufficiente infatti definire una qualsiasi numerazione τ degli archi e nel caso in cui $W(e)=W(e')$ assumere per convenzione che e precede e' se e solo se $\tau(e)<\tau(e')$.

Introduciamo quindi la seguente utile nozione. Dato un grafo $G=(V,E)$, un **taglio** (*cut*) su G è un qualunque sottoinsieme $C \subseteq E$ di archi la cui rimozione disconnette il grafo, nel senso che il grafo $G'=(V,E-C)$ è tale che esistono almeno due nodi $u, v \in V$ tra i quali non esiste un cammino.

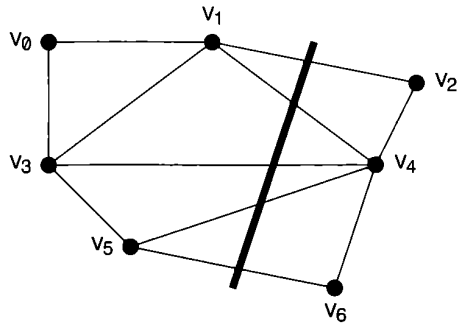


Figura 7.18 Esempio di taglio in un grafo.

ESEMPIO 7.10

Nella Figura 7.18 viene mostrato un taglio (riportato graficamente come una linea più spessa) che corrisponde all'insieme di archi $C = \{(v_1, v_2), (v_1, v_4), (v_3, v_4), (v_5, v_4)$ e $(v_5, v_6)\}$. Come possiamo verificare, la rimozione degli archi nel taglio disconnette il grafo in due componenti disgiunte $\{v_0, v_1, v_3, v_5\}$ e $\{v_2, v_4, v_6\}$.

Introduciamo ora due proprietà strutturali del minimo albero ricoprente, che saranno utilizzate per mostrare la correttezza degli algoritmi che considereremo nel seguito: queste proprietà fanno riferimento al concetto di ciclo, già introdotto nel Paragrafo 7.1, e alla nozione di taglio appena vista.

Teorema 7.9 *Dato un grafo* $G = (V, E, W)$ *pesato sugli archi con pesi tutti distinti e un suo minimo albero ricoprente* $T = (V, E')$, *per ogni arco* $e \in E$ *abbiamo che:*

Condizione di taglio $e \in E'$ *se e solo se esiste un taglio in* G *che comprende* e, *per il quale* e *è l'arco di peso minimo.*

Condizione di ciclo $e \notin E'$ *se e solo se esiste un ciclo in* G *che comprende* e, *per il quale* e *è l'arco di peso massimo.*

Dimostrazione Per dimostrare la prima proprietà, osserviamo che se $e \in E'$ allora la sua rimozione disconnette i nodi su T in due componenti V', V'' (ricordiamo che la rimozione di un qualunque arco di un albero disconnette l'albero stesso). L'insieme C che comprende sia e che tutti gli archi $(v', v'') \in E - E'$ tali che $v' \in V'$ e $v'' \in V''$ è un taglio in G. Per la minimalità dell'albero T, per qualunque arco $e' \neq e$ in C vale che $W(e') > W(e)$: se così non fosse, infatti, l'insieme di archi $E' - \{e\} \cup \{e'\}$ indurrebbe un diverso albero ricoprente di peso $W(T) - W(e) + W(e') < W(T)$. Al tempo stesso, con la stessa motivazione, dato un qualunque taglio in G, l'arco di peso minimo in tale taglio deve essere incluso in T.

Per quanto riguarda la seconda proprietà, per ogni arco $e \notin E'$, l'insieme $E' \cup \{e\}$ induce un ciclo che include tale arco: se in tale ciclo esistesse un arco $e' \in E'$ tale che $W(e') > W(e)$ allora l'insieme di archi $E' - \{e'\} \cup \{e\}$ indurrebbe un diverso albero ricoprente di peso $W(T) - W(e') + W(e) < W(T)$, pervenendo a una contraddizione. Al tempo stesso, se e è l'arco di peso massimo in un ciclo, non può appartenere a E': se così fosse, potremmo sostituirlo con uno del ciclo meno pesante, ottenendo per assurdo un albero di ricoprimento di costo inferiore al minimo. ❑

Nel seguito introduciamo due algoritmi classici risalenti a metà degli anni '50, l'algoritmo di Kruskal e quello di Jarník-Prim, per la ricerca del minimo albero ricoprente di un grafo. Prima di esaminarli, preannunciamo però che, in effetti, essi sono due varianti di un medesimo approccio generale alla risoluzione del problema.

In questo approccio, un algoritmo opera in modo goloso, inizializzando E' all'insieme vuoto, e aggiungendo poi archi a tale insieme finché il grafo $T = (V, E')$ resta non connesso. Un arco viene aggiunto a E' se è quello più leggero uscente da una qualche componente connessa di T, vale a dire se è l'arco più leggero che collega un nodo della componente a un nodo non appartenente a essa. Per quanto detto sopra, quindi, un arco è incluso nel minimo albero ricoprente se è più leggero di un qualunque taglio che separa la componente dal resto del grafo.

L'effetto di tutto ciò è che, a ogni passo, gli archi in E' formano un sottoinsieme (una foresta, in effetti) del minimo albero di ricoprimento. Dato che l'algoritmo termina quando tutti gli archi sono stati esaminati, ne deriva che per ogni taglio esiste almeno un arco (il più leggero, in particolare) che è stato inserito in E' e quindi il grafo $T = (V, E')$ è connesso. Inoltre, dato che per ogni arco inserito in E' due componenti connesse disgiunte di T vengono riunite e che, a partire da n componenti disgiunte (i singoli nodi) per giungere ad avere un singola componente di n nodi bisogna effettuare $n - 1$ di tali operazioni di riunione, ne deriva che il numero di archi in E' al termine dell'algoritmo è pari a $n - 1$ e che, quindi, T è un albero (essendo connesso e aciclico per costruzione).

7.5.2 Algoritmo di Kruskal

L'algoritmo di Kruskal per la ricerca del minimo albero di ricoprimento opera considerando gli archi l'uno dopo l'altro, in ordine crescente di peso, valutando se inserire ogni arco nell'insieme E' degli archi dell'albero. Nel considerare l'arco (u, v), possiamo avere due possibilità:

1. i due nodi u e v sono già collegati in $G = (V, E')$, e quindi l'arco (u, v) chiude un ciclo: in tal caso esso è l'arco più pesante nel ciclo, e quindi non appartiene al minimo albero di ricoprimento;

2. i due nodi u e v non sono già collegati in G = (V, E'), e quindi esiste almeno un taglio che separa u da v: (u, v) è il primo arco considerato tra quelli del taglio, quindi è il più leggero e di conseguenza deve essere inserito in E'.

L'algoritmo di Kruskal opera a partire da una situazione in cui esistono n componenti connesse distinte (gli n nodi isolati), ognuna con un proprio minimo albero di ricoprimento (l'insieme vuoto degli archi). L'algoritmo unisce man mano coppie di componenti disgiunte, mantenendo al tempo stesso traccia del minimo albero di ricoprimento della componente risultante. Al termine dell'esecuzione, tutte le componenti sono state riunificate in una sola, il cui minimo albero ricoprente è quindi il minimo albero ricoprente dell'intero grafo.

Il Codice 7.12 presenta un'implementazione dettagliata dell'algoritmo delineato sopra; nel codice vengono utilizzate diverse strutture di dati.

1. Una coda con priorità PQ contenente l'insieme degli archi del grafo e i loro pesi.
2. Una struttura di dati che rappresenta una partizione dell'insieme dei nodi in modo tale da consentire di verificare se due nodi appartengono allo stesso sottoinsieme e da effettuare l'unione dei sottoinsiemi di appartenenza di due elementi: a tal fine, utilizziamo un insieme di liste, così come illustrato nel Paragrafo 5.3.
3. Un array set che associa a ogni nodo del grafo un riferimento al corrispondente elemento nella struttura di dati precedente.
4. Una lista doppia mst (come definita nel Paragrafo 4.2) utilizzata per memorizzare gli archi nel minimo albero ricoprente, quando sono individuati dall'algoritmo.

Codice 7.12 Algoritmo di Kruskal per la ricerca del minimo albero di ricoprimento.

```
1  Kruskal( ):
2    FOR (u = 0; u < n; u = u + 1) {
3      FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4        v = x.dato;
5        elemento.dato = <u, v>;
6        elemento.peso = x.peso;
7        PQ.Enqueue( elemento );
8      }
9      set[u] = NuovoNodo( );
10     Crea( set[u] );
11   }
```



```
12   WHILE (!PQ.Empty( )) {
13     elemento = PQ.Dequeue( );
14     <u,v> = elemento.dato;
15     IF (!Appartieni( set[u], set[v] )) {
16       Unisci( set[u], set[v] );
17       mst.InserisciFondo( <u,v> );
18     }
19   }
```

Come possiamo vedere, l'algoritmo utilizza la coda con priorità per ottenere un ordinamento degli archi crescente rispetto al loro peso. In particolare, viene inizialmente creata una coda con priorità contenente tutti gli archi, oltre a una rappresentazione di componenti composte da liste disgiunte, ciascuna inizialmente contenente un solo nodo u (righe 2-11).

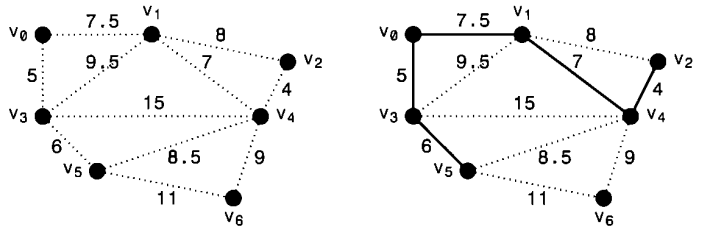
Gli archi vengono poi considerati uno dopo l'altro, in ordine crescente di peso (12-19): per ogni arco, ci chiediamo se esso collega due nodi posti in componenti diverse o, equivalentemente, se chiude un ciclo. Se ciò non avviene e, quindi, l'arco ha peso minimo per un qualche taglio che divide le due componenti, tali componenti sono unificate e l'arco viene inserito nel minimo albero di ricoprimento (righe 15-18).

Notiamo che, essendo il grafo non orientato, ogni arco viene inserito due volte nella coda con priorità senza pregiudicare l'esito dell'algoritmo quando viene estratto due volte (la prima estrazione soltanto può sortire un effetto in quanto la seconda non supera la condizione nella riga 15).

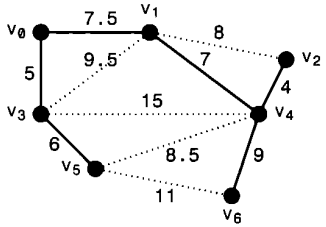
Per valutare il tempo di esecuzione dell'algoritmo di Kruskal, possiamo vedere che, su un grafo di n nodi e m archi, esso effettua al più m operazioni Enqueue, Empty e Dequeue, oltre a n operazioni Crea, Unisci e m operazioni Appartieni: il costo dell'algoritmo dipende quindi dai costi di esecuzione di tali operazioni sulle strutture di dati utilizzate. Se per esempio utilizziamo uno heap come implementazione della coda con priorità e l'insieme di liste del Paragrafo 5.3 per rappresentare partizioni di nodi, abbiamo che Enqueue e Dequeue richiedono tempo $O(\log n)$, Empty, Appartieni e Crea in tempo $O(1)$, e Unisci in tempo $O(\log n)$ ammortizzato. Da ciò deriva che il costo complessivo dell'algoritmo in tal caso è $O(m \log m + n \log n) = O((m+n)\log n)$, e quindi $O(m \log n)$ se il grafo è connesso, per cui abbiamo $m \geq n - 1$.

ESEMPIO 7.11

Mostriamo come l'algoritmo di Kruskal agisce sul grafo della Figura 7.17. Inizia con tutti nodi disgiunti, poi considera gli archi di G nell'ordine (v_2, v_4), (v_0, v_3), (v_3, v_5), (v_1, v_4), (v_0, v_1), (v_1, v_2), (v_4, v_5), (v_4, v_6), (v_1, v_3), (v_5, v_6) e (v_3, v_4). L'inserimento dei primi 5 archi non crea cicli e la soluzione parziale è mostrata nella figura a destra.



Quindi prende in considerazione gli archi (v_1, v_2) e (v_4, v_5): ognuno di questi chiude un ciclo e quindi vengono scartati. L'arco successivo, (v_4, v_6), viene aggiunto alla soluzione.



Tutti gli archi rimanenti chiudono cicli, quindi questa è la soluzione finale.

7.5.3 Algoritmo di Jarník-Prim

Come abbiamo visto, l'algoritmo di Kruskal costruisce un minimo albero ricoprente facendo crescere un insieme di minimi alberi ricoprenti relativi a sottoinsiemi dei nodi: gli alberi sono man mano unificati fino a ottenere quello relativo all'intero grafo.

L'algoritmo di Jarník-Prim opera in modo più "centralizzato": esso parte da un qualunque nodo s e fa crescere un minimo albero ricoprente a partire da tale nodo, aggiungendo man mano nuovi nodi e archi all'albero stesso: se T indica la porzione di minimo albero ricoprente attualmente costruita, l'algoritmo sceglie l'arco (u, v) tale che esso è di peso minimo nel taglio tra T e V – T, aggiungendo v a T e (u, v) all'insieme E′, fino a coprire tutti i nodi del grafo.

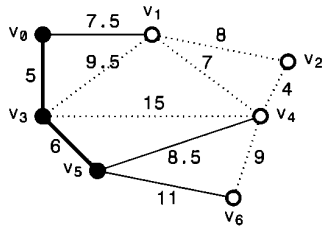
Non è difficile rendersi conto che l'insieme E′ ottenuto al termine dell'algoritmo è l'insieme degli archi nel minimo albero ricoprente. Infatti, ogni arco aggiunto a E′ è il più leggero nel taglio che separa T da V – T e quindi, per quanto detto sopra, deve far parte del minimo albero ricoprente del grafo. Dato che, inoltre, al termine dell'algoritmo abbiamo che $|E'| = n - 1$, tutti gli archi dell'albero compaiono in E′.

Come implementare in modo efficiente l'algoritmo presentato sopra? Il punto critico di un'implementazione è rappresentato dal come realizzare efficientemente la selezione dell'arco di peso minimo tra T a V – T. La soluzione banale, consistente nell'effettuare tale selezione scandendo ogni volta tutti gli m archi porterebbe a ottenere un tempo di esecuzione O(nm), peggiore quindi di quello ottenuto dall'algoritmo di Kruskal.

Una soluzione più efficiente è fornita dall'utilizzo di una coda con priorità PQ all'interno della quale mantenere, a ogni istante, l'insieme dei nodi in V – T, utilizzando come peso di ogni nodo v ∈ V – T il peso dell'arco più leggero che collega v a un qualche nodo in T.

ESEMPIO 7.12

Consideriamo il grafo rappresentato nella Figura 7.17. I primi due passi dell'algoritmo di Jarník-Prim conducono alla situazione rappresentata nella figura.

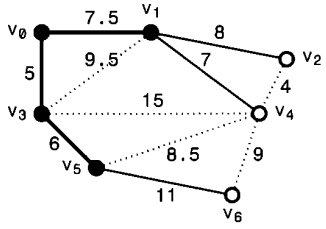


I nodi pieni sono quelli in T e gli altri sono in V – T. Gli archi più spessi sono quelli del minimo albero di ricoprimento mentre quelli più sottili rappresentano, per ogni nodo v in V – T, l'arco più leggero che lo collega a qualche nodo in T. Il peso di questo arco è il peso che ha v nella coda con priorità PQ. Quindi essendo $V - T = \{v_1, v_2, v_4, v_6\}$, i pesi associati a ognuno di tali nodi saranno, rispettivamente, 7.5, +∞, 8.5 e 11, dove il peso +∞ per il nodo v_2 sta a rappresentare il fatto che tale nodo non è adiacente a nessun nodo di T.

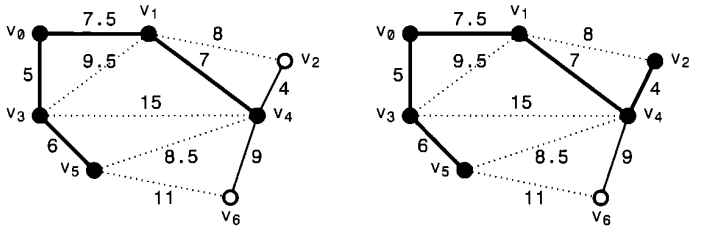
Utilizzando una coda con priorità di questo tipo, la selezione dell'arco viene effettuata mediante una Dequeue, ma, al tempo stesso, è necessario prevedere l'aggiornamento, quando necessario, dei pesi associati ai nodi in PQ. Tale aggiornamento può derivare dal passaggio di un nodo v da V – T a T, passaggio che fa sì che modifichiamo l'insieme degli archi tra T e V – T considerati per la selezione.

ESEMPIO 7.13

Prendiamo in considerazione la situazione lasciata nell'Esempio 7.12. Il passo successivo dell'algoritmo di Jarník-Prim prevede il passaggio di v_1 da V – T a T. Prendiamo in considerazione i nodi adiacenti a v_1: il nodo v_2 ora risulta adiacente a un nodo in T, quindi il suo peso in PQ passa da +∞ a 8 (il peso dell'arco (v_1, v_2)); il peso del nodo v_4 in PQ, prima uguale a $W(v_4, v_5) = 8.5$, con la presenza di v_1 in T, viene decrementato a $W(v_1, v_5) = 7$.



Nella figura in alto è rappresentata la situazione corrente. Nei due passi successivi viene inserito in T prima il nodo v_4 (figura in basso a sinistra) e poi il nodo v_2 creando la situazione descritta nella figura in basso a destra.



Nell'ultimo passo viene aggiunto a T il nodo v_6 con l'arco (v_4, v_6) creando la soluzione finale.

Da quanto osservato, possiamo concludere che l'utilizzo di PQ richiede che, in corrispondenza a ogni passaggio di un nodo v da T a V – T, vengano esaminati tutti gli archi incidenti a v. Per ogni arco (u, v) di questo tipo, se u ∈ V – T allora il peso di u in PQ viene posto pari a W(u, v), quando tale valore è minore del peso attuale di u. In tal modo, se esiste ora un modo "più economico" per collegare un nodo in T a V – T, ciò viene riportato nella coda con priorità. Quindi, le operazioni che vanno effettuate su PQ sono Enqueue per costruire la coda, Dequeue per estrarre man mano i nodi da inserire in T e DecreaseKey per aggiornare il peso dei nodi ancora in T (notiamo che l'aggiornamento può essere soltanto un decremento). Il Codice 7.13 realizza tale strategia impiegando un array booleano incluso per marcare i vertici in T, mentre quelli in V – T sono nella coda con priorità PQ (l'inizializzazione è nelle righe 2-8). Nel ciclo principale (righe 9-22), l'algoritmo estrae il nodo v da includere in T e decrementa il peso dei suoi vertici adiacenti u in V – T, quando questi hanno peso superiore a quello dell'arco (v, u) che li collega a v. L'algoritmo esegue in particolare n operazioni Enqueue e Dequeue, e al più 2m operazioni DecreaseKey. Il suo costo, come nei casi precedenti degli algoritmi di Dijkstra e di Kruskal, dipende dal costo di tali operazioni sull'implementazione della coda con priorità adottata. Per esempio, utilizzando uno heap abbiamo, come già visto, che le tre operazioni in questione hanno ciascuna costo O(log n), per cui il costo complessivo dell'algoritmo è O((n + m) log n), e quindi O(m log n) in quanto consideriamo il grafo connesso: l'utilizzo di uno heap di grado d > 2, così come per l'algoritmo di Dijkstra, fornisce un costo $O(m \log_{m/n} n)$. Infine, l'utilizzo di heap di Fibonacci (che, ricordiamo, hanno costi ammortizzati per le operazioni Enqueue, Dequeue e DecreaseKey pari a O(1), O(log n) e O(1), rispettivamente) fa sì che il costo complessivo dell'algoritmo sia O(n log n + m), migliore quindi di quanto ottenuto per mezzo dall'algoritmo di Kruskal.

Codice 7.13 Algoritmo di Jarník-Prim per la ricerca del minimo albero di ricoprimento.

```
1  Jarník-Prim( ):
2    FOR (u = 0; u < n; u = u + 1) {
3      incluso[u] = FALSE;
4      pred[u] = u;
5      elemento.peso = peso[u] = +∞;
6      elemento.dato = u;
7      PQ.Enqueue( elemento );
8    }
9    WHILE (!PQ.Empty( )) {
10     elemento = PQ.Dequeue( );
11     v = elemento.dato;
12     incluso[v] = TRUE;
13     mst.InserisciFondo( <pred[v], v> );
14     FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
15       u = x.dato;
16       IF (!incluso[u] && x.peso < peso[u]) {
17         pred[u] = v;
18         peso[u] = x.peso;
19         PQ.DecreaseKey( u, peso[u] );
20       }
21     }
22   }
```

7.6 Esercizi

7.1 Progettare un algoritmo per costruire il ciclo euleriano di un grafo non orientato.

7.2 Discutere gli algoritmi per inserire o cancellare un nodo o un arco in un grafo in relazione alle due rappresentazioni dei grafi discusse (nel Paragrafo 7.1.2).

7.3 Un grafo a torneo è un grafo orientato G in cui per ogni coppia di vertici x e y esiste un solo arco che li collega, (x, y) oppure (y, x), ma non entrambi. L'interpretazione è che nella partita del torneo tra x e y uno dei due ha vinto. Mostrare che un grafo a torneo ammette sempre un cammino hamiltoniano.

7.4 Descrivere un algoritmo lineare per stabilire se un grafo è bipartito, tentando di usare il colore opposto del vertice corrente quando scoprite un nuovo vertice.

7.5 Progettare un algoritmo che restituisca un ciclo di un grafo orientato ciclico.

7.6 Usare la visita DFS per mostrare che un grafo non orientato con grado minimo d ammette un cammino semplice di lunghezza maggiore di d.

7.7 Sia X un insieme di n variabili intere e D un insieme di disequazioni tutte della forma x > y dove x e y sono variabili in X. Progettare un algoritmo di complessità lineare in n e m che decida se D ammette una soluzione, utilizzando un'opportuna rappresentazione di D come grafo. In caso affermativo l'algoritmo deve produrre una soluzione.

7.8 Prendere il DAG risultante dalle componenti fortemente connesse (che sono i macro-vertici) di un grafo a torneo (Esercizio 7.3) e mostrare che l'ordinamento topologico del DAG induce una classifica dei partecipanti al torneo.

7.9 Sia G' il grafo ottenuto dal grafo orientato G invertendo il verso degli archi. Dimostrare che G è fortemente connesso se e solo se un qualsiasi algoritmo di visita applicato a G e a G' a partire dallo stesso nodo raggiunge tutti i vertici.

7.10 Nella trattazione dell'algoritmo di Dijkstra abbiamo implicitamente supposto che tra ogni coppia di nodi esista un solo cammino minimo. Dimostrare che la correttezza e l'efficienza dell'algoritmo valgono anche nel caso generale in cui possono esistere più cammini minimi tra due nodi. Mostrare anche che i cammini selezionati dall'algoritmo tramite l'array pred formano un albero dei cammini minimi con radice nel nodo di partenza s.

7.11 Mostrare come utilizzare l'algoritmo di Dijkstra per determinare, dato un nodo v, il ciclo semplice di lunghezza minima che include tale nodo. Implementare quindi un algoritmo per ottenere il **girovita** (*girth*) di un grafo, vale a dire la lunghezza del ciclo semplice di lunghezza minima nel grafo stesso.

7.12 Modificare il codice dell'algoritmo di Bellman-Ford per verificare la presenza di cicli negativi, così come illustrato nel testo.

7.13 Implementare un semplice algoritmo di clustering basato sul minimo albero di ricoprimento. L'algoritmo, dato un insieme S di punti in uno spazio a d = 3 dimensioni, suddivide tale insieme in k cluster, dove k è un parametro passato all'algoritmo stesso.

7.14 Fornire un controesempio per mostrare che gli algoritmi di Kruskal e Jarník-Prim non calcolano sempre correttamente l'albero minimo di ricoprimento per un grafo *orientato*: in tal caso, gli archi dell'albero devono seguire la direzione padre-figlio.

7.15 Dimostrare che il minimo albero di ricoprimento è unico se il grafo ha pesi distinti.

7.16 Dimostrare che l'arco di costo massimo di un minimo albero di ricoprimento è minimo rispetto agli archi di costo massimo degli altri alberi ricoprenti dello stesso grafo.

7.17 Progettare un algoritmo di complessità lineare per il problema dello *single-source shortest path* nel caso in cui i pesi degli archi ricadono nell'insieme {1, 2, 3}.

7.18 Siano W e W' due funzioni peso sugli archi di un grafo G non orientato tali che $W(e_1) \leq W(e_2)$ implica $W'(e_1) < W'(e_2)$. Dimostrare che T* è un minimo albero di ricoprimento per G rispetto alla funzione W se e solo lo è per il grafo G rispetto a W'.

7.19 Sia $P = \{p_1, ..., p_n\}$ un insieme di n punti del piano cartesiano, dato un numero non negativo r, definiamo il grafo $G_r = (P, E_r)$ dove E_r contiene coppie di punti la cui distanza (Euclidea) è al più r. Progettare un algoritmo che calcoli il minimo valore di r per il quale G_r è connesso.

7.20 Progettare un algoritmo di complessità lineare che decida se un dato arco di un grafo non orientato, pesato e connesso appartiene a un qualche minimo albero di ricoprimento.

CAPITOLO

8 NP-completezza e approssimazione

In questo capitolo introduciamo il concetto di riduzione polinomiale e quello di problema NP-completo, dimostrando la NP-completezza di alcuni problemi computazionali e fornendo alcuni suggerimenti su come dimostrare un nuovo risultato di NP-completezza. Infine, mostriamo come affrontare la difficoltà computazionale intrinseca di un problema facendo uso di algoritmi polinomiali di approssimazione.

8.1 Problemi intrattabili
8.2 Classi P e NP
8.3 Riducibilità polinomiale
8.4 Problemi NP-completi
8.5 Teorema di Cook-Levin
8.6 Problemi di ottimizzazione
8.7 Generazione esaustiva e backtrack
8.8 Esempi e tecniche di NP-completezza
8.9 Come dimostrare risultati di NP-completezza
8.10 Algoritmi di approssimazione
8.11 Opus libri: il problema del commesso viaggiatore
8.12 Esercizi

8.1 Problemi intrattabili

Il concetto di problema NP-completo consente di evidenziare la difficoltà di decine di migliaia di problemi computazionali interessanti che, purtroppo, non riusciamo a risolvere in tempo polinomiale: conosciamo solo algoritmi esponenziali o pseudo-polinomiali di risoluzione, ma non c'è ancora evidenza in termini di limiti inferiori che ciò sia necessario, creando di fatto un limbo computazionale per la loro complessità in tempo che potrebbe variare dal polinomiale all'esponenziale. Da un lato, il nostro obiettivo è chiaramente quello di progettare un algoritmo di risoluzione efficiente per il problema in esame; dall'altro, cerchiamo di scoprire i limiti computazionali che emergono nella sua risoluzione. Non riusciamo ad applicare questo approccio a tali problemi ma, avendo a disposizione la nozione di problema NP-completo, essi risultano essere *intrinsecamente difficili* e uniti da un destino comune: (a) l'esistenza di un algoritmo polinomiale per uno qualunque dei problemi NP-completi avrebbe conseguenze molto significative, perché tutti gli altri problemi NP-completi risulterebbero immediatamente risolvibili in tempo polinomiale per transitività; (b) se invece fosse possibile dimostrare un limite inferiore esponenziale per uno di essi, allora varrebbe per tutti gli altri, sempre per transitività.

Un risultato di NP-completezza ha l'indubbio vantaggio di far rivolgere i nostri sforzi verso obiettivi meno ambiziosi di ottenere un algoritmo polinomiale di risoluzione: come vedremo al termine di questo capitolo, esistono diversi modi per affrontare la difficoltà di un problema NP-completo, tra cui uno dei più esplorati consiste nella progettazione di algoritmi di approssimazione, ovvero algoritmi efficienti le cui prestazioni, in termini di qualità della soluzione calcolata (non necessariamente ottima), siano in qualche modo garantite.

8.2 Classi P e NP

Restringiamo la nostra attenzione (per ora) ai soli problemi di decisione, ossia ai problemi la cui soluzione è una risposta binaria – sì o no. Utilizzando i meccanismi di codifica binaria delle istanze di un problema decisionale, dove la dimensione di un'istanza di ingresso è il numero di bit utilizzati per rappresentarla, identifichiamo un problema decisionale con il corrispettivo insieme (potenzialmente infinito) di istanze la cui risposta è "sì": risolvere tale problema consiste quindi nel decidere l'appartenenza di una sequenza binaria all'insieme (osserviamo che non è riduttivo restringersi alle sole sequenze binarie, in quanto il calcolatore stesso opera solo su tale tipo di sequenze). Pertanto, un **problema di decisione** Π non è altro che un sottoinsieme dell'insieme di tutte le possibili sequenze binarie, in particolare di quelle che soddisfano una determinata proprietà.

ESEMPIO 8.1

Il problema di decidere se un dato numero intero è primo consiste di tutte le sequenze binarie che codificano numeri interi primi, per cui la sequenza `1101` (13 in decimale) appartiene a tale problema mentre la stringa `1100` (12 in decimale) non vi appartiene. Il problema di decidere se un grafo può essere colorato con tre colori consiste di tutte le sequenze binarie che codificano grafi che possono essere colorati con tre colori.

Nel seguito, per motivi di chiarezza, continueremo a definire un problema decisionale facendo uso di descrizioni formulate in linguaggio naturale, anche se implicitamente intenderemo definirlo come uno specifico insieme di sequenze binarie, facendo riferimento a opportuni meccanismi di codifica: per esempio, operando su un grafo, la sua codifica potrebbe essere ottenuta memorizzando per righe la corrispondente matrice di adiacenza.

Un problema decisionale Π appartiene alla **classe P** se esiste un algoritmo polinomiale `A` che, presa in ingresso una sequenza binaria `x`, determina se `x` appartiene a Π o meno.

ESEMPIO 8.2

Sappiamo che il problema di decidere se, dato un grafo (orientato) `G` e due suoi nodi `s` e `t`, esiste un cammino semplice da `s` a `t` appartiene a P, in quanto abbiamo visto nel Capitolo 7 come visitare tutti i nodi raggiungibili da `s` operando una visita in ampiezza del grafo: se `t` è incluso tra questi vertici, allora la risposta al problema è affermativa, altrimenti è negativa.

Non sappiamo se il problema di decidere se un grafo può essere colorato con tre colori appartiene a P, ma possiamo mostrare che lo stesso problema ristretto a due colori vi appartiene (Esercizio 7.2): a tale scopo, mostriamo come utilizzare il fatto che se un vertice è colorato con il primo colore, allora tutti i suoi vicini devono essere colorati con il secondo colore. L'idea dell'algoritmo consiste nel colorare un vertice `i` con uno dei due colori e dedurre tutte le colorazioni degli altri vertici che ne conseguono visitando in ampiezza il grafo a partire da `i`: se riusciamo a colorare tutti i vertici, possiamo concludere che il grafo è colorabile con due colori. Altrimenti, se arriviamo a una contraddizione (ovvero, siamo costretti a colorare un vertice con il colore diverso da quello già precedentemente assegnato), possiamo dedurre che il grafo non è colorabile con due colori (in realtà, questo algoritmo deve essere applicato a ogni componente connessa del grafo).

Codice 8.1 Algoritmo per decidere se un grafo connesso è colorabile con due colori.

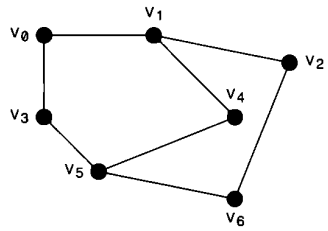
```
1  DueColorazione( ):
2    FOR (i = 0; i < n; i = i+1)
3      colore[i] = -1;
4    FOR (i = 0; i < n; i = i+1) {
5      IF (colore[i] == -1 && !Colora(i)) RETURN FALSE;
6    }
7    RETURN TRUE;
```

```
1  Colora( s ):
2    colore[s] = 0;
3    FOR (x = listaAdiacenza[s].inizio; x != null; x = x.succ)
4      Q.Enqueue( (s, x.dato) );
5    WHILE (!Q.Empty( )) {
6      (u', u) = Q.Dequeue( );
7      IF (colore[u] == -1) {
8        colore[u] = 1 - colore[u'];
9        FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ)
10         Q.Enqueue( (u, x.dato) );
11     } ELSE IF (colore[u] == colore[u']) {
12       RETURN FALSE;
13     }
14   }
15   RETURN TRUE;
```

Il Codice 8.1 realizza l'algoritmo `DueColorazione` con una visita in ampiezza: l'array `colore` ha lo scopo di memorizzare la soluzione, in particolare `colore[i]` vale `-1` se al nodo `i` non è stato assegnato alcun colore e vale `0` o `1` se il nodo è stato visitato e quindi colorato con `0` o `1` rispettivamente. Dopo aver inizializzato `colore` a `-1` per tutti i vertici (righe 2 e 3), il codice esegue una visita a partire dal primo nodo non ancora colorato (riga 5). La funzione `Colora` è una visita in ampiezza a partire da una coda vuota e un vertice `s`, a cui viene assegnato colore `0` (riga 2): a ogni nodo visitato $u \neq s$ viene assegnato un colore che è il colore complementare a quello assegnato a suo padre u' nell'albero BFS. Infatti, se dalla coda si estrae la coppia (u', u), sappiamo che il colore di u' è già stato assegnato: se u non è stato ancora visitato, necessariamente u non può avere il colore di u' (riga 8); se invece u risulta già visitato e il colore a esso assegnato è lo stesso di u' allora siamo giunti a una contraddizione e pertanto il grafo non è colorabile con due colori (righe 11-12). Al termine del ciclo `while`, se non troviamo una contraddizione, concludiamo che la componente connessa di `s` è colorabile con due colori in quanto tutti i nodi sono stati colorati (riga 15). La complessità dell'algoritmo è quello della visita in ampiezza, ovvero tempo $O(n + m)$.

ESEMPIO 8.3

Cerchiamo di colorare con due colori il grafo della figura utilizzando l'algoritmo nel Codice 8.1 partendo dal nodo v_0.



Dopo la visita dei nodi v_0, v_1 e v_3 la coda contiene i vicini di v_1 e quelli di v_3 nel seguente ordine.

$$(v_1,v_0), (v_1,v_2), (v_1,v_4), (v_3,v_0), (v_3,v_5).$$

Inoltre i nodi v_1 e v_3 hanno colore 1 e v_0 colore 0. La coppia successiva è (v_1, v_0); il nodo v_0 è già colorato e ha un colore diverso da v_1 quindi l'algoritmo può proseguire. Vengono visitati nell'ordine i nodi v_2, v_4 e v_5 ai quali viene assegnato il colore 0: dopo la visita di v_5 la coda contiene gli archi uscenti da questi ultimi nodi.

$$(v_2,v_1), (v_2,v_6), (v_4,v_1), (v_4,v_5), (v_5,v_3), (v_5,v_4), (v_5,v_6).$$

L'arco (v_2, v_1) viene estratto dalla coda senza conseguenze in quanto v_1 è già colorato con 1; il che è compatibile col colore di v_2 che è 0. L'arco (v_2, v_6) induce la colorazione di v_6 di 1 e l'arco (v_4, v_1) viene scartato. Ora tocca all'arco (v_4, v_5): v_5 risulta già colorato, ma dello stesso colore di v_4, quindi concludiamo che il grafo non è colorabile con due colori.

Migliaia di problemi decisionali appartengono alla classe P e in questo libro ne abbiamo incontrati diversi. Da questo punto di vista, uno dei risultati recenti più interessanti è stato ottenuto da un gruppo di ricercatori indiani e consiste nella dimostrazione che appartiene a P il problema di decidere se un dato numero intero è primo: tale problema aveva resistito all'attacco di centinaia di valenti ricercatori matematici e informatici, senza che nessuno fosse stato in grado di progettare un algoritmo di risoluzione polinomiale o di dimostrare che un tale algoritmo non poteva esistere. La classe P, tuttavia, non esaurisce l'intera gamma dei problemi decisionali: esistono molti problemi per i quali non conosciamo un algoritmo di risoluzione polinomiale e ve ne sono alcuni per i quali siamo sicuri che tale algoritmo non esiste.[1]

[1] Per chi conosce il gioco delle Torri di Hanoi, è un esempio di problema provatamente esponenziale: dati tre pioli e n dischetti sul primo di essi, occorre spostare tutti i dischetti dal primo al terzo piolo uno alla volta, evitando che un dischetto di diametro inferiore stia sotto a uno di diametro maggiore.

Introduciamo ora la classe NP che include, oltre a tutti i problemi in P, molti altri problemi computazionali.[2] Intuitivamente, un problema Π in NP non necessariamente ammette un algoritmo di risoluzione polinomiale, ma è tale che se una sequenza x appartiene a Π allora deve esistere una *dimostrazione* breve di questo fatto, la quale può essere *verificata* in tempo polinomiale. Formalmente, la **classe NP** include tutti i problemi di decisione Π per i quali esiste un algoritmo polinomiale V e un polinomio p che, per ogni sequenza binaria x, soddisfano le seguenti due condizioni:

completezza: se x appartiene a Π, allora esiste una sequenza y di lunghezza p(|x|) (detta **certificato polinomiale**) tale che V con x e y in ingresso termina restituendo il valore TRUE;

consistenza: se x non appartiene a Π, allora, per ogni sequenza y, V con x e y in ingresso termina restituendo il valore FALSE.

Teorema 8.1 *La classe P è contenuta in NP.*

Dimostrazione Dato un problema Π in P, sia A un algoritmo di risoluzione polinomiale per Π. Possiamo allora definire V nel modo seguente: per ogni x e y, l'algoritmo V con x e y in ingresso restituisce il valore TRUE se A con x in ingresso risponde in modo affermativo, altrimenti restituisce il valore FALSE. Chiaramente, V (assieme a un qualunque polinomio e senza aver bisogno di usare y) soddisfa le condizioni di completezza e consistenza sopra descritte: quindi Π appartiene a NP. ❑

Non sappiamo invece se NP è contenuta in P e questa non è cosa di poco conto vista l'importanza della classe NP, data l'enorme quantità di problemi in essa contenuti. La congettura più accreditata è che P sia diversa da NP. Non avendo una dimostrazione di quest'affermazione, possiamo solamente individuare all'interno della classe NP i problemi decisionali che maggiormente si prestano a fungere da problemi "separatori" delle due classi: tali problemi sono i problemi NP-completi, che costituiscono i problemi "più difficili" all'interno della classe NP.

8.3 Riducibilità polinomiale

Per poter definire il concetto di problema NP-completo, abbiamo bisogno della nozione di riducibilità polinomiale. Intuitivamente, un problema di decisione Π è riducibile polinomialmente a un altro problema di decisione Π' se l'esistenza di un algoritmo di risoluzione polinomiale per Π' implica l'esistenza di un tale algoritmo anche per Π. Vediamo un esempio dettagliato per meglio illustrare questo concetto.

[2] L'acronimo NP sta per *non-deterministico polinomiale*, motivato storicamente dalla definizione della classe NP usando la macchina di Turing non-deterministica.

Sia $X = \{x_0, x_1, ..., x_{n-1}\}$ un insieme di n variabili booleane. Una *formula booleana in forma normale congiuntiva* su X è un insieme $C = \{c_0, c_1, ..., c_{m-1}\}$ di m *clausole*, dove ciascuna clausola c_i, per $0 \le i < m$, è a sua volta un insieme di *letterali*, ovvero un insieme di variabili in X e/o di loro negazioni (indicate con $\bar{x}_i$). Un'*assegnazione di valori* per X è una funzione $\tau : X \to \{TRUE, FALSE\}$ che assegna a ogni variabile un valore di verità. Un letterale l è soddisfatto da τ se $l = x_j$ e $\tau(x_j) = TRUE$ oppure se $l = \bar{x}_j$ e $\tau(x_j) = FALSE$, per qualche $0 \le j < n$. Una clausola è soddisfatta da τ se *almeno* un suo letterale lo è; infine, una formula è soddisfatta da τ se *tutte* le sue clausole lo sono.

Il **problema della soddisfacibilità** (indicato con SAT) consiste nel decidere se una formula booleana in forma normale congiuntiva è soddisfacibile. In particolare, il problema 2-SAT è la restrizione di SAT al caso in cui le clausole contengano esattamente due letterali.

Per mostrare che 2-SAT è risolvibile in tempo polinomiale, definiamo una riduzione da 2-SAT al problema di trovare le componenti fortemente connesse in un grafo orientato G, visto nel Capitolo 7: poiché quest'ultimo problema ammette un algoritmo di risoluzione polinomiale, anche 2-SAT ammette un tale algoritmo.

Data una formula booleana φ in forma normale congiuntiva formata da m clausole $c_0, c_1, ..., c_{m-1}$ su n variabili $x_0, x_1, ..., x_{n-1}$, costruiamo un grafo orientato $G = (V, E)$ contenente $2n$ vertici (due per ogni variabile booleana) e $2m$ archi (due per ogni clausola).[3] In particolare, per ogni variabile x_i, G include due vertici v_i^{pos} e v_i^{neg} corrispondenti, rispettivamente, ai letterali x_i e $\bar{x}_i$. Inoltre, per ogni clausola $\{l_1, l_2\}$ della formula, G include un arco tra il vertice corrispondente a $\bar{l}_1$ e quello corrispondente a l_2 e un arco tra il vertice corrispondente a $\bar{l}_2$ e quello corrispondente a l_1 (Figura 8.1). Poiché $\bar{a} \vee b$ può essere equivalentemente vista come l'implicazione logica $a \to b$, la formula $l_1 \vee l_2$ viene tradotta come le implicazioni $\bar{l}_1 \to l_2$ e $\bar{l}_2 \to l_1$: i corrispondenti due archi modellano il fatto che se uno dei due letterali della clausola non è soddisfatto, allora lo deve essere l'altro.

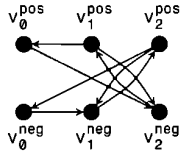


Figura 8.1 Un esempio di riduzione da 2-SAT al problema delle componenti fortemente connesse in un grafo: le clausole sono $\{x_0, \bar{x}_1\}$, $\{\bar{x}_0, \bar{x}_2\}$, $\{x_1, x_2\}$ e $\{\bar{x}_1, \bar{x}_2\}$.

[3] Senza perdita di generalità, possiamo supporre che nessuna clausola sia formata da una variabile e dalla sua negazione: in effetti, una tale clausola è soddisfatta da qualunque assegnazione di valori e può, quindi, essere eliminata dalla formula.

Prima di approfondire la nostra conoscenza sulle proprietà del grafo G introduciamo una notazione: se p è un vertice di G, con l_p indichiamo il letterale associato a p, ovvero se per esempio p = v_i^{neg} allora $l_p = \bar{x}_i$. Inoltre, i cammini presi in considerazione sono anch'essi orientati. Notiamo che il grafo G soddisfa la seguente proprietà: se esiste un cammino tra il vertice corrispondente a un letterale l e il vertice corrispondente a un letterale l′, allora esiste un cammino tra il vertice corrispondente a $\bar{l}'$ e il vertice corrispondente a $\bar{l}$.

ESEMPIO 8.4

Nella Figura 8.1 esiste il cammino da v_0^{neg} a v_2^{pos} (che passa per v_1^{neg}) ed esiste anche il cammino da v_2^{neg} a v_0^{pos} (che passa per v_1^{pos}).

Osserviamo anche che condizione necessaria e sufficiente affinché un'assegnazione τ soddisfi la formula φ è che per ogni arco (p, q) di G, τ non assegni al letterale l_p il valore TRUE e a l_q il valore FALSE in quanto non verrebbe soddisfatta la corrispondente clausola $\{\bar{l}_p, l_q\}$ di φ. Questa condizione si estende a un cammino tra p e q per transitività: nel caso in cui $\tau(l_p)$ = TRUE e $\tau(l_q)$ = FALSE, il cammino dovrebbe contenere due vertici p′ e q′ adiacenti per i quali $\tau(l'_p)$ = TRUE e $\tau(l'_q)$ = FALSE e quindi non verrebbe soddisfatta la corrispondente clausola $\{\bar{l}'_p, l'_q\}$.

Infine notiamo che un'assegnazione τ alle variabili deve per forza di cose assegnare lo stesso valore ai letterali corrispondenti ai nodi contenuti nella medesima componente fortemente connessa C. Se così non fosse dovrebbero esistere due nodi p e q in C tale che $\tau(l_p)$ = TRUE e $\tau(l_q)$ = FALSE. Per quanto detto sopra ciò non è possibile in quanto C contiene un cammino tra p e q.

Il seguente risultato lega ancora più profondamente il problema 2-SAT al problema della ricerca delle componenti fortemente connesse del grafo G e mostra come sfruttando questo legame si può ottenere un algoritmo di complessità lineare che risolve il problema di 2-SAT.

Teorema 8.2 *La formula φ è soddisfacibile se e solo se* v_i^{pos} *e* v_i^{neg} *appartengono a due componenti fortemente connesse distinte di* G *per ogni* $0 \le i < n$.

Dimostrazione La condizione necessaria è immediata: supponiamo per assurdo che v_i^{pos} e v_i^{neg} appartengano alla stessa componente fortemente connessa. Allora τ dovrebbe assegnare ai letterali x_i e $\bar{x}_i$ lo stesso valore e quindi φ non è soddisfacibile.

Per la condizione sufficiente, supponiamo che v_i^{pos} e v_i^{neg} appartengano a due diverse componenti fortemente connesse per ogni variabile x_i con $0 \le i < n$. Mostriamo come costruire un'assegnazione di valori τ che soddisfa φ.

Siano C_0, C_1, ..., C_{k-1} le componenti fortemente connesse di G. Costruiamo il grafo D = (V′, E′), dove V′ contiene un nodo c_i per ogni componente fortemente connessa C_i e E′ contiene l'arco (c_i, c_j) se esiste in G qualche arco tra un nodo in C_i e uno in C_j. Il grafo D è un grafo diretto aciclico (DAG) pertanto possiamo considerare il suo ordinamento topologico η visto nel Capitolo 7. Per comodità assumiamo che $\eta(c_i) < \eta(c_{i+1})$ per $0 \le i < k - 1$.

L'algoritmo mostrato nel Codice 8.2 costruisce l'assegnazione τ attraverso l'array booleano tau. Riceve in input un array di k liste C, una per ogni componente fortemente connessa, ognuna di queste liste contiene l'elenco dei letterali corrispondenti ai vertici appartenenti alla componente connessa associata. Inoltre l'ordinamento sull'array delle componenti corrisponde all'ordinamento topologico del grafo D. I 2n letterali sono rappresentati con un intero da 0 a 2n – 1: i rappresenta x_i se $0 \le i \le n - 1$, altrimenti rappresenta $\bar{x}_{i-n}$: quindi, dato l, il suo complementare $\bar{l}$ è l + n % 2n. L'array P indica, per ogni letterale, la componente fortemente connessa al quale appartiene: P[l] = j vuol dire che l è nella componente fortemente connessa C_j.

Codice 8.2 Algoritmo per 2-SAT basato sulle componenti fortemente connesse.

```
1  2Sat( C, P ):
2    FOR (i = 0; i < n; i = i+1)
3      tau[i] = NULL;
4    FOR (i = 0; i < k; i = i+1)
5      marcato[i] = FALSE;
6    FOR (i = k-1; i >= 0; i = i-1) {
7      IF (!marcato[i]) {
8        marcato[i] = TRUE;
9        FOR (x = C[i].inizio; x != null; x = x.succ) {
10         l = x.dato;
11         IF (tau[l]] == FALSE) RETURN FALSE
12         tau[l] = TRUE;
13         notl = (l + n) % 2n;
14         FOR (y = C[P[notl]].inizio; y != null; y = y.succ) {
15           IF (tau[y.dato] == TRUE) RETURN FALSE
16           tau[y.dato] = FALSE;
17         }
18         marcato[P[notl]] = TRUE;
19       }
20     }
21   }
22   RETURN TRUE
```

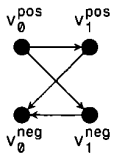
L'algoritmo analizza le componenti in ordine inverso rispetto l'ordinamento topologico: se trova una componente non marcata (riga 7), la marca (riga 8) e assegna TRUE a ogni letterale `l` contenuto in essa (riga 12). Invece, a ogni letterale nella componente fortemente connessa a cui appartiene $\bar{l}$ viene assegnato FALSE (righe 14-17) e tale componente viene marcata (riga 18). Osserviamo che ai letterali corrispondenti a nodi nella stessa componente connessa è assegnato lo stesso valore, inoltre $\tau(l)$ = TRUE se e solo se $\tau(\bar{l})$ = FALSE. In generale, prima di assegnare un valore di verità, occorre verificare che non sia stato già assegnato il valore complementare (e in tal caso la formula non è soddisfacibile).

Per induzione su `i = k - 1, ..., 0` dimostriamo che non esistono archi (l, l') tra due diverse componenti in $C_i, C_{i+1}, \ldots, C_{k-1}$ se `tau[l]` = TRUE e `tau[l']` = FALSE.

Se `i = k - 1` il risultato è vero, in quanto nell'intervallo abbiamo un'unica componente. Supponiamo che il risultato sia vero per l'intervallo $C_i, C_{i+1}, \ldots, C_{k-1}$ e sia `j < i` il massimo indice per cui ai letterali in C_j sia stato assegnato il valore TRUE. Osserviamo che tutte le componenti che seguono C_j nell'ordinamento erano marcate nel momento in cui ai letterali in C_j è stato assegnato TRUE. Dimostriamo che non esistono archi (l, l') con `l` in C_j e `l'` tale che `tau[l']` = FALSE. Se così fosse, per via dell'ordinamento topologico, `l'` deve appartenere a una componente `B` in $\{C_{j+1}, \ldots, C_{k-1}\}$. Ai nodi di `B` è stato assegnato FALSE in conseguenza all'assegnazione del valore TRUE ai nodi di una componente `D` in $\{C_i, \ldots, C_{k-1}\}$ che segue `B` nell'ordinamento. Allora in `D` esiste un letterale `l''` e in `B` il letterale $\bar{l}''$. Considerato l'arco (l, l') deduciamo che esiste un cammino da `l` a $\bar{l}''$ e quindi anche un cammino da `l''` a $\bar{l}$. Poiché `tau[l]` = TRUE, quest'ultimo cammino implicherebbe l'esistenza di un arco che contraddice l'ipotesi induttiva. ❑

ESEMPIO 8.5

Il grafo mostrato nella Figura 8.1 ha solo due componenti fortemente connesse senza archi tra di loro, il Codice 8.2 assegna ai letterali di una di queste TRUE e agli altri FALSE. Un esempio ancora molto semplice ma più interessante è dato dalla formula composta dalle clausole $\{\bar{x}_0, x_1\}$, $\{\bar{x}_0, \bar{x}_1\}$: il grafo G che ne risulta è il seguente.



Ogni nodo è una componente fortemente connessa e $C_0 = \{x_0\}$, $C_1 = \{x_1\}$, $C_2 = \{\bar{x}_1\}$ e $C_3 = \{\bar{x}_0\}$ è un ordinamento topologico delle componenti. L'algoritmo assegna `tau[2]` = TRUE, `marcato[3]` = TRUE, quindi `tau[0]` = FALSE e `marcato[0]` = TRUE. La componente C_2 non è marcata quindi si assegna `tau[3]` = TRUE e `marcato[2]` = TRUE e di conseguenza `tau[1]` = FALSE e `marcato[1]` = TRUE.

Ogni componente fortemente connessa e ogni letterale viene preso in considerazione soltanto una volta dall'algoritmo mostrato nel Codice 8.2, quindi questo ha costo lineare in tempo. Considerando che gli array `C` e `P` possono essere costruiti in tempo lineare usando gli algoritmi per il calcolo delle componenti fortemente connesse (Paragrafo 7.3.2) e dell'ordinamento topologico (Paragrafo 7.3.1) concludiamo che 2-SAT può essere risolto in tempo lineare.

8.4 Problemi NP-completi

In questo capitolo, siamo principalmente interessati a utilizzare il concetto di riducibilità per ottenere risultati negativi piuttosto che positivi, ovvero per dimostrare che un problema *non è* risolvibile facendo uso di risorse temporali limitate. Un semplice esempio di tale applicazione consiste nel dimostrare un limite inferiore alla complessità temporale per il problema geometrico del minimo insieme convesso. Tale problema consiste nel trovare, dato un insieme di punti sul piano, il più piccolo (rispetto all'inclusione insiemistica) insieme convesso `S` che li contiene tutti:[4] nella Figura 8.2 mostriamo due esempi di insiemi convessi minimi. Un punto `p` ∈ `S` è un *estremo* di `S`, se esiste un semipiano passante per `p` tale che `p` è l'unico punto che giace sulla retta che delimita il semipiano: il problema del **minimo insieme convesso** consiste nel calcolare gli estremi di `S` come una lista (ciclicamente) ordinata di punti (per esempio, la soluzione nella parte sinistra della Figura 8.2 è data da p_0, p_1, p_2 e p_3).

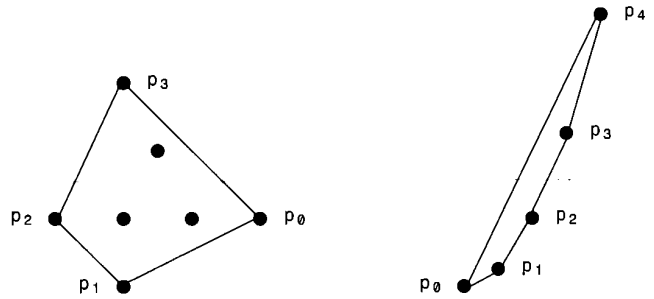


Figura 8.2 Due esempi di minimo insieme convesso.

Teorema 8.3 *Nel caso generale il problema del minimo insieme convesso ha complessità* $\Omega(n \log n)$.

Dimostrazione Riduciamo il problema dell'ordinamento di `n` numeri interi $a_0, a_1, \ldots, a_{n-1}$ a quello del calcolo del minimo insieme convesso nel modo seguente: per ogni `i` con $0 \le i \le n-1$, definiamo un punto di coordinate (a_i, a_i^2). Gli `n` punti

[4] Ricordiamo che un insieme `S` è detto *convesso* se, per ogni coppia di punti in `S`, il segmento che li unisce è interamente contenuto in `S`.

così costruiti giacciono sulla parabola di equazione $y = x^2$ (come nella parte destra della Figura 8.2 quindi il loro minimo insieme convesso consiste nella lista dei punti ordinata in base alle loro ascisse: tale elenco è dunque l'ordinamento degli n numeri interi. Poiché la costruzione suddetta può essere eseguita in tempo $O(n)$, se il problema del minimo insieme convesso fosse risolvibile in tempo $o(n \log n)$, allora anche il problema dell'ordinamento di n numeri interi sarebbe risolvibile in tempo $o(n \log n)$, ma il Teorema 2.4 ci dice che ciò non è in generale possibile. ❑

In generale, se un problema Π è riducibile polinomialmente a un problema Π' e se sappiamo che Π non ammette un algoritmo di risoluzione polinomiale, allora possiamo concludere che neanche Π' ammette un tale algoritmo.

In altre parole, Π' è almeno tanto difficile quanto Π (notiamo che l'uso "positivo" del concetto di riducibilità consiste nell'affermare che Π è almeno tanto facile quanto Π'): quindi, le cattive notizie, ovvero, la non trattabilità di un problema, si propagano da sinistra verso destra (mentre le buone notizie lo fanno da destra verso sinistra).

Per definire la nozione di NP-completezza, introduciamo una restrizione del concetto di riducibilità in cui ogni istanza del problema di partenza viene trasformata in un'istanza del problema di arrivo, in modo che le due istanze siano entrambe positive oppure entrambe negative. Formalmente, un problema Π è **polinomialmente trasformabile** in un problema Π' se esiste un algoritmo polinomiale T tale che, per ogni sequenza binaria x, vale $x \in \Pi$ se e solo se T con x in ingresso restituisce una sequenza binaria in Π'. Chiaramente, se Π è polinomialmente trasformabile in Π', allora Π è polinomialmente riducibile a Π': infatti, se esiste un algoritmo polinomiale A di risoluzione per Π', allora la composizione di T con A fornisce un algoritmo polinomiale di risoluzione per Π.

Un problema di decisione Π è **NP-completo** se Π appartiene a NP e se ogni altro problema in NP è polinomialmente trasformabile in Π. Quindi, Π è almeno tanto difficile quanto ogni altro problema in NP: in altre parole, se dimostriamo che Π è in P, allora abbiamo che l'intera classe NP è contenuta in P (e quindi le due classi coincidono).

È naturale a questo punto chiedersi se esistono problemi NP-completi (anche se il lettore avrà già intuito la risposta a tale domanda). Inoltre, se $P \neq NP$, possono esistere problemi che né appartengono a P né sono NP-completi (un possibile candidato di questo tipo è il **problema aperto** dell'isomorfismo tra grafi, discusso nel Capitolo 7, del quale non conosciamo un algoritmo polinomiale di risoluzione e nemmeno una dimostrazione di NP-completezza).

Prima di discutere l'esistenza di un problema NP-completo, però, osserviamo che una volta dimostratane l'esistenza, possiamo sfruttare la proprietà di transitività della trasformabilità polinomiale per estendere l'insieme di problemi siffatti. La definizione di trasformabilità soddisfa infatti la seguente proprietà: se Π_0 è polinomialmente trasformabile in Π_1 e Π_1 è polinomialmente trasformabile in Π_2, allora Π_0 è polinomialmente trasformabile in Π_2. A questo punto, volendo dimostrare che un certo problema computazionale Π è NP-completo, possiamo procedere in tre passi: prima dimostriamo che Π appartiene a NP mostrando l'esistenza del suo certificato polinomiale; poi individuiamo un altro problema Π', che già sappiamo essere NP-completo; infine, trasformiamo polinomialmente Π' in Π.

8.5 Teorema di Cook-Levin

Per applicare la strategia sopra esposta dobbiamo necessariamente trovare un primo problema NP-completo. Il **teorema di Cook-Levin** afferma che SAT è NP-completo. Osserviamo che SAT appartiene a NP, in quanto ogni formula soddisfacibile ammette una dimostrazione breve e facile da verificare che consiste nella specifica di un'assegnazione di valori che soddisfa la formula. La parte difficile del teorema di Cook-Levin consiste, quindi, nel mostrare che ogni problema in NP è polinomialmente trasformabile in SAT.

Non diamo la dimostrazione del suddetto teorema, ma ci limitiamo a fornire una breve descrizione dell'approccio utilizzato. Dato un problema $\Pi \in$ NP, sappiamo che esiste un algoritmo polinomiale V e un polinomio p tali che, per ogni sequenza binaria x, se $x \in \Pi$, allora esiste una sequenza y di lunghezza $p(|x|)$ tale che V con x e y in ingresso termina restituendo il valore TRUE, mentre se $x \notin \Pi$, allora, per ogni sequenza y, V con x e y in ingresso termina restituendo il valore FALSE. L'idea della dimostrazione consiste nel costruire, per ogni x, in tempo polinomiale una formula booleana φ_x le cui uniche variabili *libere* sono $p(|x|)$ variabili $y_0, y_1, \ldots, y_{p(|x|)-1}$, intendendo con ciò che la soddisfacibilità della formula dipende solo dai valori assegnati a tali variabili: intuitivamente, la variabile y_i corrisponde al valore del bit in posizione i all'interno della sequenza y. La formula φ_x in un certo senso *simula* il comportamento di V con x e y in ingresso ed è soddisfacibile solo se tale computazione termina in modo affermativo (ovvero, se y è una dimostrazione che x appartiene a Π). Il fatto che possiamo costruire una tale formula non dovrebbe sorprenderci più di tanto, se consideriamo che, in fin dei conti, l'esecuzione di un algoritmo all'interno di un calcolatore avviene attraverso circuiti logici le cui componenti di base sono porte logiche che realizzano la disgiunzione (*or*), la congiunzione (*and*) e la negazione (*not*).

Per convincerci ulteriormente che la dimostrazione del teorema di Cook-Levin così tracciata può effettivamente essere realizzata, mostriamo, per esempio, come un'istanza del problema di decidere se un grafo G può essere colorato con tre colori (che è chiaramente un problema in NP) può essere descritta mediante una formula booleana φ_G. In particolare, φ_G non sarà in forma normale congiuntiva: tuttavia, possiamo facilmente dimostrare che una formula booleana ψ può essere trasformata in tempo polinomiale in una formula booleana ψ' in forma normale congiuntiva, tale che ψ è soddisfacibile se e solo se ψ' è soddisfacibile.

Sappiamo che G = (V, E) può essere rappresentato mediante la matrice di adiacenza A tale che A[i][j] = 1 se e solo se (i, j) ∈ E. A tale matrice facciamo corrispondere n × n variabili booleane $a_{i,j}$ e usiamo in φ_G le seguenti formule booleane, per $0 \le i, j < n$:

$$A_{i,j} = \begin{cases} a_{i,j} & \text{se } A[i][j] = 1 \\ \bar{a}_{i,j} & \text{altrimenti} \end{cases}$$

(in altre parole, queste formule forzano le variabili $a_{i,j}$ a rappresentare la matrice di adiacenza di G). Per ogni vertice $i \in V$, introduciamo poi tre variabili booleane r_i, g_i e b_i che corrispondono ai tre possibili colori che possono essere assegnati al vertice (quindi, queste sono le variabili libere i cui valori di verità forniscono la dimostrazione y). Per impedire che due colori vengano assegnati allo stesso vertice, usiamo in φ_G le seguenti formule booleane, per $0 \le i < n$:

$$B_i = (r_i \wedge \bar{g}_i \wedge \bar{b}_i) \vee (\bar{r}_i \wedge g_i \wedge \bar{b}_i) \vee (\bar{r}_i \wedge \bar{g}_i \wedge b_i)$$

Infine, per verificare che l'assegnazione dei colori ai vertici sia compatibile con gli archi del grafo, φ_G usa le seguenti formule booleane, per $0 \le i, j < n$:

$$C_{i,j} = a_{i,j} \Rightarrow \left[(r_i \wedge \bar{r}_j) \vee (g_i \wedge \bar{g}_j) \vee (b_i \wedge \bar{b}_j)\right] = \bar{a}_{i,j} \vee (r_i \wedge \bar{r}_j) \vee (g_i \wedge \bar{g}_j) \vee (b_i \wedge \bar{b}_j)$$

(informalmente, $C_{i,j}$ afferma che se vi è un arco tra i due vertici i e j, allora questi due vertici non possono avere lo stesso colore). In conclusione, la formula φ_G è la seguente:

$$\bigwedge_{0 \le i,j < n} A_{i,j} \wedge \bigwedge_{0 \le i < n} B_i \wedge \bigwedge_{0 \le i,j < n} C_{i,j}$$

Chiaramente, φ_G è soddisfacibile se e solo se i vertici di G possono essere colorati con tre colori. Il teorema di Cook-Levin afferma sostanzialmente che quanto abbiamo appena fatto per il problema della colorazione può in realtà essere fatto per qualunque problema in NP.

Notiamo che la NP-completezza di SAT non implica che il problema della soddisfacibilità di formule booleane in forma normale disgiuntiva sia anch'esso NP-completo: se una formula è in **forma normale disgiuntiva**, allora una clausola è soddisfatta da un'assegnazione di valori τ se tutti i suoi letterali lo sono e la formula è soddisfatta da τ se almeno una sua clausola lo è.

In tal caso, possiamo mostrare che il problema della soddisfacibilità è risolvibile in tempo polinomiale e, quindi, è molto probabilmente non NP-completo.

8.6 Problemi di ottimizzazione

Prima di passare a dimostrare diversi risultati di NP-completezza, introduciamo il concetto di problema di ottimizzazione, inteso in qualche modo come estensione di quello di problema decisionale.

In un **problema di ottimizzazione**, a ogni istanza del problema associamo un insieme di soluzioni possibili e a ciascuna soluzione associamo una misura (che può essere un costo oppure un profitto): il problema consiste nel trovare, data un'istanza, una soluzione ottima, ovvero una soluzione di misura minima se la misura è un costo, una di misura massima altrimenti.

Abbiamo già incontrato diversi problemi di ottimizzazione nei capitoli precedenti (per esempio, il problema del minimo albero di ricoprimento del Paragrafo 7.5).

Osserviamo che a ogni problema di ottimizzazione corrisponde in modo abbastanza naturale un problema di decisione definito nel modo seguente: data un'istanza del problema e dato un valore k, decidere se la misura della soluzione ottima è inferiore a k (nel caso di costi) oppure superiore a k (nel caso di profitti).

Nella maggior parte dei problemi di ottimizzazione che sorgono nella realtà, abbiamo che se il corrispondente problema di decisione è risolvibile in tempo polinomiale, allora anche il problema di ottimizzazione è risolvibile in tempo polinomiale.

Ciò è principalmente dovuto al fatto che il valore massimo che la misura di una soluzione può assumere è limitato esponenzialmente dalla lunghezza dell'istanza: questa osservazione ci consente di ridurre polinomialmente un problema di ottimizzazione al suo corrispondente problema di decisione, operando in base a un meccanismo simile a quello di ricerca binaria descritto nel Paragrafo 3.3.

Quindi, se il problema di decisione associato a un problema di ottimizzazione è NP-completo, abbiamo che quest'ultimo non può essere risolto in tempo polinomiale a meno che P = NP: nel paragrafo finale di questo capitolo, analizzeremo in dettaglio uno dei più famosi problemi di ottimizzazione intrinsecamente difficili.

8.7 Generazione esaustiva e backtrack

Nelle applicazioni si può avere la necessità di trovare soluzioni esatte a problemi NP-completi, tale ricerca è agevolata anche dal fatto che, nella pratica, la dimensione degli input di problemi specifici potrebbe essere molto contenuta. Pertanto anche l'utilizzo di algoritmi non efficienti può portare a buoni risultati.

La tecnica più immediata di progettazione di algoritmi per problemi NP-completi è la **generazione esaustiva**: questa consiste nel verificare una alla volta tutte le potenziali soluzioni fino a trovarne una che sia effettiva soluzione del problema; nel caso in cui questa non venga trovata si conclude che la particolare istanza del problema non ammette soluzioni.

ESEMPIO 8.6

Una soluzione potenziale per SAT è un'assegnazione dei valori TRUE o FALSE alle n variabili, in altri termini è una sequenza di n elementi in {TRUE, FALSE}. L'algoritmo di generazione esaustiva non deve far altro che prendere in considerazioni una alla volta tutte le 2^n soluzioni parziali, non appena ne trova una che soddisfa φ termina la ricerca restituendo la soluzione trovata; se non ne trova termina l'esecuzione in uno stato di insuccesso. Poiché verificare che una sequenza generata soddisfa φ ha una costo polinomiale $p(n)$, concludiamo che l'algoritmo descritto ha complessità $O(2^n p(n))$.

Questa tecnica può essere utilizzata per tutti i problemi in NP. Infatti, dalla definizione di problema NP, generare tutti i potenziali certificati polinomiali y di una istanza x del problema ha un costo $O(2^{p(|x|)})$ per un polinomio p; infine ogni certificato può essere verificato in tempo polinomiale.

Una tecnica che potrebbe rivelarsi più efficiente per la risoluzione di alcuni problemi in NP è quella del **backtrack** (in italiano, tornare indietro). L'idea è la seguente: si parte da una soluzione parziale e, a ogni passo, si cerca di completarla eseguendo una determinata scelta; se si giunge in una posizione in cui non è più possibile andare aventi perché nessuna scelta possibile porterebbe all'individuazione di una soluzione, si torna indietro (backtrack), scartando parte della soluzione costruita, fino a giungere in uno stato dal quale si può procedere con altre scelte praticabili.

Il Codice 8.3 mostra un algoritmo che utilizza la tecnica del backtrack per risolvere il problema della soddisfacibilità. La funzione `Sat` invoca la funzione ricorsiva `SatRec` con input `0`, se questa restituisce `TRUE` la formula è soddisfatta dall'assegnazione descritta nell'array `tau` altrimenti la formula non è soddisfacibile.

Codice 8.3 Algoritmo per SAT che utilizza la tecnica del backtrack.

```
1  Sat( ):                    ⟨pre: una istanza del problema della soddisfacibilità⟩
2    IF (SatRec( 0 )) {
3      RETURN TRUE;
4    } ELSE {
5      RETURN FALSE;
6    }
7  SatRec( i ):
8    IF ( i == n ) RETURN TRUE;
9    tau[i] = FALSE;
10   IF (Test(i)) {
11     IF (SatRec(i+1)) RETURN TRUE;
12   }
13   tau[i] = TRUE;
14   IF (Test(i)) {
15     IF (SatRec(i+1)) RETURN TRUE;
16   }
17   RETURN FALSE;
```


La funzione `SatRec` con input `i` fa quanto segue: se `i` è n, tutte le variabili sono state assegnate con successo e quindi restituisce `TRUE` (riga 8); altrimenti estende l'assegnazione parziale assegnando alla variabile `i` `FALSE` (riga 9); se questa assegnazione non contraddice nessuna clausola (verifica lasciata alla funzione `Test` nella riga 10) viene invocata la ricorsione su `i + 1` (riga 11), altrimenti si esegue lo stesso processo assegnando alla variabile `i` il valore `TRUE` (righe 13-16). Se nemmeno questo porta a una soluzione, la funzione restituisce `FALSE` (riga 17). Questo provoca il backtrack su qualche valore di `i` più piccolo.

ESEMPIO 8.7

Utilizziamo il Codice 8.3 per trovare una soluzione della formula composta della clausole $\{\bar{x}_0, x_1, x_2\}$, $\{x_0, x_1, x_3\}$, $\{x_0, x_2\}$ e $\{\bar{x}_2, \bar{x}_3\}$.

L'invocazione delle prime due chiamate ricorsive non crea problemi in quanto l'assegnazione parziale `tau[0]` = FALSE e `tau[1]` = FALSE non nega nessuna clausola. Invece, quando per `i` = 2 viene verificata l'assegnazione estesa a `tau[2]` = FALSE, risulta che la terza clausola è insoddisfatta. Quindi si prova con `tau[2]` = TRUE; l'assegnazione risultante è consistente con tutte le clausole quindi può essere estesa. Tuttavia se `tau[3]` = FALSE la seconda clausola è insoddisfatta, mentre se `tau[3]` = TRUE è insoddisfatta la quarta. Si fa backtrack fino a ritornare alla chiamata di `SatRec` per `i` = 1, da qui si riparte assegnando `tau[1]` = TRUE (ricordiamo che `tau[0]` = FALSE). Nessuna clausola risulta insoddisfatta per cui proseguiamo con la chiamata successiva. Estendiamo la soluzione parziale con `tau[2]` = FALSE, ma la terza clausola è insoddisfatta quindi si passa a `tau[2]` = TRUE. Con l'invocazione di `SatRec` estendiamo quest'ultima assegnazione con `tau[3]` = FALSE, nessuna clausola è insoddisfatta quindi si invoca `SatRec` con input 4, ovvero n, che restituisce TRUE.

Si osservi che il backtrack non è altro che una ricerca esaustiva in cui non vengono considerate soluzioni potenziali già parzialmente generate, ma di cui si sa che non potranno diventare soluzioni del problema. Questo rende la tecnica del backtrack mediamente più efficiente della generazione esaustiva. Comunque nel caso peggiore siamo costretti a generare tutte o quasi le soluzioni potenziali, quindi la complessità temporale resta esponenziale.

8.8 Esempi e tecniche di NP-completezza

A partire da SAT, mostriamo ora come sia possibile verificare la NP-completezza di altri problemi computazionali: alcuni che abbiamo esaminato nei capitoli precedenti e che abbiamo dichiarato essere NP-completi e altri che useremo come problemi di passaggio nelle catene di trasformazioni polinomiali.

Per prima cosa, dimostriamo che la restrizione 3-SAT di SAT a clausole formate da *esattamente* tre letterali è anch'esso un problema NP-completo (chiaramente 3-SAT è in NP per lo stesso motivo per cui lo è SAT).

8.8.1 Tecnica di sostituzione locale

Sia $C = \{c_0, ..., c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di variabili booleane $\{x_0, ..., x_{n-1}\}$. Vogliamo costruire, in tempo polinomiale, un nuovo insieme D di clausole, ciascuna di cardinalità 3, costruite a partire da un insieme Z di variabili booleane e tali che C è soddisfacibile se e solo se D è soddisfacibile. A tale scopo usiamo una tecnica di trasformazione detta di **sostituzione locale**, in base alla quale costruiremo D e Z sostituendo ogni clausola $c \in C$ con un sottoinsieme D_c di D in modo indipendente dalle altre clausole di C: l'insieme D è quindi uguale a $\cup_{c \in C} D_c$ e Z è l'unione di tutte le variabili booleane che appaiono in D.

Data una clausola $c = \{l_0, ..., l_{k-1}\}$ dell'insieme C, definiamo D_c distinguendo i seguenti quattro casi:

1. $k = 1$: in questo caso, $D_c = \left\{\left\{l_0, y_0^c, y_1^c\right\}, \left\{l_0, y_0^c, \overline{y}_1^c\right\}, \left\{l_0, \overline{y}_0^c, y_1^c\right\}, \left\{l_0, \overline{y}_0^c, \overline{y}_1^c\right\}\right\}$ (chiaramente D_c è soddisfacibile se e solo se l_0 è soddisfatto);
2. $k = 2$: in questo caso, $D_c = \left\{\left\{l_0, l_1, y_0^c\right\}, \left\{l_0, l_1, \overline{y}_0^c\right\}\right\}$ (chiaramente D_c è soddisfacibile se e solo se l_0 oppure l_1 è soddisfatto);
3. $k = 3$: in questo caso, D_c è formato dalla sola clausola c;
4. $k > 3$: in questo caso, che è il più difficile, l'insieme D_c contiene un insieme di $k-2$ clausole collegate tra di loro attraverso nuove variabili booleane e tali che la loro soddisfacibilità sia equivalente a quella di c. Formalmente, D_c è l'insieme

$$\left\{\left\{l_0, l_1, y_0^c\right\}, \left\{\overline{y}_0^c, l_2, y_1^c\right\}, \left\{\overline{y}_1^c, l_3, y_2^c\right\}, \ldots, \left\{\overline{y}_{k-5}^c, l_{k-3}, y_{k-4}^c\right\}, \left\{\overline{y}_{k-4}^c, l_{k-2}, l_{k-1}\right\}\right\}$$

Dalla definizione di D_c abbiamo che

$$Z = X \cup \bigcup_{c \in C \wedge |c|=1} \{y_0^c, y_1^c\} \cup \bigcup_{c \in C \wedge |c|=2} \{y_0^c\} \cup \bigcup_{c \in C \wedge |c|>3} \{y_0^c, \ldots, y_{|c|-4}^c\}$$

e che la costruzione dell'istanza di 3-SAT può essere eseguita in tempo polinomiale.

Teorema 8.4 *La formula C è soddisfacibile se e solo se D è soddisfacibile.*

Dimostrazione Supponiamo che esista un'assegnazione τ di verità alle variabili di X che soddisfa C. Quindi, τ soddisfa c per ogni clausola $c \in C$: mostriamo che tale assegnazione può essere estesa alle nuove variabili di tipo y^c introdotte nel definire D_c, in modo che tutte le clausole in esso contenute siano soddisfatte (da quanto detto sopra, possiamo supporre che $|c| > 3$). Poiché c è soddisfatta da τ, deve esistere h tale che τ soddisfa l_h con $0 \le h \le |c| - 1$: estendiamo τ assegnando il valore TRUE a tutte le variabili y_i^c con $0 \le i \le h - 2$ e il valore FALSE alle rimanenti variabili. In questo modo, siamo sicuri che la clausola di D_c contenente l_h è soddisfatta (da l_h stesso), le clausole che la precedono sono soddisfatte grazie al loro terzo letterale e quelle che la seguono lo sono grazie al loro primo letterale.

Viceversa, supponiamo che esista un'assegnazione τ di verità alle variabili di Z che soddisfi tutte le clausole in D e, per assurdo, che tale assegnazione ristretta alle sole variabili di X non soddisfi almeno una clausola $c \in C$, ovvero che tutti i letterali contenuti in c non siano soddisfatti (di nuovo, ipotizziamo che $|c| > 3$). Ciò implica che tutte le variabili di tipo y^c devono essere vere, perché altrimenti una delle prime $|c| - 3$ clausole in D_c non sarebbe soddisfatta, contraddicendo l'ipotesi che τ soddisfa tutte le clausole in D. Quindi, $\tau(y_{|c|-4}^c) = \text{TRUE}$, ovvero $\tau(\overline{y}_{|c|-4}^c) = \text{FALSE}$: poiché abbiamo supposto che anche $l_{|c|-2}$ e $l_{|c|-1}$ non sono soddisfatti, l'ultima clausola in D_c non è soddisfatta, contraddicendo nuovamente l'ipotesi che τ soddisfa tutte le clausole in D. ❑

Come conseguenza del risultato appena dimostrato abbiamo che il problema SAT è trasformabile in tempo polinomiale nel problema 3-SAT: quindi, quest'ultimo è NP-completo. Notiamo che, in modo simile a quanto fatto per 3-SAT, possiamo mostrare la NP-completezza del problema della soddisfacibilità nel caso in cui le clausole contengono esattamente k letterali, per ogni $k \ge 3$: tale affermazione non si estende però al caso in cui $k = 2$, in quanto, come abbiamo visto nel Paragrafo 8.3, in questo caso il problema diviene risolvibile in tempo polinomiale e, quindi, difficilmente esso è anche NP-completo.

La NP-completezza di 3-SAT ha un duplice valore: da un lato essa mostra che la difficoltà computazionale del problema della soddisfacibilità non dipende dalla lunghezza delle clausole (fintanto che queste contengono almeno tre letterali), dall'altro ci consente nel seguito di usare 3-SAT come problema di partenza, il quale, avendo istanze più regolari, è più facile da utilizzare per sviluppare trasformazioni volte a dimostrare risultati di NP-completezza.

8.8.2 Tecnica di progettazione di componenti

Per dimostrare la NP-completezza del problema del massimo insieme indipendente in un grafo (o meglio della sua versione decisionale), mostriamo prima che il seguente problema, detto **minimo ricoprimento tramite vertici** è NP-completo: dato un grafo $G = (V, E)$ e un intero $k \ge 0$, esiste un sottoinsieme V' di V con $|V'| \le k$, tale che ogni arco del grafo è coperto da V' ovvero, per ogni arco $(u, v) \in E$, $u \in V'$ oppure $v \in V'$? Nella Figura 8.3 mostriamo un esempio di ricoprimento tramite 3 vertici del grafo delle conoscenze discusso nel Paragrafo 7.1.1. Notiamo che, in questo caso, un qualunque sottoinsieme di vertici di cardinalità minore di 3, non può essere un ricoprimento: in effetti, due vertici sono necessari per coprire

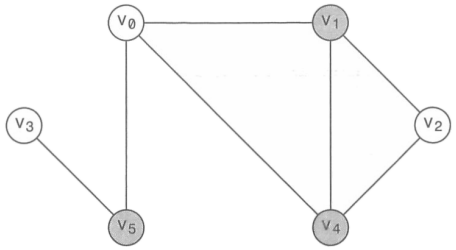


Figura 8.3 Un esempio di minimo ricoprimento tramite vertici, i nodi del ricoprimento sono in grigio.

il triangolo formato da v_1, v_2 e v_4 e un ulteriore vertice è necessario per coprire l'arco tra v_3 e v_5.

Il problema del minimo ricoprimento tramite vertici ammette dimostrazioni brevi e verificabili in tempo polinomiale: tali dimostrazioni sono i sottoinsiemi dell'insieme dei vertici del grafo che costituiscono un ricoprimento degli archi di cardinalità al più k.

Mostriamo ora che 3-SAT è trasformabile in tempo polinomiale nel problema del minimo ricoprimento tramite vertici: a tale scopo, faremo uso di una tecnica più sofisticata di quella vista nel paragrafo precedente, che viene generalmente indicata con il nome di **progettazione di componenti**. In particolare, la trasformazione opera definendo, per ogni variabile, una componente (*gadget*) del grafo il cui scopo è quello di modellare l'assegnazione di verità alla variabile e, per ogni clausola, una componente il cui scopo è quello di modellare la soddisfacibilità della clausola. I due insiemi di componenti sono poi collegati tra di loro per garantire che l'assegnazione alle variabili soddisfi tutte le clausole.

Più precisamente, sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di variabili booleane $\{x_0, \ldots, x_{n-1}\}$, tali che $|c_i| = 3$ per $0 \leq i < m$. Vogliamo definire un grafo G e un intero k tale che C è soddisfacibile se e solo se G include un ricoprimento di esattamente k vertici.

Per ogni variabile x_i con $0 \leq i < n$, G include due vertici v_i^{vero} e v_i^{falso} collegati tra di loro mediante un arco. Queste sono le componenti di verità del grafo, in quanto ogni ricoprimento di G deve necessariamente includere almeno un vertice tra v_i^{vero} e v_i^{falso} per $0 \leq i < n$: il valore di k sarà scelto in modo tale che ne includa esattamente uno, ovvero quello corrispondente al valore di verità della variabile corrispondente.

Per ogni clausola c_j con $0 \leq j < m$, G include una cricca di tre vertici v_j^0, v_j^1 e v_j^2. Queste sono le componenti corrispondenti alla soddisfacibilità delle clausole, in quanto ogni ricoprimento di G deve necessariamente includere almeno due vertici tra v_j^0, v_j^1 e v_j^2 per $0 \leq j < m$: il valore di k sarà scelto in modo tale che ne includa esattamente due, in modo che quello non selezionato corrisponda a un letterale certamente soddisfatto all'interno della clausola corrispondente.

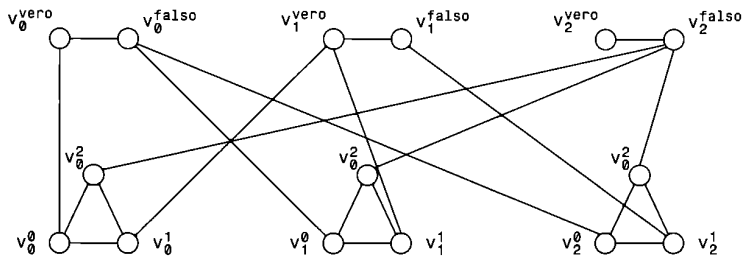


Figura 8.4 Un esempio di riduzione da 3-SAT al problema del minimo ricoprimento tramite vertici: le clausole sono $\{x_0, x_1, \bar{x}_2\}$, $\{\bar{x}_0, x_1, \bar{x}_2\}$ e $\{\bar{x}_0, \bar{x}_1, \bar{x}_2\}$.

Le componenti di verità e quelle di soddisfacibilità sono collegate tra di loro aggiungendo un arco tra i vertici contenuti nelle prime componenti con i corrispondenti vertici contenuti nelle seconde componenti. Più precisamente, per ogni i, j e k con $0 \leq i < n$, $0 \leq j < m$ e $0 \leq h < 3$:

- il vertice v_i^{vero} è collegato al vertice v_j^h se e solo se l'$(h+1)$-esimo letterale della clausola c_j è x_i;
- il vertice v_i^{falso} è collegato al vertice v_j^h se e solo se l'$(h+1)$-esimo letterale della clausola c_j è $\bar{x}_i$.

Nella Figura 8.4 mostriamo il grafo così ottenuto a partire dal seguente insieme di clausole: $\{x_0, x_1, \bar{x}_2\}$, $\{\bar{x}_0, x_1, \bar{x}_2\}$ e $\{\bar{x}_0, \bar{x}_1, \bar{x}_2\}$. Rimane da definire il valore di k: come abbiamo già detto, vogliamo che tale valore ci costringa a prendere esattamente un vertice per ogni componente di verità ed esattamente due vertici per ogni componente di soddisfacibilità. Poiché abbiamo n componenti del primo tipo e m componenti del secondo tipo, poniamo $k = n + 2m$.

Teorema 8.5 *L'insieme di clausole C è soddisfacibile se e solo se esiste un ricoprimento di G di dimensione $n + 2m$.*

Dimostrazione Sia τ un'assegnazione di verità che soddisfa C, ovvero tale che, per ogni clausola c_j con $0 \leq j < m$, esiste un letterale soddisfatto contenuto in c_j: indichiamo con p_j la posizione del primo tale letterale, dove $0 \leq p_j < 3$. Costruiamo un ricoprimento V' nel modo seguente: per $0 \leq i < n$, V' include v_i^{vero} se $\tau(x_i) =$ TRUE, altrimenti include v_i^{falso}; inoltre, per $0 \leq j < m$, V' include v_j^h dove $0 \leq h < 3$ e $h \neq p_j$ (notiamo che l'arco tra $v_j^{p_j}$ e il corrispondente vertice contenuto in una componente di verità è coperto da quest'ultimo). Poiché, per ogni componente di verità, V' include un vertice e, per ogni componente di soddisfacibilità, ne include due, abbiamo che V' è un ricoprimento e che $|V'| = n + 2m = k$.

Viceversa, supponiamo che V' sia un ricoprimento di G che include esattamente $n+2m$ nodi. Ciò implica che V' deve includere un vertice per ogni componente di verità e due vertici per ogni componente di soddisfacibilità. Definiamo un'assegnazione di verità τ tale che $\tau(x_i) = \text{TRUE}$ se e solo se $v_i^{vero} \in V'$ per $0 \leq i < n$: chiaramente, τ è un'assegnazione di verità corretta (ovvero, non assegna alla stessa variabile due valori di verità diversi). Inoltre, per ogni clausola c_j con $0 \leq j < m$, deve esistere h con $0 \leq h < 3$ tale che $v_j^h \notin V'$: l'arco che unisce v_j^h al vertice corrispondente contenuto in una componente di verità deve, quindi, essere coperto da quest'ultimo che è incluso in V'. Pertanto, l'$(h+1)$-esimo letterale in c_j è soddisfatto da τ e la clausola c_j è anch'essa soddisfatta. ❑

ESEMPIO 8.8

Facendo riferimento all'esempio mostrato nella Figura 8.4, supponiamo che τ assegni il valore TRUE alla sola variabile x_0: in tal caso, V' include i vertici v_0^{vero}, v_1^{falso} e v_2^{falso}. Il primo letterale soddisfatto contenuto nella prima clausola è x_0, che si trova nella posizione 0: quindi, V' include i due vertici v_0^1 e v_0^2. Analogamente, possiamo mostrare che V' include i vertici v_1^0, v_1^1, v_2^0 e v_2^2 e che, quindi, è un ricoprimento del grafo di cardinalità 3 + 6 = 9.

Da'altro canto, supponiamo che V' includa i vertici v_0^{falso}, v_1^{falso}, v_2^{falso}, v_0^0, v_0^1, v_1^0, v_1^1, v_2^0 e v_2^1: in tal caso, τ assegna il valore FALSE a tutte e tre le variabili booleane. Tale assegnazione soddisfa tutte le clausole di C: per esempio, della componente corrispondente alla prima clausola V' non include il vertice v_0^2 ma della componente corrispondente a x_2 include il vertice v_2^{falso}, per cui $\tau(\overline{x}_2) = \text{TRUE}$ e la clausola è soddisfatta.

Poiché il grafo G può essere costruito in tempo polinomiale a partire dall'insieme di clausole C, abbiamo che 3-SAT è polinomialmente trasformabile nel problema del minimo ricoprimento tramite vertici e, quindi, che quest'ultimo è NP-completo.

8.8.3 Tecnica di similitudine

A partire dal problema del minimo ricoprimento tramite vertici siamo ora in grado di dimostrare la NP-completezza del seguente: dato un grafo $G = (V, E)$ e un intero $k \geq 0$, esiste un sottoinsieme V' di V con $|V'| \geq k$, tale che V' è un insieme indipendente ovvero, per ogni arco $(u, v) \in E$, $u \notin V'$ oppure $v \notin V'$? In questo caso, la trasformazione è molto più semplice di quelle viste finora e si basa sulla tecnica della **similitudine**, che consiste appunto nel mostrare come un problema sia simile a uno già precedentemente dimostrato essere NP-completo.

Nel nostro caso, dato un grafo $G = (V, E)$, il concetto di similitudine si manifesta nell'equivalenza tra il fatto che un sottoinsieme dei vertici è un insieme indipendente di G e quello che il suo complemento è un ricoprimento tramite vertici dello stesso G.

Teorema 8.6 *L'insieme $V' \subseteq V$ è un insieme indipendente di G se e solo se $V - V'$ è un ricoprimento tramite vertici di G.*

Dimostrazione Se V' è un insieme indipendente, allora $V - V'$ è un ricoprimento in quanto, se così non fosse, esisterebbe un arco $(u, v) \in E$ tale che $u \notin V - V'$ e $v \notin V - V'$: quindi, esisterebbe un arco $(u, v) \in E$ tale che $u \in V'$ e $v \in V'$ contraddicendo l'ipotesi che V' è un insieme indipendente. Viceversa, se $V - V'$ è un ricoprimento tramite vertici, allora V' è un insieme indipendente in quanto, se così non fosse, esisterebbe un arco $(u, v) \in E$ tale che $u \in V'$ e $v \in V'$: quindi, esisterebbe un arco $(u, v) \in E$ tale che $u \notin V - V'$ e $v \notin V - V'$ contraddicendo l'ipotesi che $V - V'$ è un ricoprimento. ❑

ESEMPIO 8.9

Nel caso della Figura 8.3, abbiamo che l'insieme formato dai vertici v_1, v_4 e v_5 è un ricoprimento tramite vertici, mentre l'insieme complementare formato dai vertici v_0, v_2 e v_3 è un insieme indipendente.

Pertanto, il problema del minimo ricoprimento tramite vertici è trasformabile in quello del massimo insieme indipendente e viceversa.

Notiamo che il problema del massimo insieme indipendente ammette dimostrazioni brevi e verificabili in tempo polinomiale, che altro non sono se non i sottoinsiemi di vertici che formano un insieme indipendente. Abbiamo pertanto aggiunto, alla nostra lista di problemi NP-completi, il problema del massimo insieme indipendente.

8.8.4 Tecnica di restrizione

L'ultimo esempio di dimostrazione di NP-completezza che forniamo si basa sulla tecnica più semplice in assoluto, detta della **restrizione**, che consiste nel mostrare come un problema già noto essere NP-completo sia un caso speciale di un altro problema in NP: da ciò ovviamente deriva la NP-completezza di quest'ultimo.

Come esempio, consideriamo il problema del **minimo insieme di campionamento**, che è definito nel modo seguente: dato un insieme C di sottoinsiemi di un insieme A e dato un numero intero $k \geq 0$, esiste un sottoinsieme A' di A tale che $|A'| \leq k$ e A' è un campionamento di C ovvero, per ogni insieme $c \in C$, $c \cap A' \neq \phi$? Possiamo restringere questo problema a quello del minimo ricoprimento tramite vertici, limitandoci a considerare istanze in cui ciascun elemento di C contiene esattamente due elementi di A: intuitivamente, A corrisponde all'insieme dei vertici del grafo e C all'insieme degli archi.

Poiché il problema del minimo ricoprimento tramite vertici è NP-completo e poiché quello del minimo insieme di campionamento ammette dimostrazioni brevi e verificabili in tempo polinomiale (costituite dal campione A'), abbiamo

che anche quest'ultimo problema è NP-completo: d'altra parte, se riuscissimo a progettare un algoritmo polinomiale per questo, potremmo ugualmente risolvere il problema del minimo ricoprimento tramite vertici in tempo polinomiale, applicando tale algoritmo alle sole istanze ristrette.

8.9 Come dimostrare risultati di NP-completezza

Il concetto di NP-completezza è stato introdotto alla metà degli anni '70. Da allora, migliaia di problemi computazionali sono stati dimostrati essere NP-completi, di tipologie diverse e provenienti da molte aree applicative. Un punto cruciale nel cercare di dimostrare che un nuovo problema Π è NP-completo consiste nella scelta del problema da cui partire, ovvero il problema NP-completo Π' che deve essere trasformato in Π (notiamo che un tipico errore che si commette inizialmente è quello di pensare che Π deve essere trasformato in Π' e non viceversa). A tale scopo, nel loro libro *Algorithm Design*, Jon Kleinberg e Eva Tardos identificano i seguenti sei tipi primitivi di problema, suggerendo per ciascuno uno o più potenziali candidati a svolgere il ruolo del problema computazionale Π'.

Problemi di sottoinsiemi massimali. Dato un insieme di oggetti, cerchiamo un suo sottoinsieme di cardinalità massima che soddisfi determinati requisiti: un tipico esempio di problemi siffatti è il problema del massimo insieme indipendente.

Problemi di sottoinsiemi minimali. Dato un insieme di oggetti, cerchiamo un suo sottoinsieme di cardinalità minima che soddisfi determinati requisiti: due tipici esempi di problemi siffatti sono il problema del minimo ricoprimento tramite vertici e quello del minimo insieme di campionamento.

Problemi di partizionamento. Dato un insieme di oggetti, cerchiamo una sua partizione nel minor numero possibile di sottoinsiemi disgiunti che soddisfino determinati requisiti (in alcuni casi, viene anche richiesto che i sottoinsiemi della partizione siano scelti tra una collezione specificata nell'istanza del problema): un tipico esempio di problemi siffatti è il problema della colorazione di grafi.

Problemi di ordinamento. Dato un insieme di oggetti, cerchiamo un suo ordinamento che soddisfi determinati requisiti: tipici esempi di problemi di questo tipo sono il problema del circuito hamiltoniano e quello del commesso viaggiatore (di cui parleremo nel prossimo paragrafo).

Problemi numerici. Dato un insieme di numeri interi, cerchiamo un suo sottoinsieme che soddisfi determinati requisiti: tipici esempi di problemi di questo tipo sono il problema della partizione e quello della bisaccia. Notiamo che la difficoltà di questi problemi risiede principalmente nel dover trattare numeri arbitrariamente grandi: in effetti, i due problemi suddetti, ristretti a istanze in cui i numeri in gioco sono polinomialmente limitati rispetto alla lunghezza dell'istanza, sono risolvibili in tempo polinomiale (Paragrafo 6.5).

Problemi di soddisfacimento di vincoli. Dato un sistema di vincoli espressi, generalmente, mediante formule booleane o equazioni lineari su uno specifico insieme di variabili, cerchiamo un'assegnazione alle variabili che soddisfi il sistema: un tipico esempio di problemi siffatti è il problema della soddisfacibilità (eventualmente ristretto a istanze con clausole contenenti esattamente tre letterali).

Una volta scelto il problema Π' da cui partire, la progettazione della trasformazione di Π' in Π è un compito difficile tanto quanto quello di progettare un algoritmo polinomiale di risoluzione per Π: in effetti, in *Computers and Intractability. A Guide to the Theory of NP-Completeness*, Michael Garey e David Johnson suggeriscono di procedere parallelamente nelle due attività, in quanto le difficoltà che si incontrano nella progettazione di un algoritmo possono fornire suggerimenti alla progettazione della trasformazione e viceversa. Sebbene la capacità di dimostrare risultati di NP-completezza sia un'abilità che, una volta acquisita, può risultare poi di facile applicazione, non è certo possibile, come nel caso della capacità di sviluppare algoritmi efficienti, spiegarla in modo formale. Ciò nondimeno, Steven Skiena, nelle sue dispense di un corso su algoritmi, fornisce i seguenti suggerimenti di cui possiamo tener conto quando ci accingiamo a voler dimostrare che un dato problema è NP-completo:

- rendiamo Π' il più semplice possibile (per esempio, conviene usare 3-SAT invece di SAT);
- rendiamo Π il più difficile possibile, eventualmente aggiungendo (temporaneamente) vincoli ulteriori;
- identifichiamo in Π le soluzioni *canoniche* e introduciamo qualche forma di penalizzazione nei confronti di una qualunque soluzione che non sia canonica (per esempio, nel caso del problema del minimo ricoprimento tramite vertici, una soluzione canonica è formata da un vertice per ogni componente di verità e due vertici per ogni componente di soddisfacibilità);
- prima di produrre gadget (nel caso della tecnica della progettazione di componenti), ragioniamo ad alto livello chiedendoci cosa e come intendiamo fare per forzare a scegliere soluzioni canoniche.

Per quanto utili, questi suggerimenti sono abbastanza vaghi: nella realtà, non esiste altro modo di imparare a progettare trasformazioni tra problemi computazionali se non facendolo. Per questo motivo, in questo capitolo abbiamo preferito fornire pochi esempi di tali trasformazioni, lasciando al lettore il compito di cimentarsi con altri problemi, presi magari dalla lista di problemi NP-completi presenti nel libro di Garey e Johnson.

8.10 Algoritmi di approssimazione

Dimostrare che un problema è NP-completo significa rinunciare a progettare per esso un algoritmo polinomiale di risoluzione (a meno che non crediamo che P sia uguale a NP). A questo punto, però, ci chiediamo come dobbiamo comportarci: dopo tutto, il problema deve essere risolto. A questo interrogativo possiamo rispondere in diversi modi. Il primo e il più semplice, già trattato nel Paragrafo 8.7, consiste nell'ignorare la complessità temporale intrinseca del problema, sviluppare comunque un algoritmo di risoluzione e sperare che nella pratica, ovvero con istanze provenienti dal mondo reale, il tempo di risoluzione sia significativamente minore di quello previsto: dopo tutto, l'analisi nel caso pessimo, in quanto tale, non ci dice come si comporterà il nostro algoritmo nel caso di specifiche istanze.

Un secondo approccio, in linea con quello precedente, ma matematicamente più fondato, consiste nell'analizzare l'algoritmo da noi progettato nel caso medio rispetto a una specifica distribuzione di probabilità: questo è quanto abbiamo fatto nel caso dell'algoritmo di ordinamento per distribuzione (Paragrafo 3.4). Vi sono due tipi di problematiche che sorgono quando vogliamo perseguire tale approccio. La prima consiste nel fatto che un'analisi probabilistica del comportamento dell'algoritmo è quasi sempre difficile e richiede strumenti di calcolo delle probabilità talvolta molto sofisticati. La seconda e, probabilmente, più grave questione è che l'analisi probabilistica richiede la conoscenza della distribuzione di probabilità con cui le istanze si presentano nel mondo reale: purtroppo, quasi mai conosciamo tale distribuzione e, pertanto, siamo costretti a ipotizzare che essa sia una di quelle a noi più familiari, come la distribuzione uniforme.

Un terzo approccio si applica al caso di problemi di ottimizzazione, per i quali a ogni soluzione è associata una misura e il cui scopo consiste nel trovare una soluzione di misura ottimale: in tal caso, possiamo rinunciare alla ricerca di soluzioni ottime e accontentarci di progettare algoritmi efficienti che producano sì soluzioni peggiori, ma non troppo. In particolare, diremo che A è un **algoritmo di r-approssimazione** per il problema di ottimizzazione Π se, per ogni istanza x di Π, abbiamo che A con x in ingresso restituisce una soluzione di x la cui misura è al più r volte quella di una soluzione ottima (nel caso la misura sia un costo) oppure almeno $\frac{1}{r}$-esimo di quella di una soluzione ottima (nel caso la misura sia un profitto), dove r è una costante reale strettamente maggiore di 1.

Per chiarire meglio tale concetto, consideriamo il problema del minimo ricoprimento tramite vertici, che nella sua versione di ottimizzazione consiste nel trovare un sottoinsieme dei vertici di un grafo di cardinalità minima che copra tutti gli archi del grafo stesso. Il Codice 8.4 realizza un algoritmo di approssimazione per tale problema basato sul paradigma dell'algoritmo goloso. In particolare, dopo aver inizializzato la soluzione ponendola uguale all'insieme vuoto (righe 3 e 4), il codice esamina uno dopo l'altro tutti gli archi del grafo (righe 5-11): ogni qualvolta ne trova uno i cui due estremi non sono stati selezionati (riga 7), include entrambi gli estremi nella soluzione (righe 8 e 9).

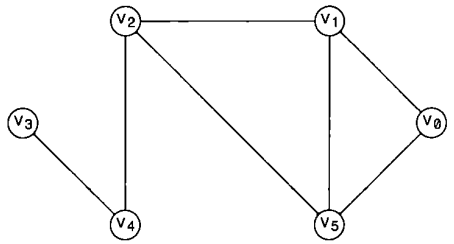
Codice 8.4 Algoritmo per il calcolo di un ricoprimento tramite vertici.

```
1  RicoprimentoVertici( A ):
2                          ⟨pre: A è la matrice di adiacenza di un grafo di n nodi⟩
3    FOR (i = 0; i < n; i = i + 1)
4      preso[i] = FALSE;
5    FOR (i = 0; i < n; i = i + 1)
6      FOR (j = 0; j < n; j = j + 1) {
7        IF (A[i][j] == 1 && !preso[i] && !preso[j]) {
8          preso[i] = TRUE;
9          preso[j] = TRUE;
10       }
11     }
12   RETURN preso;
```

ESEMPIO 8.10

Considerando il grafo mostrato nella Figura 8.3, la soluzione prodotta dall'algoritmo include tutti e sei i vertici; infatti prima vengono inseriti nella soluzione i vertici v_0 e v_1, in quanto (v_0, v_1) non è coperto, poi la coppia v_2 e v_4 e infine v_3 e v_4, poiché l'arco (v_3, v_5) risulta ancora scoperto.

Dovrebbe essere evidente che la soluzione costruita dal Codice 8.4 dipende dall'ordine in cui si elaborano gli archi. Infatti applichiamo lo stesso algoritmo al grafo seguente che è un isomorfismo di quello rappresentato nella Figura 8.3.



Nella soluzione entrano prima i nodi v_0 e v_1 e poi i nodi v_2 e v_4, in quanto l'arco (v_2, v_4) sarà il primo arco esaminato incidente in v_2 e non ancora coperto. In questo caso il ricoprimento finale ha dimensione 4 mentre quello di cardinalità minima ha solamente tre nodi.

Non possiamo garantire che tale soluzione sia di cardinalità minima, ma possiamo però mostrare che la soluzione calcolata dal Codice 8.4 include un numero di vertici che è sempre minore oppure uguale al doppio della cardinalità di una soluzione ottima.

Teorema 8.7 *Il Codice 8.4 è un algoritmo di 2-approssimazione per il problema del minimo ricoprimento tramite vertici.*

Dimostrazione Dato un grafo G, la soluzione S prodotta dall'algoritmo è un ricoprimento tramite vertici, poiché ogni arco viene coperto con due vertici se, al momento in cui viene esaminato, questi sono entrambi non inclusi nella soluzione e con almeno un vertice in caso contrario.

Il sottografo indotto dalla soluzione S calcolata dall'algoritmo con G in ingresso, è formato da $\frac{|S|}{2}$ archi a due a due disgiunti, ovvero senza estremi in comune. Chiaramente, un qualunque ricoprimento di tale sottografo (e quindi di G) deve includere almeno $\frac{|S|}{2}$ vertici: pertanto, |S| è minore oppure uguale al doppio della cardinalità di un qualunque ricoprimento tramite vertici di G e, quindi, della cardinalità minima. ❑

La complessità temporale del Codice 8.4 è $O(n^2)$, in quanto ogni iterazione dei due cicli annidati uno dentro l'altro richiede un numero costante di operazioni.

8.11 Opus libri: il problema del commesso viaggiatore

Concludiamo questo capitolo e il libro con un'ultima opera algoritmica, relativa a uno dei problemi di ottimizzazione più analizzati (in tutte le sue varianti) nel campo dell'informatica e della ricerca operativa, ovvero il **problema del commesso viaggiatore**.

Dato un insieme di città e specificato, per ogni coppia di città, la distanza chilometrica per andare dall'una all'altra o viceversa, un commesso viaggiatore si chiede quale sia il modo più breve per visitare tutte le città una e una sola volta, tornando al termine del giro alla città di partenza.

ESEMPIO 8.11

Consideriamo la seguente istanza del problema in cui 9 città olandesi sono analizzate e in cui le distanze chilometriche sono tratte da una nota guida turistica internazionale:

	A	B	D	E	H	L	M	R	U
Amsterdam	0	101	98	121	20	55	213	73	37
Breda		0	30	57	121	72	146	51	73
Dordrecht			0	92	94	45	181	24	61
Eindhoven				0	136	134	89	113	88
Haarlem					0	51	228	70	54
L'Aia						0	223	21	62
Maastricht							0	202	180
Rotterdam								0	57
Utrecht									0

Se il commesso viaggiatore decide di percorrere le città secondo il loro ordine alfabetico, allora percorre un numero di chilometri pari a

$$101 + 30 + 92 + 136 + 51 + 223 + 202 + 57 + 37 = 929$$

Supponiamo, invece, che decida di percorrerle nel seguente ordine: Amsterdam, Haarlem, L'Aia, Rotterdam, Dordrecht, Breda, Maastricht, Eindhoven e Utrecht. In tal caso, il commesso viaggiatore percorre un numero di chilometri pari a

$$20 + 51 + 21 + 24 + 30 + 146 + 89 + 88 + 37 = 506$$

Mediante una ricerca esaustiva di tutte le possibili 9! = 362880 permutazioni delle nove città, possiamo verificare che quest'ultima è la soluzione migliore possibile.

Sfortunatamente, il commesso viaggiatore non ha altra scelta che applicare un algoritmo esaustivo per trovare la soluzione al suo problema, in quanto la sua versione decisionale è un problema NP-completo. Più precisamente, consideriamo il seguente problema di decisione: dati un grafo completo G = (V, E), una funzione p che associa a ogni arco del grafo un numero intero non negativo e un numero intero k ≥ 0, esiste un *tour* del commesso viaggiatore di peso non superiore a k, ovvero un ciclo hamiltoniano in G (Paragrafo 7.1.1) la somma dei cui archi è minore oppure uguale a k?

Per dimostrare che tale problema è NP-completo, consideriamo il problema del circuito hamiltoniano che consiste nel decidere se un grafo qualsiasi include un ciclo hamiltoniano. Utilizzando la tecnica della progettazione di componenti possiamo dimostrare che tale problema è NP-completo.

Teorema 8.8 *Ciclo hamiltoniano è trasformabile in tempo polinomiale nella versione decisionale del problema del commesso viaggiatore.*

Dimostrazione Dato un grafo G = (V, E), definiamo un grafo completo G′ = (V, E′) e una funzione p tale che, per ogni arco e in E′, p(e) = 1 se e ∈ E, altrimenti p(e) = 2. Scegliendo k = |V|, abbiamo che se esiste un ciclo hamiltoniano in G, allora esiste un tour in G′ il cui costo è uguale a k. Viceversa, se non esiste un ciclo hamiltoniano in G, allora ogni tour in G′ deve includere almeno un arco il cui peso sia pari a 2, per cui ogni tour ha un costo almeno pari a k + 1. ❑

Conseguenza del risultato appena provato è che il problema del commesso viaggiatore (nella sua forma decisionale) è NP-completo.

Sfortunatamente, possiamo mostrare che il problema di ottimizzazione non ammette neanche un algoritmo efficiente di approssimazione. A tale scopo, consideriamo nuovamente il problema del circuito hamiltoniano e, facendo uso della tecnica detta del **gap**, dimostriamo che se il problema del commesso viaggiatore ammette un algoritmo efficiente di approssimazione, allora il problema del circuito hamiltoniano è risolvibile in tempo polinomiale.

Teorema 8.9 *Sia* $r > 1$ *una qualsiasi costante, non esiste alcun algoritmo di* r*-approssimazione di complessità polinomiale per il problema del commesso viaggiator a meno che NP coincida con P.*

Dimostrazione Per assurdo, sia $r > 1$ una costante e sia A un algoritmo polinomiale di r-approssimazione per il problema del commesso viaggiatore. Dato un grafo $G = (V, E)$, definiamo un grafo completo $G' = (V, E')$ e una funzione p tale che, per ogni arco e in E', $p(e) = 1$ se $e \in E$, altrimenti $p(e) = 1 + s|V|$ dove $s > r - 1$. Notiamo che G' ammette un tour del commesso viaggiatore di costo pari a $|V|$ se e solo se G include un circuito hamiltoniano: infatti, un tale tour deve necessariamente usare archi di peso pari a 1, ovvero archi contenuti in E.

Sia T il tour del commesso viaggiatore che viene restituito da A con G' in ingresso. Dimostriamo che T può essere usato per decidere se G ammette un ciclo hamiltoniano, distinguendo i seguenti due casi.

1. Il costo di T è uguale a $|V|$, per cui T è un tour ottimo. Quindi, G ammette un ciclo hamiltoniano.
2. Il costo di T è maggiore di $|V|$, per cui il suo costo deve essere almeno pari a $|V| - 1 + 1 + s|V| = (1 + s)|V| > r|V|$. In questo caso, il tour ottimo non può avere costo pari a $|V|$, in quanto altrimenti il costo di T è maggiore di r volte il costo ottimo, contraddicendo il fatto che A è un algoritmo di r-approssimazione: quindi, non esiste un ciclo hamiltoniano in G.

In conclusione, applicando l'algoritmo A al grafo G' e verificando se la soluzione restituita da A ha un costo pari oppure maggiore al numero dei vertici, possiamo decidere in tempo polinomiale se G ammette un circuito hamiltoniano, da cui si deduce che il problema del circuito hamiltoniano è in P e quindi P coincide con NP. ❑

8.11.1 Problema del commesso viaggiatore su istanze metriche

Sebbene il problema del commesso viaggiatore non sia, in generale, risolvibile in modo approssimato mediante un algoritmo polinomiale, possiamo mostrare che tale problema, ristretto al caso in cui la funzione che specifica la distanza tra due città soddisfi la disuguaglianza triangolare, ammette un algoritmo di 2-approssimazione.

Un'istanza del problema del commesso viaggiatore soddisfa la **disuguaglianza triangolare** se, per ogni tripla di vertici i, j e k, $p(i, j) \leq p(i, k) + p(k, j)$: intuitivamente, ciò vuol dire che andare in modo diretto da una città i a una città j non può essere più costoso che andare da i a j passando prima per un'altra città k. L'istanza delle città olandesi dell'Esempio 8.11 soddisfa la disuguaglianza triangolare, anche se tale disuguaglianza non sempre è soddisfatta quando si tratta di distanze stradali.

La disuguaglianza triangolare è invece sempre soddisfatta se i vertici del grafo rappresentano punti del piano euclideo e le distanze tra due vertici corrispondono alle loro distanze nel piano, in quanto in ogni triangolo la lunghezza di un lato è sempre minore della somma delle lunghezze degli altri due lati. Inoltre, la disuguaglianza è soddisfatta nel caso in cui il grafo completo G sia ottenuto nel modo seguente, a partire da un grafo G' connesso non necessariamente completo: G ha gli stessi vertici di G' e la distanza tra due suoi vertici è uguale alla lunghezza del cammino minimo tra i corrispondenti vertici di G'. Per questo motivo, la risoluzione del problema del commesso viaggiatore, ristretto al caso di istanze che soddisfano la disuguaglianza triangolare, è un problema di per sé interessante che sorge abbastanza naturalmente in diverse aree applicative.

La versione decisionale di tale problema è NP-completo, in quanto la trasformazione a partire dal problema del circuito hamiltoniano nella dimostrazione del Teorema 8.8 genera istanze che soddisfano la disuguaglianza triangolare: se i pesi degli archi sono solo 1 e 2, ovviamente tale disuguaglianza è sempre soddisfatta. Mostriamo ora un algoritmo polinomiale di 2-approssimazione per il problema del commesso viaggiatore, ristretto al caso di istanze che soddisfano la disuguaglianza triangolare.

L'idea alla base dell'algoritmo è che la somma dei pesi degli archi di un minimo albero ricoprente R di un grafo completo G costituisce un limite inferiore al costo di un tour ottimo. Infatti, cancellando un arco di un qualsiasi tour T, otteniamo un cammino hamiltoniano e, quindi, un albero ricoprente: la somma dei pesi degli archi di questo cammino deve essere, per definizione, non inferiore a quella dei pesi degli archi di R. Quindi, il costo di T (che include anche il peso dell'arco cancellato) è certamente non inferiore alla somma dei pesi degli archi di R.

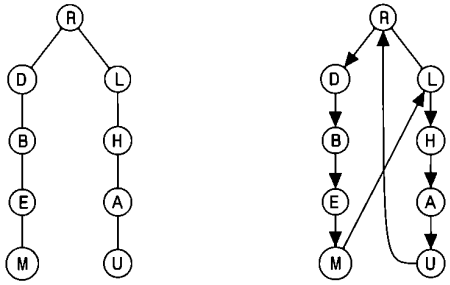
L'algoritmo (realizzato nel Codice 8.5) costruisce un minimo albero ricoprente di G (riga 3) e restituisce i nodi in un array `visitato` in cui questi compaiono secondo l'ordine di una visita DFS. Si ottiene un tour del grafo supponendo implicitamente che il vertice in ultima posizione (ovvero, posizione `n - 1`) deve essere connesso a quello in prima posizione (ovvero, posizione `0`).

Codice 8.5 Algoritmo per il calcolo di un tour approssimato del commesso viaggiatore.

```
1  CommessoViaggiatore( P ):
2        ⟨pre: P è la matrice di adiacenza e dei pesi di un grafo G completo di n nodi⟩
3  mst = Jarník-Prim( P );
4  visitato = DFS( mst );
5  RETURN visitato;
```

ESEMPIO 8.12

Nella parte sinistra della seguente figura è rappresentato il minimo albero ricoprente del grafo dell'Esempio 8.11.



Se la visita dell'albero inizia dal nodo R, il Codice 8.5 restituisce la sequenza (R, D, B, E, M, L, H, A, U) che corrisponde al tour mostrato nella parte destra della figura in alto evidenziato con archi diretti.

Poiché il calcolo del minimo albero ricoprente e l'esecuzione della visita dell'albero possono essere realizzati in $O(n^2 \log n)$, abbiamo che l'algoritmo appena descritto è polinomiale. Inoltre vale il seguente risultato.

Teorema 8.10 *Il Codice 8.5 è un algoritmo di 2-approssimazione per il problema del commesso viaggiatore metrico.*

Dimostrazione Quello restituito è un tour in quanto ogni nodo compare una volta soltanto e nodi consecutivi sono collegati da un arco in quanto il grafo è completo.

Ora consideriamo un arco $e = (u, v)$ del tour T che non appartiene al minimo albero ricoprente R: questo è un arco trasversale oppure, nel caso in cui sia l'ultimo arco del tour, all'indietro (si veda il Paragrafo 7.2.2). Sia R_e l'insieme degli archi appartenenti al percorso tra u e v nel minimo albero di ricoprimento. A causa della disuguaglianza triangolare la somma dei pesi degli archi in R_e non può essere inferiore al peso dell'arco e. Inoltre se e' è un altro arco di T ma non in R e diverso da e, allora $R_e \cap R_{e'} = \phi$, in quanto, altrimenti, l'algoritmo DFS avrebbe visitato una componente dell'albero più di una volta. Riassumendo

$$\sum_{e \in T} p(e) \leq \sum_{e \in R} p(e) + \sum_{e \in T-R} p(e) \leq \sum_{e \in R} p(e) + \sum_{e \in R_e} p(e) \leq 2 \sum_{e \in R} p(e).$$

Dove l'ultima disuguaglianza segue perché gli R_e sono disgiunti e la penultima dalla disuguaglianza triangolare. Infine giungiamo al risultato cercato tenendo conto che il costo del minimo albero di ricoprimento non può essere maggiore del costo del tour ottimale. ❑

Per concludere, osserviamo che l'algoritmo di approssimazione realizzato dal Codice 8.5 non è il migliore possibile. In effetti, con un opportuno accorgimento nello scegliere gli archi del minimo albero ricoprente da duplicare, possiamo modificare tale algoritmo ottenendone uno di 1,5-approssimazione. Inoltre, nel caso di istanze formate da punti sul piano euclideo, possiamo dimostrare che, per ogni $r > 1$, esiste un algoritmo polinomiale di r-approssimazione: in altre parole, il problema del commesso viaggiatore sul piano può essere approssimato tanto bene quanto vogliamo (ovviamente al prezzo di una complessità temporale che, pur mantenendosi polinomiale, cresce al diminuire di r).

8.11.2 Paradigma della ricerca locale

Un algoritmo per la risoluzione di un problema di ottimizzazione basato sul **paradigma della ricerca locale** opera nel modo seguente: a partire da una soluzione iniziale del problema, esplora un insieme di soluzioni "vicine" a quella corrente e si sposta in una soluzione che è migliore di quella corrente, fino a quando non giunge a una che non ha nessuna soluzione vicina migliore. Pertanto, il comportamento di un tale algoritmo dipende dalla nozione di vicinato di una soluzione (solitamente generato applicando operazioni di cambiamento locale alla soluzione corrente), dalla soluzione iniziale (che può essere calcolata mediante un altro algoritmo) e dalla strategia di selezione delle soluzioni (per esempio, scegliendo la prima soluzione vicina migliore di quella corrente oppure selezionando la migliore tra tutte quelle vicine alla corrente).

Non esistono regole generali per decidere quali siano le regole di comportamento migliori: per questo, ci limitiamo in questo paragrafo finale a descrivere due algoritmi basati sul paradigma della ricerca locale per la risoluzione (non ottima) del problema del commesso viaggiatore. Notiamo sin d'ora che non siamo praticamente in grado di formulare nessuna affermazione (non banale) relativamente alle prestazioni di questi algoritmi né in termini di complessità temporale né in termini di qualità della soluzione ottenuta. Tuttavia, questo tipo di strategie (dette anche **euristiche**) risultano nella pratica estremamente valide e, per questo, molto utilizzate.

Entrambi gli algoritmi che descriviamo fanno riferimento a operazioni locali di cambiamento: essi tuttavia differiscono tra di loro per quello che riguarda la lunghezza massima della sequenza di tali operazioni. In particolare, i due algoritmi modificano la soluzione corrente selezionando un numero fissato di archi e sostituendoli con un altro insieme di archi (della stessa cardinalità) in modo da ottenere un nuovo tour.

Il primo algoritmo, detto **2-opt**, opera nel modo seguente: dato un tour T del commesso viaggiatore, il suo vicinato è costituito da tutti i tour che possono essere ottenuti cancellando due archi (x, y) e (u, v) di T e sostituendoli con due nuovi archi (x, u) e (y, v) in modo da ottenere un tour differente T' (notiamo che

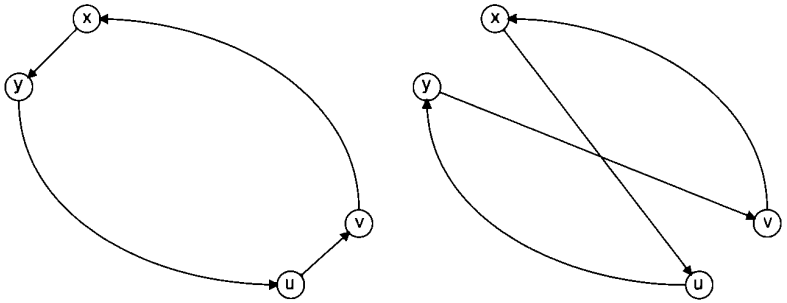


Figura 8.5 L'operazione di modifica di un tour realizzata dall'algoritmo 2-opt.

questo equivale a invertire la percorrenza di una parte del tour T, come mostrato nella Figura 8.5).

Se il nuovo tour T′ ha un costo minore di quello di T, allora T′ diviene la soluzione corrente, altrimenti l'algoritmo procede con una diversa coppia di archi: il procedimento ha termine nel momento in cui giungiamo a un tour che non può essere migliorato.

Il Codice 8.6 realizza l'algoritmo 2-opt. Dopo aver inizializzato il tour iniziale e il relativo costo, visitando i vertici nell'ordine in cui appaiono nel grafo (righe 3-5), il codice esamina tutte le coppie di archi `(tour[i], tour[i + 1])` e `(tour[j], tour[j + 1])` con $0 \le i < j - 1 < n - 1$ (righe 6-26) e per ognuna di esse genera il nuovo tour operando la sostituzione precedentemente descritta (righe 8-15). Quindi, il codice calcola il costo del nuovo tour (righe 16-19): se tale costo è minore del costo precedente, allora il tour corrente viene aggiornato e il ciclo `for` più esterno viene fatto ripartire dall'inizio (righe 21 e 23). Se non troviamo nessun tour migliore di quello attuale, allora il codice restituisce il tour attuale come soluzione del problema (riga 27).

Codice 8.6 Algoritmo 2-opt.

```
1  2-Opt( P ):
2        ⟨pre: P è la matrice di adiacenza e dei pesi di un grafo G completo di n nodi⟩
3    costo = 0;
4    FOR (i = 0; i < n; i = i +1)
5      { tour[i] = i; costo = costo + P[i][(i+1) % n]; }
6    FOR (i = 0; i < n; i = i +1) {
7      FOR (j = i+2; j < n-1; j = j + 1) {
8        FOR (h = 0; h <= i; h = h + 1)
9          nuovo[h] = tour[h];
10       nuovo[i+1] = tour[j];
11       FOR (h = 1; h < j-i; h = h + 1)
12         nuovo[i+1+h] = tour[j-h];
13       nuovo[j+1] = tour[j+1];
14       FOR (h = j+2; h < n; h = h + 1)
15         nuovo[h] = tour[h];
16       nuovoCosto = 0;
17       FOR (h = 0; h < n; h = h + 1) {
18         nuovoCosto = nuovoCosto + P[nuovo[h]][nuovo[(h+1) % n]];
19       }
20       IF (nuovoCosto < costo) {
21         { costo = nuovoCosto; i = 0; }
22         FOR (h = 0; h < n; h = h+1)
23           tour[h] = nuovo[h];
24       }
25     }
26   }
27   RETURN tour;
```


Poiché, ogni qualvolta viene trovato un tour di costo minore, tale costo diminuisce almeno di un'unità, abbiamo che il numero totale di iterazioni del ciclo `for` più esterno è limitato dal costo del tour iniziale: pertanto, il Codice 8.6 termina in tempo $O(n^3C)$ dove C indica il costo del tour iniziale.

Il secondo algoritmo di risoluzione del problema del commesso viaggiatore basato sul paradigma della ricerca locale è detto **3-opt**, in quanto opera in modo analogo a 2-opt, ma considera come vicinato di un tour T tutti i tour che possono essere ottenuti scambiando tre archi di T.

In particolare, se $i_0, i_1, \ldots, i_{n-1}$ è il tour corrente, l'algoritmo 3-opt sceglie tre indici j_0, j_1 e j_2 con $j_0 < j_1 - 1 < j_2 - 2$ e sostituisce i tre archi (i_{j_0}, i_{j_0+1}), (i_{j_1}, i_{j_1+1}) e (i_{j_2}, i_{j_2+1}) con gli archi (i_{j_0}, i_{j_1+1}), (i_{j_2}, i_{j_0+1}) e (i_{j_1}, i_{j_2+1}) (notiamo che in questo caso non abbiamo bisogno di invertire una parte del tour corrente, come mostrato nella Figura 8.6).

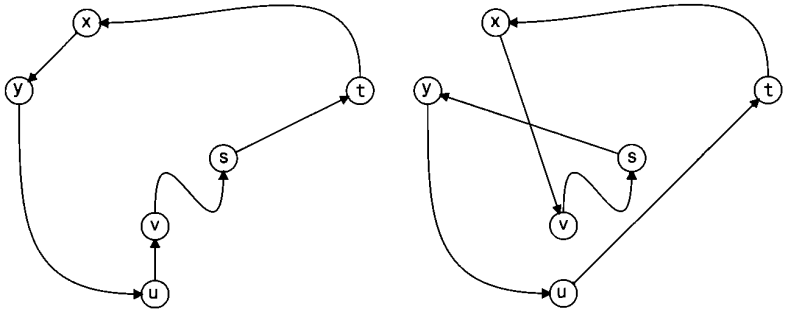


Figura 8.6 L'operazione di modifica di un tour realizzata dall'algoritmo 3-opt.

Possiamo verificare sperimentalmente che 3-opt ha delle prestazioni migliori di 2-opt per quello che riguarda la qualità della soluzione, ma richiede un tempo di calcolo superiore. Sebbene, in linea di principio, possiamo pensare di generalizzare i due algoritmi appena esposti definendo una strategia k-opt per qualunque $k \geq 2$, il miglioramento che si ottiene nella qualità della soluzione calcolata nel passare da 3-opt a 4-opt non sembra giustificare il significativo peggioramento delle prestazioni in termini di tempo di esecuzione.

ESEMPIO 8.13

Per meglio chiarire il funzionamento dell'algoritmo descritto nel Codice 8.6 descriviamo la creazione di un nuovo tour a partire da `tour` ottenuto scambiando gli archi `(tour[i], tour[i + 1])` e `(tour[j], tour[j + 1])` con gli archi `(tour[i], tour[j])` e `(tour[i + 1], tour[j + 1])`. Questa operazione viene eseguita all'interno del blocco di istruzioni tra la riga 8 e la riga 15 in cui vale che $j \geq i + 2$. Per comodità scomponiamo `tour` nel seguente modo:

$$\texttt{tour} = \alpha_0 x y \alpha_1 u v \alpha_2.$$

dove α_0 è la sequenza `tour[0] ... tour[i - 1]`, α_1 è la sequenza `tour[i + 2] ... tour[j - 1]` e α_2 è la sequenza `tour[j + 2] ... tour[n - 1]`.

Il ciclo nella riga 8 non fa altro che copiare la sequenza $\alpha_0 x$ nella prima parte dell'array `nuovo`, mentre l'istruzione successiva copia `u` in `nuovo[i + 1]`. Dopo questa prima fase nuovo = $\alpha_0 x u$. Il ciclo successivo, per $1 \leq h \leq j - i - 1$ copia in `nuovo[i + 1 + h]` il valore in `tour[j - h]` ovvero in `nuovo[i + 2]` copia `tour[j - 1]`, in `nuovo[i + 3]` copia `tour[j - 2]` e così via fino a copiare in `nuovo[j]` il valore in `tour[i + 1] = y`. L'istruzione successiva nella riga 13 copia `v` in posizione `j + 1` di nuovo. Dopo questa fase nuovo = $\alpha_0 x u \bar{\alpha}_1 y v$ dove $\bar{\alpha}_1$ indica la sequenza α_1 invertita. Il ciclo successivo esegue la copia di α_2 nelle ultime posizioni libere di nuovo che, alla fine, apparirà come segue

$$\text{nuovo} = \alpha_0 x u \bar{\alpha}_1 y v \alpha_2.$$

Le operazioni che seguono non fanno altro che calcolare il costo del nuovo tour che, se minore di quello vecchio, verrà preso come soluzione attuale.

Gli algoritmi 2-opt e 3-opt risultano efficaci nella pratica, ma possono avere prestazioni molto scarse nel caso pessimo (anche in dipendenza della scelta del tour iniziale): in effetti, non conosciamo alcun limite superiore all'approssimazione raggiunta dalla soluzione calcolata da questi algoritmi nel caso generale.

Ciò nonostante, il fatto che in situazioni reali i due algoritmi si comportino relativamente bene fa sì che essi, insieme ad altre euristiche basate sul paradigma della ricerca locale, siano diffusamente utilizzati.

8.12 Esercizi

8.1 Facendo riferimento alla rappresentazione mediante liste di adiacenza e utilizzando una variante della procedura di visita in ampiezza di un grafo vista nel Paragrafo 7.2.1, mostrare che il problema di decidere se un grafo sia colorabile con due colori è risolvibile in tempo $O(n + m)$, dove n e m indicano, rispettivamente, il numero di nodi e il numero di archi del grafo.

8.2 Osservando che, date due formule booleane φ e φ' che non contengono la variabile booleana x, $\varphi \vee \varphi'$ è soddisfacibile se e solo se $(\varphi \vee x) \wedge (\varphi' \vee \bar{x})$, mostrare che una formula booleana ψ può essere trasformata in tempo polinomiale in una formula booleana ψ' in forma normale congiuntiva, tale che ψ è soddisfacibile se e solo se ψ' è soddisfacibile.

8.3 Mostrare che il problema di decidere se, dato un insieme di clausole ciascuna in forma congiuntiva, una di esse sia soddisfacibile, può essere risolto in tempo lineare.

8.4 Sia Π un problema di ottimizzazione tale che, per ogni sua istanza x di dimensione n, la misura della soluzione ottima è limitata da 2^n. Utilizzando la tecnica della ricerca binaria, dimostrare che se il problema di decisione associato a Π è in P, allora Π è risolvibile in tempo polinomiale.

8.5 Mostrare mediante la tecnica di progettazione delle componenti che il seguente problema è NP-completo: dato un grafo $G = (V, E)$, esiste una colorazione dei vertici in V mediante tre colori?

8.6 Dato un grafo G, sia $G^c = (V, E^c)$ il grafo *complementare* di G, tale che $(u, v) \in E^c$ se e solo se $(u, v) \notin E$. Facendo uso di tale nozione, mostrare per similitudine che il seguente problema è NP-completo: dato un grafo $G = (V, E)$ e un intero $k \geq 0$, esiste un sottoinsieme V' di V con $|V'| \geq k$, tale che V' induce un grafo completo in G ovvero, per ogni $u, v \in V'$, $(u, v) \in E$?

8.7 Considerare una variante del Codice 8.4 in cui, ogni qualvolta esaminiamo un arco i cui estremi non sono inclusi nella soluzione, decidiamo di inserire nella soluzione uno solo dei due estremi (scelto a piacere). Mostrare che non è un algoritmo di approssimazione per il problema del minimo ricoprimento tramite vertici.

8.8 Scrivere un codice che implementa l'algoritmo 3-opt.

Indice analitico

1-bilanciato, 119
2-opt, 303
3-opt, 305

A

abbinamento, 210
accesso
  diretto, 3, 7, 9, 18
  sequenziale, 7, 8, 18
accoppiamento perfetto, 210
aggregazioni, 113, 168
alberi
  altezza, 46
  AVL, 118
  BFS, 223
  binari, 7, 23, 25, 46, 48, 88, 114
    1-bilanciato, 119
    completi a sinistra, 46
    di ricerca, 114, 118
  cardinali, 25, 26, 27, 140
  dei cammini minimi, 247, 268
  DFS, 225, 301
  di Fibonacci, 119, 146
  di ricerca ottimi, 197
  di ricoprimento, 223, 258, 261
  euclidei, 258
  memorizzazione binarizzata, 27
  minimo ricoprimento, 195
    tramite vertici, 289
  ordinali, 26, 27
algoritmi
  2-opt, 303
  3-opt, 305
  di approssimazione, 177, 272
  di Bellman-Ford, 251
  di Dijkstra, 244
  di distribuzione, 71, 148
    in loco, 70
  di Floyd-Warshall, 255
  di fusione, 62, 128, 151
    in loco, 60
  di Jarník-Prim, 264
  di Kruskal, 261
  di ordinamento stabile, 28
  di r-approssimazione, 296
  goloso, 190, 191, 296
  in linea, 162
  non in linea, 163
  offline, 163
  online, 162
  polinomiale, 272
  pseudo-polinomiale, 200
  random, 152, 154
all pairs shortest path, 243
analisi ammortizzata, 160, 167
approssimazione, 80, 272
architettura di von Neumann, 3
arco, 204
  back, 228
  cross, 228
  estremi di un, 204
  forward, 228
  terminali di un, 204
array, 8
  bidimensionale, 77, 190
auto-organizzazione, 162, 167

B

backtrack, 286
backup, file di, 185
BFS, 219
bisaccia, 190, 294
bitmap, 125
Breadth-First Search, 219

C

cammino, 216
  euleriano, 211
  hamiltoniano, 211
  minimo, 207
campionamento, 293
casualità, 148
chaining, 108
chiusura transitiva, 216
ciclo
  euleriano, 211
  hamiltoniano, 211
clausole, 277
clique, 208
cluster, 113, 257
cluster analysis, 257
collisione, 107, 113
componente
  connessa, 207
  fortemente connessa, 210
costo ammortizzato, 12, 50, 160, 161, 168
costo
  medio, 71, 111, 157
  ottimo, 172
crawler, 219
cricca, 208, 290

D

DAG, 230, 279
Depth-First Search, 225, 301
DFS, 225, 301
dimezzamento, 12
Directed Acyclic Graph, 230, 279
distance vector, 242
distanza pesata, 207
distribuzione, 71, 148
  in loco, 70
disuguaglianza triangolare, 300
divide et impera, 60

E

embedding planare, 215
euristiche, 303

F

Fibonacci
  alberi di, 119, 146
  numeri di, 119, 172
FIFO (First In First Out), 42
finger search, 131
foresta di ricoprimento, 258
forma normale
  congiuntiva, 277
  disgiuntiva, 284
formula booleana, 277
Fourier, trasformata veloce di, 77

G

gadget, 290
gap di complessità, 81, 299
generazione esaustiva, 285

geometria computazionale, 83
grafo
  aciclico, 248, 256, 258
  bipartito, 211, 215
  completo, 208, 215
  componente connessa di un, 207
  contrazione del, 215
  denso, 204
  diametro del, 224
  dimensione del, 204
  diretto aciclico, 279
  etichettato, 205
  fortemente connesso, 210
  isomorfo, 216
  ordine del, 204
  orientato (diretto), 218, 230
  pesato, 211, 213
  planare, 215
  sottografo, 207
    aciclico, 223
    indotto, 207, 298
    massimale, 207
  sparso, 204
  taglio (cut), 259
  visita del, 219
greedy, 177

H

hash, 105, 107, 108, 125
  open addressing, 110
  universo, 102, 105, 106
heap, 47
  rappresentazione implicita, 49
heapsort, 53
  in loco, 56
heaptree, 46, 48

I

information retrieval, 124, 132
in linea, 162
inorder, 91
insertion sort, 16
insieme
  convesso, 281
  indipendente, 193, 289
inversione, 164, 165

K

kernel, 114
knapsack, 188

L

lati, 204
LCS (Longest Common Subsequence), 180
letterali, 277
LIFO (Last In First Out), 32
limite
  inferiore, 6, 56, 81
  superiore, 6, 56, 67, 81, 156, 160
lista di adiacenza, 213
lista doppia, 21, 103, 104, 125
liste, 8
  a salti, 152
    altezza, 152
  auto-organizzazione delle, 162
  invertite (posting), 125, 126
LRU (Least Recently Used), 163

M

macchina di Turing, 276
macro-vertici, 233
massimo insieme indipendente, 193
master theorem, 63
matrici
  di adiacenza, 212
  moltiplicazione veloce di due, 81
memoization, 187
memoria
  principale, 61, 71, 114
  secondaria, 61, 114
  virtuale, 114
mergesort, 60, 128
minimo insieme
  convesso, 281
  di campionamento, 293
min-max heap, 57
motori di ricerca, 125
MTF (Move-To-Front), 162
multigrafo, 211

N

nodi, 204
  adiacenti, 208
  connessi, 207
  critici, 121
  interni, 23, 46, 93, 140, 254
  profondità dei, 46
non in linea, 163, 170
numeri di Fibonacci, 172

O

Odifreddi, Piergiorgio, 3
opus libri, 12, 35, 77, 83, 105, 114, 124, 218, 240, 257, 298
ordinamento
  randomizzato, 148
  topologico, 230, 279
ottimizzazione, 285
  problemi di, 172

P

partizione, 185
peer-to-peer, 106
perfect matching, 210
permutazione, 110
pivot, 70, 148, 152
postorder, 91
potenziale, 169
preorder, 91
probing, 110
problema
  computazionale, 210, 276, 287
  del commesso viaggiatore, 298
  della soddisfacibilità, 277
  di decisione, 276, 282
  di ottimizzazione, 188, 285
  intrinsecamente difficile, 272
  NP-completo, 200, 272
  polinomialmente trasformabile, 282
  scheduling, 12, 201
  sotto-problema, 60, 173
  sotto-sequenze, 69, 180
  zaino, 188
programmazione dinamica, 173

Q

quicksort, 69, 148, 152

R

raddoppio, 12
RAM (Random Access Machine), 3
recupero dei documenti, 124
relazioni di ricorrenza, 63, 150

ricerca
  binaria, 67, 108, 154
  locale, 303
ricoprimento tramite vertici, 195, 289
rotazioni, 121

S

selection sort, 14
self-adjusting o self-organizing, 162
sequenza
  LCS (Longest Common Subsequence), 180
  lineare, 7, 8, 13, 65, 77
    dinamica, 10
shortest path, 242, 243
single source shortest path, 244
skip list, 152
soddisfacibilità, 277
spanning tree, 223, 258

T

tecnica
  di restrizione, 293
  di similitudine, 292
  di sostituzione locale, 288
teorema
  di Cook-Levin, 283
  di Kuratowski-Pontryagin-Wagner, 215
  fondamentale delle ricorrenze, 63, 150, 151
tour, 299
Turing, macchina di, 276

U

union-find, 158

V

vertici, 204
visita, 219
  anticipata, 91
  in ampiezza, 219
  in profondità, 225
  posticipata, 91
  simmetrica, 91
von Neumann, architettura di, 3