Introduzione

Questa parte è un'introduzione alle tecniche di progettazione e analisi degli algoritmi. È stata ideata per presentare gradualmente il modo in cui specifichiamo gli algoritmi, alcune strategie di progettazione che saranno utilizzate in questo libro e molti dei concetti fondamentali dell'analisi degli algoritmi. Le parti successive del libro si fondano su queste basi.

Il Capitolo 1 è una panoramica degli algoritmi e del loro ruolo nei moderni sistemi di elaborazione dei dati. Questo capitolo definisce che cos'è un algoritmo e fornisce alcuni esempi. Ipotizza inoltre che gli algoritmi siano una tecnologia, esattamente come le unità hardware veloci, le interfacce grafiche, i sistemi orientati agli oggetti e le reti.

Nel Capitolo 2 presentiamo i primi algoritmi che risolvono il problema dell'ordinamento di una sequenza di n numeri. Ogni algoritmo è scritto con un semplice pseudocodice che, sebbene non sia direttamente traducibile in uno dei linguaggi di programmazione convenzionali, presenta la struttura dell'algoritmo in modo sufficientemente chiaro per consentire a un programmatore di implementarla nel suo linguaggio preferito. Fra gli algoritmi di ordinamento esaminati figurano *insertion sort*, che usa un approccio incrementale, e *merge sort*, che usa una tecnica ricorsiva detta "divide et impera". Sebbene il tempo di calcolo richiesto da ciascun algoritmo cresca con il valore di n, tuttavia il tasso di crescita di questo tempo varia fra i due algoritmi. Il Capitolo 2 descrive come calcolare i tempi di esecuzione degli algoritmi e presenta un'utile notazione per esprimerli.

Il Capitolo 3 definisce con esattezza questa notazione, che chiameremo *notazione asintotica*. Inizialmente, presenteremo varie notazioni asintotiche, che poi utilizzeremo per definire i limiti dei tempi di esecuzione degli algoritmi. La parte restante del capitolo è essenzialmente una presentazione di notazioni matematiche; lo scopo principale non è quello di insegnarvi nuovi concetti matematici, ma bensì garantire che le vostre notazioni siano conformi a quelle adottate in questo libro.

Il Capitolo 4 tratta più approfonditamente il metodo *divide et impera* introdotto nel Capitolo 2. In particolare, presenteremo i metodi per risolvere le ricorrenze, che sono utili per descrivere i tempi di esecuzione degli algoritmi ricorsivi. Una tecnica molto efficace è il "metodo dell'esperto" che può essere impiegato per risolvere le ricorrenze che derivano dagli algoritmi divide et impera. Gran

parte di questo capitolo è dedicata alla dimostrazione della correttezza del metodo dell'esperto (potete comunque tralasciare questa dimostrazione senza alcun problema).

Il Capitolo 5 introduce l'analisi probabilistica e gli algoritmi randomizzati. Tipicamente, l'analisi probabilistica viene utilizzata per determinare il tempo di esecuzione di un algoritmo nel caso in cui, per la presenza di una particolare distribuzione di probabilità, il tempo di esecuzione vari con input diversi della stessa dimensione. In alcuni casi, supponiamo che gli input siano conformi a una distribuzione di probabilità nota, in modo da mediare il tempo di esecuzione su tutti i possibili input. In altri casi, la distribuzione di probabilità non proviene dagli input, ma da scelte casuali effettuate durante lo svolgimento dell'algoritmo. Un algoritmo il cui comportamento è determinato non soltanto dai suoi input, ma anche dai valori prodotti da un generatore di numeri casuali è detto *algoritmo randomizzato*. È possibile utilizzare gli algoritmi randomizzati per imporre una distribuzione di probabilità agli input – garantendo così che nessun input possa sistematicamente provocare una riduzione delle prestazioni – o anche per limitare il tasso di errore di algoritmi cui è consentito produrre risultati affetti da un errore controllato.

Le Appendici A-C trattano altri concetti matematici che vi saranno particolarmente utili durante la lettura di questo libro. È probabile che conosciate già molti degli argomenti descritti nelle appendici (sebbene le particolari notazioni da noi adottate possano differire in alcuni casi da quelle che conoscete), quindi potete considerare le appendici come materiale di riferimento. D'altra parte, potreste non avere mai visto molti argomenti della Parte I. Tutti i capitoli della Parte I e delle appendici sono scritti con la tipica forma dei tutorial.

Ruolo degli algoritmi nell'elaborazione dei dati 1

Che cosa sono gli algoritmi? Perché è utile studiare gli algoritmi? Qual è il ruolo degli algoritmi rispetto ad altre tecnologie utilizzate nei calcolatori? In questo capitolo risponderemo a queste domande.

1.1 Algoritmi

Informalmente, un ***algoritmo*** è una procedura di calcolo ben definita che prende un certo valore, o un insieme di valori, come ***input*** e genera un valore, o un insieme di valori, come ***output***. Un algoritmo è quindi una sequenza di passi computazionali che trasforma l'input in output.

Possiamo anche considerare un algoritmo come uno strumento per risolvere un ***problema computazionale*** ben definito. La definizione del problema specifica in termini generali la relazione di input/output desiderata. L'algoritmo descrive una specifica procedura computazionale per ottenere tale relazione di input/output.

Per esempio, supponiamo di dovere ordinare una sequenza di numeri in ordine non decrescente. Questo problema si presenta spesso nella pratica e rappresenta un terreno fertile per introdurre vari strumenti di analisi e tecniche di progettazione standard. Il ***problema dell'ordinamento*** può essere formalmente definito nel seguente modo:

Input: una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: una permutazione (riordinamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Per esempio, data la sequenza di input $\langle 31, 41, 59, 26, 41, 58 \rangle$, un algoritmo di ordinamento restituisce come output la sequenza $\langle 26, 31, 41, 41, 58, 59 \rangle$. Tale sequenza di input è detta ***istanza*** del problema dell'ordinamento. In generale, l'***istanza di un problema*** è formata dall'input (che soddisfa tutti i vincoli imposti nella definizione del problema) richiesto per calcolare una soluzione del problema.

L'ordinamento è un'operazione fondamentale in informatica (molti programmi la usano come passo intermedio), per questo sono stati sviluppati vari algoritmi di ordinamento. La scelta dell'algoritmo più appropriato a una data applicazione dipende – fra l'altro – dal numero di elementi da ordinare, dal livello di ordinamento iniziale degli elementi, da eventuali vincoli sui valori degli elementi e dal tipo di unità di memorizzazione da utilizzare: memoria principale, dischi o nastri.

Un algoritmo si dice ***corretto*** se, per ogni istanza di input, termina con l'output corretto. Diciamo che un algoritmo corretto ***risolve*** il problema computazionale dato. Un algoritmo errato potrebbe non terminare affatto con qualche istanza di input o potrebbe terminare fornendo una soluzione diversa da quella desiderata. Contrariamente a quello che uno potrebbe aspettarsi, gli algoritmi errati a vol-

te possono essere utili, se il loro tasso di errore può essere controllato. Vedremo un esempio di questo nel Capitolo 31 quando studieremo gli algoritmi per trovare i numeri primi grandi. Di solito, tuttavia, ci occuperemo soltanto di algoritmi corretti.

Un algoritmo può essere specificato in lingua italiana, come un programma per computer, o perfino come un progetto hardware. L'unico requisito è che la specifica deve fornire una descrizione esatta della procedura computazionale da seguire.

Quali problemi risolvono gli algoritmi?

L'ordinamento non è affatto l'unico problema computazionale per cui sono stati sviluppati gli algoritmi (molti lo avranno intuito osservando la mole di questo libro). Le applicazioni pratiche degli algoritmi sono innumerevoli; ne citiamo alcune:

- Il Progetto Genoma Umano ha l'obiettivo di identificare tutti i 100.000 geni del DNA umano, determinando le sequenze di 3 miliardi di paia di basi chimiche che formano il DNA umano, registrando queste informazioni nei database e sviluppando gli strumenti per analizzare i dati. Ciascuno di questi passaggi richiede sofisticati algoritmi. Sebbene le soluzioni di questi problemi esulino dagli obiettivi di questo libro, i concetti esposti in molti capitoli vengono utilizzati per risolvere tali problemi biologici, consentendo così agli scienziati di svolgere i loro compiti utilizzando in modo efficiente le risorse. Si risparmia tempo (di persone e macchine) e denaro, in quanto è possibile estrarre più informazioni dalle tecniche di laboratorio.
- Internet consente agli utenti di tutto il mondo di accedere rapidamente a grandi quantità di informazioni. Per fare ciò vengono impiegati algoritmi intelligenti che gestiscono e manipolano enormi volumi di dati. Fra gli esempi di problemi che devono essere risolti citiamo la ricerca dei percorsi ottimali che i dati devono seguire (le tecniche per risolvere questi problemi sono descritte nel Capitolo 24) e l'uso di un motore di ricerca per trovare velocemente le pagine che contengono una particolare informazione (le relative tecniche sono trattate nei Capitoli 11 e 32).
- Il commercio elettronico consente di negoziare e scambiare elettronicamente beni e servizi. La capacità di mantenere riservate informazioni quali i codici delle carte di credito, le password e gli estratti conto è essenziale alla diffusione su vasta scala del commercio elettronico. La crittografia a chiave pubblica e le firme digitali (descritte nel Capitolo 31) sono le principali tecnologie utilizzate e si basano su algoritmi numerici e sulla teoria dei numeri.
- Nelle attività industriali e commerciali spesso è importante allocare poche risorse nel modo più vantaggioso. Una compagnia petrolifera potrebbe essere interessata a sapere dove disporre i propri pozzi per massimizzare i profitti. Un candidato alla presidenza degli Stati Uniti d'America potrebbe essere interessato a determinare in quale campagna pubblicitaria investire i suoi soldi per massimizzare le probabilità di vincere le elezioni. Una compagnia aerea potrebbe essere interessata ad assegnare il personale ai voli nel modo più economico possibile, verificando che ogni volo sia coperto e che siano soddisfatte le disposizioni governative sulla programmazione del personale di volo. Un

provider di servizi Internet potrebbe essere interessato a determinare dove allocare delle risorse addizionali per servire i suoi clienti in modo più efficiente. Tutti questi sono esempi di problemi che possono essere risolti utilizzando la programmazione lineare, che sarà trattata nel Capitolo 29.

Sebbene alcuni dettagli di questi esempi esulino dagli scopi di questo libro, tuttavia è opportuno descrivere le tecniche di base che si applicano a questi tipi di problemi. Spiegheremo inoltre come risolvere molti problemi concreti, inclusi i seguenti:

- Supponiamo di avere una carta stradale dove sono segnate le distanze fra ogni coppia di incroci adiacenti; il nostro obiettivo è determinare il percorso più breve da un incrocio all'altro. Il numero di percorsi possibili può essere enorme, anche se escludiamo i percorsi che passano su sé stessi. Come scegliere il più breve di tutti i percorsi? In questo caso, creiamo un modello della carta stradale (che a sua volta è un modello delle strade reali) come un grafo (che descriveremo nel Capitolo 10 e nell'Appendice B) e cerchiamo di determinare il cammino più breve da un vertice all'altro del grafo. Spiegheremo come risolvere efficientemente questo problema nel Capitolo 24.
- Data una sequenza $\langle A_1, A_2, \ldots, A_n \rangle$ di n matrici, vogliamo determinare il loro prodotto $A_1 A_2 \cdots A_n$. Poiché la moltiplicazione di matrici è associativa, ci sono vari modi di moltiplicare. Per esempio, se $n = 4$, potremmo eseguire il prodotto delle matrici in uno dei seguenti modi: $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$ o $(((A_1A_2)A_3)A_4)$. Se le matrici sono tutte quadrate (e quindi della stessa dimensione), il modo di moltiplicare le matrici non avrà effetto sul tempo richiesto per eseguire il prodotto. Se, invece, queste matrici hanno dimensioni differenti (ma compatibili con la moltiplicazione delle matrici), allora il modo di moltiplicare può determinare una differenza significativa. Il numero dei possibili modi di moltiplicare le matrici è esponenziale in n, pertanto provare tutti i possibili modi potrebbe richiedere un tempo molto lungo. Vedremo nel Capitolo 15 come utilizzare una tecnica generale, la programmazione dinamica, per risolvere questo problema in una maniera molto più efficiente.
- Data l'equazione $ax \equiv b \pmod{n}$, dove a, b e n sono interi, vogliamo determinare tutti gli interi x, modulo n, che soddisfano l'equazione. Ci possono essere zero, una o più soluzioni. Potremmo semplicemente provare $x = 0, 1, \ldots, n-1$ nell'ordine, ma il Capitolo 31 descrive un metodo più efficiente.
- Dati n punti nel piano, vogliamo determinare il guscio convesso di questi punti. Il guscio convesso è il più piccolo poligono convesso che contiene i punti. Intuitivamente, possiamo immaginare ogni punto come se fosse rappresentato da un chiodo che fuoriesce da una tavola. Il guscio convesso potrebbe essere rappresentato da un elastico teso che circonda tutti i chiodi. Ogni chiodo attorno al quale l'elastico fa un giro è un vertice del guscio convesso (un esempio è illustrato nella Figura 33.6 a pagina 805). Uno dei 2^n sottoinsiemi dei punti potrebbe essere formato dai vertici del guscio convesso. Conoscere i punti che formano i vertici del guscio convesso non è sufficiente, in quanto occorre sapere anche l'ordine in cui essi si presentano. Ci sono dunque molte possibilità di scelta per i vertici del guscio convesso. Il Capitolo 33 descrive due buoni metodi per trovare il guscio convesso.

Questo elenco non è affatto esaustivo (come probabilmente avrete immaginato dalle dimensioni di questo libro), ma presenta due caratteristiche che sono comuni a molti algoritmi.

1. Esistono numerose soluzioni possibili, molte delle quali non sono ciò che vogliamo. Trovare quella desiderata può essere un'impresa ardua.
2. Esistono varie applicazioni pratiche. Fra i problemi precedentemente elencati, determinare il percorso più breve rappresenta l'esempio più semplice. Un'azienda di trasporti su strada o rotaie è interessata a trovare i percorsi minimi nelle reti stradali o ferroviarie, perché tali percorsi consentono di risparmiare costi di manodopera e carburante. Come altro esempio, potrebbe essere necessario un nodo di routing su Internet per trovare il percorso più breve nella rete che permette di instradare rapidamente un messaggio.

Strutture dati

Questo libro contiene anche diverse strutture dati. Una ***struttura dati*** è un modo per memorizzare e organizzare i dati e semplificarne l'accesso e la modifica. Non esiste un'unica struttura dati che va bene per qualsiasi compito, quindi è importante conoscere vantaggi e svantaggi di queste strutture.

Tecnica

Sebbene possiate utilizzare questo libro come un "libro di ricette" per algoritmi, tuttavia un giorno potreste incontrare un problema per il quale non riuscite a trovare un algoritmo pubblicato (come molti esercizi e problemi di questo libro!). Il libro vi insegna le tecniche per progettare e analizzare gli algoritmi, in modo che possiate sviluppare i vostri algoritmi, dimostrare che forniscono la risposta esatta e valutare la loro efficienza.

Problemi difficili

Gran parte di questo libro è dedicata agli algoritmi efficienti. La tipica unità di misura dell'efficienza è la velocità, ovvero quanto tempo impiega un algoritmo per produrre il suo risultato. Ci sono problemi, tuttavia, per i quali non si conosce una soluzione efficiente. Il Capitolo 34 studia un interessante sottoinsieme di questi problemi, noti come problemi NP-completi.

Perché sono interessanti i problemi NP-completi? In primo luogo, sebbene non sia stato ancora trovato un algoritmo efficiente per un problema NP-completo, tuttavia nessuno ha dimostrato che non possa esistere un algoritmo efficiente per uno di questi problemi. In altre parole, non sappiamo se esistano algoritmi efficienti per i problemi NP-completi. In secondo luogo, l'insieme dei problemi NP-completi gode dell'importante proprietà che, se esiste un algoritmo efficiente per uno di essi, allora esistono algoritmi efficienti per tutti questi problemi. Questa relazione fra i problemi NP-completi rende molto più attraente la mancanza di soluzioni efficienti. In terzo luogo, molti problemi NP-completi sono simili, non identici, ai problemi per i quali conosciamo gli algoritmi efficienti. Una piccola variazione della definizione del problema può causare una grande variazione dell'efficienza del migliore algoritmo conosciuto.

È importante conoscere i problemi NP-completi perché spesso alcuni di essi si presentano in modo inaspettato nelle applicazioni reali. Se vi chiedessero di creare

un algoritmo efficiente per un problema NP-completo, rischiereste di sprecare molto del vostro tempo in ricerche inutili. Se riuscite a dimostrare che il problema è NP-completo, allora potrete impiegare il vostro tempo a sviluppare un algoritmo efficiente che fornisce una buona soluzione, non la migliore possibile.

Come esempio concreto, considerate un'impresa di trasporti che abbia un magazzino centrale. Tutte le mattine un autocarro viene caricato presso il magazzino e poi indirizzato alle varie destinazioni per consegnare le merci. Alla fine della giornata l'autocarro deve ritornare al magazzino per essere pronto per il giorno successivo. Per ridurre i costi, l'azienda intende scegliere un ordine di fermate per le consegne che consenta all'autocarro di percorrere la distanza minima. Si tratta del cosiddetto "problema del commesso viaggiatore" ed è un problema NP-completo. Non esiste un algoritmo efficiente. Sotto opportune ipotesi, tuttavia, ci sono algoritmi efficienti che forniscono una distanza complessiva che non è molto diversa da quella minima. Il Capitolo 35 tratta questi "algoritmi di approssimazione".

Esercizi

1.1-1
Indicate un esempio nel mondo reale in cui si presenta uno dei seguenti problemi computazionali: ordinamento, determinare il modo ottimale di moltiplicare le matrici o trovare il guscio convesso.

1.1-2
Oltre alla velocità, quali altri indici di efficienza potrebbero essere utilizzati in uno scenario del mondo reale?

1.1-3
Scegliete una struttura dati che avete visto in precedenza e analizzatene vantaggi e svantaggi.

1.1-4
In che modo sono simili i problemi del percorso minimo e del commesso viaggiatore? In che modo differiscono?

1.1-5
Descrivete un problema del mondo reale in cui è ammissibile soltanto la soluzione ideale. Poi indicatene uno in cui è accettabile una soluzione che "approssima" quella ideale.

1.2 Algoritmi come tecnologia

Se i computer fossero infinitamente veloci e la memoria dei computer fosse gratuita, avremmo ancora qualche motivo per studiare gli algoritmi? La risposta è sì, se non altro perché vorremmo ugualmente dimostrare che il nostro metodo di risoluzione termina e fornisce la soluzione esatta.

Se i computer fossero infinitamente veloci, qualsiasi metodo corretto per risolvere un problema andrebbe bene. Probabilmente, vorremmo che la nostra implementazione rispettasse le buone norme dell'ingegneria del software (ovvero fosse ben progettata e documentata), ma il più delle volte adotteremmo il metodo più semplice da implementare. Ovviamente, i computer possono essere veloci, ma non infinitamente veloci. La memoria può costare poco, ma non può essere gra-

tuita. Il tempo di elaborazione e lo spazio nella memoria sono risorse limitate, che devono essere saggiamente utilizzate; gli algoritmi che sono efficienti in termini di tempo o spazio ci aiuteranno a farlo.

Efficienza

Algoritmi progettati per risolvere lo stesso problema spesso sono notevolmente diversi nella loro efficienza. Queste differenze possono essere molto più significative di quelle dovute all'hardware e al software.

Per esempio, nel Capitolo 2 esamineremo due algoritmi di ordinamento. Il primo, detto ***insertion sort***, impiega un tempo pari a circa $c_1 n^2$ per ordinare n elementi, dove c_1 è una costante che non dipende da n; ovvero occorre un tempo all'incirca proporzionale a n^2. Il secondo algoritmo, ***merge sort***, richiede un tempo pari a circa $c_2 n \lg n$, dove $\lg n$ sta per $\log_2 n$ e c_2 è un'altra costante che non dipende da n. Insertion sort, di solito, ha un fattore costante più piccolo di merge sort ($c_1 < c_2$). Vedremo come i fattori costanti possano avere meno influenza sul tempo di esecuzione rispetto alla dimensione n dell'input. Quando merge sort ha un fattore $\lg n$ nel suo tempo di esecuzione, insertion sort ha un fattore n, che è molto più grande. Sebbene insertion sort, di solito, sia più veloce di merge sort per input di piccole dimensioni, tuttavia quando la dimensione dell'input n diventa relativamente grande, il vantaggio di merge sort, $\lg n$ su n, compensa abbondantemente la differenza fra i fattori costanti. Indipendentemente da quanto c_1 sia più piccola rispetto a c_2, ci sarà sempre un punto oltre il quale merge sort è più rapido.

Come esempio concreto, mettiamo a confronto un computer veloce (computer A) che esegue insertion sort e un computer lento (computer B) che esegue merge sort. Entrambi devono ordinare un array di un milione di numeri. Supponiamo che il computer A esegua un miliardo di istruzioni al secondo e il computer B esegua soltanto 10 milioni di istruzioni al secondo, ovvero il computer A è 100 volte più veloce del computer B in termini di potenza di calcolo. Per rendere la differenza ancora più evidente, supponiamo che il miglior programmatore del mondo abbia codificato insertion sort nel linguaggio macchina del computer A e che il codice risultante richieda $2n^2$ istruzioni per ordinare n numeri (in questo caso, $c_1 = 2$). Merge sort, invece, è stato programmato per il computer B da un programmatore medio con un linguaggio di alto livello e un compilatore inefficiente; il codice risultante richiede $50n \lg n$ istruzioni ($c_2 = 50$). Per ordinare un milione di numeri, il computer A impiega

$$\frac{2 \cdot (10^6)^2 \text{ istruzioni}}{10^9 \text{ istruzioni/secondo}} = 2000 \text{ secondi}$$

mentre il computer B impiega

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ istruzioni}}{10^7 \text{ istruzioni/secondo}} \approx 100 \text{ secondi}$$

Utilizzando un algoritmo il cui tempo di esecuzione cresce più lentamente, perfino con un compilatore scadente, il computer B è 20 volte più veloce del computer A! Il vantaggio di merge sort è ancora più significativo se ordiniamo dieci milioni di numeri: insertion sort impiega più di 2 giorni, merge sort meno di 20 minuti. In generale, al crescere della dimensione del problema, aumenta il vantaggio relativo di merge sort.

Algoritmi e altre tecnologie

L'esempio dimostra che gli algoritmi, come l'hardware, sono una ***tecnologia***. Le prestazioni globali di un sistema dipendono tanto dalla scelta di algoritmi efficienti quanto dalla scelta di un hardware veloce. Analogamente a quanto è avvenuto in altre tecnologie per calcolatori, anche gli algoritmi hanno avuto un loro rapido progresso. Qualcuno potrebbe chiedersi se gli algoritmi siano davvero così importanti per i moderni calcolatori alla luce di altre tecnologie avanzate, quali

- hardware con alte frequenze di clock, pipeline e architetture superscalari
- interfacce grafiche (GUI) intuitive e facili da usare
- sistemi orientati agli oggetti
- reti locali (LAN) e geografiche (WAN)

La risposta è sì. Sebbene ci siano applicazioni che non richiedano esplicitamente un contenuto algoritmico a livello dell'applicazione (per esempio, alcune semplici applicazioni web), molte richiedono una certa dose di contenuto algoritmico. Per esempio, considerate un servizio web che determina come spostarsi da un sito all'altro (esistevano molti di questi servizi quando abbiamo scritto questo libro). La sua implementazione dovrebbe fare affidamento su un hardware veloce, un'interfaccia grafica utente, una rete WAN e, possibilmente, anche su sistemi orientati agli oggetti; ma richiederebbe anche gli algoritmi per svolgere determinate operazioni, come trovare i percorsi (utilizzando un algoritmo per il percorso più breve), rappresentare le mappe e interpolare gli indirizzi.

Inoltre, anche un'applicazione che non richiede un contenuto algoritmico a livello applicazione fa affidamento sugli algoritmi. L'applicazione fa affidamento su un hardware veloce? Il progetto dell'hardware utilizza gli algoritmi. L'applicazione fa affidamento su un'interfaccia grafica utente? Il progetto dell'interfaccia fa affidamento sugli algoritmi. L'applicazione fa affidamento sulle reti? Il routing delle reti impiega gli algoritmi. L'applicazione è stata scritta in un linguaggio diverso dal codice machina? Allora è stata elaborata da un compilatore, un interprete o un assembler, ciascuno dei quali utilizza ampiamente gli algoritmi. Gli algoritmi sono il nucleo delle principali tecnologie utilizzate nei moderni calcolatori. Grazie alle loro sempre crescenti capacità, i calcolatori vengono utilizzati per risolvere problemi più complicati che in passato. Come abbiamo visto nel precedente confronto fra gli algoritmi insertion sort e merge sort, è con i problemi di dimensioni maggiori che le differenze di efficienza fra gli algoritmi diventano particolarmente evidenti.

Avere una solida base di conoscenza degli algoritmi e delle tecniche è una caratteristica che contraddistingue i programmatori esperti dai principianti. Con i moderni calcolatori, potete svolgere alcuni compiti senza sapere molto di algoritmi, ma con una buona conoscenza degli algoritmi, potete fare molto, molto di più.

Esercizi

1.2-1

Indicate l'esempio di un'applicazione che richiede un contenuto algoritmico a livello dell'applicazione e descrivete la funzione degli algoritmi richiesti.

1.2-2
Supponete di confrontare le implementazioni di insertion sort e merge sort sulla stessa macchina. Con un input di dimensione n, insertion sort viene eseguito in $8n^2$ passi, mentre merge sort viene eseguito in $64n \lg n$ passi. Per quali valori di n insertion sort batte merge sort?

1.2-3
Qual è il più piccolo valore di n per cui un algoritmo il cui tempo di esecuzione è $100n^2$ viene eseguito più velocemente di un algoritmo il cui tempo di esecuzione è 2^n sulla stessa macchina?

Problemi

1-1 Confronto fra i tempi di esecuzione
Per ogni funzione $f(n)$ e tempo t della seguente tabella, determinate la massima dimensione n di un problema che può essere risolto nel tempo t, supponendo che l'algoritmo che risolve il problema impieghi $f(n)$ microsecondi.

	1 secondo	1 minuto	1 ora	1 giorno	1 mese	1 anno	1 secolo
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Note

Ci sono molti testi eccellenti che trattano in generale gli algoritmi, fra i quali citiamo: Aho, Hopcroft e Ullman [5, 6], Baase e Van Gelder [26], Brassard e Bratley [46, 47], Goodrich e Tamassia [128], Horowitz, Sahni e Rajasekaran [158], Kingston [179], Knuth [182, 183, 185], Kozen [193], Manber [210], Mehlhorn [217, 218, 219], Purdom e Brown [252], Reingold, Nievergelt e Deo [257], Sedgewick [269], Skiena [280], e Wilf [315]. Alcuni degli aspetti più pratici della progettazione degli algoritmi sono descritti da Bentley [39, 40] e Gonnet [126]. I manuali *Handbook of Theoretical Computer Science*, Volume A [302] e *CRC Handbook on Algorithms and Theory of Computation* [24] riportano alcuni studi sugli algoritmi. Una panoramica sugli algoritmi utilizzati nella biologia computazionale si trova nei libri di testo di Gusfield [136], Pevzner [240], Setubal e Meidanis [272], e Waterman [309].

Introduzione 2

Questo capitolo consente ai lettori di acquisire familiarità con i concetti fondamentali della progettazione e dell'analisi degli algoritmi. È un capitolo autonomo, sebbene includa riferimenti ad argomenti che saranno introdotti nei Capitoli 3 e 4 (presenta anche alcune sommatorie che saranno descritte nell'Appendice A).

Inizieremo a esaminare l'algoritmo insertion sort per risolvere il problema dell'ordinamento introdotto nel Capitolo 1. Definiremo uno "pseudocodice" che dovrebbe essere familiare ai lettori che hanno studiato la programmazione dei computer e lo utilizzeremo per mostrare come specificheremo i nostri algoritmi. Dopo avere specificato l'algoritmo, supporremo che esso effettui correttamente l'ordinamento e analizzeremo il suo tempo di esecuzione. L'analisi introduce una notazione che si concentra su come cresce questo tempo con il numero di elementi da ordinare. Successivamente, introdurremo l'approccio *divide et impera* per progettare algoritmi e sviluppare un algoritmo detto *merge sort*. Il capitolo termina con un'analisi del tempo di esecuzione di merge sort.

2.1 Insertion sort

Il nostro primo algoritmo, insertion sort, risolve il ***problema dell'ordinamento*** introdotto nel Capitolo 1:

Input: una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: una permutazione (riordinamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

I numeri da ordinare sono anche detti ***chiavi***.

In questo libro, tipicamente, descriveremo gli algoritmi come programmi scritti in uno ***pseudocodice*** che è simile per molti aspetti ai linguaggi C, Pascal e Java. Se conoscete uno di questi linguaggi, non dovreste incontrare molte difficoltà a leggere i nostri algoritmi. Ciò che distingue lo pseudocodice dal codice "reale" è che nello pseudocodice impieghiamo qualsiasi mezzo espressivo che specifichi nel modo più chiaro e conciso un determinato algoritmo. A volte, il mezzo più chiaro è l'italiano, quindi non sorprendetevi se incontrate una frase in italiano all'interno di una sezione di codice "reale". Un'altra differenza fra pseudocodice e codice reale è che il primo, tipicamente, non si occupa dei problemi di ingegneria del software. Problemi quali l'astrazione dei dati, la modularità e la gestione degli errori, di solito, vengono ignorati per potere esprimere in modo più conciso l'essenza di un algoritmo.

Iniziamo con ***insertion sort***, che è un algoritmo efficiente per ordinare un piccolo numero di elementi. Questo algoritmo opera nello stesso modo in cui molte persone ordinano le carte da gioco. Iniziamo con la mano sinistra vuota e le carte

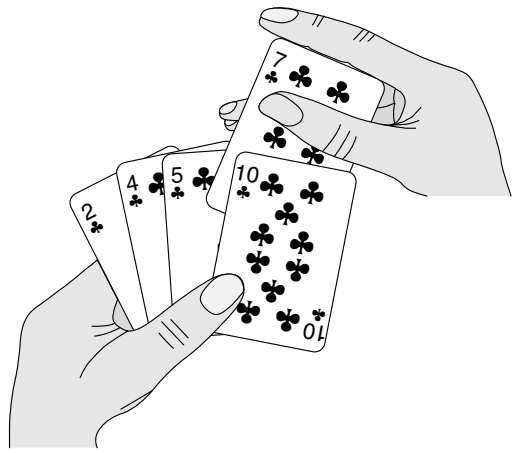


Figura 2.1 Ordinare una mano di carte mediante insertion sort.

coperte poste sul tavolo. Prendiamo una carta alla volta dal tavolo e la inseriamo nella posizione corretta nella mano sinistra. Per trovare la posizione corretta di una carta, la confrontiamo con le singole carte che abbiamo già in mano, da destra a sinistra, come illustra la Figura 2.1. In qualsiasi momento, le carte che teniamo nella mano sinistra sono ordinate; originariamente queste carte erano le prime della pila di carte che erano sul tavolo.

Il nostro pseudocodice per insertion sort è presentato come una procedura chiamata INSERTION-SORT, che prende come parametro un array $A[1\,..\,n]$ contenente una sequenza di lunghezza n che deve essere ordinata (nel codice il numero n di elementi di A è indicato da $lunghezza[A]$). I numeri di input vengono ***ordinati sul posto***: i numeri sono risistemati all'interno dell'array A, con al più un numero costante di essi memorizzati all'esterno dell'array in qualsiasi istante. L'array di input A contiene la sequenza di output ordinata quando la procedura INSERTION-SORT è completata.

```
INSERTION-SORT(A)
1  for j ← 2 to lunghezza[A]
2       do chiave ← A[j]
3          ▷ Inserisce A[j] nella sequenza ordinata A[1 .. j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > chiave
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← chiave
```

Invarianti di ciclo e correttezza di insertion sort

La Figura 2.2 mostra come opera questo algoritmo con $A = \langle 5, 2, 4, 6, 1, 3\rangle$. L'indice j identifica la "carta corrente" che viene inserita nelle altre. All'inizio di ogni iterazione del ciclo **for** "esterno", il cui indice è j, il sottoarray che è formato dagli elementi $A[1\,..\,j-1]$ costituisce la mano di carte correntemente ordinate e gli elementi $A[j+1\,..\,n]$ corrispondono alla pila delle carte che si trovano ancora sul tavolo. In effetti, gli elementi $A[1\,..\,j-1]$ sono quelli che *originariamente* occupavano le posizioni da 1 a $j-1$, ma che adesso sono ordinati. Definiamo formalmente queste proprietà di $A[1\,..\,j-1]$ come ***invariante di ciclo***:

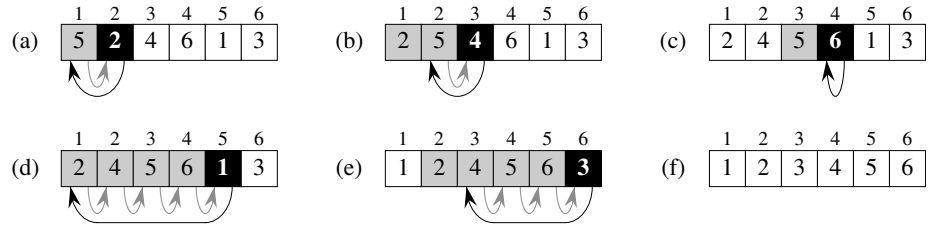


Figura 2.2 Il funzionamento di INSERTION-SORT con l'array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Gli indici dell'array sono indicati sopra i rettangoli; i valori memorizzati nelle posizioni dell'array sono indicati all'interno dei rettangoli. **(a)–(e)** Le iterazioni del ciclo **for** (righe 1–8). In ogni iterazione, il rettangolo nero contiene la chiave estratta da $A[j]$, che viene confrontata con i valori nei rettangoli grigi alla sua sinistra nel test della riga 5. Le frecce grige mostrano i valori dell'array che vengono spostati di una posizione verso destra nella riga 6; le frecce nere indicano dove viene spostata la chiave nella riga 8. **(f)** L'array finale ordinato.

> All'inizio di ogni iterazione del ciclo **for** (righe 1–8), il sottoarray $A[1 \,..\, j\text{–}1]$ è formato dagli elementi ordinati che originariamente erano in $A[1 \,..\, j\text{–}1]$.

Utilizziamo le invarianti di ciclo per aiutarci a capire perché un algoritmo è corretto. Dobbiamo dimostrare tre cose su un'invariante di ciclo:

Inizializzazione: è vera prima della prima iterazione del ciclo.

Conservazione: se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva iterazione.

Conclusione: quando il ciclo termina, l'invariante fornisce un'utile proprietà che ci aiuta a dimostrare che l'algoritmo è corretto.

Quando le prime due proprietà sono valide, l'invariante di ciclo è vera prima di ogni iterazione del ciclo. Notate l'analogia con l'induzione matematica, dove per provare che una proprietà è valida, si prova un caso base e un passaggio induttivo. Qui, dimostrare che l'invariante è vera prima della prima iterazione equivale al caso base e dimostrare che l'invariante resta vera da un'iterazione all'altra equivale al passaggio induttivo.

La terza proprietà è forse la più importante, perché utilizziamo l'invariante di ciclo per dimostrare la correttezza. C'è anche una differenza con l'uso consueto dell'induzione matematica, dove il passaggio induttivo viene utilizzato all'infinito; qui invece interrompiamo il "processo induttivo" quando il ciclo termina.

Vediamo se queste proprietà sono valide per insertion sort.

Inizializzazione: iniziamo dimostrando che l'invariante di ciclo è vera prima della prima iterazione del ciclo, quando $j = 2$.[1] Il sottoarray $A[1 \,..\, j-1]$, quindi, è formato dal solo elemento $A[1]$, che infatti è l'elemento originale in $A[1]$. Inoltre, questo sottoarray è ordinato (banale, ovviamente) e ciò dimostra che l'invariante di ciclo è vera prima della prima iterazione del ciclo.

[1] Quando il ciclo è un ciclo **for**, il punto in cui verifichiamo l'invariante di ciclo appena prima della prima iterazione è immediatamente dopo l'assegnazione iniziale del contatore del ciclo e appena prima del primo test del ciclo. Nel caso di INSERTION-SORT, questo punto è dopo l'assegnazione di 2 alla variabile j, ma prima del primo test $j \leq \mathit{lunghezza}[A]$.

Conservazione: passiamo alla seconda proprietà: dimostrare che ogni iterazione conserva l'invariante di ciclo. Informalmente, il corpo del ciclo **for** esterno opera spostando $A[j-1]$, $A[j-2]$, $A[j-3]$ e così via di una posizione verso destra, finché non troverà la posizione appropriata per $A[j]$ (righe 4–7), dove inserirà il valore di $A[j]$ (riga 8). Un trattamento più formale della seconda proprietà richiederebbe di definire e dimostrare un'invariante di ciclo per il ciclo **while** "interno". A questo punto, tuttavia, preferiamo non impantanarci in simili formalismi; quindi, confidiamo nella nostra analisi informale per dimostrare che la seconda proprietà è vera per il ciclo esterno.

Conclusione: infine, esaminiamo che cosa accade quando il ciclo termina. Per insertion sort, il ciclo **for** esterno termina quando j supera n, ovvero quando $j = n+1$. Sostituendo j con $n+1$ nella formulazione dell'invariante di ciclo, otteniamo che il sottoarray $A[1\,..\,n]$ è formato dagli elementi *ordinati* che si trovavano originariamente in $A[1\,..\,n]$. Ma il sottoarray $A[1\,..\,n]$ è l'intero array! Dunque, tutto l'array è ordinato; ciò significa che l'algoritmo è corretto.

Applicheremo questo metodo delle invarianti di ciclo per dimostrare la correttezza più avanti in questo capitolo e in altri capitoli.

Convenzioni di pseudocodifica

Adotteremo le seguenti convenzioni nelle nostre pseudocodifiche.

1. L'indentazione (rientro verso destra delle righe) serve a indicare la struttura a blocchi dello pseudocodice. Per esempio, il corpo del ciclo **for**, che inizia nella riga 1, è formato dalla righe 2–8 e il corpo del ciclo **while**, che inizia nella riga 5, contiene le righe 6–7, ma non la riga 8. Il nostro stile di indentazione si applica anche alle istruzioni **if**-**then**-**else**. Utilizzando l'indentazione, anziché gli indicatori convenzionali della struttura a blocchi, come le istruzioni **begin** e **end**, si riduce molto la confusione, preservando o perfino migliorando la chiarezza.[2]
2. I costrutti iterativi **while**, **for** e **repeat** e i costrutti condizionali **if**, **then** ed **else** hanno interpretazioni simili a quelle del Pascal.[3] C'è tuttavia una piccola differenza nei cicli **for**: nel Pascal il valore del contatore del ciclo è indefinito dopo la conclusione del ciclo, mentre in questo libro il contatore del ciclo mantiene il suo valore dopo la fine del ciclo. Quindi, immediatamente dopo un ciclo **for**, il valore del contatore del ciclo è quello che ha appena superato il limite del ciclo **for**. Abbiamo utilizzato questa proprietà nella nostra analisi della correttezza di insertion sort. La prima istruzione del ciclo **for** (riga 1) è **for** $j \leftarrow 2$ **to** $lunghezza[A]$; quindi, alla fine di questo ciclo, $j = lunghezza[A] + 1$ (che equivale a $j = n+1$, in quanto $n = lunghezza[A]$).
3. Il simbolo "$\triangleright$" indica che il resto della riga è un commento.

---

[2] Nei linguaggi di programmazione reali, in generale, non è consigliabile utilizzare soltanto l'indentazione per indicare la struttura a blocchi, in quanto i livelli di indentazione sono difficili da determinare quando il codice è distribuito su più pagine.

[3] Molti linguaggi con strutture a blocchi hanno costrutti equivalenti, anche se la sintassi esatta può differire da quella del Pascal.

4. Un'assegnazione multipla della forma $i \leftarrow j \leftarrow e$ assegna a entrambe le variabili i e j il valore dell'espressione e; deve essere considerata equivalente all'assegnazione $j \leftarrow e$ seguita dall'assegnazione $i \leftarrow j$.
5. Le variabili (come i, j e $chiave$) sono locali a una determinata procedura. Non dovremmo utilizzare variabili globali senza un'esplicita indicazione.
6. Per identificare un elemento di un array, specifichiamo il nome dell'array seguito dall'indice dell'elemento fra parentesi quadre. Per esempio, $A[i]$ indica l'elemento i-esimo dell'array A. La notazione ".." è utilizzata per indicare un intervallo di valori all'interno di un array. Quindi, $A[1\,..\,j]$ indica il sottoarray di A che è composto da j elementi: $A[1], A[2], \ldots, A[j]$.
7. I dati composti sono tipicamente organizzati in ***oggetti***, che sono formati da ***attributi*** o ***campi***. Un particolare campo è identificato utilizzando il nome del campo seguito dal nome del suo oggetto fra parentesi quadre. Per esempio, noi trattiamo un array come un oggetto con l'attributo $lunghezza$ che indica il numero di elementi contenuti nell'array. Per specificare il numero di elementi di un array A, scriviamo $lunghezza[A]$. Anche se utilizziamo le parentesi quadre sia per gli indici degli array sia per gli attributi degli oggetti, di solito, è chiaro dal contesto a cosa intendiamo riferirci.

   Una variabile che rappresenta un array o un oggetto è trattata come un puntatore ai dati che costituiscono l'array o l'oggetto. Per tutti i campi f di un oggetto x, l'assegnazione $y \leftarrow x$ implica che $f[y] = f[x]$. Inoltre, se poi impostiamo $f[x] \leftarrow 3$, allora non soltanto sarà $f[x] = 3$, ma anche $f[y] = 3$. In altre parole, x e y puntano allo stesso oggetto dopo l'assegnazione $y \leftarrow x$.

   Un puntatore può non fare riferimento ad alcun oggetto; in questo caso daremo ad esso il valore speciale NIL.
8. I parametri vengono passati a una procedura ***per valore***: la procedura chiamata riceve la sua copia dei parametri e, se viene assegnato un valore a un parametro, la modifica *non* viene vista dalla procedura chiamante. Quando viene passato un oggetto, viene copiato il puntatore ai dati che costituiscono l'oggetto, ma non vengono copiati i campi dell'oggetto. Per esempio, se x è un parametro di una procedura chiamata, l'assegnazione $x \leftarrow y$ all'interno della procedura chiamata non è visibile alla procedura chiamante. L'assegnazione $f[x] \leftarrow 3$, invece, è visibile.
9. Gli operatori booleani "and" e "or" sono ***operatori di cortocircuito***. Questo significa che, quando valutiamo l'espressione "x and y", prima dobbiamo valutare x. Se x è FALSE, allora l'intera espressione non può essere TRUE, quindi non occorre valutare y. Se, invece, x è TRUE, dobbiamo valutare y per determinare il valore dell'intera espressione. Analogamente, se abbiamo l'espressione "x or y", valutiamo y soltanto se x è FALSE. Gli operatori di cortocircuito ci consentono di scrivere espressioni booleane come "$x \neq$ NIL and $f[x] = y$" senza preoccuparci di ciò che accade quando tentiamo di valutare $f[x]$ quando x è NIL.

Esercizi

2.1-1

Utilizzando la Figura 2.2 come modello, illustrate l'operazione di INSERTION-SORT sull'array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Modificate la procedura INSERTION-SORT per disporre gli elementi in ordine non crescente, anziché non decrescente.

2.1-3

Considerate il seguente ***problema di ricerca***:

Input: una sequenza di n numeri $A = \langle a_1, a_2, \ldots, a_n \rangle$ e un valore v.

Output: un indice i tale che $v = A[i]$ o il valore speciale NIL se v non figura in A.

Scrivere uno pseudocodice di ***ricerca lineare*** che esamina gli elementi della sequenza alla ricerca di v. Utilizzando un'invariante di ciclo, dimostrate che il vostro algoritmo è corretto. Verificate che la vostra invariante di ciclo soddisfa le tre proprietà richieste.

2.1-4

Considerate il problema di sommare due numeri interi binari di n-bit, memorizzati in due array A e B di n elementi. La somma dei due interi deve essere memorizzata in forma binaria nell'array C di $(n + 1)$ elementi. Definite formalmente il problema e scrivete lo pseudocodice per sommare i due interi.

2.2 Analisi degli algoritmi

Analizzare un algoritmo significa prevedere le risorse che l'algoritmo richiede. Raramente sono di primaria importanza risorse come la memoria, la larghezza di banda nelle comunicazioni o l'hardware nei computer, mentre più frequentemente è importante misurare il tempo di elaborazione. In generale, analizzando più algoritmi candidati a risolvere un problema, è facile identificare quello più efficiente. Tale analisi potrebbe indicare più di un candidato, ma di solito in questo processo vengono scartati diversi algoritmi inefficienti.

Prima di analizzare un algoritmo, dobbiamo avere un modello della tecnologia di implementazione che sarà utilizzata, incluso un modello per le risorse di tale tecnologia e dei loro costi. Nella maggior parte dei casi di questo libro, considereremo come tecnologia di implementazione un generico modello di calcolo a un processore, che chiameremo modello ***random-access machine*** (***RAM***); inoltre, i nostri algoritmi saranno implementati come programmi per computer. Nel modello RAM, le istruzioni sono eseguite una dopo l'altra, senza operazioni contemporanee. Nei capitoli successivi, tuttavia, avremo modo di esaminare modelli per hardware digitale.

A rigor di termini, dovremmo definire con precisione le istruzioni del modello RAM e i loro costi. Purtroppo, tutto questo risulterebbe noioso e non gioverebbe molto a illustrare il processo di analisi e progettazione degli algoritmi. Eppure dobbiamo stare attenti a non abusare del modello RAM. Per esempio, che cosa accadrebbe se un modello RAM avesse un'istruzione di ordinamento? Potremmo ordinare gli elementi con una sola istruzione. Tale modello non sarebbe realistico, in quanto i computer reali non hanno simili istruzioni. La nostra guida, dunque, è come i computer reali sono progettati. Il modello RAM contiene istruzioni che si trovano comunemente nei computer reali: istruzioni aritmetiche (addizione, sottrazione, moltiplicazione, divisione, resto, floor, ceiling), istruzioni per spostare i

dati (load, store, copy) e istruzioni di controllo (salto condizionato e incondizionato, chiamata di subroutine e return). Ciascuna di queste istruzioni richiede una quantità costante di tempo.

I tipi di dati nel modello RAM sono integer (numeri interi) e floating point (numeri in virgola mobile). Sebbene di solito la precisione non sia problema in questo libro, tuttavia, in alcune applicazioni potrebbe essere un fattore cruciale. Inoltre, supporremo che ci sia un limite alla dimensione di ogni parola (word) di dati. Per esempio, quando operiamo con input di dimensione n, tipicamente, supponiamo che i numeri interi siano rappresentati da $c \lg n$ bit per una costante $c \geq 1$. Noi richiediamo $c \geq 1$ in modo che ogni parola possa contenere il valore di n, consentendoci di indicizzare i singoli elementi di input, e imponiamo che c sia una costante in modo che la dimensione della parola non cresca in modo arbitrario (se questa dimensione potesse crescere arbitrariamente, potremmo memorizzare enormi quantità di dati in una parola e operare con essa sempre in un tempo costante – chiaramente uno scenario irreale).

I computer reali contengono istruzioni non elencate in precedenza; tali istruzioni rappresentano un'area grigia nel modello RAM. Per esempio, l'elevamento a potenza è un'istruzione a tempo costante? Nel caso generale, no; occorrono varie istruzioni per calcolare x^y quando x e y sono numeri reali. In casi limitati, invece, l'elevamento a potenza è un'operazione a tempo costante. Molti computer hanno un'istruzione "shift left" (scorrimento a sinistra), che fa scorrere in un tempo costante i bit di un numero intero di k posizioni a sinistra. In molti computer, lo scorrimento dei bit di un intero di una posizione a sinistra equivale a moltiplicare per 2. Lo scorrimento dei bit di k posizioni a sinistra equivale a moltiplicare per 2^k. Di conseguenza, tali computer possono calcolare 2^k in un'istruzione a tempo costante facendo scorrere l'intero 1 di k posizioni a sinistra, purché k non superi il numero di bit di una parola del computer. Cercheremo di evitare tali aree grige nel modello RAM, tuttavia considereremo il calcolo di 2^k come un'operazione a tempo costante quando k è un intero positivo sufficientemente piccolo.

Nel modello RAM non tenteremo di modellare la struttura gerarchica della memoria che è comune nei computer attuali, ovvero non modelleremo la memoria cache o virtuale, che molto spesso viene implementata con la paginazione su richiesta (demand paging). Vari modelli computazionali tentano di tenere conto degli effetti della gerarchia della memoria, che a volte sono significativi nei programmi o nelle macchine reali. In pochi problemi di questo libro esamineremo gli effetti della gerarchia della memoria, ma nella maggior parte dei casi l'analisi ignorerà tali effetti. I modelli che includono la struttura gerarchica della memoria sono un po' più complessi del modello RAM, quindi è difficile operare con essi. Inoltre, l'analisi del modello RAM di solito è un eccellente strumento per prevedere le prestazioni delle macchine reali.

L'analisi di un algoritmo nel modello RAM può risultare complessa anche se l'algoritmo è semplice. Gli strumenti matematici richiesti possono includere la teoria delle probabilità, la topologia combinatoria, destrezza algebrica e capacità di identificare i termini più significativi in una formula. Poiché il comportamento di un algoritmo può essere diverso per ogni possibile input, occorrono strumenti per sintetizzare tale comportamento in formule semplici e facili da capire.

Anche se di solito selezioniamo soltanto un modello di macchina per analizzare un determinato algoritmo, avremo a disposizione varie scelte per decidere come esprimere la nostra analisi. Preferiremmo un metodo che sia semplice da scrive-

re e manipolare, mostri le caratteristiche importanti delle risorse richieste da un algoritmo ed elimini i dettagli più noiosi.

Analisi di insertion sort

Il tempo richiesto dalla procedura INSERTION-SORT dipende dall'input: occorre più tempo per ordinare un migliaio di numeri che tre numeri. Inoltre, INSERTION-SORT può richiedere quantità di tempo differenti per ordinare due sequenze di input della stessa dimensione a seconda di come gli elementi siano già ordinati. In generale, il tempo richiesto da un algoritmo cresce con la dimensione dell'input, quindi è tradizione descrivere il tempo di esecuzione di un programma come una funzione della dimensione del suo input. Per farlo, dobbiamo definire più correttamente i termini "tempo di esecuzione" e "dimensione dell'input".

La definizione migliore della ***dimensione dell'input*** dipende dal problema che si sta studiando. Per la maggior parte dei problemi, come l'ordinamento o il calcolo delle trasformate discrete di Fourier, la misura più naturale è il *numero di elementi dell'input* – per esempio, la dimensione n dell'array per l'ordinamento. Per molti altri problemi, come la moltiplicazione di due interi, la misura migliore della dimensione dell'input è il *numero totale di bit* richiesti per rappresentare l'input nella normale notazione binaria. A volte, è più appropriato descrivere la dimensione dell'input con due numeri, anziché con un uno. Per esempio, se l'input di un algoritmo è un grafo, la dimensione dell'input può essere descritta dal numero di vertici e dal numero di lati del grafo. Per ogni problema analizzato dovremo indicare quale misura della dimensione di input sarà adottata.

Il ***tempo di esecuzione*** di un algoritmo per un particolare input è il numero di operazioni primitive che vengono eseguite o "passi". Conviene definire il concetto di passo nel modo più indipendente possibile dal tipo di macchina. Per il momento, adottiamo il seguente quadro di ipotesi. Per eseguire una riga del nostro pseudocodice occorre una quantità costante di tempo. Una riga può richiedere una quantità di tempo diversa da un'altra riga, tuttavia supporremo che ogni esecuzione dell'i-esima riga richieda un tempo c_i, dove c_i è una costante. Questa ipotesi è conforme al modello RAM e rispecchia anche il modo in cui lo pseudocodice può essere implementato in molti computer reali.[4]

Nella discussione che segue, la nostra espressione del tempo di esecuzione per INSERTION-SORT si evolverà da una formula grezza che usa tutti i costi c_i delle istruzioni a una notazione molto più semplice, concisa e facilmente manipolabile. Questa notazione semplificata renderà anche più facile determinare se un algoritmo è più efficiente di un altro.

Presentiamo, innanzi tutto, la procedura INSERTION-SORT con il tempo impiegato da ogni istruzione (costo) e il numero di volte che vengono eseguite le singole istruzioni. Per ogni $j = 2, 3, \ldots, n$, dove $n = lunghezza[A]$, indichiamo con t_j il

[4]Ci sono alcuni particolari da chiarire. I passi computazionali che specifichiamo in italiano spesso sono varianti di una procedura che richiede più di una quantità di tempo costante. Per esempio, più avanti in questo libro potremmo dire di "ordinare i punti in funzione della coordinata x"; come vedremo, questa operazione richiede più di una quantità di tempo costante. Notiamo inoltre che un'istruzione che chiama una subroutine impiega un tempo costante, sebbene la subroutine, una volta chiamata, possa impiegare di più. In altre parole, separiamo il processo della ***chiamata*** della subroutine – passare i parametri alla subroutine, ecc. – dal processo di ***esecuzione*** della subroutine.

numero di volte che il test del ciclo **while** nella riga 5 viene eseguito per quel valore di j. Quando un ciclo **for** o **while** termina nel modo consueto (come stabilito dal test all'inizio del ciclo), il test viene eseguito una volta di più del corpo del ciclo. Noi supponiamo che i commenti non siano istruzioni eseguibili e, quindi, il loro costo è nullo.

INSERTION-SORT(A)		*costo*	*numero di volte*
1	**for** $j \leftarrow 2$ **to** $lunghezza[A]$	c_1	n
2	**do** $chiave \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Inserisce $A[j]$ nella sequenza ordinata $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > chiave$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow chiave$	c_8	$n-1$

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione per ogni istruzione eseguita; un'istruzione che richiede c_i passi e viene eseguita n volte contribuirà con $c_i n$ al tempo di esecuzione totale.[5] Pe calcolare $T(n)$, il tempo di esecuzione di INSERTION-SORT, sommiamo i prodotti delle colonne *costo* e *numero di volte*, ottenendo

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) \\ &\quad + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1) \end{aligned}$$

Anche per più input della stessa dimensione, il tempo di esecuzione di un algoritmo può dipendere da *quale* input di quella dimensione viene scelto. Per esempio, in INSERTION-SORT il caso migliore si verifica se l'array è già ordinato. Per ogni $j = 2, 3, \ldots, n$, troviamo che $A[i] \leq chiave$ nella riga 5, quando i ha il suo valore iniziale $j-1$. Quindi $t_j = 1$ per $j = 2, 3, \ldots, n$ e il tempo di esecuzione nel caso migliore è

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Questo tempo di esecuzione può essere espresso come $an + b$, con le *costanti* a e b che dipendono dai costi delle istruzioni c_i; quindi è una ***funzione lineare*** di n.

Se l'array è ordinato in senso inverso – cioè in ordine decrescente – allora si verifica il caso peggiore. Dobbiamo confrontare ogni elemento $A[j]$ con ogni elemento dell'intero sottoarray ordinato $A[1\,..\,j-1]$, e quindi $t_j = j$ per $j = 2, 3, \ldots, n$.

[5] Questa caratteristica non è necessariamente valida per una risorsa come la memoria. Un'istruzione che fa riferimento a m parole di memoria e viene eseguita n volte non necessariamente consuma mn parole di memoria in totale.

Poiché

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

e

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

(consultate l'Appendice A per sapere come risolvere queste sommatorie), il tempo di esecuzione di INSERTION-SORT nel caso peggiore è

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Questo tempo di esecuzione può essere espresso come $an^2 + bn + c$, con le costanti a, b e c che, anche in questo caso, dipendono dei costi delle istruzioni c_i; quindi è una ***funzione quadratica*** di n. Tipicamente, come per insertion sort, il tempo di esecuzione di un algoritmo è fisso per un dato input, sebbene nei successivi capitoli vedremo alcuni interessanti algoritmi "randomizzati" il cui comportamento può variare anche con un input fisso.

Analisi del caso peggiore e del caso medio

Nell'analisi di insertion sort, abbiamo esaminato sia il caso migliore, in cui l'array di input era già ordinato, sia il caso peggiore, in cui l'array di input era ordinato alla rovescia. Nel seguito del libro, di solito, sono descritte le tecniche per determinare soltanto il ***tempo di esecuzione nel caso peggiore***, ovvero il tempo di esecuzione più lungo per *qualsiasi* input di dimensione n. Ci sono tre ragioni alla base di questo orientamento.

- Il tempo di esecuzione nel caso peggiore di un algoritmo è un limite superiore al tempo di esecuzione per qualsiasi input. Conoscendo questo tempo, abbiamo la garanzia che l'algoritmo non potrà impiegare di più. Non abbiamo bisogno di fare altre ipotesi sul tempo di esecuzione e sperare che questo tempo non venga mai superato.
- Per alcuni algoritmi, il caso peggiore si verifica molto spesso. Per esempio, nella ricerca di una particolare informazione in un database, il caso peggiore dell'algoritmo di ricerca si verifica ogni volta che l'informazione non è presente nel database. In alcune applicazioni di ricerca potrebbe essere frequente ricercare informazioni assenti.
- Il "caso medio" spesso è brutto quasi quanto quello peggiore. Supponete di avere scelto a caso n numeri e di applicare l'algoritmo insertion sort. Quanto tempo impiegherà l'algoritmo per determinare dove inserire l'elemento $A[j]$ nel sottoarray $A[1\,..\,j-1]$? In media, metà degli elementi di $A[1\,..\,j-1]$ sono più piccoli di $A[j]$, mentre gli altri elementi sono più grandi. In media, quindi, verifichiamo metà del sottoarray $A[1\,..\,j-1]$, pertanto t_j vale circa $j/2$.

Il tempo di esecuzione nel caso medio risulta dunque una funzione quadratica della dimensione dell'input, proprio come il tempo di esecuzione nel caso peggiore.

In alcuni casi particolari saremo interessati a determinare il tempo di esecuzione nel ***caso medio*** di un algoritmo, detto anche tempo di esecuzione ***previsto***. Nel Capitolo 5 vedremo la tecnica dell'***analisi probabilistica*** che permette di determinare il tempo di esecuzione previsto. Una difficoltà dell'analisi del caso medio, tuttavia, è che non è sempre evidente ciò che costituisce un input "medio" per un particolare problema. Spesso supporremo che tutti gli input di una data dimensione abbiano la stessa probabilità. In pratica questa ipotesi potrebbe essere invalidata, tuttavia in alcuni casi possiamo utilizzare un ***algoritmo randomizzato***, che effettua delle scelte casuali, per consentirci di svolgere l'analisi probabilistica.

Tasso di crescita

In precedenza abbiamo fatto alcune ipotesi per semplificare l'analisi della procedura INSERTION-SORT. Innanzi tutto, abbiamo ignorato il costo effettivo di ogni istruzione, utilizzando le costanti c_i per rappresentare questi costi. Poi, abbiamo osservato che anche queste costanti ci danno più dettagli del necessario: il tempo di esecuzione nel caso peggiore è $an^2 + bn + c$, con le costanti a, b e c che dipendono dai costi delle istruzioni c_i. Quindi, abbiamo ignorato non soltanto i costi effettivi delle istruzioni, ma anche i costi astratti c_i. Adesso faremo un'altra astrazione esemplificativa. È il ***tasso*** o ***livello di crescita*** del tempo di esecuzione che effettivamente ci interessa. Di conseguenza, consideriamo soltanto il termine iniziale di una formula (per esempio an^2), in quanto i termini di ordine inferiore sono relativamente insignificanti per grandi valori di n. Ignoriamo anche il coefficiente costante del termine iniziale, in quanto i fattori costanti sono meno significativi del tasso di crescita nel determinare l'efficienza computazionale per grandi input. Quindi, scriviamo che insertion sort, per esempio, ha un tempo di esecuzione nel caso peggiore pari a $\Theta(n^2)$ (che si pronuncia "teta di n al quadrato"). In questo capitolo adotteremo informalmente la notazione Θ, che sarà definita più precisamente nel Capitolo 3. Di solito, un algoritmo è considerato più efficiente di un altro se il suo tempo di esecuzione nel caso peggiore ha un tasso di crescita inferiore. A causa dei fattori costanti e dei termini di ordine inferiore, questa valutazione potrebbe essere errata per piccoli input. Tuttavia, per input sufficientemente grandi, un algoritmo $\Theta(n^2)$, per esempio, sarà eseguito più velocemente nel caso peggiore di un algoritmo $\Theta(n^3)$.

Esercizi

2.2-1

Esprimete la funzione $n^3/1000 - 100n^2 - 100n + 3$ nella notazione Θ.

2.2-2

Supponete di ordinare n numeri memorizzati nell'array A trovando prima il più piccolo elemento di A e scambiandolo con l'elemento in $A[1]$; poi, trovate il secondo elemento più piccolo di A e scambiatelo con $A[2]$. Continuate in questo modo per i primi $n - 1$ elementi di A. Scrivete lo pseudocodice per questo algoritmo, che è noto come ***selection sort*** (ordinamento per selezione). Quale invariante di ciclo conserva questo algoritmo? Perché basta eseguirlo soltanto per i primi $n - 1$ elementi, anziché per tutti gli n elementi? Esprimete nella notazione Θ i tempi di esecuzione nei casi migliore e peggiore dell'algoritmo selection sort.

2.2-3
Considerate di nuovo la ricerca lineare (Esercizio 2.1-3). Quanti elementi della sequenza di input devono essere esaminati in media, supponendo che l'elemento cercato ha la stessa probabilità di essere un elemento qualsiasi dell'array? Quanti elementi nel caso peggiore? Quali sono i tempi di esecuzione nei casi migliore e peggiore della ricerca lineare nella notazione Θ. Spiegate le vostre risposte.

2.2-4
Come possiamo modificare quasi tutti gli algoritmi in modo da avere un buon tempo di esecuzione nel caso migliore?

2.3 Progettare gli algoritmi

Ci sono varie tecniche per progettare gli algoritmi. Insertion sort usa un approccio ***incrementale***: dopo avere ordinato il sottoarray $A[1 \mathinner{..} j-1]$, inseriamo il singolo elemento $A[j]$ nella posizione appropriata, ottenendo il sottoarray ordinato $A[1 \mathinner{..} j]$. Nel prossimo paragrafo esamineremo un metodo di progettazione alternativo, noto come "divide et impera". Utilizzeremo questo metodo per progettare un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore è molto più piccolo di quello di insertion sort. Un vantaggio degli algoritmi divide et impera è che i loro tempi di esecuzione, spesso, possono essere facilmente determinati applicando le tecniche che saranno presentate nel Capitolo 4.

2.3.1 Il metodo divide et impera

Molti utili algoritmi sono ***ricorsivi*** nella struttura: per risolvere un determinato problema, questi algoritmi chiamano sé stessi in modo ricorsivo, una o più volte, per trattare sottoproblemi strettamente correlati. Tipicamente, gli algoritmi ricorsivi adottano un approccio ***divide et impera***: suddividono il problema in vari sottoproblemi, che sono simili al problema originale, ma di dimensioni più piccole, risolvono i sottoproblemi in modo ricorsivo e, poi, combinano le soluzioni per creare una soluzione del problema originale.

Il paradigma divide et impera prevede tre passi a ogni livello di ricorsione:

Divide: il problema viene diviso in un certo numero di sottoproblemi.

Impera: i sottoproblemi vengono risolti in modo ricorsivo. Tuttavia, se i sottoproblemi hanno una dimensione sufficientemente piccola, possono essere risolti in maniera semplice.

Combina: le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale.

L'algoritmo ***merge sort*** è conforme al paradigma divide et impera; intuitivamente, opera nel modo seguente.

Divide: divide la sequenza degli n elementi da ordinare in due sottosequenze di $n/2$ elementi ciascuna.

Impera: ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo merge sort.

Combina: fonde le due sottosequenze ordinate per generare la sequenza ordinata.

La ricorsione "tocca il fondo" quando la sequenza da ordinare ha lunghezza 1, nel qual caso non c'è più nulla da fare, in quanto ogni sequenza di lunghezza 1 è già ordinata.

L'operazione chiave dell'algoritmo merge sort è la fusione di due sottosequenze ordinate nel passo "combina". Per effettuare la fusione, utilizziamo una procedura ausiliaria MERGE(A, p, q, r), dove A è un array e p, q e r sono indici di numerazione degli elementi dell'array tali che $p \leq q < r$. La procedura suppone che i sottoarray $A[p\,..\,q]$ e $A[q+1\,..\,r]$ siano ordinati; li ***fonde*** per formare un unico sottoarray ordinato che sostituisce il sottoarray corrente $A[p\,..\,r]$.

La procedura MERGE impiega un tempo $\Theta(n)$, dove $n = r - p + 1$ è il numero di elementi da fondere, e opera nel modo seguente. Riprendendo l'esempio delle carte da gioco, supponiamo di avere sul tavolo due mazzi di carte scoperte. Ogni mazzo è ordinato, con le carte più piccole in alto. Vogliamo "fondere" i due mazzi in un unico mazzo ordinato di output, con le carte coperte. Il passo base consiste nello scegliere la più piccola fra le carte scoperte in cima ai due mazzi, togliere questa carta dal suo mazzo (scoprendo così una nuova carta in cima al mazzo) e deporla coperta sul mazzo di output. Ripetiamo questo passo finché un mazzo di input sarà vuoto; a questo punto, prendiamo le carte rimanenti del mazzo di input e le poniamo coperte sopra il mazzo di output. Da un punto di vista computazionale, ogni passo base impiega un tempo costante, in quanto operiamo soltanto con le due carte in cima ai mazzi di input. Poiché svolgiamo al massimo n passi base, la fusione dei mazzi impega un tempo $\Theta(n)$.

Il seguente pseudocodice implementa la precedente idea, con un espediente aggiuntivo che evita di dover controllare se i mazzi sono vuoti in ogni passo base.

MERGE(A, p, q, r)

```
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  crea gli array L[1 .. n1 + 1] e R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else  A[k] ← R[j]
17                  j ← j + 1
```

L'idea consiste nel porre in fondo a ogni mazzo una carta ***sentinella***, che contiene un valore speciale che usiamo per semplificare il nostro codice. In questo esempio usiamo ∞ come valore sentinella, in modo che quando si presenta una carta con ∞, essa non può essere la carta più piccola, a meno che entrambi i mazzi non abbiano esposto le loro sentinelle. Ma quando accade questo, tutte le carte non-sentinella sono state già poste nel mazzo di output. Poiché sappiamo in anticipo

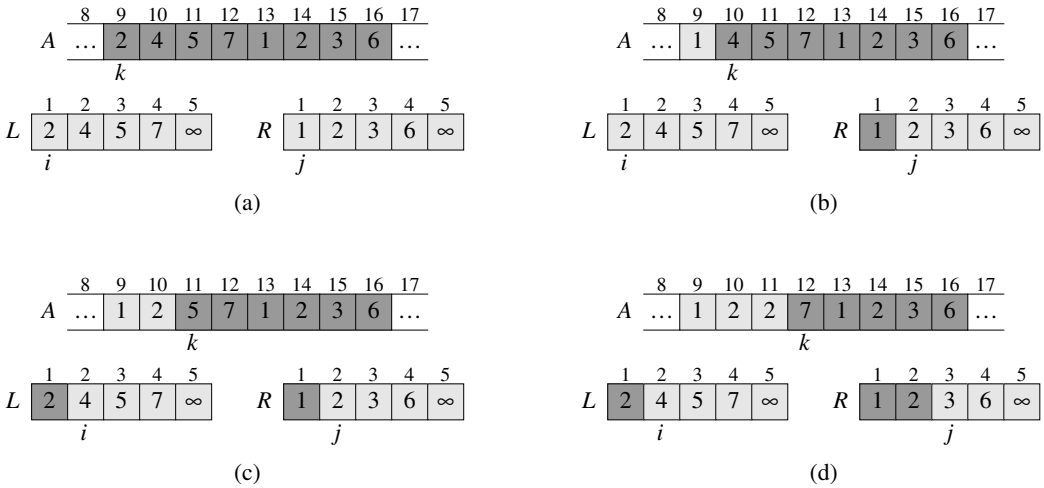


Figura 2.3 Il funzionamento delle righe 10–17 nella chiamata MERGE$(A, 9, 12, 16)$, quando il sottoarray $A[9\,..\,16]$ contiene la sequenza $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. Una volta copiate e inserite le sentinelle, l'array L contiene $\langle 2, 4, 5, 7, \infty\rangle$ e l'array R contiene $\langle 1, 2, 3, 6, \infty\rangle$. Le posizioni di colore grigio chiaro di A contengono i loro valori finali; le posizioni di colore grigio chiaro di L e R contengono i valori che devono essere copiati in A. Tutte insieme, le posizioni di colore grigio chiaro contengono sempre i valori che originariamente erano in $A[9\,..\,16]$ e le due sentinelle. Le posizioni di colore grigio scuro di A contengono i valori sui quali saranno copiati altri valori; le posizioni di colore grigio scuro di L e R contengono i valori che sono stati già copiati in A. **(a)–(h)** Gli array A, L e R e i loro rispettivi indici k, i e j prima di ogni iterazione del ciclo 12–17. **(i)** Gli array e gli indici alla fine del ciclo. A questo punto, il sottoarray in $A[9\,..\,16]$ è ordinato e le due sentinelle in L e R sono gli unici due elementi in questi array che non sono stati copiati in A.

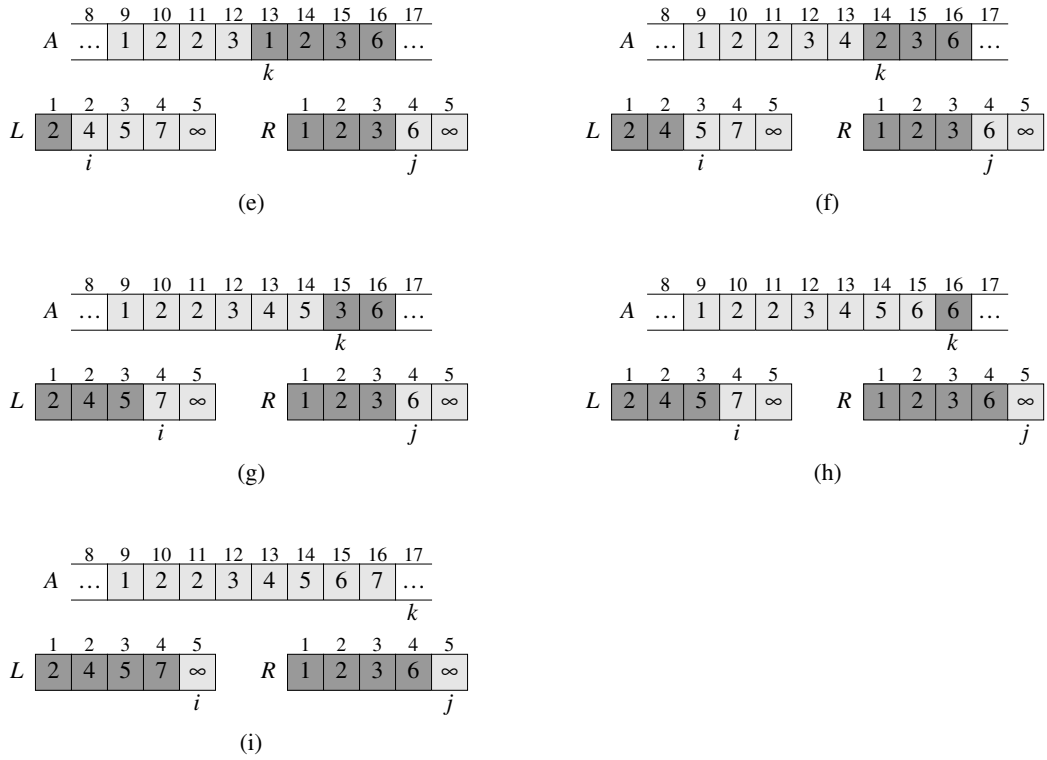
che saranno poste esattamente $r - p + 1$ carte nel mazzo di output, possiamo fermare il processo una volta che abbiamo svolto tutti i passi base.

In dettaglio, la procedura MERGE opera nel modo seguente. La riga 1 calcola la lunghezza n_1 del sottoarray $A[p..q]$; la riga 2 calcola la lunghezza n_2 del sottoarray $A[q + 1..r]$. Nella riga 3 creiamo gli array L e R (L sta per "left" o sinistro e R sta per "right" o destro), rispettivamente, di lunghezza $n_1 + 1$ e $n_2 + 1$. Il ciclo **for**, righe 4–5, copia il sottoarray $A[p\,..\,q]$ in $L[1\,..\,n_1]$; il ciclo **for**, righe 6–7, copia il sottoarray $A[q + 1\,..\,r]$ in $R[1\,..\,n_2]$. Le righe 8–9 pongono le sentinelle alla fine degli array L e R. Le righe 10–17, illustrate nella Figura 2.3, eseguono $r - p + 1$ passi base mantenendo la seguente invariante di ciclo:

> All'inizio di ogni iterazione del ciclo **for**, righe 12–17, il sottoarray $A[p\,..\,k-1]$ contiene $k - p$ elementi ordinati che sono i più piccoli di $L[1\,..\,n_1+1]$ e $R[1\,..\,n_2+1]$. Inoltre, $L[i]$ e $R[j]$ sono i più piccoli elementi dei loro array che non sono stati copiati in A.

Dobbiamo dimostrare che questa invariante di ciclo è valida prima della prima iterazione del ciclo **for**, righe 12–17, che ogni iterazione del ciclo conserva l'invariante e che l'invariante fornisce un'utile proprietà per dimostrare la correttezza quando il ciclo termina.

Inizializzazione: prima della prima iterazione del ciclo, abbiamo $k = p$, quindi il sottoarray $A[p\,..\,k-1]$ è vuoto. Questo sottoarray vuoto contiene $k - p = 0$ elementi, che sono i più piccoli di L e R; poiché $i = j = 1$, $L[i]$ e $R[j]$ sono gli elementi più piccoli dei loro array che non sono stati copiati in A.



Conservazione: per verificare che ogni iterazione conserva l'invariante di ciclo, supponiamo innanzi tutto che $L[i] \leq R[j]$; quindi $L[i]$ è l'elemento più piccolo che non è stato ancora copiato in A. Poiché $A[p\,..\,k-1]$ contiene i $k-p$ elementi più piccoli, dopo che la riga 14 ha copiato $L[i]$ in $A[k]$, il sottoarray $A[p\,..\,k]$ conterrà i $k-p+1$ elementi più piccoli. Incrementando k (ciclo **for**) e i (riga 15), si ristabilisce l'invariante di ciclo per la successiva iterazione. Se, invece, $L[i] > R[j]$, allora le righe 16–17 svolgono l'azione appropriata per conservare l'invariante di ciclo.

Conclusione: alla fine del ciclo, $k = r+1$. Per l'invariante di ciclo, il sottoarray $A[p\,..\,k-1]$, che è $A[p\,..\,r]$, contiene $k-p = r-p+1$ elementi ordinati che sono i più piccoli di $L[1\,..\,n_1+1]$ e $R[1\,..\,n_2+1]$. Gli array L e R contengono $n_1+n_2+2 = r-p+3$ elementi. Tutti gli elementi, tranne i due più grandi, sono stati copiati in A; questi due elementi sono le sentinelle.

Per verificare che la procedura MERGE viene eseguita nel tempo $\Theta(n)$, con $n = r-p+1$, notate che ciascuna delle righe 1–3 e 8–11 impiega un tempo costante, i cicli **for** (righe 4–7) impiegano un tempo $\Theta(n_1+n_2) = \Theta(n)$,[6] e ci sono n iterazioni del ciclo **for** (righe 12–17), ciascuna delle quali impiega un tempo costante. Adesso possiamo utilizzare la procedura MERGE come subroutine nell'algoritmo merge sort. La procedura MERGE-SORT(A, p, r) ordina gli elementi nel sottoarray $A[p\,..\,r]$. Se $p \geq r$, il sottoarray ha al massimo un elemento e, quindi, è già ordinato; altrimenti, il passo "divide" calcola semplicemente un indice q

[6] Il Capitolo 3 spiega come interpretare formalmente le equazioni che contengono la notazione Θ.

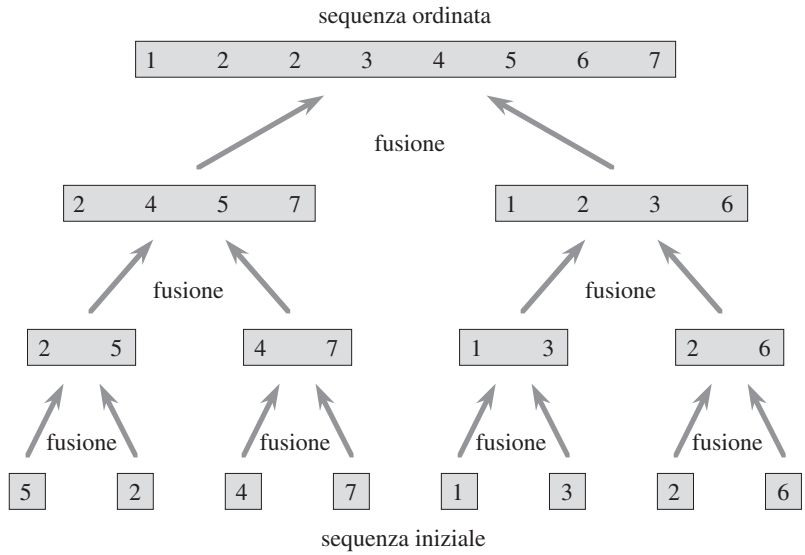


Figura 2.4
Funzionamento di merge sort con l'array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Le lunghezze delle sequenze ordinate da fondere aumentano via via che l'algoritmo procede dal basso verso l'alto.

che separa $A[p\,..\,r]$ in due sottoarray: $A[p\,..\,q]$, che contiene $\lceil n/2 \rceil$ elementi, e $A[q+1\,..\,r]$, che contiene $\lfloor n/2 \rfloor$ elementi.[7]

MERGE-SORT(A, p, r)

```
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Per ordinare l'intera sequenza $A = \langle A[1], A[2], \ldots, A[n] \rangle$, effettuiamo la chiamata iniziale MERGE-SORT$(A, 1, length[A])$, dove ancora una volta $length[A] = n$. La Figura 2.4 illustra il funzionamento della procedura dal basso verso l'alto, quando n è una potenza di 2. L'algoritmo consiste nel fondere coppie di sequenze di un elemento per formare sequenze ordinate di lunghezza 2, fondere coppie di sequenze di lunghezza 2 per formare sequenze ordinate di lunghezza 4 e così via, finché non si fonderanno due sequenze di lunghezza $n/2$ per formare l'ultima sequenza ordinata di lunghezza n.

2.3.2 Analisi degli algoritmi divide et impera

Quando un algoritmo contiene una chiamata ricorsiva a sé stesso, il suo tempo di esecuzione spesso può essere descritto con una ***equazione di ricorrenza*** o ***ricorrenza***, che esprime il tempo di esecuzione totale di un problema di dimensione n in funzione del tempo di esecuzione per input più piccoli. Poi è possibile utilizzare gli strumenti matematici per risolvere l'equazione di ricorrenza e stabilire i limiti delle prestazioni dell'algoritmo.

[7] L'espressione $\lceil x \rceil$ indica il più piccolo numero intero che è maggiore o uguale a x; $\lfloor x \rfloor$ indica il più grande numero intero che è minore o uguale a x. Queste notazioni sono definite nel Capitolo 3. Il sistema più semplice per verificare che impostando q a $\lfloor (p+r)/2 \rfloor$ si ottengono i sottoarray $A[p\,..\,q]$ e $A[q+1\,..\,r]$, rispettivamente, di dimensione $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$, consiste nell'esaminare i quattro casi che si presentano a seconda se p e r siano pari o dispari.

Una ricorrenza per il tempo di esecuzione di un algoritmo divide et impera si basa sui tre passi del paradigma di base. Come in precedenza, supponiamo che $T(n)$ sia il tempo di esecuzione di un problema di dimensione n. Se la dimensione del problema è sufficientemente piccola, per esempio $n \leq c$ per qualche costante c, la soluzione semplice richiede un tempo costante, che indichiamo con $\Theta(1)$. Supponiamo che la nostra suddivisione del problema generi a sottoproblemi e che la dimensione di ciascun sottoproblema sia $1/b$ la dimensione del problema originale (per merge sort, i valori di a e b sono entrambi pari a 2, ma vedremo vari algoritmi divide et impera in cui $a \neq b$). Se impieghiamo un tempo $D(n)$ per dividere il problema in sottoproblemi e un tempo $C(n)$ per combinare le soluzioni dei sottoproblemi nella soluzione del problema originale, otteniamo la ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{negli altri casi} \end{cases}$$

Nel Capitolo 4 vedremo come risolvere le ricorrenze comuni di questa forma.

Analisi di merge sort

Sebbene lo pseudocodice di MERGE-SORT funzioni correttamente quando il numero di elementi non è pari, la nostra analisi basata sulla ricorrenza si semplifica se supponiamo che la dimensione del problema originale sia una potenza di 2. Ogni passo *divide* genera due sottosequenze di dimensione esattamente pari a $n/2$. Nel Capitolo 4, vedremo che questa ipotesi non influisce sul tasso di crescita della soluzione della ricorrenza.

Per stabilire la ricorrenza per $T(n)$, il tempo di esecuzione nel caso peggiore di merge sort con n numeri, possiamo fare il seguente ragionamento. L'algoritmo merge sort applicato a un solo elemento impiega un tempo costante. Se abbiamo $n > 1$ elementi, suddividiamo il tempo di esecuzione nel modo seguente.

Divide: questo passo calcola semplicemente il centro del sottoarray. Ciò richiede un tempo costante, quindi $D(n) = \Theta(1)$.

Impera: risolviamo in modo ricorsivo i due sottoproblemi, ciascuno di dimensione $n/2$; ciò contribuisce con $2T(n/2)$ al tempo di esecuzione.

Combina: abbiamo già notato che la procedura MERGE con un sottoarray di n elementi richiede un tempo $\Theta(n)$, quindi $C(n) = \Theta(n)$.

Quando sommiamo le funzioni $D(n)$ e $C(n)$ per l'analisi di merge sort, stiamo sommando una funzione che è $\Theta(1)$ e una funzione che è $\Theta(n)$. Questa somma è una funzione lineare di n, cioè $\Theta(n)$. Sommandola al termine $2T(n/2)$ del passo "impera", si ottiene la ricorrenza per il tempo di esecuzione $T(n)$ nel caso peggiore di merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases} \tag{2.1}$$

Nel Capitolo 4 vedremo il "teorema dell'esperto", che possiamo utilizzare per dimostrare che $T(n)$ è $\Theta(n \lg n)$, dove $\lg n$ sta per $\log_2 n$. Poiché la funzione logaritmica cresce più lentamente di qualsiasi funzione lineare, per input sufficientemente grandi, l'algoritmo merge sort, con il suo tempo di esecuzione $\Theta(n \lg n)$, supera le prestazioni di insertion sort, il cui tempo di esecuzione è $\Theta(n^2)$, nel caso peggiore. Non occorre il teorema dell'esperto per capire perché la soluzione della ricorrenza (2.1) è $T(n) = \Theta(n \lg n)$.

Riscriviamo la ricorrenza (2.1) così

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases} \tag{2.2}$$

La costante c rappresenta sia il tempo richiesto per risolvere i problemi di dimensione 1 sia il tempo per elemento dell'array dei passi *divide* e *combina*.[8]

La Figura 2.5 mostra come possiamo risolvere la ricorrenza (2.2). Per comodità, supponiamo che n sia una potenza esatta di 2. La parte (a) della figura mostra $T(n)$, che nella parte (b) è stato espanso in un albero equivalente che rappresenta la ricorrenza. Il termine cn è la radice (il costo al primo livello di ricorsione) e i due sottoalberi della radice sono le due ricorrenze più piccole $T(n/2)$. La parte (c) mostra questo processo un passo più avanti con l'espansione di $T(n/2)$. Il costo per ciascuno dei due sottonodi al secondo livello di ricorsione è $cn/2$. Continuiamo a espandere i nodi nell'albero suddividendolo nelle sue componenti come stabilisce la ricorrenza, finché le dimensioni dei problemi si riducono a 1, ciascuno con un costo c. La parte (d) mostra l'albero risultante.

Sommiamo i costi per ogni livello dell'albero. Il primo livello ha un costo totale cn, il secondo livello ha un costo totale $c(n/2) + c(n/2) = cn$, il terzo livello ha un costo totale $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ e così via. In generale, il livello i sotto il primo ha 2^i nodi, ciascuno dei quali ha un costo $c(n/2^i)$, quindi l'i-esimo livello sotto il primo ha un costo totale $2^i\, c(n/2^i) = cn$. A livello più basso ci sono n nodi, ciascuno con un costo c, per un costo totale cn.

Il numero totale di livelli dell'*albero di ricorsione* nella Figura 2.5 è $\lg n + 1$. Questo può essere facilmente dimostrato con un ragionamento induttivo informale. Il caso base si verifica quando $n = 1$, nel qual caso c'è un solo livello. Poiché $\lg 1 = 0$, abbiamo che $\lg n + 1$ fornisce il numero corretto di livelli. Adesso supponiamo, come ipotesi induttiva, che il numero di livelli di un albero di ricorsione per 2^i nodi sia $\lg 2^i + 1 = i + 1$ (poiché per qualsiasi valore di i, si ha $\lg 2^i = i$). Poiché stiamo supponendo che la dimensione dell'input originale sia una potenza di 2, la successiva dimensione da considerare è 2^{i+1}. Un albero con 2^{i+1} nodi ha un livello in più di un albero con 2^i nodi; quindi il numero totale di livelli è $(i + 1) + 1 = \lg 2^{i+1} + 1$.

Per calcolare il costo totale rappresentato dalla ricorrenza (2.2), basta sommare i costi di tutti i livelli. Ci sono $\lg n + 1$ livelli, ciascuno di costo cn, per un costo totale di $cn(\lg n + 1) = cn \lg n + cn$. Ignorando il termine di ordine inferiore e la costante c, si ottiene il risultato desiderato $\Theta(n \lg n)$.

Esercizi

2.3-1

Utilizzando la Figura 2.4 come modello, illustrate l'operazione di merge sort sull'array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

---

[8] È improbabile che la stessa costante rappresenti esattamente sia il tempo richiesto per risolvere i problemi di dimensione 1 sia il tempo per elemento dell'array dei passi *divide* e *combina*. Possiamo aggirare questo problema, assegnando a c il valore più grande di questi tempi e accettando che la nostra ricorrenza impone un limite superiore al tempo di esecuzione oppure assegnando a c il valore più piccolo di questi tempi e accettando che la nostra ricorrenza impone un limite inferiore al tempo di esecuzione. Entrambi i limiti saranno nell'ordine di $n \lg n$ e, presi insieme, danno un tempo di esecuzione $\Theta(n \lg n)$.

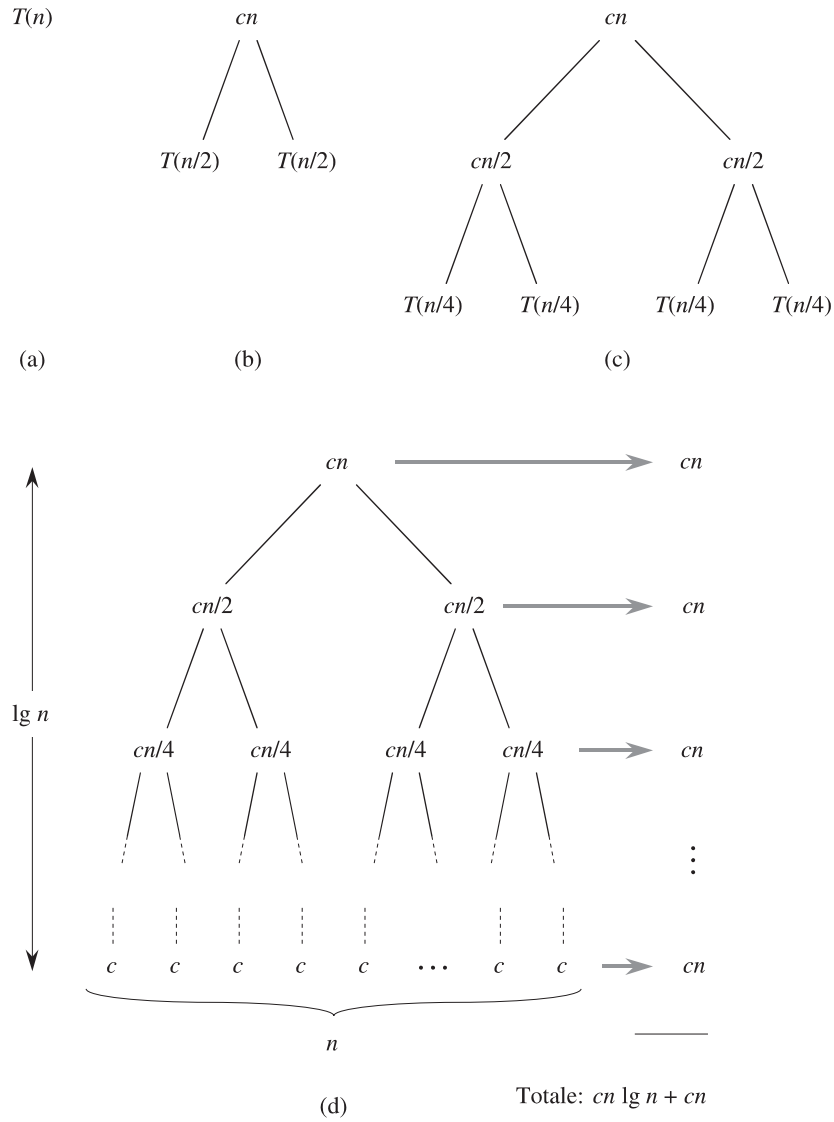


Figura 2.5
La costruzione di un albero di ricorsione per la ricorrenza $T(n) = 2T(n/2) + cn$. La parte **(a)** mostra $T(n)$, che viene progressivamente espanso in **(b)–(d)** per formare l'albero di ricorsione. L'albero completamente espanso nella parte (d) ha $\lg n + 1$ livelli (cioè ha un'altezza $\lg n$, come indicato) e ogni livello ha un costo cn. Di conseguenza, il costo totale è $cn \lg n + cn$, che è $\Theta(n \lg n)$.

2.3-2
Modificate la procedura MERGE in modo da non utilizzare le sentinelle; interrompete il processo quando tutti gli elementi di uno dei due array L e R sono stati copiati in A; poi copiate il resto dell'altro array in A.

2.3-3
Applicate l'induzione matematica per dimostrare che, se n è una potenza esatta di 2, allora $T(n) = n \lg n$ è la soluzione della ricorrenza

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

2.3-4
Insertion sort può essere espresso come una procedura ricorsiva nel modo seguente: per ordinare $A[1 \,..\, n]$, si ordina in modo ricorsivo $A[1 \,..\, n-1]$ e poi si inserisce $A[n]$ nell'array ordinato $A[1 \,..\, n-1]$. Scrivete una ricorrenza per il tempo di esecuzione di questa versione ricorsiva di insertion sort.

2.3-5
Riprendendo il problema della ricerca (Esercizio 2.1-3), notiamo che, se la sequenza A è ordinata, possiamo confrontare il punto centrale della sequenza con v ed escludere metà sequenza da ulteriori considerazioni. La ***ricerca binaria*** è un algoritmo che ripete questa procedura, dimezzando ogni volta la dimensione della porzione restante della sequenza. Scrivete uno pseudocodice, iterativo o ricorsivo, per la ricerca binaria. Dimostrate che il tempo di esecuzione nel caso peggiore della ricerca binaria è $\Theta(\lg n)$.

2.3-6
Il ciclo **while**, righe 5–7, della procedura INSERTION-SORT nel Paragrafo 2.1 usa la ricerca lineare per esplorare (a ritroso) il sottoarray ordinato $A[1 \,..\, j-1]$. È possibile usare la ricerca binaria (Esercizio 2.3-5) per migliorare il tempo di esecuzione complessivo nel caso peggiore di insertion sort fino a $\Theta(n \lg n)$?

2.3-7 $\star$
Descrivete un algoritmo con tempo $\Theta(n \lg n)$ che, dato un insieme S di n numeri interi e un altro intero x, determini se esistono due elementi in S la cui somma è esattamente x.

2.4 Problemi

2-1 Insertion sort su piccoli arrays in merge sort
Sebbene merge sort venga eseguito nel tempo $\Theta(n \lg n)$ nel caso peggiore e insertion sort venga eseguito nel tempo $\Theta(n^2)$ nel caso peggiore, i fattori costanti di insertion sort lo rendono più veloce per piccoli valori di n. Quindi, ha senso usare insertion sort all'interno di merge sort quando i sottoproblemi diventano sufficientemente piccoli. Considerate una versione modificata di merge sort in cui n/k sottoliste di lunghezza k siano ordinate utilizzando insertion sort e poi fuse mediante il meccanismo standard di merge sort; k è un valore da determinare.

a. Dimostrare che le n/k sottoliste, ciascuna di lunghezza k, possono essere ordinate da insertion sort nel tempo $\Theta(nk)$ nel caso peggiore.

b. Dimostrare che le sottoliste possono essere fuse nel tempo $\Theta(n \lg(n/k))$ nel caso peggiore.

c. Dato che l'algoritmo modificato viene eseguito nel tempo $\Theta(nk + n \lg(n/k))$ nel caso peggiore, qual è il massimo valore asintotico di k (notazione Θ) come funzione di n per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del meccanismo standard di merge sort?

d. In pratica, come dovrebbe essere scelto il valore di k?

2-2 Correttezza di bubblesort
Bubblesort è un noto algoritmo di ordinamento; opera scambiando ripetutamente gli elementi adiacenti che non sono ordinati.

BUBBLESORT(A)
1 **for** $i \leftarrow 1$ **to** $lunghezza[A]$
2 **do for** $j \leftarrow lunghezza[A]$ **downto** $i + 1$
3 **do if** $A[j] < A[j-1]$
4 **then** scambia $A[j] \leftrightarrow A[j-1]$

a. Indichiamo con A' l'output di BUBBLESORT(A). Per dimostrare che la procedura BUBBLESORT è corretta, bisogna verificare che termina e che

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n] \tag{2.3}$$

dove $n = \mathit{lunghezza}[A]$. Che altro bisogna verificare per dimostrare che BUBBLESORT ordina effettivamente gli elementi?

I prossimi due punti dimostrano la disuguaglianza (2.3).

b. Definite con precisione un'invariante di ciclo per il ciclo **for**, righe 2–4, e dimostrate che tale invariante è vera. La vostra dimostrazione dovrebbe usare la struttura della verifica delle invarianti di ciclo presentata in questo capitolo.

c. Utilizzando la condizione di conclusione dell'invariante di ciclo dimostrata nel punto (b), definite un'invariante di ciclo per il ciclo **for**, righe 1–4, che vi consentirà di dimostrare la disuguaglianza (2.3). La vostra dimostrazione dovrebbe usare la struttura della verifica delle invarianti di ciclo presentata in questo capitolo.

d. Qual è il tempo di esecuzione nel caso peggiore di bubblesort? Confrontatelo con il tempo di esecuzione di insertion sort.

2-3 Correttezza della regola di Horner

Il seguente frammento di codice implementa la regola di Horner per il calcolo di un polinomio

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots)) \end{aligned}$$

dati i coefficienti $a_0, a_1, \ldots, a_n$ e un valore di x:

```
1  y ← 0
2  i ← n
3  while i ≥ 0
4        do y ← a_i + x · y
5           i ← i − 1
```

a. Qual è il tempo di esecuzione asintotico di questo frammento di codice per la regola di Horner?

b. Scrivete uno pseudocodice per implementare un semplice algoritmo che calcola i termini del polinomio da zero. Qual è il tempo di esecuzione di questo algoritmo? Confrontatelo con la regola di Horner?

c. Dimostrate che la seguente definizione è un'invariante di ciclo per il ciclo **while**, righe 3 –5:

All'inizio di ogni iterazione del ciclo **while**, righe 3–5,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \; .$$

La vostra dimostrazione dovrebbe usare la struttura della verifica delle invarianti di ciclo presentata in questo capitolo e dovrebbe verificare che, alla fine, $y = \sum_{k=0}^{n} a_k x^k$.

d. Concludete dimostrando che il frammento di codice dato calcola correttamente un polinomio caratterizzato dai coefficienti $a_0, a_1, \ldots, a_n$.

2-4 Inversioni

Sia $A[1\,..\,n]$ un array di n numeri distinti. Se $i < j$ e $A[i] > A[j]$, allora la coppia (i, j) è detta ***inversione*** di A.

a. Elencate le cinque inversioni dell'array $\langle 2, 3, 8, 6, 1 \rangle$.

b. Quale array con elementi estratti dall'insieme $\{1, 2, \ldots, n\}$ ha più inversioni? Quante inversioni ha?

c. Qual è la relazione fra il tempo di esecuzione di insertion sort e il numero di inversioni nell'array di input? Spiegate la vostra risposta.

d. Create un algoritmo che determina il numero di inversioni in una permutazione di n elementi nel tempo $\Theta(n \lg n)$ nel caso peggiore (*suggerimento*: modificate merge sort).

Note

Nel 1968 Knuth pubblicò il primo di tre volumi con il titolo generale *The Art of Computer Programming* [182, 183, 185]. Il primo volume diede l'avvio allo studio moderno degli algoritmi per calcolatori, con particolare enfasi sull'analisi dei tempi di esecuzione; l'opera completa resta un importante riferimento per molti argomenti trattati nel nostro libro. Secondo Knuth, la parola "algoritmo" deriva da "al-Khowârizmî", un matematico persiano del nono secolo.

Aho, Hopcroft e Ullman [5] sostennero l'analisi asintotica degli algoritmi come uno strumento per confrontare prestazioni relative. Diffusero inoltre l'uso delle relazioni di ricorrenza per descrivere i tempi di esecuzione degli algoritmi ricorsivi.

Knuth [185] è autore di un trattato enciclopedico su molti algoritmi di ordinamento. Il confronto di questi algoritmi (pagina 381) include l'analisi del conteggio esatto dei passi, come quella che abbiamo fatto in questo capitolo per insertion sort. La discussione di Knuth sull'algoritmo insertion sort include numerose varianti dell'algoritmo. La più importante di queste è *Shell's sort*, introdotta da D. L. Shell, che applica insertion sort a sottosequenze periodiche dell'input per produrre un algoritmo di ordinamento più veloce.

Knuth ha descritto anche merge sort. Egli ricorda che nel 1938 fu inventato un collazionatore meccanico in grado di fondere due pacchi di schede perforate in un solo passo. Nel 1945 J. von Neumann, uno dei pionieri dell'informatica, apparentemente scrisse un programma di merge sort nel calcolatore EDVAC.

Gries [133] ha scritto la storia recente delle dimostrazioni della correttezza dei programmi; attribuisce a P. Naur il primo articolo in questo campo e a R. W. Floyd le invarianti di ciclo. Il libro di Mitchell [222] descrive i più recenti progressi nella dimostrazione della correttezza dei programmi.

Crescita delle funzioni 3

Il tasso di crescita del tempo di esecuzione di un algoritmo, definito nel Capitolo 2, fornisce una semplice caratterizzazione dell'efficienza dell'algoritmo; inoltre, ci consente di confrontare le prestazioni relative di algoritmi alternativi. Se la dimensione dell'input n diventa sufficientemente grande, l'algoritmo merge sort, con il suo tempo di esecuzione $\Theta(n \lg n)$ nel caso peggiore, batte insertion sort, il cui tempo di esecuzione nel caso peggiore è $\Theta(n^2)$. Sebbene a volte sia possibile determinare il tempo esatto di esecuzione di un algoritmo, come abbiamo fatto con insertion sort nel Capitolo 2, tuttavia l'estrema precisione, di solito, non compensa lo sforzo per ottenerla. Per input sufficientemente grandi, le costanti moltiplicative e i termini di ordine inferiore di un tempo esatto di esecuzione sono dominati dagli effetti della dimensione stessa dell'input.

Quando operiamo con dimensioni dell'input abbastanza grandi da rendere rilevante soltanto il tasso di crescita del tempo di esecuzione, stiamo studiando l'efficienza ***asintotica*** degli algoritmi. In altre parole, ci interessa sapere come aumenta il tempo di esecuzione di un algoritmo al crescere della dimensione dell'input *al limite*, quando la dimensione dell'input cresce senza limitazioni. Di solito, un algoritmo che è asintoticamente più efficiente sarà il migliore con tutti gli input, tranne con quelli molto piccoli.

Questo capitolo descrive diversi metodi standard per semplificare l'analisi asintotica degli algoritmi. Il prossimo paragrafo inizia definendo vari tipi di "notazioni asintotiche", di cui abbiamo già visto un esempio nella notazione Θ. Poi saranno presentate alcune convenzioni sulle notazioni che saranno adottate in tutto il libro. Infine, esamineremo il comportamento delle funzioni che comunemente si usano nell'analisi degli algoritmi.

3.1 Notazione asintotica

Le notazioni che usiamo per descrivere il tempo di esecuzione asintotico di un algoritmo sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbf{N} = \{0, 1, 2, \ldots\}$. Tali notazioni sono comode per descrivere la funzione del tempo di esecuzione nel caso peggiore $T(n)$, che di solito è definita soltanto con dimensioni intere dell'input. A volte, però, è lecito *abusare* della notazione asintotica in vari modi. Per esempio, la notazione viene facilmente estesa al dominio dei numeri reali o limitata a un sottoinsieme dei numeri naturali. È importante capire il significato esatto della notazione in modo che, quando se ne abusa, non venga *utilizzata male*. Questo paragrafo definisce le notazioni asintotiche di base e presenta anche alcuni tipici abusi.

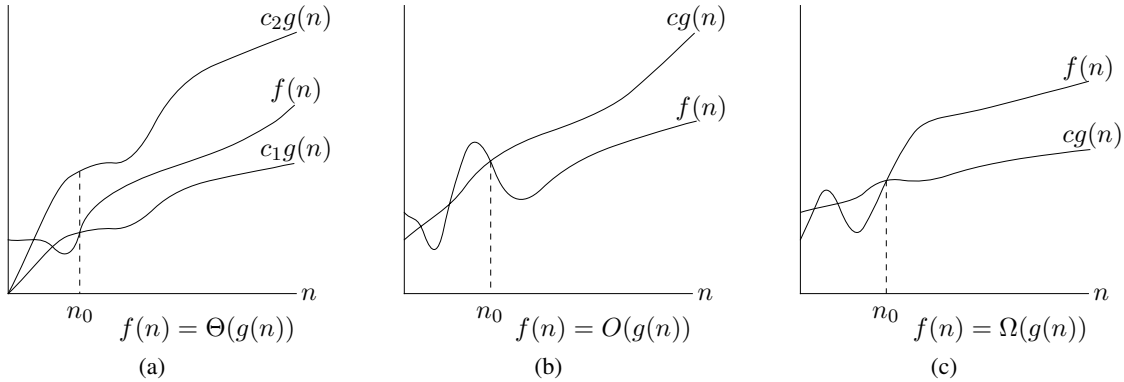


Figura 3.1 Esempi grafici delle notazioni Θ, O, e Ω. Nei tre casi il valore di n_0 mostrato è il più piccolo possibile; è accettabile anche un valore più grande. **(a)** La notazione Θ limita una funzione a meno di fattori costanti. Si scrive $f(n) = \Theta(g(n))$ se esistono delle costanti positive n_0, c_1 e c_2 tali che, a destra di n_0, il valore di $f(n)$ è sempre compreso fra $c_1g(n)$ e $c_2g(n)$, estremi inclusi. **(b)** La notazione O fornisce un limite superiore a una funzione a meno di un fattore costante. Si scrive $f(n) = O(g(n))$ se esistono delle costanti positive n_0 e c tali che, a destra di n_0, il valore di $f(n)$ è sempre uguale o minore di $cg(n)$. **(c)** La notazione Ω fornisce un limite inferiore a una funzione a meno di un fattore costante. Si scrive $f(n) = \Omega(g(n))$ se esistono delle costanti positive n_0 e c tali che, a destra di n_0, il valore di $f(n)$ è sempre uguale o maggiore di $cg(n)$.

Notazione Θ

Nel Capitolo2 abbiamo determinato che il tempo di esecuzione nel caso peggiore di insertion sort è $T(n) = \Theta(n^2)$. Definiamo adesso il significato di questa notazione. Per una data funzione $g(n)$, indichiamo con $\Theta(g(n))$ l'*insieme delle funzioni*

$$\Theta(g(n)) = \{f(n) : \text{ esistono delle costanti positive } c_1, c_2 \text{ e } n_0 \text{ tali che } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ per ogni } n \geq n_0\}\ .^1$$

Una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$ se esistono delle costanti positive c_1 e c_2 tali che essa possa essere compresa fra $c_1g(n)$ e $c_2g(n)$, per un valore sufficientemente grande di n. Poiché $\Theta(g(n))$ è un insieme, potremmo scrivere "$f(n) \in \Theta(g(n))$" per indicare che $f(n)$ è un membro di $\Theta(g(n))$. Invece, di solito, scriveremo "$f(n) = \Theta(g(n))$" per esprimere lo stesso concetto. Questo abuso del simbolo di uguaglianza per denotare l'appartenenza a un insieme, inizialmente, potrebbe sembrare poco chiaro, ma vedremo più avanti che ha i suoi vantaggi.

La Figura 3.1(a) presenta un quadro intuitivo delle funzioni $f(n)$ e $g(n)$, dove $f(n) = \Theta(g(n))$. Per tutti i valori di n a destra di n_0, il valore di $f(n)$ coincide o sta sopra $c_1g(n)$ e coincide o sta sotto $c_2g(n)$. In altre parole, per ogni $n \geq n_0$, la funzione $f(n)$ è uguale a $g(n)$ a meno di un fattore costante. Si dice che $g(n)$ è un ***limite asintoticamente stretto*** per $f(n)$.

La definizione di $\Theta(g(n))$ richiede che ogni membro di $f(n) \in \Theta(g(n))$ sia ***asintoticamente non negativo***, ovvero che $f(n)$ sia non negativa quando n è sufficientemente grande (una funzione ***asintoticamente positiva*** è positiva per qualsiasi valore sufficientemente grande di n). Di conseguenza, la funzione $g(n)$ stessa

[1] All'interno della notazione dell'insieme, i due punti (:) vanno letti "tale che".

deve essere asintoticamente non negativa, altrimenti l'insieme $\Theta(g(n))$ è vuoto. Pertanto, supporremo che ogni funzione utilizzata nella notazione Θ sia asintoticamente non negativa. Questa ipotesi vale anche per le altre notazioni asintotiche definite in questo capitolo.

Nel Capitolo 2 abbiamo introdotto un concetto informale della notazione Θ che equivaleva a escludere i termini di ordine inferiore e a ignorare il coefficiente del termine di ordine più elevato. Giustifichiamo brevemente questa intuizione utilizzando la definizione formale per dimostrare che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Per farlo, dobbiamo determinare le costanti positive c_1, c_2 e n_0 in modo che

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

per qualsiasi $n \geq n_0$. Dividendo per n^2, si ha

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

La disuguaglianza destra può essere resa valida per qualsiasi valore di $n \geq 1$ scegliendo $c_2 \geq 1/2$. Analogamente, la disuguaglianza sinistra può essere resa valida per qualsiasi valore di $n \geq 7$ scegliendo $c_1 \leq 1/14$. Quindi, scegliendo $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, possiamo verificare che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certamente è possibile scegliere altri valori delle costanti, ma la cosa importante è che esiste *qualche* scelta. Notate che queste costanti dipendono dalla funzione $\frac{1}{2}n^2 - 3n$; un'altra funzione che appartiene a $\Theta(n^2)$, di solito, richiede costanti differenti. Possiamo applicare anche la definizione formale per verificare che $6n^3 \neq \Theta(n^2)$. Supponiamo per assurdo che esistano le costanti c_2 e n_0 tali che $6n^3 \leq c_2 n^2$ per ogni $n \geq n_0$; ma allora $n \leq c_2/6$ e questo non può essere assolutamente vero per n arbitrariamente grande, in quanto c_2 è costante.

Intuitivamente, i termini di ordine inferiore di una funzione asintoticamente positiva possono essere ignorati nel determinare i limiti asintoticamente stretti, perché sono insignificanti per grandi valori di n. Una piccola frazione del termine di ordine più elevato è sufficiente a dominare i termini di ordine inferiore. Quindi, assegnando a c_1 un valore che è leggermente più piccolo del coefficiente del termine di ordine più elevato e a c_2 un valore che è leggermente più grande, è possibile soddisfare le disuguaglianze nella definizione della notazione Θ. In modo analogo, può essere ignorato il coefficiente del termine di ordine più elevato, in quanto esso cambia soltanto c_1 e c_2 di un fattore costante pari al coefficiente.

Come esempio consideriamo una funzione quadratica $f(n) = an^2 + bn + c$, dove a, b e c sono costanti e $a > 0$. Escludendo i termini di ordine inferiore e ignorando la costante, si ha $f(n) = \Theta(n^2)$. Formalmente, per dimostrare la stessa cosa, prendiamo le costanti $c_1 = a/4$, $c_2 = 7a/4$ e $n_0 = 2 \cdot \max((|b|/a), \sqrt{(|c|/a)})$. Il lettore può verificare che $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ per ogni $n \geq n_0$. In generale, per qualsiasi polinomio $p(n) = \sum_{i=0}^{d} a_i n^i$, dove i coefficienti a_i sono costanti e $a_d > 0$, si ha $p(n) = \Theta(n^d)$ (vedere Problema 3-1).

Poiché qualsiasi costante è un polinomio di grado 0, possiamo esprimere qualsiasi funzione costante come $\Theta(n^0)$ o $\Theta(1)$. Quest'ultima notazione, tuttavia, è

un abuso di second'ordine, perché non è chiaro quale variabile tende all'infinito.[2] Adotteremo spesso la notazione $\Theta(1)$ per indicare una costante o una funzione costante rispetto a qualche variabile.

Notazione *O*

La notazione Θ limita asintoticamente una funzione da sopra e da sotto. Quando abbiamo soltanto un ***limite asintotico superiore***, utilizziamo la notazione O. Per una data funzione $g(n)$, denotiamo con $O(g(n))$ (si legge "O grande di g di n" o semplicemente "O di g di n") l'insieme delle funzioni

$$O(g(n)) = \{f(n) : \text{ esistono delle costanti positive } c \text{ e } n_0 \text{ tali che } 0 \leq f(n) \leq cg(n) \text{ per ogni } n \geq n_0\}$$

La notazione O si usa per assegnare un limite superiore a una funzione, a meno di un fattore costante. La Figura 3.1(b) illustra il concetto intuitivo che sta dietro questa notazione. Per qualsiasi valore n a destra di n_0, il valore della funzione $f(n)$ coincide o sta sotto $cg(n)$.

Si scrive $f(n) = O(g(n))$ per indicare che una funzione $f(n)$ è un membro dell'insieme $O(g(n))$. Notate che $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, in quanto la notazione Θ è una nozione più forte della notazione O. Secondo la notazione della teoria degli insiemi possiamo scrivere $\Theta(g(n)) \subseteq O(g(n))$. Quindi, la nostra dimostrazione che qualsiasi funzione quadratica $an^2 + bn + c$, con $a > 0$, è in $\Theta(n^2)$ implica anche che tali funzioni quadratiche sono in $O(n^2)$. Ciò che può sembrare più sorprendente è che qualsiasi funzione *lineare* $an + b$ è in $O(n^2)$; questo è facilmente verificabile ponendo $c = a + |b|$ e $n_0 = 1$.

I lettori che hanno già visto la notazione O potrebbero trovare strano che noi scriviamo, per esempio, $n = O(n^2)$. Nella letteratura, la notazione O viene a volte utilizzata informalmente per descrivere i limiti asintoticamente stretti, ovvero ciò che noi abbiamo definito con la notazione Θ. In questo libro, tuttavia, quando scriviamo $f(n) = O(g(n))$, stiamo semplicemente affermando che qualche costante multipla di $g(n)$ è un limite asintotico superiore di $f(n)$, senza specificare quanto sia stretto il limite superiore. La distinzione fra limiti superiori asintotici e limiti asintoticamente stretti è diventata standard nella letteratura degli algoritmi.

Utilizzando la notazione O, spesso, è possibile descrivere il tempo di esecuzione di un algoritmo, esaminando semplicemente la struttura complessiva dell'algoritmo. Per esempio, la struttura con i cicli doppiamente annidati dell'algoritmo insertion sort del Capitolo 2 genera immediatamente un limite superiore $O(n^2)$ sul tempo di esecuzione nel caso peggiore: il costo di ogni iterazione del ciclo interno è limitato superiormente da $O(1)$ (costante), gli indici i e j sono entrambi al massimo n; il ciclo interno viene eseguito al massimo una volta per ognuna delle n^2 coppie di valori i e j.

Poiché la notazione O descrive un limite superiore, quando la utilizziamo per limitare il tempo di esecuzione nel caso peggiore di un algoritmo, abbiamo un limite sul tempo di esecuzione dell'algoritmo per ogni input. Quindi, il limite

[2] Il problema reale è che la nostra notazione ordinaria per le funzioni non distingue le funzioni dai valori. Nel lambda-calcolo, i parametri di una funzione sono specificati in modo chiaro: la funzione n^2 può essere scritta $\lambda n.n^2$ o anche $\lambda r.r^2$. Tuttavia, se adottassimo una notazione più rigorosa, complicheremmo le manipolazioni algebriche; per questo abbiamo scelto di tollerare l'abuso.

$O(n^2)$ sul tempo di esecuzione nel caso peggiore di insertion sort si applica anche al suo tempo di esecuzione per qualsiasi input. Il limite $\Theta(n^2)$ sul tempo di esecuzione nel caso peggiore di insertion sort, tuttavia, non implica un limite $\Theta(n^2)$ sul tempo di esecuzione di insertion sort per *qualsiasi* input. Per esempio, abbiamo visto nel Capitolo 2 che, quando l'input è già ordinato, insertion sort viene eseguito nel tempo $\Theta(n)$.

Tecnicamente, è un abuso dire che il tempo di esecuzione di insertion sort è $O(n^2)$, in quanto, per un dato n, il tempo effettivo di esecuzione varia a seconda del particolare input di dimensione n. Quando scriviamo "il tempo di esecuzione è $O(n^2)$", intendiamo dire che c'è una funzione $f(n)$ che è $O(n^2)$ tale che, per qualsiasi valore di n, indipendentemente da quale input di dimensione n venga scelto, il tempo di esecuzione con quell'input è limitato superiormente dal valore $f(n)$. In modo equivalente, intendiamo che il tempo di esecuzione nel caso peggiore è $O(n^2)$.

Notazione Ω

Così come la notazione O fornisce un limite asintotico *superiore* a una funzione, la notazione Ω fornisce un ***limite asintotico inferiore***. Per una data funzione $g(n)$, denotiamo con $\Omega(g(n))$ (si legge "Omega grande di g di n" o semplicemente "Omega di g di n") l'insieme delle funzioni

$$\Omega(g(n)) = \{f(n) : \text{ esistono delle costanti positive } c \text{ e } n_0 \text{ tali che } 0 \leq cg(n) \leq f(n) \text{ per ogni } n \geq n_0\}$$

Il concetto intuitivo che sta dietro la notazione Ω è illustrato nella Figura 3.1(c). Per tutti i valori di n a destra di n_0, il valore di $f(n)$ coincide o sta sopra $cg(n)$.

Dalle definizioni delle notazioni asintotiche che abbiamo visto finora, è facile dimostrare il seguente importante teorema (vedere Esercizio 3.1-5).

Teorema 3.1
Per ogni coppia di funzioni $f(n)$ e $g(n)$, si ha $f(n) = \Theta(g(n))$, se e soltanto se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$. ■

Come esempio applicativo di questo teorema, la nostra dimostrazione che $an^2 + bn + c = \Theta(n^2)$ per qualsiasi costante a, b e c, con $a > 0$, implica immediatamente che $an^2 + bn + c = \Omega(n^2)$ e $an^2 + bn + c = O(n^2)$. In pratica, anziché usare il Teorema 3.1 per ottenere i limiti asintotici superiore e inferiore dai limiti asintoticamente stretti, come abbiamo fatto per questo esempio, di solito lo usiamo per dimostrare i limiti asintoticamente stretti dai limiti asintotici superiore e inferiore.

Poiché la notazione Ω descrive un limite inferiore, quando la usiamo per limitare il tempo di esecuzione nel caso migliore di un algoritmo, implicitamente limitiamo anche il tempo di esecuzione dell'algoritmo con input arbitrari. Per esempio, il tempo di esecuzione nel caso migliore di insertion sort è $\Omega(n)$, che implica che il tempo di esecuzione di insertion sort è $\Omega(n)$.

Il tempo di esecuzione di insertion sort quindi è compreso fra $\Omega(n)$ e $O(n^2)$, in quanto si trova in una zona compresa tra una funzione lineare di n e una funzione quadratica di n. Inoltre, questi limiti sono asintoticamente i più stretti possibili: per esempio, il tempo di esecuzione di insertion sort non è $\Omega(n^2)$, in quanto esiste un input per il quale insertion sort viene eseguito nel tempo $\Theta(n)$ (per esempio,

quando l'input è già ordinato). Tuttavia, non è contraddittorio affermare che il tempo di esecuzione nel *caso peggiore* di insertion sort è $\Omega(n^2)$, in quanto esiste un input con il quale l'algoritmo impiega un tempo $\Omega(n^2)$. Quando diciamo che il *tempo di esecuzione* di un algoritmo è $\Omega(g(n))$, intendiamo che *indipendentemente da quale particolare input di dimensione n sia scelto per qualsiasi valore di n*, il tempo di esecuzione con quell'input è pari ad almeno una costante moltiplicata per $g(n)$, con n sufficientemente grande.

Notazione asintotica nelle equazioni e nelle disuguaglianze

Abbiamo già visto come la notazione asintotica possa essere utilizzata all'interno di formule matematiche. Per esempio, presentando la notazione O, abbiamo scritto "$n = O(n^2)$". Avremmo potuto scrivere anche $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Come vanno interpretate queste formule?

Quando la notazione asintotica sta da sola sul lato destro di un'equazione (o disuguaglianza), come in $n = O(n^2)$, abbiamo già definito il segno uguale per indicare l'appartenenza all'insieme: $n \in O(n^2)$. In generale, però, quando la notazione asintotica appare in una formula, va interpretata come se indicasse qualche funzione anonima, di cui non è importante fare il nome. Per esempio, la formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ significa che $2n^2 + 3n + 1 = 2n^2 + f(n)$, dove $f(n)$ è qualche funzione dell'insieme $\Theta(n)$. In questo caso, $f(n) = 3n + 1$, che appartiene effettivamente a $\Theta(n)$.

Utilizzando la notazione asintotica in questo modo, è possibile eliminare dettagli superflui e poco chiari da un'equazione. Per esempio, nel Capitolo 2 abbiamo espresso il tempo di esecuzione nel caso peggiore di merge sort come la ricorrenza

$$T(n) = 2T(n/2) + \Theta(n)$$

Se siamo interessati soltanto al comportamento asintotico di $T(n)$, non è importante specificare con esattezza tutti i termini di ordine inferiore; è sottointeso che essi siano tutti inclusi nella funzione anonima indicata dal termine $\Theta(n)$.

Il numero di funzioni anonime in un'espressione è sottointeso che sia uguale al numero di volte che appare la notazione asintotica; per esempio, nell'espressione

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

c'è una sola funzione anonima (una funzione di i). Questa espressione *non* è quindi la stessa cosa di $O(1) + O(2) + \cdots + O(n)$ che, in effetti, non ha una chiara interpretazione.

In alcuni casi, la notazione asintotica si trova sul lato sinistro di un'equazione, come in

$$2n^2 + \Theta(n) = \Theta(n^2)$$

Per interpretare simili equazioni, applichiamo la seguente regola: *Indipendentemente dal modo in cui vengano scelte le funzioni anonime a sinistra del segno uguale, c'è un modo di scegliere le funzioni anonime a destra del segno uguale per rendere valida l'equazione*. Quindi, il significato del nostro esempio è che per *qualsiasi* funzione $f(n) \in \Theta(n)$, c'è *qualche* funzione $g(n) \in \Theta(n^2)$ tale che $2n^2 + f(n) = g(n)$ per ogni n. In altre parole, il lato destro di un'equazione fornisce un livello di dettaglio più grossolano del lato sinistro.

Simili relazioni potrebbero essere concatenate in questo modo

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Applicando la precedente regola, possiamo interpretare separatamente le singole equazioni. La prima equazione indica che esiste *qualche* funzione $f(n) \in \Theta(n)$ tale che $2n^2 + 3n + 1 = 2n^2 + f(n)$ per ogni n. La seconda equazione indica che per *qualsiasi* funzione $g(n) \in \Theta(n)$ (come la funzione $f(n)$ appena citata), c'è *qualche* funzione $h(n) \in \Theta(n^2)$ tale che $2n^2 + g(n) = h(n)$ per ogni n. Notate che questa interpretazione implica che $2n^2 + 3n + 1 = \Theta(n^2)$, che è quanto intuitivamente indicano le equazioni concatenate.

Notazione *o*

Il limite asintotico superiore fornito dalla notazione O può essere asintoticamente stretto oppure no. Il limite $2n^2 = O(n^2)$ è asintoticamente stretto, mentre il limite $2n = O(n^2)$ non lo è. Utilizziamo la notazione o per denotare un limite superiore che non è asintoticamente stretto. Definiamo formalmente $o(g(n))$ (si legge "o piccolo di g di n") come l'insieme

$$\begin{aligned} o(g(n)) = \{f(n) :\ &\text{per qualsiasi costante } c > 0, \text{ esiste una costante} \\ &n_0 > 0 \text{ tale che } 0 \leq f(n) < cg(n) \text{ per ogni } n \geq n_0\} \end{aligned}$$

Per esempio, $2n = o(n^2)$, ma $2n^2 \neq o(n^2)$.

Le definizioni delle notazioni O e o sono simili. La differenza principale è che in $f(n) = O(g(n))$ il limite $0 \leq f(n) \leq cg(n)$ vale per *qualche* costante $c > 0$, mentre in $f(n) = o(g(n))$ il limite $0 \leq f(n) < cg(n)$ vale per *tutte* le costanti $c > 0$. Intuitivamente, nella notazione o la funzione $f(n)$ diventa insignificante rispetto a $g(n)$ quando n tende all'infinito; ovvero

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \tag{3.1}$$

Alcuni autori usano questo limite come definizione della notazione o; la definizione in questo libro vincola anche le funzioni anonime a essere asintoticamente non negative.

Notazione ω

Per analogia, la notazione ω sta alla notazione Ω come la notazione o sta alla notazione O. Utilizziamo la notazione ω per indicare un limite inferiore che non è asintoticamente stretto. Un modo per definirla è il seguente

$f(n) \in \omega(g(n))$ se e soltanto se $g(n) \in o(f(n))$

Formalmente, tuttavia, definiamo $\omega(g(n))$ ("omega piccolo di g di n") come l'insieme

$$\begin{aligned} \omega(g(n)) = \{f(n) :\ &\text{per qualsiasi costante } c > 0, \text{ esiste una costante} \\ &n_0 > 0 \text{ tale che } 0 \leq cg(n) < f(n) \text{ per ogni } n \geq n_0\} \end{aligned}$$

Per esempio, $n^2/2 = \omega(n)$, ma $n^2/2 \neq \omega(n^2)$. La relazione $f(n) = \omega(g(n))$ implica che

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

se il limite esiste; cioè $f(n)$ diventa arbitrariamente grande rispetto a $g(n)$ quando n tende all'infinito.

Confronto di funzioni

Molte delle proprietà delle relazioni dei numeri reali si applicano anche ai confronti asintotici. Supponiamo che $f(n)$ e $g(n)$ siano asintoticamente positive.

Proprietà transitiva:

$f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ implicano $f(n) = \Theta(h(n))$
$f(n) = O(g(n))$ e $g(n) = O(h(n))$ implicano $f(n) = O(h(n))$
$f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ implicano $f(n) = \Omega(h(n))$
$f(n) = o(g(n))$ e $g(n) = o(h(n))$ implicano $f(n) = o(h(n))$
$f(n) = \omega(g(n))$ e $g(n) = \omega(h(n))$ implicano $f(n) = \omega(h(n))$

Proprietà riflessiva:

$f(n) = \Theta(f(n))$
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

Proprietà simmetrica:

$f(n) = \Theta(g(n))$ se e soltanto se $g(n) = \Theta(f(n))$

Simmetria trasposta:

$f(n) = O(g(n))$ se e soltanto se $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ se e soltanto se $g(n) = \omega(f(n))$

Poiché queste proprietà sono valide per le notazioni asintotiche, è possibile definire un'analogia fra il confronto asintotico di due funzioni f e g e il confronto di due numeri reali a e b:

$f(n) = O(g(n)) \approx a \leq b$
$f(n) = \Omega(g(n)) \approx a \geq b$
$f(n) = \Theta(g(n)) \approx a = b$
$f(n) = o(g(n)) \approx a < b$
$f(n) = \omega(g(n)) \approx a > b$

Diciamo che $f(n)$ è ***asintoticamente più piccola*** di $g(n)$ se $f(n) = o(g(n))$ e $f(n)$ è ***asintoticamente più grande*** di $g(n)$ se $f(n) = \omega(g(n))$. C'è una proprietà dei numeri reali che non può essere trasferita alla notazione asintotica:

Tricotomia: se a e b sono due numeri reali qualsiasi, deve essere valida una sola delle seguenti relazioni: $a < b$, $a = b$, $a > b$.

Sebbene sia possibile confrontare due numeri reali qualsiasi, non tutte le funzioni sono asintoticamente confrontabili; ovvero, date due funzioni $f(n)$ e $g(n)$, potrebbe accadere che non sia vero che $f(n) = O(g(n))$ né che $f(n) = \Omega(g(n))$. Per esempio, le funzioni n e $n^{1+\sin n}$ non possono essere confrontate mediante la notazione asintotica, in quanto il valore dell'esponente di $n^{1+\sin n}$ oscilla fra 0 e 2, assumendo tutti i valori intermedi.

Esercizi

3.1-1
Se $f(n)$ e $g(n)$ sono funzioni asintoticamente non negative, usate la definizione di base della notazione Θ per dimostrare che $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1-2
Dimostrate che per qualsiasi costante reale a e b, con $b > 0$,

$$(n + a)^b = \Theta(n^b) \tag{3.2}$$

3.1-3
Spiegate perché l'asserzione "il tempo di esecuzione dell'algoritmo A è almeno $O(n^2)$" è priva di significato.

3.1-4
È vero che $2^{n+1} = O(2^n)$? È vero che $2^{2n} = O(2^n)$?

3.1-5
Dimostrate il Teorema 3.1.

3.1-6
Dimostrate che il tempo di esecuzione di un algoritmo è $\Theta(g(n))$ se e soltanto se il suo tempo di esecuzione nel caso peggiore è $O(g(n))$ e quello nel caso migliore è $\Omega(g(n))$.

3.1-7
Dimostrate che $o(g(n)) \cap \omega(g(n))$ è l'insieme vuoto.

3.1-8
Possiamo estendere la nostra notazione al caso di due parametri n e m che possono tendere indipendentemente all'infinito con tassi di crescita differenti. Per una data funzione $g(n, m)$, indichiamo con $O(g(n, m))$ l'insieme delle funzioni

$$O(g(n,m)) = \{f(n,m) : \text{ esistono delle costanti positive } c,\ n_0 \text{ e } m_0 \text{ tali che } 0 \le f(n,m) \le cg(n,m) \text{ per ogni } n \ge n_0 \text{ o } m \ge m_0\}$$

Date le corrispondenti definizioni per $\Omega(g(n, m))$ e $\Theta(g(n, m))$.

3.2 Notazioni standard e funzioni comuni

Questo paragrafo presenta alcune funzioni e notazioni matematiche standard ed esamina le loro relazioni; descrive anche l'uso delle notazioni asintotiche.

Funzioni monotòne

Una funzione $f(n)$ è ***monotonicamente crescente*** se $m \le n$ implica $f(m) \le f(n)$; analogamente, $f(n)$ è ***monotonicamente decrescente*** se $m \le n$ implica $f(m) \ge f(n)$. Una funzione $f(n)$ è ***strettamente crescente*** se $m < n$ implica $f(m) < f(n)$ e ***strettamente decrescente*** se $m < n$ implica $f(m) > f(n)$.

Floor e ceiling

Dato un numero reale x, indichiamo con $\lfloor x \rfloor$ (si legge "floor di x") l'intero più grande che è minore o uguale a x e con $\lceil x \rceil$ (si legge "ceiling di x") l'intero più piccolo che è maggiore o uguale a x. Per qualsiasi numero reale x

$$x - 1 \ < \ \lfloor x \rfloor \ \le \ x \ \le \ \lceil x \rceil \ < \ x + 1 \tag{3.3}$$

Per qualsiasi numero intero n

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

Per qualsiasi numero reale $n \geq 0$ e due interi $a, b > 0$

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \quad (3.4)$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor \quad (3.5)$$

$$\lceil a/b \rceil \leq (a + (b-1))/b \quad (3.6)$$

$$\lfloor a/b \rfloor \geq (a - (b-1))/b \quad (3.7)$$

La funzione floor $f(x) = \lfloor x \rfloor$ e la funzione ceiling $f(x) = \lceil x \rceil$ sono monotonicamente crescenti.

Aritmetica modulare

Per qualsiasi intero a e qualsiasi intero positivo n, il valore $a \bmod n$ è il ***resto*** (o ***residuo***) del quoziente a/n:

$$a \bmod n = a - \lfloor a/n \rfloor n \quad (3.8)$$

Una volta definito il resto della divisione fra due numeri interi, è comodo utilizzare una notazione speciale per indicare l'uguaglianza dei resti. Se $(a \bmod n) = (b \bmod n)$, scriviamo $a \equiv b \pmod n$ e diciamo che a è ***equivalente*** a b, modulo n. In altre parole, $a \equiv b \pmod n$ se a e b hanno lo stesso resto quando sono divisi per n. In modo equivalente, $a \equiv b \pmod n$ se e soltanto se n è un divisore di $b - a$. Scriviamo $a \not\equiv b \pmod n$ se a non è equivalente a b, modulo n.

Polinomi

Dato un intero non negativo d, un ***polinomio in n di grado d*** è una funzione $p(n)$ della forma

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

dove le costanti $a_0, a_1, \ldots, a_d$ sono i ***coefficienti*** del polinomio e $a_d \neq 0$. Un polinomio è asintoticamente positivo se e soltanto se $a_d > 0$. Per un polinomio asinoticamente positivo $p(n)$ di grado d, si ha $p(n) = \Theta(n^d)$. Per qualsiasi costante reale $a \geq 0$, la funzione n^a è monotonicamente crescente e per qualsiasi costante reale $a \leq 0$, la funzione n^a è monotonicamente decrescente. Si dice che una funzione $f(n)$ è ***polinomialmente limitata*** se $f(n) = O(n^k)$ per qualche costante k.

Esponenziali

Per qualsiasi reale $a > 0$, m e n, si hanno le seguenti identità:

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^{-1} &= 1/a \\ (a^m)^n &= a^{mn} \\ (a^m)^n &= (a^n)^m \\ a^m a^n &= a^{m+n} \end{aligned}$$

Per qualsiasi n e $a \geq 1$, la funzione a^n è monotonicamente crescente in n. Per comodità, assumeremo $0^0 = 1$.

Le velocità di crescita delle funzioni polinomiali ed esponenziali possono essere correlate per il seguente fatto. Per tutte le costanti a e b, con $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 \tag{3.9}$$

da cui possiamo concludere che

$$n^b = o(a^n)$$

Quindi, qualsiasi funzione esponenziale con una base strettamente maggiore di 1 cresce più rapidamente di qualsiasi funzione polinomiale.

Se usiamo e per indicare $2.71828\ldots$, la base dei logaritmi naturali, otteniamo per qualsiasi valore reale x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{3.10}$$

Il simbolo "!" indica la funzione fattoriale (definita successivamente). Per ogni reale x, abbiamo la disuguaglianza

$$e^x \geq 1 + x \tag{3.11}$$

Qui l'uguaglianza vale soltanto se $x = 0$. Quando $|x| \leq 1$, abbiamo l'approssimazione

$$1 + x \leq e^x \leq 1 + x + x^2 \tag{3.12}$$

Quando $x \to 0$, l'approssimazione di e^x con $1 + x$ è abbastanza buona:

$$e^x = 1 + x + \Theta(x^2)$$

(In questa equazione la notazione asintotica è usata per descrivere il comportamento al limite per $x \to 0$, anziché per $x \to \infty$.) Per ogni x si ha

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x \tag{3.13}$$

Logaritmi

Adotteremo le seguenti notazioni:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(logaritmo binario)} \\
\ln n &= \log_e n && \text{(logaritmo naturale)} \\
\lg^k n &= (\lg n)^k && \text{(elevamento a potenza)} \\
\lg \lg n &= \lg(\lg n) && \text{(composizione)}
\end{aligned}$$

Un'importante convenzione che adotteremo con queste notazioni è che *le funzioni logaritmiche si applicano soltanto al termine successivo nella formula*, quindi $\lg n + k$ significa $(\lg n) + k$, non $\lg(n + k)$. Per $b > 1$ e $n > 0$, la funzione $\log_b n$ è strettamente crescente. Per qualsiasi reale $a > 0$, $b > 0$, $c > 0$ e n, si ha

$$\begin{aligned}
a &= b^{\log_b a} \\
\log_c(ab) &= \log_c a + \log_c b
\end{aligned}$$

$$\begin{aligned}
\log_b a^n &= n \log_b a \\
\log_b a &= \frac{\log_c a}{\log_c b} \qquad (3.14) \\
\log_b(1/a) &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b c} &= c^{\log_b a} \qquad (3.15)
\end{aligned}$$

In tutte queste equazioni le basi dei logaritmi sono diverse da 1.

Per l'equazione (3.14), cambiando la base di un logaritmo da una costante all'altra, cambia soltanto il valore del logaritmo per un fattore costante, quindi useremo spesso la notazione "$\lg n$" quando i fattori costanti non sono importanti, come nella notazione O. Gli scienziati informatici ritengono che 2 sia la base più naturale dei logaritmi, perché molti algoritmi e strutture dati richiedono la suddivisione di un problema in due parti.

C'è un semplice sviluppo in serie di $\ln(1+x)$ quando $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots$$

Abbiamo anche le seguenti disuguaglianze per $x > -1$:

$$\frac{x}{1+x} \;\le\; \ln(1+x) \;\le\; x \qquad (3.16)$$

L'uguaglianza vale soltanto se $x = 0$.

Una funzione $f(n)$ è detta ***polilogaritmicamente limitata*** se $f(n) = O(\lg^k n)$ per qualche costante k. Per correlare la crescita dei polinomi con quella dei polilogaritmi, sostituiamo nell'equazione (3.9) n con $\lg n$ e a con 2^a; otteniamo

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0$$

Da questo limite, possiamo concludere che

$$\lg^b n = o(n^a)$$

per qualsiasi costante $a > 0$. Quindi, una funzione polinomiale positiva cresce più rapidamente di una funzione polilogaritmica.

Fattoriali

La notazione $n!$ (si legge "n fattoriale") è definita per i numeri interi $n \ge 0$:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

Quindi, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Un limite superiore meno stretto per la funzione fattoriale è $n! \le n^n$, in quanto ciascuno degli n termini nel prodotto fattoriale è al massimo n. L'***approssimazione di Stirling***

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \qquad (3.17)$$

dove e è la base del logaritmo naturale, fornisce un limite superiore più stretto e anche un limite inferiore. È possibile dimostrare che (vedere Esercizio 3.2-3)

$$\begin{aligned} n! &= o(n^n) \\ n! &= \omega(2^n) \\ \lg(n!) &= \Theta(n \lg n) \end{aligned} \tag{3.18}$$

dove l'approssimazione di Stirling è utile per dimostrare l'equazione (3.18). La seguente equazione vale anche per qualsiasi $n \geq 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.19}$$

dove

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \tag{3.20}$$

Iterazione di funzione

Usiamo la notazione $f^{(i)}(n)$ per denotare la funzione $f(n)$ applicata iterativamente i volte a un valore iniziale di n. Formalmente, sia $f(n)$ una funzione definita sui reali. Per interi non negativi i, definiamo in modo ricorsivo

$$f^{(i)}(n) = \begin{cases} n & \text{se } i = 0 \\ f(f^{(i-1)}(n)) & \text{se } i > 0 \end{cases}$$

Per esempio, se $f(n) = 2n$, allora $f^{(i)}(n) = 2^i n$.

La funzione logaritmica iterata

Utilizziamo la notazione $\lg^* n$ (si legge "log stella di n") per denotare l'algoritmo iterato, che è definito nel modo seguente. Sia $\lg^{(i)} n$ una funzione definita come nel precedente paragrafo, con $f(n) = \lg n$. Poiché l'algoritmo di un numero non positivo non è definito, la funzione $\lg^{(i)} n$ è definita soltanto se $\lg^{(i-1)} n > 0$. Non bisogna confondere $\lg^{(i)} n$ (la funzione logaritmica applicata i volte in successione, a partire dall'argomento n) con $\lg^i n$ (il logaritmo di n elevato alla i-esima potenza). La funzione logaritmica iterata è definita così

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

Il logaritmo iterato è una funzione che cresce *molto* lentamente:

$$\begin{aligned} \lg^* 2 &= 1 \\ \lg^* 4 &= 2 \\ \lg^* 16 &= 3 \\ \lg^* 65536 &= 4 \\ \lg^*(2^{65536}) &= 5 \end{aligned}$$

Poiché si stima che il numero di atomi nell'universo visibile sia pari a circa 10^{80}, che è molto più piccolo di 2^{65536}, raramente potremo incontrare un input di dimensione n tale che $\lg^* n > 5$.

Numeri di Fibonacci

I ***numeri di Fibonacci*** sono definiti dalla seguente ricorrenza:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 \end{aligned} \tag{3.21}$$

Poiché ogni numero di Fibonacci è la somma dei due numeri precedenti, si ottiene la sequenza

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

I numeri di Fibonacci sono correlati al ***rapporto aureo*** ϕ e al suo coniugato $\widehat{\phi}$, che sono dati dalle seguenti formule:

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} = 1.61803\ldots \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} = -.61803\ldots \end{aligned} \tag{3.22}$$

Più precisamente, si ha

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} \tag{3.23}$$

Questa relazione può essere dimostrata per induzione (vedere Esercizio 3.2-6). Poiché $|\widehat{\phi}| < 1$, si ha $|\widehat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, quindi l'$i$-esimo numero di Fibonacci F_i è uguale a $\phi^i/\sqrt{5}$ arrotondato all'intero più vicino. Dunque, i numeri di Fibonacci crescono in modo esponenziale.

Esercizi

3.2-1
Dimostrate che, se $f(n)$ e $g(n)$ sono funzioni monotonicamente crescenti, allora lo sono anche le funzioni $f(n) + g(n)$ e $f(g(n))$; se $f(n)$ e $g(n)$ sono anche non negative, allora $f(n) \cdot g(n)$ è monotonicamente crescente.

3.2-2
Dimostrate l'equazione (3.15).

3.2-3
Dimostrate l'equazione (3.18). Dimostrate inoltre che $n! = \omega(2^n)$ e $n! = o(n^n)$.

3.2-4 $\star$
La funzione $\lceil \lg n \rceil!$ è polinomialmente limitata? La funzione $\lceil \lg \lg n \rceil!$ è polinomialmente limitata?

3.2-5 $\star$
Quale funzione è asintoticamente più grande: $\lg(\lg^* n)$ o $\lg^*(\lg n)$?

3.2-6
Dimostrate per induzione che l'i-esimo numero di Fibonacci soddisfa la seguente relazione (ϕ è il rapporto aureo e $\widehat{\phi}$ è il suo coniugato):

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}$$

3.2-7
Dimostrate che per $i \geq 0$, l'$(i+2)$-esimo numero di Fibonacci soddisfa la relazione $F_{i+2} \geq \phi^i$.

3.3 Problemi

3-1 Comportamento asintotico di polinomi

Dato il seguente polinomio in n di grado d (con $a_d > 0$ e k costante):

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

Applicate le definizioni delle notazioni asintotiche per dimostrare le seguenti proprietà:

a. Se $k \geq d$, allora $p(n) = O(n^k)$.

b. Se $k \leq d$, allora $p(n) = \Omega(n^k)$.

c. Se $k = d$, allora $p(n) = \Theta(n^k)$.

d. Se $k > d$, allora $p(n) = o(n^k)$.

e. Se $k < d$, allora $p(n) = \omega(n^k)$.

3-2 Crescite asintotiche relative

Indicate, per ogni coppia di espressioni (A, B) della seguente tabella, se A è O, o, Ω, ω, o Θ di B. Supponete che $k \geq 1$, $\epsilon > 0$ e $c > 1$ siano costanti. Inserite le risposte ("sì" o "no") in ogni casella della tabella.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3 Classificare le funzioni per velocità di crescita

a. Ordinate le seguenti funzioni per velocità di crescita; ovvero trovate una disposizione $g_1, g_2, \ldots, g_{30}$ delle funzioni che soddisfano le relazioni $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Suddividete il vostro elenco in classi di equivalenza in modo tale che $f(n)$ e $g(n)$ si trovino nella stessa classe se e soltanto se $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Indicate una funzione $f(n)$ non negativa che, per ogni funzione $g_i(n)$ del punto (a), non sia $O(g_i(n))$ né $\Omega(g_i(n))$.

3-4 Proprietà della notazione asintotica

Siano $f(n)$ e $g(n)$ due funzioni asintoticamente positive. Dimostrate la veridicità o falsità delle seguenti congetture.

a. $f(n) = O(g(n))$ implica $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implica $\lg(f(n)) = O(\lg(g(n)))$, dove $\lg(g(n)) \geq 1$ e $f(n) \geq 1$ per ogni n sufficientemente grande.

d. $f(n) = O(g(n))$ implica $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

f. $f(n) = O(g(n))$ implica $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5 Varianti di O e Ω

Alcuni autori definiscono Ω in un modo un po' diverso dal nostro; usiamo $\overset{\infty}{\Omega}$ (si legge "Omega infinito") per questa definizione alternativa. Diciamo che $f(n) = \overset{\infty}{\Omega}(g(n))$ se esiste una costante positiva c tale che $f(n) \geq cg(n) \geq 0$ per un numero infinitamente grande di interi n.

a. Dimostrate che per ogni coppia di funzioni $f(n)$ e $g(n)$, asintoticamente non negative, valgono entrambe le relazioni $f(n) = O(g(n))$ e $f(n) = \overset{\infty}{\Omega}(g(n))$ o una sola di esse, mentre ciò non è vero se si usa Ω al posto di $\overset{\infty}{\Omega}$.

b. Descrivete i vantaggi e svantaggi potenziali di usare $\overset{\infty}{\Omega}$, anziché Ω, per caratterizzare i tempi di esecuzione dei programmi.

Alcuni autori definiscono O in modo un po' diverso; usiamo O' per la definizione alternativa. Diciamo che $f(n) = O'(g(n))$ se e soltanto se $|f(n)| = O(g(n))$.

c. Che cosa accade per ciascuna direzione della clausola "se e soltanto se" nel Teorema 3.1, se sostituiamo O con O', mantenendo Ω?

Alcuni autori definiscono $\widetilde{O}$ (si legge "O tilde") per indicare O con fattori logaritmici ignorati:

$$\widetilde{O}(g(n)) = \{f(n) : \text{ esistono delle costanti positive } c, k \text{ e } n_0 \text{ tali che } 0 \leq f(n) \leq cg(n)\lg^k(n) \text{ per ogni } n \geq n_0\}$$

d. Definite $\widetilde{\Omega}$ e $\widetilde{\Theta}$ in modo analogo. Dimostrate il corrispondente Teorema 3.1.

3-6 Funzioni iterate

L'operatore di iterazione * usato nella funzione $\lg^*$ può essere applicato a qualsiasi funzione monotonicamente crescente $f(n)$ nei numeri reali. Per una data costante $c \in \mathbf{R}$, definiamo la funzione iterata f_c^* in questo modo:

$$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\}$$

Questa funzione non ha bisogno di essere ben definita in tutti i casi. In altre parole, la quantità $f_c^*(n)$ è il numero di applicazioni ripetute della funzione f che sono necessarie per ridurre il suo argomento a un valore minore o uguale a c.

Per ciascuna delle seguenti funzioni $f(n)$ e costanti c, specificate il limite più stretto possibile per $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Note

Secondo Knuth [182] l'origine della notazione O risale a un testo sulla teoria dei numeri scritto da P. Bachmann nel 1892. La notazione o è stata inventata da E. Landau nel 1909 con le sue argomentazioni sulla distribuzione dei numeri primi. Knuth [186] sostenne l'applicazione delle notazioni Ω e Θ per correggere la pratica diffusa, ma tecnicamente poco precisa, di usare la notazione O per entrambi i limiti superiore e inferiore. Molti continuano a usare la notazione O nei casi in cui la notazione Θ sarebbe tecnicamente più precisa. Per ulteriori informazioni sulla storia e lo sviluppo delle notazioni asintotiche, consultate Knuth [182, 186] e Brassard e Bratley [46].

Non tutti gli autori definiscono le notazioni asintotiche nello stesso modo, sebbene le varie definizioni concordino nella maggior parte delle situazioni più comuni. Alcune definizioni alternative includono funzioni che non sono asintoticamente non negative, finché i loro valori assoluti sono appropriatamente limitati.

L'equazione (3.19) è dovuta a Robbins [260]. Per altre proprietà delle funzioni matematiche di base, consultate un buon testo di matematica, come Abramowitz e Stegun [1] o Zwillinger [320], o un libro di calcolo, come Apostol [18] o Thomas e Finney [296]. I testi di Knuth [182] e di Graham, Knuth e Patashnik [132] contengono materiale abbondante sulla matematica discreta applicata all'informatica.

4 Ricorrenze

Come detto nel Paragrafo 2.3.2, quando un algoritmo contiene una chiamata ricorsiva a sé stessa, il suo tempo di esecuzione spesso può essere descritto da una ricorrenza. Una ***ricorrenza*** è un'equazione o disequazione che descrive una funzione in termini del suo valore con input più piccoli. Per esempio, come visto nel Paragrafo 2.3.2, il tempo di esecuzione $T(n)$ nel caso peggiore della procedura MERGE-SORT può essere descritto dalla ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases} \tag{4.1}$$

la cui soluzione è $T(n) = \Theta(n \lg n)$.

Questo capitolo presenta tre metodi per risolvere le ricorrenze – cioè per ottenere i limiti asintotici "Θ" o "O" nella soluzione. Nel ***metodo di sostituzione***, ipotizziamo un limite e poi usiamo l'induzione matematica per dimostrare che la nostra ipotesi è corretta. Il ***metodo dell'albero di ricorsione*** converte la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione; per risolvere la ricorrenza, adotteremo delle tecniche che limitano le sommatorie. Il ***metodo dell'esperto*** fornisce i limiti sulle ricorrenze nella forma

$$T(n) = aT(n/b) + f(n)$$

dove $a \geq 1$, $b > 1$ e $f(n)$ è una funzione data. Questo metodo richiede la memorizzazione di tre casi, ma fatto questo, è facile determinare i limiti asintotici per molte ricorrenze semplici.

Dettagli tecnici

In pratica, quando definiamo e risolviamo le ricorrenze, trascuriamo alcuni dettagli tecnici. Un buon esempio di dettaglio che viene spesso ignorato è supporre che gli argomenti delle funzioni siano numeri interi. Normalmente, il tempo di esecuzione $T(n)$ di un algoritmo è definito soltanto quando n è un intero, in quanto per la maggior parte degli algoritmi, la dimensione dell'input è sempre un numero intero. Per esempio, la ricorrenza che descrive il tempo di esecuzione nel caso peggiore di MERGE-SORT è effettivamente

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{4.2}$$

Le condizioni al contorno rappresentano un'altra classe di dettagli che tipicamente trascuriamo. Poiché il tempo di esecuzione di un algoritmo con un input di dimensione costante è una costante, le ricorrenze che derivano dai tempi di esecuzione degli algoritmi, generalmente, hanno $T(n) = \Theta(1)$ per valori sufficientemente piccoli di n. Per comodità, quindi, di solito ometteremo le definizioni delle

condizioni al contorno delle ricorrenze e assumeremo che $T(n)$ sia costante per n piccolo. Per esempio, normalmente definiamo la ricorrenza (4.1) così

$$T(n) = 2T(n/2) + \Theta(n) \tag{4.3}$$

senza dare esplicitamente i valori per n piccolo. La ragione sta nel fatto che, sebbene una variazione di $T(1)$ cambi la soluzione della ricorrenza, tuttavia la soluzione tipicamente non cambia per più di un fattore costante, quindi il tasso di crescita resta immutato.

Quando definiamo e risolviamo le ricorrenze, spesso omettiamo le condizioni al contorno, floor e ceiling. Procederemo senza questi dettagli e in seguito determineremo se sono importanti oppure no; di solito non lo sono, tuttavia è bene sapere quando lo sono. L'esperienza aiuta, come pure alcuni teoremi che stabiliscono che questi dettagli non influiscono sui limiti asintotici di molte ricorrenze che si incontrano nell'analisi degli algoritmi (vedere il Teorema 4.1). In questo capitolo, tuttavia, ci occuperemo di alcuni di questi dettagli per dimostrare le caratteristiche più peculiari dei metodi di risoluzione delle ricorrenze.

4.1 Il metodo di sostituzione

Il metodo di sostituzione per risolvere le ricorrenze richiede due passaggi:

1. Ipotizzare la forma della soluzione.
2. Usare l'induzione matematica per trovare le costanti e dimostrare che la soluzione funziona.

Il nome del metodo deriva dalla sostituzione della soluzione ipotizzata nella funzione quando l'ipotesi induttiva viene applicata a valori più piccoli. Questo metodo è potente, ma ovviamente può essere applicato soltanto nei casi in cui sia facile immaginare la forma della soluzione.

Il metodo di sostituzione può essere usato per determinare il limite inferiore o superiore di una ricorrenza. Come esempio, determiniamo un limite superiore per la ricorrenza

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{4.4}$$

che è simile alle ricorrenze (4.2) e (4.3). Supponiamo che la soluzione sia $T(n) = O(n \lg n)$. Il nostro metodo consiste nel dimostrare che $T(n) \leq cn \lg n$ per una scelta appropriata della costante $c > 0$. Supponiamo, innanzi tutto, che questo limite sia valido per $\lfloor n/2 \rfloor$, ovvero che $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Facendo le opportune sostituzioni nella ricorrenza, si ha

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

L'ultimo passo è vero finché $c \geq 1$.

A questo punto, l'induzione matematica richiede di dimostrare che la nostra soluzione vale per le condizioni al contorno. Tipicamente, questo è fatto dimostran-

do che le condizioni al contorno sono appropriate come casi base della dimostrazione induttiva. Per la ricorrenza (4.4), dobbiamo dimostrare che è possibile scegliere una costante c sufficientemente grande in modo che il limite $T(n) \leq cn \lg n$ sia valido anche per le condizioni al contorno. Questa necessità a volte può creare dei problemi. Supponiamo per esempio che $T(1) = 1$ sia l'unica condizione al contorno della ricorrenza. Allora per $n = 1$, il limite $T(n) \leq cn \lg n$ diventa $T(1) \leq c1 \lg 1 = 0$, che è in contrasto con $T(1) = 1$. Di conseguenza, il caso base della nostra dimostrazione induttiva non risulta valido.

Questa difficoltà nel dimostrare un'ipotesi induttiva per una specifica condizione al contorno può essere facilmente superata. Per esempio, nella ricorrenza (4.4), sfruttiamo la notazione asintotica che ci richiede soltanto di provare che $T(n) \leq cn \lg n$ per $n \geq n_0$, dove n_0 è una costante arbitrariamente scelta. L'idea è quella di escludere la difficile condizione al contorno $T(1) = 1$ dalla dimostrazione induttiva. Osservate che per $n > 3$, la ricorrenza non dipende direttamente da $T(1)$. Pertanto, possiamo sostituire $T(1)$ con $T(2)$ e $T(3)$ come casi base della dimostrazione induttiva, ponendo $n_0 = 2$. Notate che facciamo una distinzione fra il caso base della ricorrenza ($n = 1$) e i casi base della dimostrazione induttiva ($n = 2$ e $n = 3$). Dalla ricorrenza otteniamo che $T(2) = 4$ e $T(3) = 5$. La dimostrazione induttiva che $T(n) \leq cn \lg n$ per qualche costante $c \geq 1$ adesso può essere completata scegliendo c sufficientemente grande in modo che $T(2) \leq c2 \lg 2$ e $T(3) \leq c3 \lg 3$. Come si può vedere, è sufficiente scegliere un valore $c \geq 2$ per rendere validi i casi base di $n = 2$ e $n = 3$. Per la maggior parte delle ricorrenze che esamineremo, è facile estendere le condizioni al contorno in modo che l'ipotesi induttiva sia valida per piccoli valori di n.

Formulare una buona ipotesi

Purtroppo non esiste un metodo generale per formulare l'ipotesi della soluzione corretta di una ricorrenza. Per indovinare una soluzione bisogna avere esperienza e, a volte, creatività. Fortunatamente, esistono alcune euristiche che ci aiutano a diventare buoni risolutori. Per formulare delle buone ipotesi, è anche possibile utilizzare gli alberi di ricorsione, che descriveremo nel Paragrafo 4.2.

Se una ricorrenza è simile a una che avete già visto, allora ha senso provare una soluzione analoga. Per esempio, considerate la ricorrenza

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$$

che sembra difficile per l'aggiunta del "17" nell'argomento di T. Intuitivamente, però, questo termine aggiuntivo non può influire in modo sostanziale sulla soluzione della ricorrenza. Quando n è grande, la differenza fra $T(\lfloor n/2 \rfloor)$ e $T(\lfloor n/2 \rfloor + 17)$ non è così grande: in entrambi i casi, n viene diviso all'incirca a metà. Di conseguenza, facciamo l'ipotesi che $T(n) = O(n \lg n)$, la cui validità può essere dimostrata applicando il metodo di sostituzione (vedere l'Esercizio 4.1-5).

Un altro modo per fare una buona ipotesi consiste nel dimostrare la validità di limiti meno severi sulla ricorrenza e, poi, nel ridurre progressivamente il grado di incertezza. Per esempio, potremmo iniziare con un limite inferiore di $T(n) = \Omega(n)$ per la ricorrenza (4.4), in quanto abbiamo il termine n nella ricorrenza, e provare un limite superiore iniziale pari a $T(n) = O(n^2)$. Poi, potremmo ridurre gradualmente il limite superiore e alzare quello inferiore fino a convergere alla soluzione corretta e asintoticamente stretta $T(n) = \Theta(n \lg n)$.

Finezze

Ci sono casi in cui è possibile ipotizzare correttamente un limite asintotico sulla soluzione di una ricorrenza, ma in qualche modo sembra che i calcoli matematici non tornino nell'induzione. Di solito, il problema è che l'ipotesi induttiva non è abbastanza forte per dimostrare il limite dettagliato. Quando ci si imbatte in simili ostacoli, spesso basta correggere l'ipotesi sottraendo un termine di ordine inferiore per far tornare i conti. Consideriamo la seguente ricorrenza

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

Supponiamo che la soluzione sia $O(n)$; proviamo a dimostrare che $T(n) \leq cn$ per una costante c appropriatamente scelta. Sostituendo la nostra ipotesi nella ricorrenza, otteniamo

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

che non implica $T(n) \leq cn$ per qualsiasi c. Saremmo tentati di provare un'ipotesi più ampia, per esempio $T(n) = O(n^2)$, che può funzionare, ma in effetti la nostra ipotesi che la soluzione sia $T(n) = O(n)$ è corretta. Per provarlo, però, dobbiamo formulare un'ipotesi induttiva più forte.

Intuitivamente, la nostra ipotesi è quasi esatta: non vale soltanto per la costante 1, un termine di ordine inferiore. Nonostante questo, l'induzione matematica non funziona, a meno che non dimostriamo la forma esatta dell'ipotesi induttiva. Superiamo questa difficoltà *sottraendo* un termine di ordine inferiore dalla precedente ipotesi. La nuova ipotesi è $T(n) \leq cn - b$, dove $b \geq 0$ è costante. Adesso abbiamo

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

finché $b \geq 1$. Come prima, la costante c deve essere scelta sufficientemente grande per gestire le condizioni al contorno.

Molti ritengono che l'idea si sottrarre un termine di ordine inferiore non sia intuitiva. Dopo tutto, se i calcoli matematici non tornano, non dovremmo ampliare la nostra ipotesi? Il segreto per capire questo passaggio sta nel ricordarsi che stiamo applicando l'induzione matematica: possiamo dimostrare qualcosa di più forte per un dato valore supponendo qualcosa di più forte per valori più piccoli.

Evitare i tranelli

È facile sbagliare a usare la notazione asintotica. Per esempio, nella ricorrenza (4.4) potremmo "dimostrare" erroneamente che $T(n) = O(n)$, supponendo che $T(n) \leq cn$ e poi deducendo che

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \qquad \Longleftarrow \textit{sbagliato!} \end{aligned}$$

in quanto c è una costante. L'errore sta nel fatto che non abbiamo dimostrato la *forma esatta* dell'ipotesi induttiva, ovvero che $T(n) \leq cn$.

Sostituzione di variabili

A volte, una piccola manipolazione algebrica può rendere una ricorrenza incognita simile a una che avete già visto. Per esempio, la seguente ricorrenza sembra difficile da risolvere

$$T(n) = 2T\left(\lfloor\sqrt{n}\rfloor\right) + \lg n$$

Tuttavia, è possibile semplificare questa ricorrenza con una sostituzione di variabili. Per comodità, ignoreremo l'arrotondamento agli interi di valori come $\sqrt{n}$. Ponendo $m = \lg n$ si ottiene

$$T(2^m) = 2T(2^{m/2}) + m$$

Adesso poniamo $S(m) = T(2^m)$ per ottenere la nuova ricorrenza

$$S(m) = 2S(m/2) + m$$

che è molto simile alla ricorrenza (4.4). In effetti, questa nuova ricorrenza ha la stessa soluzione: $S(m) = O(m \lg m)$. Ripristinando $T(n)$, otteniamo $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg\lg n)$.

Esercizi

4.1-1
Dimostrate che la soluzione di $T(n) = T(\lceil n/2\rceil) + 1$ è $O(\lg n)$.

4.1-2
Abbiamo visto che la soluzione di $T(n) = 2T(\lfloor n/2\rfloor) + n$ è $O(n \lg n)$. Dimostrate che la soluzione di questa ricorrenza è anche $\Omega(n \lg n)$. In conclusione, la soluzione è $\Theta(n \lg n)$.

4.1-3
Dimostrate che, formulando una diversa ipotesi induttiva, è possibile superare la difficoltà della condizione al contorno $T(1) = 1$ per la ricorrenza (4.4), senza bisogno di modificare le condizioni al contorno per la dimostrazione induttiva.

4.1-4
Dimostrate che $\Theta(n \lg n)$ è la soluzione della ricorrenza (4.2) "esatta" per merge sort.

4.1-5
Dimostrate che la soluzione di $T(n) = 2T(\lfloor n/2\rfloor + 17) + n$ è $O(n \lg n)$.

4.1-6
Risolvete la ricorrenza $T(n) = 2T(\sqrt{n}) + 1$ mediante una sostituzione di variabili. Dovreste ottenere una soluzione asintoticamente stretta. Non preoccupatevi se i valori sono interi.

4.2 Il metodo dell'albero di ricorsione

Sebbene il metodo di sostituzione possa fornire una prova succinta che una soluzione di una ricorrenza sia corretta, a volte è difficile formulare una buona ipotesi per la soluzione. Disegnare un albero di ricorsione, come abbiamo fatto nella nostra analisi della ricorrenza di merge sort nel Paragrafo 2.3.2, è una tecnica semplice per ideare una buona ipotesi.

In un ***albero di ricorsione*** ogni nodo rappresenta il costo di un singolo sottoproblema da qualche parte nell'insieme delle chiamate ricorsive di funzione. Sommiamo i costi all'interno di ogni livello dell'albero per ottenere un insieme di costi per livello; poi sommiamo tutti i costi per livello per determinare il costo totale di tutti i livelli della ricorsione. Gli alberi di ricorsione sono particolarmente utili quando la ricorrenza descrive il tempo di esecuzione di un algoritmo divide et impera.

Un albero di ricorsione è un ottimo metodo per ottenere una buona ipotesi, che poi viene verificata con il metodo di sostituzione. Quando si usa un albero di ricorsione per generare una buona ipotesi, spesso si tollera una certa dose di "approssimazione", in quanto l'ipotesi sarà verificata in un secondo momento. Tuttavia, se prestate particolare attenzione quando create l'albero di ricorsione e sommate i costi, potete usare l'albero di ricorsione come prova diretta di una soluzione della ricorrenza. In questo paragrafo, utilizzeremo l'albero di ricorsione per generare buone ipotesi; nel Paragrafo 4.4, utilizzeremo gli alberi di ricorsione direttamente per dimostrare il teorema che forma la base del metodo dell'esperto.

Per esempio, vediamo come un albero di ricorsione possa fornire una buona ipotesi per la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Iniziamo a ricercare un limite superiore per la soluzione. Poiché sappiamo che floor e ceiling di solito non influiscono sulla risoluzione delle ricorrenze (ecco un esempio di approssimazione che possiamo tollerare), creiamo un albero di ricorsione per la ricorrenza $T(n) = 3T(n/4) + cn^2$, ricordando che il coefficiente $c > 0$ è costante.

Nella Figura 4.1 è illustrata la derivazione dell'albero di ricorsione per $T(n) = 3T(n/4) + cn^2$. Per comodità, supponiamo che n sia una potenza esatta di 4 (altro esempio di approssimazione tollerabile). La parte (a) della figura mostra $T(n)$, che viene espanso nella parte (b) in un albero equivalente che rappresenta la ricorrenza. Il termine cn^2 nella radice rappresenta il costo al livello più alto della ricorsione; i tre sottoalberi della radice rappresentano i costi richiesti dai sottoproblemi di dimensione $n/4$. La parte (c) mostra il passaggio successivo di questo processo, dove ogni nodo è espanso con il costo $T(n/4)$ dalla parte (b). Il costo di ogni albero figlio della radice è $c(n/4)^2$. Continuiamo a espandere ogni nodo dell'albero, suddividendolo nelle sue parti costituenti, come stabilisce la ricorrenza.

Poiché le dimensioni dei sottoproblemi diminuiscono via via che ci allontaniamo dalla radice, alla fine dovremo raggiungere una condizione al contorno. A quale distanza dalla radice ne troveremo una? La dimensione del sottoproblema per un nodo alla profondità i è $n/4^i$. Quindi, la dimensione del sottoproblema diventa $n = 1$ quando $n/4^i = 1$ ovvero quando $i = \log_4 n$. Dunque, l'albero ha $\log_4 n + 1$ livelli $(0, 1, 2, \ldots, \log_4 n)$.

Adesso determiniamo il costo a ogni livello dell'albero. Ogni livello ha tre volte i nodi del livello precedente; quindi il numero di nodi alla profondità i è 3^i. Poiché le dimensioni dei sottoproblemi diminuiscono di un fattore 4 ogni volta che si scende di un livello rispetto alla radice, ogni nodo alla profondità i (per $i = 0, 1, 2, \ldots, \log_4 n - 1$) ha un costo di $c(n/4^i)^2$. Moltiplicando, notiamo che il costo totale di tutti i nodi alla profondità i (per $i = 0, 1, 2, \ldots, \log_4 n - 1$) è $3^i c(n/4^i)^2 = (3/16)^i cn^2$. L'ultimo livello, alla profondità $\log_4 n$, ha $3^{\log_4 n} = n^{\log_4 3}$ nodi, ciascuno con un costo $T(1)$, per un costo totale pari a $n^{\log_4 3} T(1)$, che è $\Theta(n^{\log_4 3})$.

A questo punto, sommiamo i costi di tutti i livelli per determinare il costo dell'albero intero:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}\,cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1}\,cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

Quest'ultima formula si presenta alquanto complicata, finché non realizziamo che possiamo di nuovo tollerare una piccola dose di approssimazione e usare come limite superiore una serie geometrica decrescente. Facendo un passo indietro e applicando l'equazione (A.6), otteniamo

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)}\,cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13}\,cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) \end{aligned}$$

Quindi, abbiamo ricavato l'ipotesi $T(n) = O(n^2)$ per la nostra ricorrenza originale $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. In questo esempio, i coefficienti di cn^2 formano una serie geometrica decrescente e, per l'equazione (A.6), la somma di questi coefficienti è limitata superiormente della costante $16/13$. Poiché il contributo della radice al costo totale è cn^2, la radice contribuisce con una frazione costante del costo totale. In altre parole, il costo totale dell'albero è dominato dal costo della radice.

In effetti, se $O(n^2)$ è davvero un limite superiore per la ricorrenza (come verificheremo subito), allora deve essere un limite stretto. Perché? La prima chiamata ricorsiva contribuisce con un costo $\Theta(n^2)$, quindi $\Omega(n^2)$ deve essere un limite inferiore per la ricorrenza.

Adesso possiamo usare il metodo di sostituzione per verificare che la nostra ipotesi era corretta, ovvero $T(n) = O(n^2)$ è un limite superiore per la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Intendiamo dimostrare che $T(n) \le dn^2$ per qualche costante $d > 0$. Utilizzando la stessa costante $c > 0$ di prima, otteniamo

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\,\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}\,dn^2 + cn^2 \\ &\le dn^2 \end{aligned}$$

L'ultimo passaggio è vero finché $d \ge (16/13)c$.

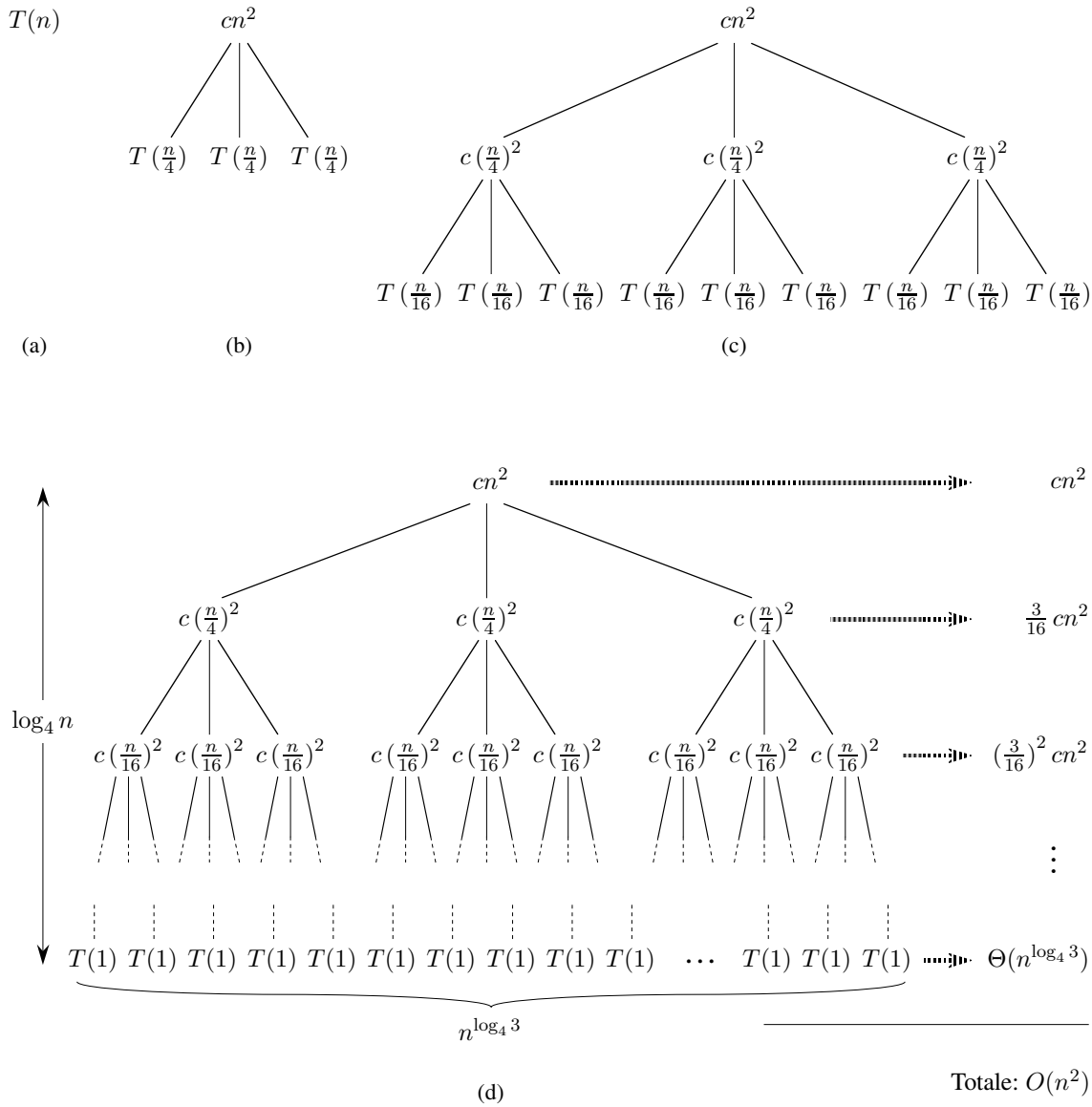


Figura 4.1 La costruzione di un albero di ricorsione per la ricorrenza $T(n) = 3T(n/4) + cn^2$. La parte **(a)** mostra $T(n)$, che viene progressivamente espanso nelle parti **(b)–(d)** per formare l'albero di ricorsione. L'albero completamente espanso nella parte (d) ha un'altezza $\log_4 n$ (con un numero di livelli pari a $\log_4 n + 1$).

Come altro esempio, più complicato, esaminate la Figura 4.2 che illustra l'albero di ricorsione per

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

Anche qui, per semplificare, abbiamo omesso le funzioni floor e ceiling. Come in precedenza, c è il fattore costante nel termine $O(n)$. Quando sommiamo i valori nei singoli livelli dell'albero di ricorsione, otteniamo un valore pari a cn per ogni livello. Il cammino più lungo dalla radice a una foglia è $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Poiché $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, l'altezza dell'albero è $\log_{3/2} n$.

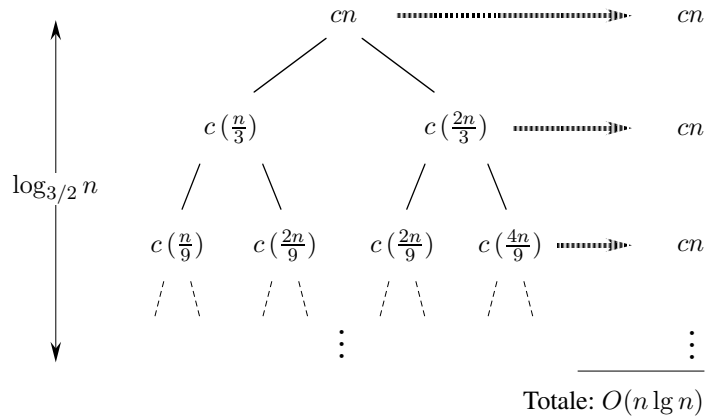


Figura 4.2 Un albero di ricorsione per la ricorrenza $T(n) = T(n/3) + T(2n/3) + cn$.

Intuitivamente, prevediamo che la soluzione della ricorrenza sia al massimo pari al numero di livelli per il costo di ciascun livello, ovvero $O(cn \log_{3/2} n) = O(n \lg n)$. Il costo totale è equamente distribuito fra i livelli dell'albero di ricorsione. Qui c'è una complicazione: dobbiamo ancora considerare il costo delle foglie. Se questo albero di ricorsione fosse un albero binario completo di altezza $\log_{3/2} n$, ci sarebbero $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ foglie. Poiché il costo di ogni foglia è una costante, il costo totale di tutte le foglie sarebbe $\Theta(n^{\log_{3/2} 2})$, che è $\omega(n \lg n)$. Tuttavia, questo albero di ricorsione non è un albero binario completo, pertanto ha meno di $n^{\log_{3/2} 2}$ foglie. Inoltre, via via che si scende dalla radice, mancano sempre più nodi interni. Di conseguenza, non tutti i livelli contribuiscono esattamente con un costo cn; i livelli più bassi contribuiscono in misura minore. Potremmo calcolare con precisione tutti i costi, ma vi ricordiamo che stiamo semplicemente tentando di trovare un'ipotesi da utilizzare nel metodo di sostituzione. Accettando un certo grado di approssimazione, proviamo a dimostrare che l'ipotesi $O(n \lg n)$ per il limite superiore è corretta. In effetti, possiamo applicare il metodo di sostituzione per verificare che $O(n \lg n)$ è un limite superiore per la soluzione della ricorrenza. Dimostriamo che $T(n) \leq dn \lg n$, dove d è un'opportuna costante positiva. Finché $d \geq c/(\lg 3 - (2/3))$, abbiamo che

$$\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\leq dn\lg n
\end{aligned}$$

Quindi, non è necessario svolgere un calcolo più accurato dei costi nell'albero di ricorsione.

Esercizi

4.2-1

Utilizzate un albero di ricorsione per determinare un buon limite superiore asintotico per la ricorrenza $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Applicate il metodo di sostituzione per verificare la vostra soluzione.

4.2-2
Utilizzando un albero di ricorsione, dimostrate che la soluzione della ricorrenza $T(n) = T(n/3) + T(2n/3) + cn$ (c è una costante) è $\Omega(n \lg n)$.

4.2-3
Disegnate l'albero di ricorsione per $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, dove c è una costante, e determinate un limite asintotico stretto sulla sua soluzione. Verificate il limite ottenuto con il metodo di sostituzione.

4.2-4
Utilizzate un albero di ricorsione per trovare una soluzione asintoticamente stretta della ricorrenza $T(n) = T(n-a)+T(a)+cn$, dove $a \geq 1$ e $c > 0$ sono costanti.

4.2-5
Utilizzate un albero di ricorsione per trovare una soluzione asintoticamente stretta della ricorrenza $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, dove α è una costante nell'intervallo $0 < \alpha < 1$ e $c > 0$ è un'altra costante.

4.3 Il metodo dell'esperto

Il metodo dell'esperto rappresenta un "ricettario" per risolvere le ricorrenze della forma

$$T(n) = aT(n/b) + f(n) \tag{4.5}$$

dove $a \geq 1$ e $b > 1$ sono costanti e $f(n)$ è una funzione asintoticamente positiva. Il metodo dell'esperto richiede la memorizzazione di tre casi, ma poi la soluzione di molte ricorrenze può essere facilmente determinata, spesso senza carta e penna.

La ricorrenza (4.5) descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sottoproblemi, ciascuno di dimensione n/b, dove a e b sono costanti positive. I sottoproblemi vengono risolti in modo ricorsivo, ciascuno nel tempo $T(n/b)$. Il costo per dividere il problema e combinare i risultati dei sottoproblemi è descritto dalla funzione $f(n)$ (che è $f(n) = D(n)+C(n)$, applicando la notazione del Paragrafo 2.3.2). Per esempio, la ricorrenza che risulta dalla procedura MERGE-SORT ha $a = 2$, $b = 2$ e $f(n) = \Theta(n)$.

Tecnicamente parlando, la ricorrenza non è effettivamente ben definita perché n/b potrebbe non essere un intero. Tuttavia, la sostituzione di ciascuno degli a termini $T(n/b)$ con $T(\lfloor n/b \rfloor)$ o $T(\lceil n/b \rceil)$ non influisce sul comportamento asintotico della ricorrenza (questo sarà dimostrato nel prossimo paragrafo). Pertanto, di solito, omettiamo per comodità le funzioni floor e ceiling quando scriviamo ricorrenze di divide et impera di questa forma.

Il teorema dell'esperto

Il metodo dell'esperto dipende dal seguente teorema.

Teorema 4.1 (Teorema dell'esperto)
Date le costanti $a \geq 1$ e $b > 1$ e la funzione $f(n)$; sia $T(n)$ una funzione definita sugli interi non negativi dalla ricorrenza

$$T(n) = aT(n/b) + f(n)$$

dove n/b rappresenta $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Allora $T(n)$ può essere asintoticamente limitata nei seguenti modi:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$ e se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$. ■

Prima di applicare il teorema dell'esperto a qualche esempio, cerchiamo di capire che cosa dice. In ciascuno dei tre casi, confrontiamo la funzione $f(n)$ con la funzione $n^{\log_b a}$. Intuitivamente, la soluzione della ricorrenza è determinata dalla più grande delle due funzioni. Se, come nel caso 1, la funzione $n^{\log_b a}$ è la più grande, allora la soluzione è $T(n) = \Theta(n^{\log_b a})$. Se, come nel caso 3, la funzione $f(n)$ è la più grande, allora la soluzione è $T(n) = \Theta(f(n))$. Se, come nel caso 2, le due funzioni hanno la stessa dimensione, moltiplichiamo per un fattore logaritmico e la soluzione è $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Oltre a questo concetto intuitivo, ci sono alcuni dettagli tecnici da capire. Nel primo caso, $f(n)$ non soltanto deve essere più piccola di $n^{\log_b a}$, ma deve essere *polinomialmente* più piccola; ovvero, $f(n)$ deve essere asintoticamente più piccola di $n^{\log_b a}$ per un fattore n^{ϵ} per qualche costante $\epsilon > 0$. Nel terzo caso, $f(n)$ non soltanto deve essere più grande di $n^{\log_b a}$, ma deve essere polinomialmente più grande e soddisfare anche la condizione di "regolarità" $af(n/b) \leq cf(n)$. Questa condizione è soddisfatta dalla maggior parte delle funzioni polinomialmente limitate che incontreremo.

È importante capire che i tre casi non coprono tutte le versioni possibili di $f(n)$. C'è un intervallo fra i casi 1 e 2 in cui $f(n)$ è minore di $n^{\log_b a}$, ma non in modo polinomiale. Analogamente, c'è un intervallo fra i casi 2 e 3 in cui $f(n)$ è maggiore di $n^{\log_b a}$, ma non in modo polinomiale. Se la funzione $f(n)$ ricade in uno di questi intervalli o se la condizione di regolarità nel caso 3 non è soddisfatta, il metodo dell'esperto non può essere usato per risolvere la ricorrenza.

Applicazione del metodo dell'esperto

Per utilizzare il metodo dell'esperto, determiniamo semplicemente quale caso (se esiste) del teorema dell'esperto possiamo applicare e scriviamo la soluzione. Come primo esempio, consideriamo

$$T(n) = 9T(n/3) + n$$

Per questa ricorrenza abbiamo $a = 9$, $b = 3$, $f(n) = n$, quindi $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Poiché $f(n) = O(n^{\log_3 9 - \epsilon})$, dove $\epsilon = 1$, possiamo applicare il caso 1 del teorema dell'esperto e concludere che la soluzione è $T(n) = \Theta(n^2)$. Adesso consideriamo

$$T(n) = T(2n/3) + 1$$

dove $a = 1$, $b = 3/2$, $f(n) = 1$ e $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Si applica il caso 2, in quanto $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ e, quindi, la soluzione della ricorrenza è $T(n) = \Theta(\lg n)$. Nella ricorrenza

$$T(n) = 3T(n/4) + n \lg n$$

abbiamo $a = 3$, $b = 4$, $f(n) = n \lg n$ e $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Poiché $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, dove $\epsilon \approx 0.2$, si applica il caso 3, se dimostriamo

che la condizione di regolarità è soddisfatta per $f(n)$. Per n sufficientemente grande, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n\lg n = cf(n)$ per $c = 3/4$. Di conseguenza, per il caso 3, la soluzione della ricorrenza è $T(n) = \Theta(n\lg n)$.

Il metodo dell'esperto non si applica alla ricorrenza

$$T(n) = 2T(n/2) + n\lg n$$

anche se ha la forma appropriata: $a = 2$, $b = 2$, $f(n) = n\lg n$ e $n^{\log_b a} = n$. Sembra che si possa applicare il caso 3, in quanto $f(n) = n\lg n$ è asintoticamente più grande di $n^{\log_b a} = n$; il problema è che non è *polinomialmente* più grande. Il rapporto $f(n)/n^{\log_b a} = (n\lg n)/n = \lg n$ è asintoticamente minore di n^ϵ per qualsiasi costante positiva ϵ. Quindi, la ricorrenza ricade nell'intervallo fra i casi 2 e 3 (vedere l'Esercizio 4.4-2 per una soluzione).

Esercizi

4.3-1

Applicare il metodo dell'esperto per determinare i limiti asintotici stretti per le seguenti ricorrenze:

a. $T(n) = 4T(n/2) + n$

b. $T(n) = 4T(n/2) + n^2$

c. $T(n) = 4T(n/2) + n^3$

4.3-2

La ricorrenza $T(n) = 7T(n/2) + n^2$ descrive il tempo di esecuzione di un algoritmo A. Un algoritmo concorrente A' ha un tempo di esecuzione $T'(n) = aT'(n/4)+n^2$. Qual è il più grande valore intero di a che rende A' asintoticamente più veloce di A?

4.3-3

Applicate il metodo dell'esperto per dimostrare che la soluzione della ricorrenza $T(n) = T(n/2) + \Theta(1)$ della ricerca binaria è $T(n) = \Theta(\lg n)$ (la ricerca binaria è descritta nell'Esercizio 2.3-5).

4.3-4

Il metodo dell'esperto può essere applicato alla ricorrenza $T(n) = 4T(n/2) + n^2\lg n$? Perché o perché no? Determinate un limite asintotico superiore per questa ricorrenza.

4.3-5 $\star$

Considerate la condizione di regolarità $af(n/b) \leq cf(n)$ per qualche costante $c < 1$, che è parte del caso 3 del teorema dell'esperto. Indicate due costanti $a \geq 1$ e $b > 1$ e una funzione $f(n)$ che soddisfa tutte le condizioni del caso 3 del teorema dell'esperto, tranne quella di regolarità.

$\star$ 4.4 Dimostrazione del teorema dell'esperto

Questo paragrafo contiene una dimostrazione del teorema dell'esperto (Teorema 4.1). Non occorre comprendere la dimostrazione per applicare il teorema.

La dimostrazione si divide in due parti. La prima parte analizza la ricorrenza "principale" (4.5), sotto l'ipotesi esemplificativa che $T(n)$ sia definita soltanto con potenze esatte di $b > 1$, ovvero per $n = 1, b, b^2, \ldots$. Questa parte presenta tutti i concetti intuitivi necessari per capire perché il teorema dell'esperto è vero. La seconda parte mostra come l'analisi possa essere estesa a tutti gli interi positivi n ed è una semplice tecnica matematica applicata al problema della gestione di floor e ceiling.

In questo paragrafo, a volte, abuseremo un po' della nostra notazione asintotica utilizzandola per descrivere il comportamento di funzioni che sono definite soltanto con potenze esatte di b. Ricordiamo che le definizioni delle notazioni asintotiche richiedono che i limiti siano dimostrati per tutti i numeri sufficientemente grandi, non solo per quelli che sono potenze di b. Dal momento che potremmo definire nuove notazioni asintotiche che si applicano all'insieme $\{b^i : i = 0, 1, \ldots\}$, anziché agli interi non negativi, questo è un abuso di second'ordine.

Nonostante ciò, dobbiamo sempre stare in guardia quando utilizziamo la notazione asintotica su un dominio limitato per non incorrere in conclusioni improprie. Per esempio, dimostrare che $T(n) = O(n)$ quando n è una potenza esatta di 2 non garantisce che $T(n) = O(n)$. La funzione $T(n)$ potrebbe essere definita in questo modo

$$T(n) = \begin{cases} n & \text{se } n = 1, 2, 4, 8, \ldots \\ n^2 & \text{negli altri casi} \end{cases}$$

Nel qual caso il limite superiore migliore che possiamo dimostrare è $T(n) = O(n^2)$. A causa di questo genere di conseguenze drastiche, non useremo mai la notazione asintotica su un dominio limitato, a meno che non sia assolutamente chiaro dal contesto che lo stiamo facendo.

4.4.1 La dimostrazione per le potenze esatte

La prima parte della dimostrazione del teorema dell'esperto analizza la ricorrenza (4.5)

$$T(n) = aT(n/b) + f(n)$$

per il metodo dell'esperto, nell'ipotesi che n sia una potenza esatta di $b > 1$, dove b non deve essere necessariamente un intero. L'analisi è suddivisa in tre lemmi. Il primo lemma riduce il problema di risolvere la ricorrenza principale al problema di valutare un'espressione che contiene una sommatoria. Il secondo lemma determina i limiti su questa sommatoria. Il terzo lemma riunisce i primi due per dimostrare una versione del teorema dell'esperto nel caso in cui n sia una potenza esatta di b.

Lemma 4.2
Siano $a \geq 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita sulle potenze esatte di b. Se $T(n)$ è definita sulle potenze esatte di b dalla ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ aT(n/b) + f(n) & \text{se } n = b^i \end{cases}$$

dove i è un intero positivo, allora

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{4.6}$$

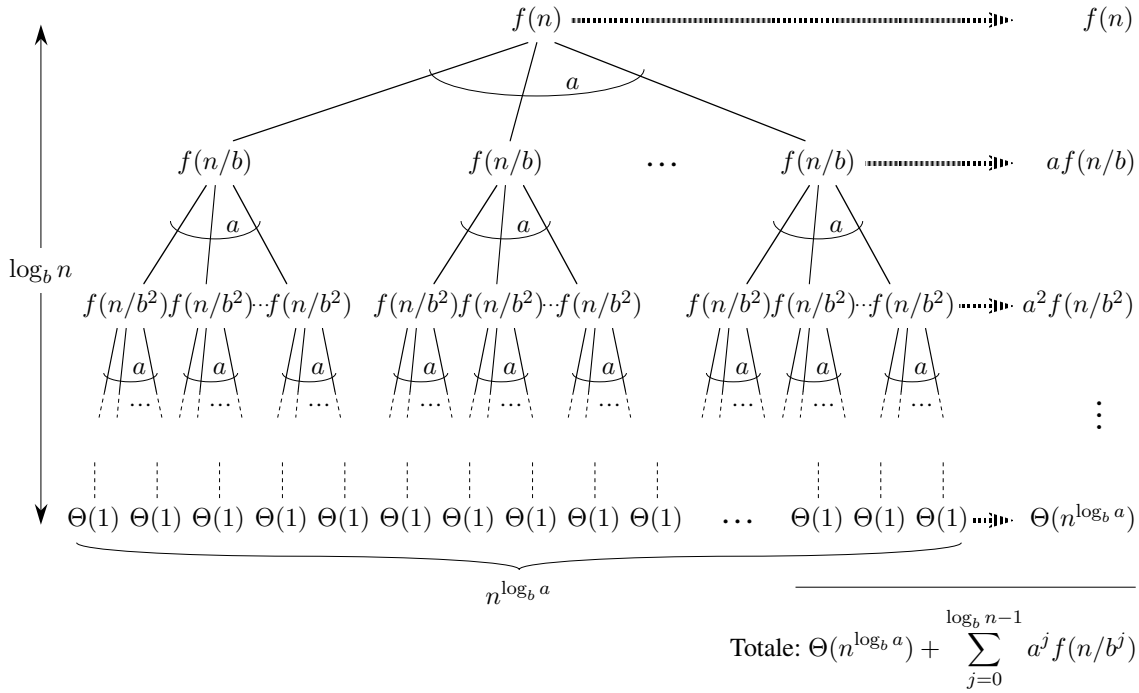


Figura 4.3 L'albero di ricorsione generato da $T(n) = aT(n/b) + f(n)$. È un albero a-ario completo con $n^{\log_b a}$ foglie e altezza $\log_b n$. Il costo di ogni livello è indicato a destra e la loro somma è data dall'equazione (4.6).

Dimostrazione Utilizziamo l'albero di ricorsione nella Figura 4.3. La radice dell'albero ha costo $f(n)$ e ha a figli, ciascuno di costo $f(n/b)$. (È comodo pensare ad a come a un numero intero, specialmente quando rappresentiamo l'albero di ricorsione, anche se matematicamente ciò non sia richiesto.) Ciascuno di questi figli ha, a sua volta, a figli con un costo di $f(n/b^2)$; quindi ci sono a^2 nodi alla distanza 2 dalla radice. In generale, ci sono a^j nodi alla distanza j dalla radice, ciascuno dei quali ha un costo di $f(n/b^j)$. Il costo di ogni foglia è $T(1) = \Theta(1)$ e ogni foglia si trova alla profondità $\log_b n$, in quanto $n/b^{\log_b n} = 1$. L'albero ha $a^{\log_b n} = n^{\log_b a}$ foglie.

Possiamo ottenere l'equazione (4.6) sommando i costi di ogni livello dell'albero, come illustra la figura. Il costo per un livello j di nodi interni è $a^j f(n/b^j)$; quindi il totale per tutti i livelli dei nodi interni è

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

Nell'algoritmo divide et impera sottostante, questa somma rappresenta i costi per dividere un problema in sottoproblemi e poi per ricombinare i sottoproblemi. Il costo di tutte le foglie, che è il costo per svolgere $n^{\log_b a}$ sottoproblemi di dimensione 1, è $\Theta(n^{\log_b a})$. ■

Nei termini dell'albero di ricorsione, i tre casi del teorema dell'esperto corrispondono ai casi in cui il costo totale dell'albero è (1) dominato dai costi delle foglie, (2) equamente distribuito fra i livelli dell'albero o (3) dominato dal costo della radice.

La sommatoria nell'equazione (4.6) descrive il costo per dividere e combinare i passaggi dell'algoritmo divide et impera sottostante. Il prossimo lemma fornisce i limiti asintotici sulla crescita della sommatoria.

Lemma 4.3
Siano $a \geq 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita sulle potenze esatte di b. Una funzione $g(n)$ definita sulle potenze esatte di b da

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.7}$$

può essere limitata asintoticamente per le potenze esatte di b nei seguenti modi.

1. Se $f(n) = O(n^{\log_b a-\epsilon})$ per qualche costante $\epsilon > 0$, allora $g(n) = O(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, allora $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni $n \geq b$, allora $g(n) = \Theta(f(n))$.

Dimostrazione Per il caso 1, abbiamo $f(n) = O(n^{\log_b a-\epsilon})$, che implica che $f(n/b^j) = O((n/b^j)^{\log_b a-\epsilon})$. Sostituendo nell'equazione (4.7), otteniamo

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \tag{4.8}$$

Limitiamo la sommatoria all'interno della notazione O mettendo in evidenza i fattori comuni e semplificando; alla fine otteniamo una serie geometrica crescente:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right)
\end{aligned}$$

Poiché b e ϵ sono costanti, possiamo riscrivere l'ultima espressione così:

$n^{\log_b a-\epsilon} O(n^\epsilon) = O(n^{\log_b a})$

Sostituendo questa espressione nella sommatoria dell'equazione (4.8), otteniamo

$g(n) = O(n^{\log_b a})$

Il caso 1 è dimostrato.

Per il caso 2, l'ipotesi $f(n) = \Theta(n^{\log_b a})$ implica $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Sostituendo nell'equazione (4.7), otteniamo

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) \tag{4.9}$$

Limitiamo la sommatoria all'interno della notazione Θ come nel caso 1; stavolta, però, non otteniamo una serie geometrica. Piuttosto, scopriamo che ogni termine della sommatoria è lo stesso:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a} \log_b n\end{aligned}$$

Sostituendo questa espressione nella sommatoria dell'equazione (4.9), otteniamo

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a} \log_b n) \\ &= \Theta(n^{\log_b a} \lg n)\end{aligned}$$

Il caso 2 è dimostrato.

Il caso 3 si dimostra in modo simile. Poiché $f(n)$ appare nella definizione (4.7) di $g(n)$ e tutti i termini di $g(n)$ sono non negativi, possiamo concludere che $g(n) = \Omega(f(n))$ per potenze esatte di b. Avendo ipotizzato che $af(n/b) \le cf(n)$ per qualche costante $c < 1$ e per ogni $n \ge b$, abbiamo $f(n/b) \le (c/a)f(n)$. Iterando j volte, otteniamo $f(n/b^j) \le (c/a)^j f(n)$ ovvero $a^j f(n/b^j) \le c^j f(n)$. Sostituendo nell'equazione (4.7) e semplificando, si ottiene una serie geometrica che, diversamente da quella del caso 1, è decrescente in quanto c è costante:

$$\begin{aligned}g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\log_b n-1} c^j f(n) \\ &\le f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n) \left(\frac{1}{1-c}\right) \\ &= O(f(n))\end{aligned}$$

Pertanto possiamo concludere che $g(n) = \Theta(f(n))$ per potenze esatte di b. Il caso 3 è dimostrato e questo completa la dimostrazione del lemma. ■

Adesso possiamo dimostrare una versione del teorema dell'esperto per il caso in cui n è una potenza esatta di b.

Lemma 4.4

Siano $a \ge 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita sulle potenze esatte di b. Se $T(n)$ è definita sulle potenze esatte di b dalla ricorrenza

$$T(n) = \begin{cases}\Theta(1) & \text{se } n = 1 \\ aT(n/b) + f(n) & \text{se } n = b^i\end{cases}$$

dove i è un intero positivo, allora $T(n)$ può essere limitata asintoticamente per le potenze esatte b nei seguenti modi.

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$ e se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

Dimostrazione Usiamo il limiti del Lemma 4.3 per valutare la sommatoria (4.6) dal Lemma 4.2. Per il caso 1 abbiamo

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \end{aligned}$$

Per il caso 2

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) \end{aligned}$$

Per il caso 3

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) \end{aligned}$$

in quanto $f(n) = \Omega(n^{\log_b a + \epsilon})$. ■

4.4.2 Floor e ceiling

Per completare la dimostrazione del teorema dell'esperto, a questo punto dobbiamo estendere la nostra analisi al caso in cui siano utilizzate le funzioni floor e ceiling nella ricorrenza principale, in modo che la ricorrenza sia definita per tutti i numeri interi, non soltanto per le potenze esatte di b. È semplice ottenere un limite inferiore per

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

e un limite superiore per

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11}$$

in quanto possiamo accettare il limite $\lceil n/b \rceil \geq n/b$ nel primo caso per ottenere il risultato desiderato e il limite $\lfloor n/b \rfloor \leq n/b$ nel secondo caso. Per limitare inferiormente la ricorrenza (4.11) occorre pressoché la stessa tecnica per limitare superiormente la ricorrenza (4.10), quindi presenteremo soltanto quest'ultimo limite.

Modifichiamo l'albero di ricorsione della Figura 4.3 per generare l'albero di ricorsione illustrato nella Figura 4.4. Procendendo verso il basso nell'albero di ricorsione, otteniamo una sequenza di chiamate ricorsive degli argomenti

$$\begin{aligned} &n \\ &\lceil n/b \rceil \\ &\lceil \lceil n/b \rceil /b \rceil \\ &\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil \\ &\qquad \vdots \end{aligned}$$

Indichiamo con n_j il j-esimo elemento della sequenza, dove

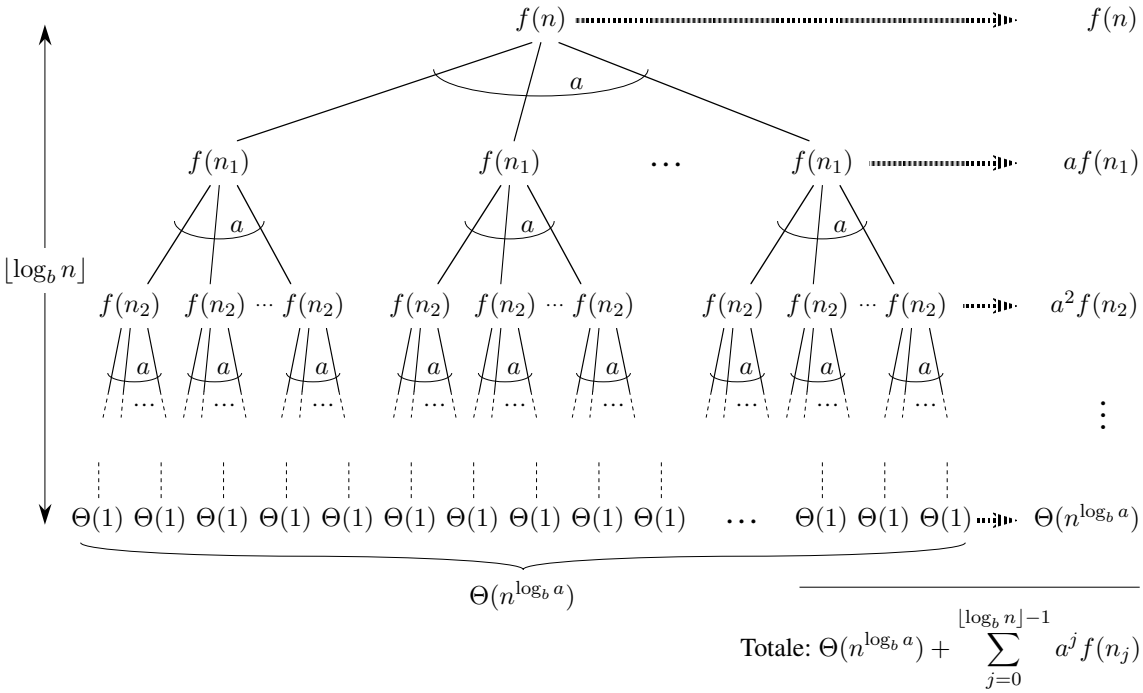


Figura 4.4 L'albero di ricorsione generato da $T(n) = aT(\lceil n/b \rceil) + f(n)$. L'argomento ricorsivo n_j è dato dall'equazione (4.12).

$$n_j = \begin{cases} n & \text{se } j = 0 \\ \lceil n_{j-1}/b \rceil & \text{se } j > 0 \end{cases} \tag{4.12}$$

Il nostro primo obiettivo è determinare la profondità k in modo tale che n_k sia una costante. Applicando la disuguaglianza $\lceil x \rceil \leq x + 1$, otteniamo

$$\begin{aligned} n_0 &\leq n \\ n_1 &\leq \frac{n}{b} + 1 \\ n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1 \\ n_3 &\leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1 \\ &\vdots \end{aligned}$$

In generale,

$$\begin{aligned} n_j &\leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1} \end{aligned}$$

Ponendo $j = \lfloor \log_b n \rfloor$, otteniamo

$$n_{\lfloor \log_b n \rfloor} < \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1}$$

$$
\begin{aligned}
&< \frac{n}{b^{\log_b n-1}} + \frac{b}{b-1} \\
&= \frac{n}{n/b} + \frac{b}{b-1} \\
&= b + \frac{b}{b-1} \\
&= O(1)
\end{aligned}
$$

Notiamo che, alla profondità $\lfloor \log_b n \rfloor$, la dimensione del problema è al massimo una costante. Dalla Figura 4.4 otteniamo la relazione

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.13}$$

che è pressoché uguale all'equazione (4.6), con la differenza che n è un numero intero arbitrario e non è vincolato a essere una potenza esatta di b.

Adesso possiamo calcolare la sommatoria

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.14}$$

dall'equazione (4.13) in modo analogo alla dimostrazione del Lemma 4.3. Iniziando dal caso 3, se $af(\lceil n/b \rceil) \leq cf(n)$ per $n > b + b/(b-1)$, dove $c < 1$ è una costante, ne consegue che $a^j f(n_j) \leq c^j f(n)$. Pertanto, la sommatoria nell'equazione (4.14) può essere calcolata proprio come nel Lemma 4.3. Nel caso 2 abbiamo $f(n) = \Theta(n^{\log_b a})$. Se dimostriamo che $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, allora seguiremo la dimostrazione del caso 2 del Lemma 4.3. Notate che $j \leq \lfloor \log_b n \rfloor$ implica $b^j/n \leq 1$. Il limite $f(n) = O(n^{\log_b a})$ implica che esiste una costante $c > 0$ tale che, per ogni n_j sufficientemente grande:

$$
\begin{aligned}
f(n_j) &\leq c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\
&= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\
&\leq c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right)
\end{aligned}
$$

in quanto $c(1 + b/(b-1))^{\log_b a}$ è una costante. Quindi, il caso 2 è dimostrato. La dimostrazione del caso 1 è quasi identica. La chiave sta nel dimostrare il limite $f(n_j) = O(n^{\log_b a - \epsilon})$, che è simile alla corrispondente dimostrazione del caso 2, sebbene i calcoli algebrici siano più complicati.

Abbiamo così dimostrato i limiti superiori nel teorema dell'esperto per tutti i numeri interi n. La dimostrazione dei limiti inferiori è simile.

Esercizi

4.4-1 $\star$

Create un'espressione semplice ed esatta per n_j nell'equazione (4.12) per il caso in cui b sia un numero intero positivo, anziché un numero reale arbitrario.

4.4-2 $\star$

Dimostrate che, se $f(n) = \Theta(n^{\log_b a} \lg^k n)$, dove $k \geq 0$, la soluzione della ricorrenza principale è $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Per semplicità, limitate la vostra analisi alle potenze esatte di b.

4.4-3 $\star$

Dimostrate che il caso 3 del teorema dell'esperto è sovradefinito, nel senso che la condizione di regolarità $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ implica che esiste una costante $\epsilon > 0$ tale che $f(n) = \Omega(n^{\log_b a + \epsilon})$.

4.5 Problemi

4-1 Esempi di ricorrenze

Indicate i limiti superiore e inferiore per $T(n)$ in ciascuna delle seguenti ricorrenze. Supponete che $T(n)$ sia costante per $n \leq 2$. I limiti devono essere i più stretti possibili; spiegate le vostre soluzioni.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

4-2 Trovare il numero intero mancante

Un array $A[1 \,..\, n]$ contiene tutti i numeri interi da 0 a n, tranne uno. Sarebbe facile trovare l'intero mancante nel tempo $O(n)$ utilizzando un array ausiliario $B[0 \,..\, n]$ per memorizzare quali numeri si trovano in A. In questo problema, invece, non è possibile accedere a un intero completo di A con una singola operazione. Gli elementi di A sono rappresentati nel formato binario, quindi l'unica operazione che possiamo usare per accedere a tali elementi consiste nel "leggere il bit j-esimo di $A[i]$", che richiede un tempo costante.

Dimostrate che, utilizzando questa sola operazione, è ancora possibile determinare l'intero mancante nel tempo $O(n)$.

4-3 Costi per passare i parametri

In tutto il libro supporremo che il passaggio dei parametri durante le chiamate delle procedure richieda un tempo costante, anche quando viene passato un array

di N elementi. Questa ipotesi è valida nella maggior parte dei sistemi, in quanto viene passato il puntatore all'array, non l'array. Questo problema esamina le implicazioni di tre strategie per passare i parametri:

1. Un array viene passato tramite un puntatore. Tempo $= \Theta(1)$.
2. Un array viene passato facendone una copia. Tempo $= \Theta(N)$, dove N è la dimensione dell'array.
3. Un array viene passato copiando soltanto la parte che potrebbe essere utilizzata dalla procedura chiamata. Tempo $= \Theta(q-p+1)$ se viene passato il sottoarray $A[p\,..\,q]$.

a. Considerate l'algoritmo di ricerca binaria per trovare un numero in un array ordinato (vedere l'Esercizio 2.3-5). Determinate le ricorrenze per i tempi di esecuzione nel caso peggiore della ricerca binaria quando gli array vengono passati utilizzando ciascuno dei tre metodi precedenti; specificate dei buoni limiti superiori sulle soluzioni delle ricorrenze. Indicate con N la dimensione del problema originale e con n la dimensione di un sottoproblema.

b. Ripetete il punto (a) per l'algoritmo MERGE-SORT del Paragrafo 2.3.1.

4-4 Altri esempi di ricorrenze

Indicate i limiti asintotici superiore e inferiore per $T(n)$ in ciascuna delle seguenti ricorrenze. Supponete che $T(n)$ sia costante per n sufficientemente piccolo. I limiti devono essere i più stretti possibili; spiegate le vostre soluzioni.

a. $T(n) = 3T(n/2) + n \lg n$.

b. $T(n) = 5T(n/5) + n/\lg n$.

c. $T(n) = 4T(n/2) + n^2\sqrt{n}$.

d. $T(n) = 3T(n/3+5) + n/2$.

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 2\lg n$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5 Numeri di Fibonacci

Questo problema sviluppa le proprietà dei numeri di Fibonacci, che sono definiti dalla ricorrenza (3.21). Utilizzeremo la tecnica delle funzioni generatrici per risolvere la ricorrenza di Fibonacci. La ***funzione generatrice*** (o ***serie di potenze formali***) $\mathcal{F}$ è definita in questo modo

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots\end{aligned}$$

Dove F_i è l'i-esimo numero di Fibonacci.

a. Dimostrate che $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Dimostrate che

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \widehat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z}\right)\end{aligned}$$

dove

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\ldots$$

e

$$\widehat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803\ldots$$

c. Dimostrate che

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \widehat{\phi}^i) z^i$$

d. Dimostrate che $F_i = \phi^i/\sqrt{5}$ per $i > 0$, arrotondato all'intero più vicino (*suggerimento:* notate che $|\widehat{\phi}| < 1$).

e. Dimostrate che $F_{i+2} \geq \phi^i$ per $i \geq 0$.

4-6 Collaudo di chip VLSI

Il professor Diogene ha n chip VLSI[1], apparentemente identici, che in linea di principio sono capaci di collaudarsi a vicenda. L'apparecchiatura di prova del professore può accogliere due chip alla volta. Dopo che l'apparecchiatura è stata avviata, ogni chip prova l'altro e indica se è buono o guasto. Un chip buono indica sempre con esattezza se l'altro chip è buono o guasto, ma l'indicazione di un chip guasto non può essere accettata. Le quattro possibili indicazioni di un test sono le seguenti:

Chip A dice	Chip B dice	Conclusione
B è buono	A è buono	entrambi sono buoni o entrambi sono guasti
B è buono	A è guasto	almeno uno è guasto
B è guasto	A è buono	almeno uno è guasto
B è guasto	A è guasto	almeno uno è guasto

a. Dimostrate che se più di $n/2$ chip sono guasti, il professore non può determinare quali chip sono buoni applicando qualsiasi strategia basata su questo tipo di collaudo a coppie. Supponete che i chip guasti possano cospirare per ingannare il professore.

[1] VLSI sta per "very large scale integration" (integrazione su larghissima scala), che è la tecnologia dei circuiti integrati utilizzata per fabbricare la maggior parte dei moderni microprocessori.

b. Considerate il problema di trovare un unico chip buono fra n chip, supponendo che più di $n/2$ chip siano buoni. Dimostrate che $\lfloor n/2 \rfloor$ collaudi a coppie sono sufficienti per ridurre il problema a uno di dimensione quasi dimezzata.

c. Dimostrate che i chip buoni possono essere identificati con $\Theta(n)$ collaudi a coppie, supponendo che più di $n/2$ chip siano buoni. Specificate e risolvete la ricorrenza che descrive il numero di collaudi.

4-7 Array di Monge

Un array A di $m \times n$ numeri reali è un ***array di Monge*** se, per ogni i, j, k e l tali che $1 \leq i < k \leq m$ e $1 \leq j < l \leq n$, si ha

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j]$$

In altre parole, ogni volta che scegliamo due righe e due colonne di un array di Monge e consideriamo i quattro elementi nelle intersezioni fra righe e colonne, la somma degli elementi superiore sinistro e inferiore destro è minore o uguale alla somma degli elementi inferiore sinistro e superiore destro. Ecco un esempio di array di Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Dimostrate che un array di Monge è tale, se e soltanto se, per ogni $i = 1, 2, ..., m-1$ e $j = 1, 2, ..., n-1$, si ha

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$$

(*Suggerimento:* per la parte "se", usate l'induzione separatamente sulle righe e le colonne.)

b. Il seguente array non è un array di Monge; cambiate un elemento per trasformarlo in array di Monge (*suggerimento:* applicate il punto (a)).

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Sia $f(i)$ l'indice della colonna che contiene il più piccolo elemento a sinistra nella riga i. Dimostrate che $f(1) \leq f(2) \leq \cdots \leq f(m)$ per qualsiasi array di Monge $m \times n$.

d. Ecco la descrizione di un algoritmo divide et impera che calcola il più piccolo elemento a sinistra in ogni riga di un array di Monge A di $m \times n$ elementi:

> Costruite una sottomatrice A' di A formata dalle righe con numeri pari di A. Determinate ricorsivamente il più piccolo elemento a sinistra per ogni riga di A'. Poi calcolate il più piccolo elemento a sinistra nelle righe con numeri dispari di A.

Spiegate come calcolare il più piccolo elemento a sinistra nelle righe con numeri dispari di A (conoscendo il più piccolo elemento a sinistra nelle righe con numeri pari) nel tempo $O(m+n)$.

e. Scrivete la ricorrenza che descrive il tempo di esecuzione dell'algoritmo presentato nel punto (d). Dimostrate che la sua soluzione è $O(m + n \log m)$.

Note

Le ricorrenze furono studiate già nel 1202 da L. Fibonacci, dal quale hanno preso il nome i numeri di Fibonacci. A. De Moivre introdusse il metodo delle funzioni generatrici (vedere il Problema 4-5) per risolvere le ricorrenze. Il metodo dell'esperto è stato adattato da Bentley, Haken e Saxe [41], che hanno descritto il metodo esteso presentato nell'Esercizio 4.4-2. Knuth [182] e Liu [205] spiegano come risolvere le ricorrenze lineari applicando il metodo delle funzioni generatrici. Purdom e Brown [252] e Graham, Knuth e Patashnik [132] trattano in modo esteso i metodi di risoluzione delle ricorrenze. Molti ricercatori, inclusi Akra e Bazzi [13], Roura [262] e Verma [306], hanno creato dei metodi per risolvere più ricorrenze divide et impera di quelle risolte dal metodo dell'esperto. Descriviamo qui il risultato di Akra e Bazzi, che funziona per ricorrenze della forma

$$T(n) = \sum_{i=1}^{k} a_i T(\lfloor n/b_i \rfloor) + f(n) \tag{4.15}$$

dove $k \geq 1$; tutti i coefficienti a_i sono positivi e la loro somma è almeno 1; ogni b_i è maggiore di 1; $f(n)$ è una funzione limitata, positiva e non decrescente; per tutte le costanti $c > 1$, esistono delle costanti n_0, $d > 0$ tali che $f(n/c) \geq df(n)$ per ogni $n \geq n_0$. Questo metodo funziona su una ricorrenza tale che $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, alla quale non si applica il metodo dell'esperto. Per risolvere la ricorrenza (4.15), troviamo prima il valore p tale che $\sum_{i=1}^{k} a_i b_i^{-p} = 1$ (il valore p esiste sempre, è unico e positivo). La soluzione della ricorrenza è quindi

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^{n} \frac{f(x)}{x^{p+1}}\, dx\right)$$

per una costante n' sufficientemente grande. Il metodo Akra-Bazzi potrebbe risultare difficile da usare, ma serve a risolvere le ricorrenze che modellano la divisione di un problema in sottoproblemi di dimensioni sostanzialmente differenti. Il metodo dell'esperto è più semplice da usare, ma si applica soltanto quando i sottoproblemi hanno la stessa dimensione.

5 Analisi probabilistica e algoritmi randomizzati

Questo capitolo introduce l'analisi probabilistica e gli algoritmi randomizzati. Se non avete dimestichezza con le basi della teoria delle probabilità, dovreste leggere l'Appendice C che tratta questi argomenti. L'analisi probabilistica e gli algoritmi randomizzati saranno utilizzati più volte in questo libro.

5.1 Il problema delle assunzioni

Supponete di dovere assumere un nuovo impiegato. Poiché i vostri precedenti tentativi di assumere un impiegato non hanno avuto successo, decidete di rivolgervi a un'agenzia di selezione del personale. L'agenzia vi invia un candidato al giorno. Venite a colloquio con questa persona e poi decidete se assumerla oppure no. Dovrete pagare all'agenzia un piccolo compenso per avere un colloquio con un candidato. L'assunzione effettiva di un candidato, invece, è un'operazione più costosa, perché dovrete licenziare l'attuale impiegato e pagare un grosso compenso all'agenzia. Intendete assumere la persona migliore possibile per ogni compito da svolgere. Di conseguenza, avete deciso che, dopo avere avuto un colloquio con un candidato, se questo è migliore dell'attuale impiegato, licenzierete l'attuale impiegato e assumerete il candidato. Siete intenzionati a pagare il prezzo derivante da questa strategia, ma volete stimare quale sarà questo prezzo.

La seguente procedura HIRE-ASSISTANT esprime questa strategia nella forma di pseudocodice. I candidati sono numerati da 1 a n. La procedura suppone che, dopo avere avuto un colloquio con il candidato i, voi siate in grado di determinare se questo candidato è il migliore fra quelli precedentemente incontrati. All'inizio, la procedura crea un candidato fittizio (con numero 0), che è il meno qualificato di tutti gli altri candidati.

```
HIRE-ASSISTANT(n)
1  best ← 0          ▷ il candidato 0 è il meno qualificato
2  for i ← 1 to n
3      do colloquio con il candidato i
4         if il candidato i è migliore del candidato best
5           then best ← i
6                assumi il candidato i
```

Il modello dei costi per questo problema è diverso dal modello descritto nel Capitolo 2. Non siamo interessati al tempo di esecuzione di HIRE-ASSISTANT, ma bensì ai costi richiesti per il colloquio e l'assunzione. In apparenza, l'analisi dei costi di questo algoritmo potrebbe sembrare molto diversa dall'analisi del

tempo di esecuzione, per esempio, di merge sort. Le tecniche analitiche adottate sono identiche sia quando valutiamo i costi sia quando valutiamo il tempo di esecuzione. In entrambi i casi, contiamo il numero di volte che vengono eseguite determinate operazioni di base.

Il colloquio ha un costo modesto (c_i), mentre l'assunzione ha un costo più alto (c_h). Se m è il numero totale di persone assunte, il costo totale associato a questo algoritmo è $O(nc_i + mc_h)$. Indipendentemente dal numero di persone assunte, dovremo sempre avere un colloquio con n candidati e, quindi, avremo sempre il costo nc_i associato ai colloqui. Quindi, concentriamo la nostra analisi sul costo di assunzione mc_h. Questa quantità varia ogni volta che viene eseguito l'algoritmo.

Questo scenario serve come modello per un tipico paradigma computazionale. Capita spesso di dover trovare il valore massimo o minimo in una sequenza, esaminando i singoli elementi della sequenza e conservando il "vincitore" corrente. Il problema delle assunzioni rappresenta il modello di quanto spesso aggiorniamo la nostra conoscenza sull'elemento che sta correntemente vincendo.

Analisi del caso peggiore

Nel caso peggiore, noi assumiamo ogni candidato con il quale abbiamo un colloquio. Questa situazione si verifica se i candidati si presentano in ordine crescente di qualità, nel qual caso effettuiamo n assunzioni, con un costo totale per le assunzioni pari a $O(nc_h)$.

Può essere ragionevole prevedere, tuttavia, che i candidati non sempre arrivino in ordine crescente di qualità. In effetti, non abbiamo alcuna idea sull'ordine in cui essi si presenteranno né abbiamo alcun controllo su tale ordine. Di conseguenza, è naturale chiedersi che cosa prevediamo che accada in un caso tipico o medio.

Analisi probabilistica

L'***analisi probabilistica*** è l'uso della probabilità nell'analisi dei problemi. Tipicamente, usiamo l'analisi probabilistica per analizzare il tempo di esecuzione di un algoritmo. A volte, la utilizziamo per analizzare altre grandezze, come il costo delle assunzioni nella procedura HIRE-ASSISTANT. Per svolgere un'analisi probabilistica, dobbiamo conoscere la distribuzione degli input oppure dobbiamo fare delle ipotesi su tale distribuzione. Poi analizziamo il nostro algoritmo, calcolando un tempo di esecuzione previsto. La previsione è fatta sulla distribuzione degli input possibili. Quindi, in effetti, stiamo mediando il tempo di esecuzione su tutti gli input possibili.

La scelta della distribuzione degli input richiede molta attenzione. Per alcuni problemi, ha senso ipotizzare un insieme di tutti i possibili input e applicare l'analisi probabilistica come tecnica per progettare algoritmi efficienti e come strumento di approfondimento dei problemi. Per altri problemi, invece, non è possibile ipotizzare una distribuzione di input accettabile; in questi casi, l'analisi probabilistica non può essere applicata.

Per il problema delle assunzioni, possiamo supporre che i candidati arrivino in ordine casuale. Che cosa significa ciò per questo problema? Noi supponiamo di poter confrontare due candidati qualsiasi e decidere quale dei due abbia i requisiti migliori; ovvero c'è un *ordine totale* nei candidati (vedere l'Appendice B per la definizione di ordine totale). Di conseguenza, possiamo classificare ogni candidato con un numero unico da 1 a n, utilizzando $rank(i)$ per in-

dicare il rango di appartenenza del candidato i, e adottare la convenzione che a un rango più alto corrisponda un candidato più qualificato. La lista ordinata $\langle rank(1), rank(2), \ldots, rank(n)\rangle$ è una permutazione della lista $\langle 1, 2, \ldots, n\rangle$. Dire che i candidati si presentano in ordine casuale equivale a dire che questa lista di ranghi ha la stessa probabilità di essere una qualsiasi delle $n!$ permutazioni dei numeri da 1 a n. In alternativa, possiamo dire che i ranghi formano una ***permutazione casuale uniforme***, ovvero ciascuna delle $n!$ possibili permutazioni si presenta con uguale probabilità. Il Paragrafo 5.2 contiene un'analisi probabilistica del problema delle assunzioni.

Algoritmi randomizzati

Per potere utilizzare l'analisi probabilistica, dobbiamo sapere qualcosa sulla distribuzione degli input. In molti casi, sappiamo molto poco su questa distribuzione. Anche quando sappiamo qualcosa sulla distribuzione degli input, potremmo non essere in grado di modellare computazionalmente questa conoscenza. Eppure, spesso, è possibile utilizzare la probabilità e la casualità come strumento per progettare e analizzare gli algoritmi, rendendo casuale il comportamento di parte dell'algoritmo.

Nel problema delle assunzioni, sembra che i candidati si presentino in ordine casuale, tuttavia non abbiamo modo di sapere se ciò sia vero o no. Quindi, per sviluppare un algoritmo randomizzato per il problema delle assunzioni, dobbiamo avere un controllo maggiore sull'ordine in cui svolgiamo i colloqui con i candidati; per questo motivo, modificheremo leggermente il modello. Supponiamo che l'agenzia di selezione del personale abbia n candidati e che ci invii in anticipo una lista di candidati. Ogni giorno, scegliamo a caso il candidato con il quale avere un colloquio. Sebbene non sappiamo nulla sui candidati (a parte i loro nomi), abbiamo fatto una significativa modifica. Anziché fare affidamento sull'ipotesi che i candidati si presentino in ordine casuale, abbiamo ottenuto il controllo del processo e imposto un ordine casuale.

In generale, diciamo che un algoritmo è ***randomizzato*** se il suo comportamento è determinato non soltanto dal suo input, ma anche dai valori prodotti da un ***generatore di numeri casuali***. Supporremo di avere a disposizione un generatore di numeri casuali, che chiameremo RANDOM. Una chiamata di RANDOM(a, b) restituisce un numero intero compreso fra a e b (estremi inclusi); ciascuno di questi numeri interi ha la stessa probabilità di essere generato. Per esempio, RANDOM$(0, 1)$ genera il numero 0 con probabilità $1/2$ e il numero 1 con probabilità $1/2$. Una chiamata di RANDOM$(3, 7)$ restituisce uno dei numeri 3, 4, 5, 6 e 7, ciascuno con probabilità $1/5$. Ogni intero generato da RANDOM è indipendente dagli interi generati nelle precedenti chiamate. Potete immaginare RANDOM come un dado con $(b - a + 1)$ facce per ottenere il suo output (in pratica, molti ambienti di programmazione hanno un ***generatore di numeri pseudocasuali***: un algoritmo deterministico che restituisce numeri che "sembrano" statisticamente casuali).

Esercizi

5.1-1

Dimostrate che l'ipotesi che siamo sempre in grado di determinare quale candidato sia migliore nella riga 4 della procedura HIRE-ASSISTANT implica che conosciamo un ordine totale sui ranghi dei candidati.

5.1-2 $\star$
Descrivete un'implementazione della procedura RANDOM(a, b) che effettua soltanto le chiamate RANDOM$(0, 1)$. Qual è il tempo di esecuzione previsto della vostra procedura in funzione di a e b?

5.1-3 $\star$
Supponete di generare il numero 0 con probabilità $1/2$ e 1 con probabilità $1/2$. Avete a disposizione una procedura BIASED-RANDOM che genera 0 o 1. Questa procedura genera 1 con probabilità p e 0 con probabilità $1-p$, dove $0 < p < 1$, ma non conoscete il valore di p. Create un algoritmo che usa BIASED-RANDOM come subroutine e restituisce una soluzione imparziale, restituendo 0 con probabilità $1/2$ e 1 con probabilità $1/2$. Qual è il tempo di esecuzione previsto del vostro algoritmo in funzione di p?

5.2 Variabili casuali indicatrici

Per potere analizzare vari algoritmi, incluso il problema delle assunzioni, utilizzeremo le variabili casuali indicatrici. Queste variabili offrono un metodo comodo per convertire probabilità e valori attesi. Supponiamo di avere lo spazio dei campioni S e un evento A. La ***variabile casuale indicatrice*** $\mathrm{I}\{A\}$ associata all'evento A è così definita

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{se si verifica l'evento } A \\ 0 & \text{se non si verifica l'evento } A \end{cases} \tag{5.1}$$

Come semplice esempio, supponiamo di lanciare in aria una moneta; vogliamo determinare il numero previsto di volte che otteniamo testa (T). Il nostro spazio dei campioni è $S = \{T, C\}$, con $\Pr\{T\} = \Pr\{C\} = 1/2$. Poi definiamo una variabile casuale indicatrice X_T, associata alla moneta che presenta testa, che è l'evento T. Questa variabile conta il numero di volte che si presenta testa in questo lancio; il suo valore è 1 se si presenta testa, altrimenti vale 0. Scriviamo

$$X_T = \mathrm{I}\{T\} = \begin{cases} 1 & \text{se si verifica } T \\ 0 & \text{se si verifica } C \end{cases}$$

Il numero atteso di T che si ottiene in un lancio della moneta è semplicemente il valore atteso della nostra variabile indicatrice X_T:

$$\begin{aligned} \mathrm{E}[X_T] &= \mathrm{E}[\mathrm{I}\{T\}] \\ &= 1 \cdot \Pr\{T\} + 0 \cdot \Pr\{C\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \end{aligned}$$

Dunque, il numero atteso di T che si ottiene in un lancio di una moneta è $1/2$. Come dimostra il seguente lemma, il valore atteso di una variabile casuale indicatrice associata a un evento A è uguale alla probabilità che si verifichi A.

Lemma 5.1
Se S è lo spazio dei campioni e A è un evento nello spazio dei campioni S, ponendo $X_A = \mathrm{I}\{A\}$, si ha $\mathrm{E}[X_A] = \Pr\{A\}$.

Dimostrazione Per le definizioni del valore atteso – equazione (C.19) – e della variabile casuale indicatrice – equazione (5.1) – possiamo scrivere

$$\begin{aligned}\mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\}\end{aligned}$$

dove $\overline{A}$ indica $S - A$, il complemento di A. ■

Sebbene possa sembrare complicato usare le variabili casuali indicatrici in applicazioni come quella di contare il numero previsto di volte che si presenta testa nel lancio di una moneta, tuttavia queste variabili sono utili per analizzare situazioni in cui effettuiamo ripetutamente delle prove casuali (*trial*). Per esempio, le variabili casuali indicatrici ci offrono un metodo semplice per arrivare al risultato dell'equazione (C.36). In questa equazione, calcoliamo il numero di volte che otteniamo testa in n lanci della moneta, considerando separatamente la probabilità di ottenere 0 volte testa, 1 volta testa, 2 volte testa e così via. Tuttavia, il metodo più semplice proposto nell'equazione (C.37), in effetti, usa implicitamente le variabili casuali indicatrici. Per essere più espliciti, indichiamo con X_i la variabile casuale indicatrice associata all'evento in cui si presenta testa nell'i-esimo lancio della moneta: $X_i = \mathrm{I}\{\text{nell'}i\text{-esimo lancio si verifica l'evento } T\}$. Sia X la variabile casuale che indica il numero totale di T in n lanci:

$$X = \sum_{i=1}^{n} X_i$$

Per calcolare il numero previsto di T, prendiamo il valore atteso di entrambi i membri della precedente equazione, ottenendo

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right]$$

Il membro sinistro di questa equazione è il valore atteso della somma di n variabili casuali. Per il Lemma 5.1, possiamo facilmente calcolare il valore atteso di ciascuna delle variabili casuali. Per l'equazione (C.20) – linearità del valore atteso – è facile calcolare il valore atteso della somma: è uguale alla somma dei valori attesi delle n variabili casuali. La linearità del valore atteso rende l'impiego delle variabili casuali indicatrici una potente tecnica analitica; si applica anche quando c'è dipendenza fra le variabili casuali. A questo punto possiamo calcolare facilmente il numero atteso di volte che si presenta testa:

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] \\ &= \sum_{i=1}^{n} 1/2 \\ &= n/2\end{aligned}$$

Quindi, paragonate al metodo utilizzato nell'equazione (C.36), le variabili casuali indicatrici semplificano notevolmente i calcoli. Utilizzeremo le variabili casuali indicatrici nella parte restante del libro.

Analisi del problema delle assunzioni con le variabili casuali indicatrici

Ritornando al problema delle assunzioni, adesso vogliamo calcolare il numero previsto di volte che assumiamo un nuovo impiegato. Per applicare l'analisi probabilistica, supponiamo che i candidati arrivino in ordine casuale, come descritto nel precedente paragrafo (vedremo nel Paragrafo 5.3 come rimuovere questa ipotesi). Sia X la variabile casuale il cui valore è uguale al numero di volte che assumiamo un nuovo impiegato. Applicando la definizione del valore atteso dell'equazione (C.19), otteniamo

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\}$$

Il calcolo di questa espressione non è semplice; utilizzeremo le variabili casuali indicatrici per semplificarlo notevolmente.

Per utilizzare le variabili casuali indicatrici, anziché calcolare $\mathrm{E}[X]$ definendo una variabile associata al numero di volte che assumiamo un nuovo impiegato, definiamo n variabili correlate al fatto che un candidato venga assunto oppure no. In particolare, indichiamo con X_i la variabile casuale indicatrice associata all'evento in cui l'i-esimo candidato sia assunto, ovvero

$$X_i = \mathrm{I}\{\text{il candidato } i \text{ è assunto}\} = \begin{cases} 1 & \text{se il candidato } i \text{ è assunto} \\ 0 & \text{se il candidato } i \text{ non è assunto} \end{cases} \tag{5.2}$$

e

$$X = X_1 + X_2 + \cdots + X_n \tag{5.3}$$

Per il Lemma 5.1, abbiamo che

$$\mathrm{E}[X_i] = \Pr\{\text{il candidato } i \text{ è assunto}\}$$

Quindi dobbiamo calcolare la probabilità che le righe 5–6 di HIRE-ASSISTANT siano eseguite.

Il candidato i è assunto, nella riga 5, esattamente quando il candidato i è migliore di tutti i candidati da 1 a $i-1$. Poiché abbiamo ipotizzato che i candidati arrivino in ordine casuale, i primi i candidati si sono presentati in ordine casuale. Uno qualsiasi dei primi i candidati ha la stessa possibilità di essere classificato come il migliore di tutti. Il candidato i ha la probabilità $1/i$ di essere qualificato il migliore dei candidati da 1 a $i-1$ e, quindi, ha la probabilità $1/i$ di essere assunto. Per il Lemma 5.1, concludiamo che

$$\mathrm{E}[X_i] = 1/i \tag{5.4}$$

Adesso possiamo calcolare $\mathrm{E}[X]$:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(per l'equazione (5.3))} && (5.5) \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(per la linearità del valore atteso)} \\ &= \sum_{i=1}^{n} 1/i && \text{(per l'equazione (5.4))} \\ &= \ln n + O(1) && \text{(per l'equazione (A.7))} && (5.6) \end{aligned}$$

Anche se abbiamo un colloquio con n candidati, assumiamo approssimativamente soltanto $\ln n$ di essi, in media. Sintetizziamo questo risultato nel seguente lemma.

Lemma 5.2
Supponendo che i candidati si presentino in ordine casuale, l'algoritmo HIRE-ASSISTANT ha un costo totale per le assunzioni pari a $O(c_h \ln n)$.

Dimostrazione Il limite si ricava immediatamente dalla nostra definizione di costo di assunzione e dall'equazione (5.6). ■

Il costo previsto per le assunzioni è un significativo miglioramento rispetto al costo $O(nc_h)$ per le assunzioni nel caso peggiore.

Esercizi

5.2-1
Nella procedura HIRE-ASSISTANT, supponendo che i candidati si presentino in ordine casuale, qual è la probabilità che venga effettuata esattamente una sola assunzione? Qual è la probabilità che vengano effettuate esattamente n assunzioni?

5.2-2
Nella procedura HIRE-ASSISTANT, supponendo che i candidati si presentino in ordine casuale, qual è la probabilità che vengano effettuate esattamente due assunzioni?

5.2-3
Utilizzate le variabili casuali indicatrici per calcolare il valore atteso della somma di n dadi.

5.2-4
Utilizzate le variabili casuali indicatrici per risolvere il seguente problema, che chiameremo ***problema dei cappelli***. Ognuno degli n clienti consegna il suo cappello a una persona che si trova all'ingresso di un ristorante. La persona restituisce i cappelli ai clienti in ordine casuale. Qual è il numero atteso di clienti che riceveranno il loro cappello?

5.2-5
Sia $A[1 \,..\, n]$ un array di n numeri distinti. Se $i < j$ e $A[i] > A[j]$, allora la coppia (i, j) è detta ***inversione*** di A (vedere il Problema 2-4 per maggiori informazioni sulle inversioni). Supponete che gli elementi di A formino una permutazione casuale uniforme di $\langle 1, 2, \ldots, n \rangle$. Utilizzate le variabili casuali indicatrici per calcolare il numero atteso di inversioni.

5.3 Algoritmi randomizzati

Nel precedente paragrafo abbiamo visto che, se conosciamo una distribuzione degli input, è più facile analizzare il comportamento nel caso medio di un algoritmo. Molte volte, non abbiamo simili conoscenze e non è possibile svolgere alcuna analisi del caso medio. Come detto nel Paragrafo 5.1, potremmo utilizzare un algoritmo randomizzato. Per un problema come quello delle assunzioni, in cui è utile ipotizzare che tutte le permutazioni dell'input siano egualmente possibili, l'analisi probabilistica guiderà lo sviluppo di un algoritmo randomizzato. Anziché ipotizzare una distribuzione degli input, imponiamo noi una distribuzione. In particolare, prima di eseguire l'algoritmo, permutiamo casualmente i candidati per imporre la proprietà che ogni permutazione sia egualmente possibile. Questa

modifica non cambia il valore atteso di assumere $\ln n$ volte circa un nuovo impiegato; essa significa, però, che *qualsiasi* input può essere considerato il caso da esaminare, non l'input ottenuto da una particolare distribuzione.

Analizziamo meglio la differenza fra analisi probabilistica e algoritmi randomizzati. Nel Paragrafo 5.2 abbiamo visto che, supponendo che i candidati si presentino in ordine casuale, il numero previsto di volte che assumiamo un nuovo impiegato è pari a circa $\ln n$. Notate che l'algoritmo qui è deterministico; per un particolare input, il numero di volte che viene assunto un nuovo impiegato sarà sempre lo stesso. Inoltre, il numero di volte che assumiamo un nuovo impiegato è diverso se cambia l'input e dipende dai ranghi dei vari candidati. Poiché questo numero dipende soltanto dai ranghi dei candidati, possiamo rappresentare un particolare input elencando, in ordine, tali ranghi, ovvero $\langle rank(1), rank(2), \ldots, rank(n)\rangle$.

Data la lista dei ranghi $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$, un nuovo impiegato sarà assunto 10 volte, in quanto ogni candidato è migliore del precedente; le righe 5–6 saranno eseguite in ogni iterazione dell'algoritmo. Data la lista dei ranghi $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$, un nuovo impiegato sarà assunto una sola volta, nella prima iterazione. Data la lista dei ranghi $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$, un nuovo impiegato sarà assunto tre volte, dopo i colloqui con i candidati dei ranghi 5, 8 e 10. Ricordando che il costo del nostro algoritmo dipende dal numero di volte che viene assunto un nuovo impiegato, notiamo che ci sono input costosi, come A_1, input economici, come A_2, e input moderatamente costosi, come A_3.

Consideriamo, d'altra parte, l'algoritmo randomizzato che prima permuta i candidati e poi determina il miglior candidato. In questo caso, la randomizzazione è nell'algoritmo, non nella distribuzione degli input. Dato un particolare input (per esempio A_3), non possiamo dire quante volte sarà aggiornato il massimo, perché questa quantità è diversa ogni volta che viene eseguito l'algoritmo. La prima volta che eseguiamo l'algoritmo su A_3, potremmo ottenere la permutazione A_1 con 10 aggiornamenti, mentre la seconda volta che eseguiamo l'algoritmo, potremmo ottenere la permutazione A_2 con un solo aggiornamento. La terza volta, l'algoritmo potrebbe effettuare qualche altro numero di aggiornamenti. Ogni volta che eseguiamo l'algoritmo, l'esecuzione dipende dalle scelte casuali fatte ed è probabile che sia diversa dalle precedenti esecuzioni. Per questo e per molti altri algoritmi randomizzati, *nessun input determina il caso peggiore nel comportamento dell'algoritmo*. Neppure il vostro peggior nemico è in grado di produrre un brutto array di input, in quanto la permutazione casuale rende irrilevante l'ordine degli input. L'algoritmo randomizzato si comporta male soltanto se il generatore di numeri casuali produce una permutazione "sventurata".

Per il problema delle assunzioni, l'unica modifica da apportare al codice è permutare casualmente l'array.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  permuta casualmente la lista dei candidati
2  best ← 0        ▷ il candidato 0 è il meno qualificato
3  for i ← 1 to n
4      do colloquio con il candidato i
5         if il candidato i è migliore del candidato best
6           then best ← i
7                assumi il candidato i
```

Con questa semplice modifica abbiamo creato un algoritmo randomizzato le cui prestazioni corrispondono a quelle ottenute supponendo che i candidati si presentino in ordine casuale.

Lemma 5.3
Il costo previsto per assumere nuovi impiegati nella procedura RANDOMIZED-HIRE-ASSISTANT è $O(c_h \ln n)$.

Dimostrazione Dopo avere permutato l'array di input, abbiamo una situazione identica a quella dell'analisi probabilistica di HIRE-ASSISTANT. ■

Il confronto fra i Lemmi 5.2 e 5.3 mette in risalto la differenza fra analisi probabilistica e algoritmi randomizzati. Nel Lemma 5.2 facciamo un'ipotesi sull'input. Nel Lemma 5.3 non facciamo alcuna ipotesi, sebbene la randomizzazione dell'input richieda un tempo aggiuntivo. Nel prossimo paragrafo analizzeremo alcuni problemi relativi alla permutazione casuale dell'input.

Permutazione casuale degli array

Molti algoritmi randomizzati effettuano la randomizzazione dell'input permutando un dato array di input (ci sono altri metodi per applicare la randomizzazione). Qui, adotteremo due metodi per farlo. Supponiamo di avere un array A che, senza perdere di generalità, contenga gli elementi da 1 a n. Il nostro obiettivo è produrre una permutazione casuale dell'array.

Un tipico metodo consiste nell'assegnare a ogni elemento $A[i]$ dell'array una priorità casuale $P[i]$ e, poi, nell'ordinare gli elementi di A in funzione di queste priorità. Per esempio, se il nostro array iniziale è $A = \langle 1, 2, 3, 4 \rangle$ e scegliamo le priorità casuali $P = \langle 36, 3, 97, 19 \rangle$, otterremo l'array $B = \langle 2, 4, 1, 3 \rangle$, in quanto la seconda priorità è la più piccola, seguita dalla quarta, poi dalla prima e, infine, dalla terza. Chiamiamo questa procedura PERMUTE-BY-SORTING:

```
PERMUTE-BY-SORTING(A)
1  n ← lunghezza[A]
2  for i ← 1 to n
3        do P[i] = RANDOM(1, n^3)
4  ordina A, utilizzando P come chiavi di ordinamento
5  return A
```

La riga 3 sceglie un numero casuale fra 1 e n^3. Utilizziamo un intervallo da 1 a n^3 per avere la possibilità che tutte le priorità di P siano uniche (l'Esercizio 5.3-5 chiede di dimostrare che la probabilità che tutte le priorità siano uniche è almeno $1 - 1/n$; l'Esercizio 5.3-6 chiede come implementare l'algoritmo anche se due o più priorità sono identiche). Supponiamo che tutte le priorità siano uniche.

Il passo che consuma tempo in questa procedura è l'ordinamento nella riga 4. Come vedremo nel Capitolo **??**, se utilizziamo un ordinamento per confronti, l'operazione di ordinamento richiede un tempo $\Omega(n \lg n)$. Possiamo raggiungere questo limite inferiore, perché abbiamo visto che merge sort richiede un tempo $\Theta(n \lg n)$ (nella Parte **??** vedremo altri ordinamenti per confronti che richiedono un tempo $\Theta(n \lg n)$). Dopo l'ordinamento, se $P[i]$ è la j-esima priorità più piccola, allora $A[i]$ sarà nella posizione j dell'output. In questo modo otteniamo una permutazione. Resta da dimostrare che la procedura genera una ***permutazione ca-***

suale uniforme, ovvero che ogni permutazione dei numeri da 1 a n ha la stessa probabilità di essere prodotta.

Lemma 5.4
Se tutte le priorità sono distinte, la procedura PERMUTE-BY-SORTING genera una permutazione casuale uniforme dell'input.

Dimostrazione Iniziamo a considerare la particolare permutazione in cui ogni elemento $A[i]$ riceve la i-esima priorità più piccola. Dovremo dimostrare che questa permutazione si verifica con una probabilità esattamente pari a $1/n!$. Per $i = 1, 2, \ldots, n$, sia X_i l'evento in cui l'elemento $A[i]$ riceve la i-esima priorità più piccola. La probabilità che si verifichi l'evento X_i per qualsiasi i è data da

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\}$$

Applicando l'Esercizio C.2-6, questa probabilità è uguale a

$$\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\}$$
$$\cdots \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\}$$

Otteniamo che $\Pr\{X_1\} = 1/n$, perché è la probabilità che una priorità scelta a caso da un insieme di n elementi sia la più piccola. Poi, osserviamo che $\Pr\{X_2 \mid X_1\} = 1/(n-1)$, in quanto, dato che l'elemento $A[1]$ ha la priorità più piccola, ciascuno dei restanti $n-1$ elementi ha la stessa probabilità di avere la seconda priorità più piccola. In generale, per $i = 2, 3, \ldots, n$, abbiamo che $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1)$, in quanto, dato che gli elementi da $A[1]$ a $A[i-1]$ hanno le $i-1$ priorità più piccole (in ordine), ciascuno dei restanti $n-(i-1)$ elementi ha la stessa probabilità di avere la i-esima priorità più piccola; quindi, abbiamo

$$\begin{aligned}\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ &= \frac{1}{n!}\end{aligned}$$

Abbiamo dimostrato che la probabilità di ottenere la permutazione identità è $1/n!$.

Possiamo estendere questa dimostrazione a qualsiasi permutazione delle priorità. Consideriamo una permutazione fissa $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$ dell'insieme $\{1, 2, \ldots, n\}$. Indichiamo con r_i il rango della priorità assegnata all'elemento $A[i]$, dove l'elemento con la j-esima priorità più piccola ha rango j. Se definiamo X_i l'evento in cui l'elemento $A[i]$ riceve la $\sigma(i)$-esima priorità più piccola, o $r_i = \sigma(i)$, si può applicare ancora la stessa dimostrazione. Ne consegue che, per determinare la probabilità di ottenere una particolare permutazione, dobbiamo svolgere gli stessi calcoli del precedente caso; quindi la probabilità di ottenere questa permutazione è ancora $1/n!$. ■

Si potrebbe pensare che per dimostrare che una permutazione sia casuale e uniforme sia sufficiente dimostrare che, per ogni elemento $A[i]$, la probabilità che esso raggiunga la posizione j sia $1/n$. L'Esercizio 5.3-4 dimostra che questa condizione più debole, in effetti, non è sufficiente.

Un metodo migliore per generare una permutazione casuale consiste nel permutare l'array sul posto. La procedura RANDOMIZE-IN-PLACE lo fa nel tempo $O(n)$. Nell'iterazione i, l'elemento $A[i]$ viene scelto casualmente fra gli elementi compresi fra $A[i]$ e $A[n]$. Dopo l'iterazione i, $A[i]$ non viene più modificato.

RANDOMIZE-IN-PLACE(A)

1 $n \leftarrow lunghezza[A]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** scambia $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

Utilizzeremo una invariante di ciclo per dimostrare che questa procedura produce una permutazione casuale uniforme. Dato un insieme di n elementi, una permutazione-k è una sequenza che contiene k degli n elementi (vedere l'Appendice C). In un insieme di n elementi ci sono $n!/(n-k)!$ permutazioni-k.

Lemma 5.5
La procedura RANDOMIZE-IN-PLACE genera una permutazione casuale uniforme.

Dimostrazione Utilizziamo la seguente invariante di ciclo:

> Appena prima della i-esima iterazione del ciclo **for**, righe 2–3, per ogni possibile permutazione-$(i-1)$, il sottoarray $A[1\,..\,i-1]$ contiene questa permutazione-$(i-1)$ con probabilità $(n-i+1)!/n!$.

Dobbiamo dimostrare che questa invariante è vera prima della prima iterazione del ciclo, che ogni iterazione del ciclo conserva l'invariante e che l'invariante fornisce un'utile proprietà per dimostrare la correttezza quando il ciclo termina.

Inizializzazione: consideriamo la situazione appena prima della prima iterazione del ciclo, con $i = 1$. L'invariante di ciclo dice che per ogni possibile permutazione-0, il sottoarray $A[1\,..\,0]$ contiene questa permutazione-0 con probabilità $(n-i+1)!/n! = n!/n! = 1$. Il sottoarray $A[1\,..\,0]$ è vuoto e una permutazione-0 non ha elementi. Quindi, $A[1\,..\,0]$ contiene qualsiasi permutazione-0 con probabilità 1 e l'invariante di ciclo è vera prima della prima iterazione.

Conservazione: supponiamo che, appena prima della i-esima iterazione, ogni possibile permutazione-$(i-1)$ appaia nel sottoarray $A[1\,..\,i-1]$ con probabilità $(n-i+1)!/n!$; dimostreremo che, dopo l'i-esima iterazione, ogni possibile permutazione-i appare nel sottoarray $A[1\,..\,i]$ con probabilità $(n-i)!/n!$. L'incremento di i per la successiva iterazione manterrà l'invariante di ciclo.

Esaminiamo la i-esima iterazione. Consideriamo una particolare permutazione-i e indichiamo i suoi gli elementi con $\langle x_1, x_2, \ldots, x_i\rangle$. Questa permutazione è formata da una permutazione-$(i-1)$ $\langle x_1, \ldots, x_{i-1}\rangle$ seguita dal valore x_i che l'algoritmo pone in $A[i]$. Indichiamo con E_1 l'evento in cui le prime $i-1$ iterazioni hanno creato la particolare permutazione-$(i-1)$ $\langle x_1, \ldots, x_{i-1}\rangle$ in $A[1\,..\,i-1]$. Per l'invariante di ciclo, $\Pr\{E_1\} = (n-i+1)!/n!$. Sia E_2 l'evento in cui la i-esima iterazione pone x_i nella posizione $A[i]$. La permutazione-i $\langle x_1, \ldots, x_i\rangle$ si forma in $A[1\,..\,i]$ precisamente quando si verificano entrambi gli eventi E_1 ed E_2, quindi calcoliamo $\Pr\{E_2 \cap E_1\}$. Applicando l'equazione (C.14), abbiamo

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\}\Pr\{E_1\}$$

La probabilità $\Pr\{E_2 \mid E_1\}$ è uguale a $1/(n-i+1)$, perché nella riga 3 l'algoritmo sceglie x_i a caso dagli $n-i+1$ valori nelle posizioni $A[i\,..\,n]$.

Dunque, otteniamo

$$\begin{aligned}\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\}\Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!}\end{aligned}$$

Conclusione: alla fine del ciclo, $i = n+1$ e il sottoarray $A[1\,..\,n]$ è una particolare permutazione-n con probabilità $(n-n)!/n! = 1/n!$.

Quindi, la procedura RANDOMIZE-IN-PLACE genera una permutazione casuale uniforme. ■

Un algoritmo randomizzato spesso è il metodo più semplice ed efficiente per risolvere un problema. In questo libro utilizzeremo saltuariamente gli algoritmi randomizzati.

Esercizi

5.3-1

Il professor Marceau non è d'accordo con l'invariante di ciclo utilizzata nella dimostrazione del Lemma 5.5. Egli dubita che sia vera prima della prima iterazione. Il suo ragionamento è che uno potrebbe altrettanto facilmente dichiarare che un sottoarray non contiene permutazioni-0. Di conseguenza, la probabilità che un sottoarray vuoto contenga una permutazione-0 dovrebbe essere 0, invalidando così l'invariante di ciclo prima della prima iterazione. Riscrivete la procedura RANDOMIZE-IN-PLACE in modo che la sua invariante di ciclo si applichi a un sottoarray non vuoto prima della prima iterazione e modificate la dimostrazione del Lemma 5.5 per la vostra procedura.

5.3-2

Il professor Kelp decide di scrivere una procedura che produrrà casualmente qualsiasi permutazione oltre alla permutazione identità; propone la seguente procedura:

```
PERMUTE-WITHOUT-IDENTITY(A)
1  n ← lunghezza[A]
2  for i ← 1 to n − 1
3       do scambia A[i] ↔ A[RANDOM(i + 1, n)]
```

Questo codice fa ciò che intende il professor Kelp?

5.3-3

Anziché scambiare l'elemento $A[i]$ con un elemento a caso del sottoarray $A[i\,..\,n]$, supponete di scambiarlo con un elemento a caso dell'array:

```
PERMUTE-WITH-ALL(A)
1  n ← lunghezza[A]
2  for i ← 1 to n
3       do scambia A[i] ↔ A[RANDOM(1, n)]
```

Questo codice genera una permutazione casuale uniforme? Perché o perché no?

5.3-4
Il professor Armstrong suggerisce la seguente procedura per generare una permutazione casuale uniforme:

PERMUTE-BY-CYCLIC(A)

```
1  n ← lunghezza[A]
2  offset ← RANDOM(1, n)
3  for i ← 1 to n
4       do dest ← i + offset
5          if dest > n
6            then dest ← dest − n
7          B[dest] ← A[i]
8  return B
```

Dimostrate che ogni elemento $A[i]$ ha probabilità $1/n$ di trovarsi in una particolare posizione in B. Poi dimostrate che il professor Armstrong sbaglia, dimostrando che la permutazione risultante non è uniformemente casuale.

5.3-5 $\star$
Dimostrate che nell'array P della procedura PERMUTE-BY-SORTING la probabilità che tutti gli elementi siano unici è almeno $1 - 1/n$.

5.3-6
Spiegate come implementare l'algoritmo PERMUTE-BY-SORTING per gestire il caso in cui due o più priorità siano identiche; ovvero il vostro algoritmo dovrebbe produrre una permutazione casuale uniforme, anche se due o più priorità sono identiche.

$\star$ 5.4 Approfondimento dell'analisi probabilistica

Quest'ultima parte del capitolo approfondisce l'analisi probabilistica illustrando quattro esempi. Il primo esempio determina la probabilità che, in una stanza di k persone, due di esse siano nate nello stesso giorno dell'anno. Il secondo esempio studia il lancio casuale delle palline nei contenitori. Il terzo esempio analizza una sequenza di teste consecutive nel lancio di una moneta. L'ultimo esempio studia una variante del problema delle assunzioni, dove bisogna prendere delle decisioni senza avere un colloquio con tutti i candidati.

5.4.1 Il paradosso del compleanno

Il nostro primo esempio è il ***paradosso del compleanno***. Quante persone devono trovarsi in una stanza prima che ci sia una probabilità del 50% che due di esse siano nate nello stesso giorno dell'anno? Sorprendentemente, la risposta è: poche. Il paradosso è che, in effetti, ne bastano molte di meno del numero di giorni di un anno o anche la metà del numero di giorni di un anno, come vedremo.

Per rispondere a questa domanda, assegniamo a ogni persona un indice intero $1, 2, \ldots, k$, dove k è il numero delle persone che si trovano nella stanza. Ignoriamo il problema degli anni bisestili e supponiamo che tutti gli anni abbiano $n = 365$ giorni. Per $i = 1, 2, \ldots, k$, sia b_i il giorno dell'anno in cui è nata la persona i, con $1 \leq b_i \leq n$. Supponiamo inoltre che i compleanni siano uniformemente distribuiti negli n giorni dell'anno, quindi $\Pr\{b_i = r\} = 1/n$ per $i = 1, 2, \ldots, k$ e $r = 1, 2, \ldots, n$.

La probabilità che due persone, i e j, siano nate nello stesso giorno dell'anno dipende dal fatto che la selezione casuale dei compleanni sia indipendente. Nell'ipotesi che i compleanni siano indipendenti, la probabilità che il compleanno di i e il compleanno di j siano nel giorno r è

$$\begin{aligned}\Pr\{b_i = r \text{ and } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= 1/n^2\end{aligned}$$

Quindi, la probabilità che entrambi i compleanni siano nello stesso giorno è

$$\begin{aligned}\Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ and } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n \qquad (5.7)\end{aligned}$$

Più intuitivamente, una volta scelto b_i, la probabilità che b_j sia scelto nello stesso giorno è $1/n$. Quindi, la probabilità che i e j abbiano lo stesso compleanno è la stessa probabilità che il compleanno di uno di essi sia in un dato giorno. Notate, tuttavia, che questa coincidenza dipende dall'ipotesi che i compleanni siano indipendenti. Possiamo analizzare la probabilità di almeno 2 su k persone che hanno lo stesso compleanno, osservando l'evento complementare. La probabilità che almeno due compleanni coincidano è 1 meno la probabilità che tutti i compleanni siano differenti. L'evento in cui k persone abbiano compleanni distinti è

$$B_k = \bigcap_{i=1}^{k} A_i$$

dove A_i è l'evento in cui il compleanno della persona i sia diverso da quello della persona j per ogni $j < i$. Poiché possiamo scrivere $B_k = A_k \cap B_{k-1}$, otteniamo dall'equazione (C.16) la ricorrenza

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \qquad (5.8)$$

dove consideriamo $\Pr\{B_1\} = \Pr\{A_1\} = 1$ come una condizione iniziale. In altre parole, la probabilità che $b_1, b_2, \ldots, b_k$ siano compleanni distinti è la probabilità che $b_1, b_2, \ldots, b_{k-1}$ siano compleanni distinti moltiplicata per la probabilità che $b_k \neq b_i$ per $i = 1, 2, \ldots, k-1$, dato che $b_1, b_2, \ldots, b_{k-1}$ sono distinti.

Se $b_1, b_2, \ldots, b_{k-1}$ sono distinti, la probabilità condizionale che $b_k \neq b_i$ per $i = 1, 2, \ldots, k-1$ è $\Pr\{A_k \mid B_{k-1}\} = (n-k+1)/n$, dal momento che, su n giorni, $n-(k-1)$ giorni non vengono considerati. Applicando iterativamente la ricorrenza (5.8), otteniamo

$$\begin{aligned}\Pr\{B_k\} &= \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \\ &= \Pr\{B_{k-2}\}\Pr\{A_{k-1} \mid B_{k-2}\}\Pr\{A_k \mid B_{k-1}\} \\ &\vdots \\ &= \Pr\{B_1\}\Pr\{A_2 \mid B_1\}\Pr\{A_3 \mid B_2\}\cdots\Pr\{A_k \mid B_{k-1}\} \\ &= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) \\ &= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)\end{aligned}$$

Dalla disuguaglianza (3.11), $1 + x \leq e^x$, si ottiene

$$\begin{aligned}\Pr\{B_k\} &\leq e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n} \\ &= e^{-\sum_{i=1}^{k-1} i/n} \\ &= e^{-k(k-1)/2n} \\ &\leq 1/2\end{aligned}$$

quando $-k(k-1)/2n \leq \ln(1/2)$. La probabilità che tutti i k compleanni siano distinti è al massimo $1/2$ quando $k(k-1) \geq 2n\ln 2$ oppure, risolvendo l'equazione quadratica, quando $k \geq (1 + \sqrt{1 + (8\ln 2)n})/2$. Per $n = 365$, dobbiamo avere $k \geq 23$. Quindi, se nella stanza ci sono almeno 23 persone, la probabilità che almeno due di esse abbiano lo stesso compleanno è almeno $1/2$. Su Marte, un anno dura 669 giorni, quindi occorrono 31 marziani per ottenere lo stesso effetto.

Un'analisi tramite le variabili casuali indicatrici

Possiamo usare le variabili casuali indicatrici per fare un'analisi più semplice, ma approssimata, del paradosso del compleanno. Per ogni coppia (i, j) delle k persone che si trovano nella stanza, definiamo la variabile casuale indicatrice X_{ij}, con $1 \leq i < j \leq k$, in questo modo

$$\begin{aligned}X_{ij} &= \mathrm{I}\{\text{la persona } i \text{ e la persona } j \text{ hanno lo stesso compleanno}\} \\ &= \begin{cases}1 & \text{se la persona } i \text{ e la persona } j \text{ hanno lo stesso compleanno} \\ 0 & \text{negli altri casi}\end{cases}\end{aligned}$$

Per l'equazione (5.7), la probabilità che due persone abbiano lo stesso compleanno è $1/n$, quindi per il Lemma 5.1, abbiamo

$$\begin{aligned}\mathrm{E}[X_{ij}] &= \Pr\{\text{la persona } i \text{ e la persona } j \text{ hanno lo stesso compleanno}\} \\ &= 1/n\end{aligned}$$

Indicando con X la variabile casuale che conta il numero di coppie di persone che hanno lo stesso compleanno, abbiamo

$$X = \sum_{i=1}^{k}\sum_{j=i+1}^{k} X_{ij}$$

Prendendo i valori attesi da entrambi i membri dell'equazione e applicando la linearità del valore atteso, otteniamo

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k}\sum_{j=i+1}^{k} X_{ij}\right] \\ &= \sum_{i=1}^{k}\sum_{j=i+1}^{k}\mathrm{E}[X_{ij}] \\ &= \binom{k}{2}\frac{1}{n} \\ &= \frac{k(k-1)}{2n}\end{aligned}$$

Se $k(k-1) \geq 2n$, il numero atteso di coppie di persone che hanno lo stesso compleanno è almeno 1. Dunque, se nella stanza ci sono almeno $\sqrt{2n} + 1$ persone, possiamo prevedere che almeno due persone siano nate nello stesso giorno

dell'anno. Per $n = 365$, se $k = 28$, il numero atteso di coppie con lo stesso compleanno è $(28 \cdot 27)/(2 \cdot 365) \approx 1,0356$. Quindi, con almeno 28 persone, ci aspettiamo di trovare almeno una coppia con lo stesso compleanno. Su Marte, dove un anno dura 669 giorni, occorrono almeno 38 marziani.

La prima analisi, che utilizzava soltanto le probabilità, ha determinato il numero di persone necessarie affinché la probabilità che ci sia una coppia con lo stesso compleanno superi $1/2$. La seconda analisi, che utilizzava le variabili casuali indicatrici, ha determinato un numero tale che il numero atteso di coppie di persone con lo stesso compleanno sia 1. Sebbene i numeri esatti di persone differiscano nei due casi, tuttavia essi sono asintoticamente uguali: $\Theta(\sqrt{n})$.

5.4.2 Lancio delle palline nei contenitori

Consideriamo il processo di lanciare a caso delle palline identiche in b contenitori, numerati $1, 2, \ldots, b$. I lanci sono indipendenti e in ogni lancio la pallina ha la stessa probabilità di finire in qualsiasi contenitore. La probabilità che una pallina lanciata finisca in un determinato contenitore è $1/b$. Quindi, il lancio delle palline è una sequenza di prove ripetute di Bernoulli (vedere l'Appendice C.4) con una probabilità $1/b$ di successo, dove successo significa che la pallina finisce in un determinato contenitore. Questo modello è particolarmente utile per analizzare l'hashing (vedere il Capitolo **??**) e noi possiamo rispondere a una serie di interessanti domande sul lancio delle palline (il Problema C-1 pone altre domande sul lancio delle palline).

Quante palline finiscono in un determinato contenitore? Il numero di palline che cadono in un determinato contenitore segue la distribuzione binomiale $b(k; n, 1/b)$. Se vengono lanciate n palline, l'equazione (C.36) ci dice che il numero atteso di palline che finiscono in un determinato contenitore è n/b.

Quante palline devono essere lanciate, in media, affinché una di esse finisca in un determinato contenitore? Il numero di lanci da effettuare affinché una pallina finisca in un determinato contenitore segue la distribuzione geometrica con probabilità $1/b$ e, per l'equazione (C.31), il numero atteso di lanci per fare centro in quel contenitore è $1/(1/b) = b$.

Quante palline devono essere lanciate affinché ogni contenitore abbia almeno una pallina? Chiamiamo "centro" un lancio in cui una pallina cade in un determinato contenitore. Vogliamo determinare il numero atteso n di lanci necessari per fare b centri.

I centri possono essere utilizzati per ripartire gli n lanci in fasi. L'i-esima fase è formata dai lanci effettuati dopo l'$(i-1)$-esimo centro fino all'i-esimo centro. La prima fase è formata dal primo lancio, perché abbiamo la certezza di fare centro quando tutti i contenitori sono vuoti. Per ogni lancio effettuato durante l'i-esima fase, ci sono $i - 1$ contenitori che contengono palline e $b - i + 1$ contenitori vuoti. Quindi, per ogni lancio nell'i-esima fase, la probabilità di fare centro è $(b - i + 1)/b$.

Se indichiamo con n_i il numero di lanci nell'i-esima fase, il numero di lanci necessari per fare b centri è $n = \sum_{i=1}^{b} n_i$. Ogni variabile casuale n_i ha una distribuzione geometrica con probabilità di successo $(b - i + 1)/b$ e, per l'equazione (C.31), abbiamo

$$\mathrm{E}[n_i] = \frac{b}{b - i + 1}$$

Applicando la linearità del valore atteso, otteniamo

$$\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b\sum_{i=1}^{b} \frac{1}{i} \\
&= b(\ln b + O(1))
\end{aligned}$$

L'ultima riga deriva dal limite (A.7) sulle serie armoniche. Dunque, occorrono circa $b \ln b$ lanci prima di prevedere che ogni contenitore abbia una pallina. Questo problema, detto anche ***problema del collezionista di figurine***, dice che una persona che tenta di collezionare b figurine differenti deve raccogliere a caso circa $b \ln b$ figurine per raggiungere il suo obiettivo.

5.4.3 Serie dello stesso evento

Supponete di lanciare n volte una moneta. Qual è la serie più lunga di teste consecutive che prevedete di ottenere? La risposta è $\Theta(\lg n)$, come dimostra la seguente analisi.

Prima dimostriamo che la lunghezza prevista della serie più lunga di teste è $O(\lg n)$. La probabilità che in ogni lancio si ottenga testa è pari a $1/2$. Indichiamo con A_{ik} l'evento in cui una serie di teste di lunghezza almeno uguale a k inizi con l'i-esimo lancio della moneta o, più precisamente, l'evento in cui per k lanci consecutivi $i, i+1, \ldots, i+k-1$ si presenti sempre testa, dove $1 \le k \le n$ e $1 \le i \le n-k+1$. Poiché i lanci della moneta sono mutuamente indipendenti, per un dato evento A_{ik}, la probabilità che in tutti i k lanci si ottenga testa è

$$\Pr\{A_{ik}\} = 1/2^k \tag{5.9}$$

Per $k = 2\lceil \lg n \rceil$, si ha

$$\begin{aligned}
\Pr\{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\
&\le 1/2^{2 \lg n} \\
&= 1/n^2
\end{aligned}$$

Quindi la probabilità che una serie di teste di lunghezza almeno pari a $2\lceil \lg n \rceil$ inizi nella posizione i è molto piccola. Ci sono al massimo $n - 2\lceil \lg n \rceil + 1$ posizioni dove tale serie può iniziare. Pertanto, la probabilità che una serie di teste di lunghezza almeno pari a $2\lceil \lg n \rceil$ inizi in qualsiasi posizione è

$$\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} 1/n^2 \\
&< \sum_{i=1}^{n} 1/n^2 \\
&= 1/n
\end{aligned} \tag{5.10}$$

In quanto, per la disuguaglianza di Boole (C.18), la probabilità dell'unione di più eventi è pari al massimo alla somma delle probabilità dei singoli eventi (notate che la disuguaglianza di Boole è valida anche per eventi come questi che non sono indipendenti).

Adesso utilizziamo la disuguaglianza (5.10) per limitare la lunghezza della serie più lunga. Per $j = 0, 1, 2, \ldots, n$, indichiamo con L_j l'evento in cui la più lunga serie di teste abbia una lunghezza esattamente pari a j e con L la lunghezza della serie più lunga. Per la definizione del valore atteso, si ha

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\} \tag{5.11}$$

Potremmo provare a calcolare questa sommatoria utilizzando i limiti superiori su ogni $\Pr\{L_j\}$ simili a quelli calcolati nella disuguaglianza (5.10). Purtroppo questo metodo genera limiti deboli. Tuttavia, possiamo sfruttare la precedente analisi per ottenere un buon limite. Informalmente, osserviamo che per nessun termine della sommatoria nell'equazione (5.11) i fattori j e $\Pr\{L_j\}$ sono entrambi grandi. Perché? Quando $j \geq 2\lceil \lg n \rceil$, allora $\Pr\{L_j\}$ è molto piccolo; quando $j < 2\lceil \lg n \rceil$, allora j è piuttosto piccolo. Più formalmente, notiamo che gli eventi L_j per $j = 0, 1, \ldots, n$ sono disgiunti; quindi la probabilità che una serie di teste di lunghezza almeno pari a $2\lceil \lg n \rceil$ inizi in qualsiasi posizione è $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. Per la disuguaglianza (5.10), abbiamo $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < 1/n$. Inoltre, notando che $\sum_{j=0}^{n} \Pr\{L_j\} = 1$, allora $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} \leq 1$. Quindi, otteniamo

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot (1/n) \\
&= O(\lg n)
\end{aligned}$$

Le probabilità che una serie di teste superi $r\lceil \lg n \rceil$ lanci diminuiscono rapidamente con r. Per $r \geq 1$, la probabilità che una serie di $r\lceil \lg n \rceil$ teste inizi nella posizione i è

$$\begin{aligned}
\Pr\{A_{i, r\lceil \lg n \rceil}\} &= 1/2^{r\lceil \lg n \rceil} \\
&\leq 1/n^r
\end{aligned}$$

Quindi, la probabilità che la serie più lunga sia almeno $r\lceil \lg n \rceil$ è al massimo $n/n^r = 1/n^{r-1}$ ovvero la probabilità che la serie più lunga abbia una lunghezza minore di $r\lceil \lg n \rceil$ è almeno $1 - 1/n^{r-1}$.

Per esempio, se i lanci della moneta sono $n = 1000$, la probabilità di avere una serie di almeno $2\lceil \lg n \rceil = 20$ teste è al massimo $1/n = 1/1000$. Le probabilità di avere una serie più lunga di $3\lceil \lg n \rceil = 30$ teste è al massimo $1/n^2 = 1/1\,000\,000$.

Adesso dimostriamo un limite inferiore complementare: la lunghezza prevista della serie più lunga di teste in n lanci della moneta è $\Omega(\lg n)$. Per provare questo limite, cerchiamo le serie di lunghezza s suddividendo gli n lanci in circa n/s gruppi, ciascuno di s lanci. Se scegliamo $s = \lfloor (\lg n)/2 \rfloor$, possiamo dimostrare che è probabile che almeno uno di questi gruppi abbia soltanto teste e, quindi, è probabile che la serie più lunga abbia una lunghezza almeno pari a $s = \Omega(\lg n)$. Poi dimostreremo che la serie più lunga ha una lunghezza prevista pari a $\Omega(\lg n)$.

Suddividiamo gli n lanci della moneta in almeno $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ gruppi di $\lfloor (\lg n)/2 \rfloor$ lanci consecutivi e limitiamo la probabilità che nessun gruppo possa avere soltanto teste. Per l'equazione (5.9), la probabilità che il gruppo che inizia nella posizione i abbia soltanto teste è

$$\begin{aligned} \Pr\{A_{i,\lfloor (\lg n)/2 \rfloor}\} &= 1/2^{\lfloor (\lg n)/2 \rfloor} \\ &\geq 1/\sqrt{n} \end{aligned}$$

La probabilità che una serie di teste di lunghezza almeno pari a $\lfloor (\lg n)/2 \rfloor$ non inizi nella posizione i è, quindi, al massimo $1 - 1/\sqrt{n}$. Poiché $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ gruppi sono formati da lanci indipendenti, che si escludono a vicenda, la probabilità che ciascuno di questi gruppi *non riesca* a essere una serie di lunghezza $\lfloor (\lg n)/2 \rfloor$ è al massimo

$$\begin{aligned} (1 - 1/\sqrt{n})^{\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor} &\leq (1 - 1/\sqrt{n})^{n/\lfloor (\lg n)/2 \rfloor - 1} \\ &\leq (1 - 1/\sqrt{n})^{2n/\lg n - 1} \\ &\leq e^{-(2n/\lg n - 1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n) \end{aligned}$$

Qui abbiamo utilizzato la disuguaglianza (3.11), $1 + x \leq e^x$, e ci siamo basati sul fatto (che dovreste verificare) che $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ per valori di n sufficientemente grandi. Ne consegue che la probabilità che la serie più lunga superi $\lfloor (\lg n)/2 \rfloor$ è

$$\sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \Pr\{L_j\} \geq 1 - O(1/n) \tag{5.12}$$

Adesso possiamo calcolare un limite inferiore sulla lunghezza prevista della serie più lunga, iniziando con l'equazione (5.11) e procedendo in modo analogo all'analisi del limite superiore:

$$\begin{aligned} \mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\ &= \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} j \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} j \Pr\{L_j\} \\ &\geq \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \lfloor (\lg n)/2 \rfloor \Pr\{L_j\} \end{aligned}$$

$$
\begin{aligned}
&= 0 \cdot \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor} \Pr\{L_j\} + \lfloor (\lg n)/2 \rfloor \sum_{j=\lfloor (\lg n)/2 \rfloor + 1}^{n} \Pr\{L_j\} \\
&\geq 0 + \lfloor (\lg n)/2 \rfloor \, (1 - O(1/n)) \qquad \text{(per la disuguaglianza (5.12))} \\
&= \Omega(\lg n)
\end{aligned}
$$

Analogamente al paradosso del compleanno, possiamo svolgere un'analisi più semplice, ma approssimata, utilizzando le variabili casuali indicatrici. Sia $X_{ik} = \mathrm{I}\{A_{ik}\}$ la variabile casuale indicatrice associata alla serie di teste di lunghezza almeno pari a k che inizia con l'i-esimo lancio della moneta. Per contare il numero totale di tali sequenze, definiamo

$$X = \sum_{i=1}^{n-k+1} X_{ik}$$

Prendendo i valori attesi da entrambi i membri dell'equazione e applicando la linearità del valore atteso, otteniamo

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1} 1/2^k \\
&= \frac{n-k+1}{2^k}
\end{aligned}
$$

Sostituendo i vari valori di k, possiamo calcolare il numero atteso di serie di lunghezza k. Se questo numero è grande (molto più grande di 1), allora si prevedono molte serie di lunghezza k e la probabilità che se ne verifichi una è alta. Se questo numero è piccolo (molto più piccolo di 1), allora si prevedono pochissime serie di lunghezza k e la probabilità che se ne verifichi una è bassa. Se $k = c \lg n$, per qualche costante positiva c, otteniamo

$$
\begin{aligned}
\mathrm{E}[X] &= \frac{n - c\lg n + 1}{2^{c \lg n}} \\
&= \frac{n - c\lg n + 1}{n^c} \\
&= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\
&= \Theta(1/n^{c-1})
\end{aligned}
$$

Se c è grande, il numero atteso di serie di lunghezza $c \lg n$ è molto piccolo e concludiamo che è poco probabile che tali serie si verifichino. D'altra parte, se $c < 1/2$, allora otteniamo $\mathrm{E}[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$ e prevediamo che ci sarà un numero elevato di serie di lunghezza $(1/2) \lg n$. Pertanto, è molto probabile che si verifichi una serie di tale lunghezza. Da queste stime approssimative possiamo concludere che la lunghezza prevista della serie più lunga è $\Theta(\lg n)$.

5.4.4 Il problema delle assunzioni online

Come esempio finale, consideriamo una variante del problema delle assunzioni. Supponiamo adesso di non incontrare tutti i candidati per selezionare il migliore. Inoltre, non intendiamo assumere e licenziare ogni volta che troviamo un candidato migliore del precedente; piuttosto, ci accontenteremo di un candidato che è vicino al migliore, col vantaggio di fare esattamente una sola assunzione. Bisogna rispettare una regola dell'azienda: dopo ogni colloquio, dobbiamo offrire immediatamente il lavoro al candidato o dobbiamo dirgli che non lo avrà. Qual è il compromesso fra minimizzare il numero di colloqui e massimizzare la qualità del candidato assunto?

Possiamo modellare questo problema nel modo seguente. Dopo avere incontrato un candidato, siamo in grado di assegnargli un punteggio. Indichiamo con $score(i)$ il punteggio dato all'i-esimo candidato; due candidati non possono avere lo stesso punteggio. Dopo avere incontrato j candidati, conosciamo il candidato con il punteggio più alto, ma non sappiamo se uno dei restanti $n-j$ candidati avrà un punteggio più alto. Decidiamo di adottare la strategia di scegliere un numero intero positivo $k < n$, incontrare e scartare i primi k candidati e, in seguito, assumere il primo candidato che ha un punteggio più alto di tutti i candidati precedenti. Se scopriamo che il candidato più qualificato era fra i primi k precedentemente incontrati, allora assumeremo l'n-esimo candidato. Questa strategia è formalizzata nella seguente procedura ON-LINE-MAXIMUM(k, n), che restituisce l'indice del candidato che intendiamo assumere.

```
ON-LINE-MAXIMUM(k, n)
1  scoremax ← −∞
2  for i ← 1 to k
3       do if score(i) > scoremax
4            then scoremax ← score(i)
5  for i ← k + 1 to n
6       do if score(i) > scoremax
7            then return i
8  return n
```

Intendiamo determinare, per ogni possibile valore di k, la probabilità che abbiamo di assumere il candidato più qualificato. Poi sceglieremo il miglior valore possibile di k e implementeremo la strategia con questo valore. Per il momento, supponiamo che k sia fisso. Indichiamo con $M(j) = \max_{1 \le i \le j} \{score(i)\}$ il punteggio massimo fra i candidati da 1 a j. Sia S l'evento in cui riusciamo a selezionare il candidato più qualificato; sia S_i l'evento in cui riusciamo a selezionare l'i-esimo candidato che abbiamo incontrato. Poiché i vari eventi S_i sono disgiunti, abbiamo $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Dal momento che non riusciremo mai a selezionare il candidato più qualificato quando questo è fra i primi k, abbiamo $\Pr\{S_i\} = 0$ per $i = 1, 2, \ldots, k$. Quindi, otteniamo

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} \tag{5.13}$$

Calcoliamo adesso $\Pr\{S_i\}$. Per riuscire a selezionare il candidato più qualificato quando questo è l'i-esimo candidato, devono accadere due cose. In primo luogo, il candidato più qualificato deve trovarsi nella posizione i, un evento che

indicheremo con B_i. In secondo luogo, l'algoritmo non deve selezionare nessun candidato nelle posizioni da $k+1$ a $i-1$; questo si verifica soltanto se, per ogni j tale che $k+1 \leq j \leq i-1$, si ha che $score(j) < scoremax$ nella riga 6 (poiché i punteggi sono unici, possiamo ignorare la possibilità che $score(j) = scoremax$). In altre parole, deve accadere che tutti i valori da $score(k+1)$ a $score(i-1)$ sono minori di $M(k)$; se, invece, un valore è maggiore di $M(k)$, sarà restituito l'indice del primo valore che è maggiore. Indichiamo con O_i l'evento in cui nessuno dei candidati nelle posizioni da $k+1$ a $i-1$ sia scelto. Fortunatamente, i due eventi B_i e O_i sono indipendenti. L'evento O_i dipende soltanto dall'ordinamento relativo dei valori nelle posizioni da 1 a $i-1$, mentre B_i dipende soltanto dal fatto che il valore nella posizione i sia maggiore dei valori in tutte le altre posizioni. L'ordinamento dei valori nelle posizioni da 1 a $i-1$ non dipende dal fatto che il valore nella posizione i sia maggiore di tutti questi valori e il valore nella posizione i non influisce sull'ordinamento dei valori nelle posizioni da 1 a $i-1$. Quindi, possiamo applicare l'equazione (C.15) per ottenere

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\}$$

La probabilità $\Pr\{B_i\}$ è chiaramente $1/n$, in quanto il valore massimo ha uguale probabilità di trovarsi in una qualsiasi delle n posizioni. Affinché possa verificarsi l'evento O_i, il valore massimo delle posizioni da 1 a $i-1$ deve trovarsi in una delle prime k posizioni ed ha uguale probabilità di trovarsi in una di queste $i-1$ posizioni. Di conseguenza, $\Pr\{O_i\} = k/(i-1)$ e $\Pr\{S_i\} = k/(n(i-1))$. Applicando l'equazione (5.13), otteniamo

$$\begin{aligned}
\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\
&= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}
\end{aligned}$$

Approssimiamo con gli integrali per limitare questa sommatoria da sopra e da sotto. Per la disuguaglianza (A.12), abbiamo

$$\int_k^n \frac{1}{x}\,dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x}\,dx$$

Calcolando questi integrali definiti, otteniamo

$$\frac{k}{n}(\ln n - \ln k) \leq \Pr\{S\} \leq \frac{k}{n}(\ln(n-1) - \ln(k-1))$$

che rappresentano dei limiti abbastanza stretti per $\Pr\{S\}$. Poiché intendiamo massimizzare la nostra probabilità di successo, concentriamo l'attenzione sulla scelta del valore k che massimizza il limite inferiore su $\Pr\{S\}$ (l'espressione del limite inferiore è più semplice da massimizzare dell'espressione del limite superiore). Derivando l'espressione $(k/n)(\ln n - \ln k)$ rispetto a k, otteniamo

$$\frac{1}{n}(\ln n - \ln k - 1)$$

Ponendo questa derivata uguale a 0, notiamo che il limite inferiore sulla probabilità è massimizzato quando $\ln k = \ln n - 1 = \ln(n/e)$ ovvero quando $k = n/e$. Quindi, se implementiamo la nostra strategia con $k = n/e$, riusciremo ad assumere il candidato più qualificato con una probabilità pari almeno a $1/e$.

Esercizi

5.4-1

Quante persone devono trovarsi in una stanza prima che ci sia una probabilità almeno pari a $1/2$ che una di esse sia nata nel vostro stesso giorno dell'anno? Quante persone devono esserci prima che la probabilità che almeno due persone festeggino il compleanno il 4 luglio sia maggiore di $1/2$?

5.4-2

Supponete che delle palline siano lanciate in b contenitori. Ogni lancio è indipendente e ogni pallina ha la stessa probabilità di finire in qualsiasi contenitore. Qual è il numero atteso di lanci prima che almeno in un contenitore ci siano due palline?

5.4-3 $\star$

Per l'analisi del paradosso del compleanno, è importante che i compleanni siano mutuamente indipendenti o è sufficiente che siano indipendenti a coppie? Spiegate la vostra risposta.

5.4-4 $\star$

Quante persone dovreste invitare a una festa per avere la probabilità di trovarne *tre* nate nello stesso giorno dell'anno?

5.4-5 $\star$

Qual è la probabilità che una stringa-k su un insieme di dimensione n sia effettivamente una permutazione-k? Qual è la relazione fra questa domanda e il paradosso del compleanno?

5.4-6 $\star$

Supponete che n palline siano lanciate in n contenitori; ogni lancio è indipendente e una pallina ha la stessa probabilità di finire in qualsiasi contenitore. Qual è il numero atteso di contenitori vuoti? Qual è il numero atteso di contenitori con una sola pallina?

5.4-7 $\star$

Affinate il limite inferiore sulla lunghezza di una serie dello stesso evento dimostrando che, in n lanci di una moneta, la probabilità che nessuna serie più lunga di $\lg n - 2\lg\lg n$ teste consecutive possa verificarsi è minore di $1/n$.

5.5 Problemi

5-1 Contatore probabilistico

Con un contatore ordinario a b bit, di solito è possibile contare fino a $2^b - 1$. Con il ***contatore probabilistico*** di R. Morris è possibile contare fino a un valore molto più grande, perdendo un po' di precisione.

Indichiamo con i il valore del contatore che rappresenta il conteggio di n_i per $i = 0, 1, \ldots, 2^b - 1$, dove n_i forma una sequenza crescente di valori non negativi. Supponiamo che il valore iniziale del contatore sia 0, che rappresenta il conteggio di $n_0 = 0$. La procedura INCREMENT usa un contatore che calcola il valore i in modo probabilistico. Se $i = 2^b - 1$, allora viene segnalato un errore di overflow, altrimenti il contatore viene incrementato di 1 con probabilità $1/(n_{i+1} - n_i)$ e rimane inalterato con probabilità $1 - 1/(n_{i+1} - n_i)$.

Se scegliamo $n_i = i$ per ogni $i \geq 0$, allora il contatore è ordinario. Situazioni più interessanti si presentano se scegliamo, per esempio, $n_i = 2^{i-1}$ per $i > 0$ o $n_i = F_i$ (l'i-esimo numero di Fibonacci – vedere il Paragrafo 3.2).

Per questo problema, supponiamo che n_{2^b-1} sia sufficientemente grande da poter trascurare la probabilità di un errore di overflow.

a. Dimostrate che il valore atteso rappresentato dal contatore dopo n operazioni di INCREMENT è esattamente pari a n.

b. L'analisi della varianza del valore rappresentato dal contatore dipende dalla sequenza di n_i. Consideriamo un semplice caso: $n_i = 100i$ per ogni $i \geq 0$. Calcolate la varianza nel valore rappresentato dal registro dopo n operazioni di INCREMENT.

5-2 Ricerca in un array non ordinato

Questo problema esamina tre algoritmi che ricercano un valore x in un array A di n elementi non ordinati.

Consideriamo la seguente strategia di randomizzazione: scegliamo un indice casuale i in A. Se $A[i] = x$, terminiamo la ricerca, altrimenti continuiamo a cercare scegliendo un nuovo indice casuale in A. Continuiamo a scegliere indici casuali in A finché non avremo trovato un indice j tale che $A[j] = x$ o finché non avremo esaminato tutti gli elementi di A. Notate che l'indice ogni volta viene scelto dall'intero insieme degli indici, quindi lo stesso elemento potrebbe essere esaminato più volte.

a. Scrivete la pseudocodifica per una procedura RANDOM-SEARCH che implementa la precedente strategia. Verificate che il vostro algoritmo termini dopo che sono stati selezionati tutti gli indici in A.

b. Supponete che ci sia un solo indice i tale che $A[i] = x$. Qual è il numero atteso di indici in A che devono essere selezionati prima che x venga trovato e la procedura RANDOM-SEARCH termini?

c. Generalizzando la soluzione che avete ottenuto nel punto (b), supponete che ci siano $k \geq 1$ indici i tali che $A[i] = x$. Qual è il numero atteso di indici in A che devono essere selezionati prima che x venga trovato e la procedura RANDOM-SEARCH termini? La risposta dovrebbe essere una funzione di n e k.

d. Supponete che non ci siano indici i tali che $A[i] = x$. Qual è il numero atteso di indici in A che devono essere selezionati prima che tutti gli elementi di A siano stati esaminati e la procedura RANDOM-SEARCH termini?

Consideriamo ora un algoritmo di ricerca lineare deterministica, che chiameremo DETERMINISTIC-SEARCH. L'algoritmo ricerca x in A, esaminando ordinatamente $A[1], A[2], A[3], \ldots, A[n]$ finché trova $A[i] = x$ o raggiunge la fine dell'array. Supponete che tutte le possibili permutazioni dell'array di input siano ugualmente probabili.

e. Supponete che ci sia un solo indice i tale che $A[i] = x$. Qual è il tempo di esecuzione previsto di DETERMINISTIC-SEARCH? Qual è il tempo di esecuzione nel caso peggiore di DETERMINISTIC-SEARCH?

f. Generalizzando la soluzione che ottenuto nel punto (e), supponete che ci siano $k \geq 1$ indici i tali che $A[i] = x$. Qual è il tempo di esecuzione previsto di DETERMINISTIC-SEARCH? Qual è il tempo di esecuzione nel caso peggiore di DETERMINISTIC-SEARCH? La risposta dovrebbe essere una funzione di n e k.

g. Supponete che non ci siano indici i tali che $A[i] = x$. Qual è il tempo di esecuzione previsto di DETERMINISTIC-SEARCH? Qual è il tempo di esecuzione nel caso peggiore di DETERMINISTIC-SEARCH?

Infine, consideriamo l'algoritmo randomizzato SCRAMBLE-SEARCH che opera in questo modo: prima permuta casualmente l'array di input e poi svolge la ricerca lineare deterministica (precedentemente descritta) nell'array permutato.

h. Indicando con k il numero di indici i tali che $A[i] = x$, determinate il caso peggiore e i tempi di esecuzione previsti di SCRAMBLE-SEARCH nei casi in cui $k = 0$ e $k = 1$. Generalizzate la vostra soluzione per includere il caso in cui $k \geq 1$.

i. Quale di questi tre algoritmi di ricerca utilizzereste? Perché?

Note

Bollobás [44], Hofri [151] e Spencer [283] descrivono numerose tecniche avanzate di analisi probabilistica. I vantaggi degli algoritmi randomizzati sono stati analizzati e classificati da Karp [174] e Rabin [253]. Il libro di testo di Motwani e Raghavan [228] tratta diffusamente gli algoritmi randomizzati.

Sono state studiate numerose varianti del problema delle assunzioni. Questo tipo di problema è riportato come "secretary problem". Uno studio in quest'area è stato fatto da Ajtai, Meggido e Waarts [12].

II Ordinamento e statistiche d'ordine

Introduzione

Questa parte presenta vari algoritmi che risolvono il seguente ***problema di ordinamento***:

Input: una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: una permutazione (riarrangiamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

La sequenza di input di solito è un array di n elementi, anche se può essere rappresentata in qualche altra forma, come una lista.

La struttura dei dati

In pratica, i numeri da ordinare raramente sono valori isolati. Ogni numero di solito appartiene a una raccolta di dati detta ***record***. Ogni record contiene una ***chiave***, che è il valore da ordinare; la parte restante del record è composta da ***dati satelliti***, che di solito si muovono con la chiave. In pratica, quando un algoritmo di ordinamento permuta le chiavi, deve permutare anche i dati satelliti. Se ogni record include una grande quantità di dati satellite, spesso permutiamo un array di puntatori ai record, anziché i record stessi, per minimizzare gli spostamenti dei dati.

In un certo senso, sono questi dettagli implementativi che distinguono un algoritmo da un programma vero e proprio. Ordinare singoli numeri o grandi record che contengono numeri non è rivelante per il *metodo* in base al quale una procedura di ordinamento determina l'ordine degli elementi. Pertanto, quando analizziamo un problema di ordinamento, tipicamente supponiamo che l'input sia composto soltanto da numeri. La conversione di un algoritmo che ordina numeri in un programma che ordina record è concettualmente semplice, sebbene in determinati casi alcuni dettagli tecnici potrebbero complicare la programmazione effettiva.

Perché ordinare?

Molti esperti di informatica considerano l'ordinamento come il problema fondamentale nello studio degli algoritmi. Le ragioni sono diverse:

- A volte l'esigenza di ordinare le informazioni è inerente a un'applicazione. Per esempio, per preparare l'estratto conto dei clienti, le banche hanno bisogno di ordinare gli assegni in base ai numeri.
- Gli algoritmi spesso usano l'ordinamento come una subroutine chiave. Per esempio, un programma che visualizza sullo schermo degli oggetti grafici che si sovrappongono uno sull'altro potrebbe avere bisogno di ordinare gli oggetti in base a una determinata relazione di "precedenza di rappresentazione", in modo che possa disegnare gli oggetti dall'ultimo al primo. In questo libro esamineremo molti algoritmi che usano l'ordinamento come subroutine.
- Esistono vari tipi di algoritmi di ordinamento, che impiegano un'ampia gamma di tecniche. In effetti, molte delle principali tecniche utilizzate nella progettazione degli algoritmi vengono rappresentate nel corpo degli algoritmi di ordinamento che sono stati sviluppati nel corso degli anni. In questo modo, l'ordinamento diventa anche un problema di interesse storico.
- L'ordinamento è un problema per il quale possiamo dimostrare un limite inferiore significativo (come vedremo nel Capitolo 8). I nostri migliori limiti superiori corrispondono asintoticamente al limite inferiore, quindi sappiamo che i nostri algoritmi di ordinamento sono asintoticamente ottimali. Inoltre, possiamo utilizzare il limite inferiore dell'ordinamento per dimostrare i limiti inferiori per certi altri problemi.
- Molti problemi di progettazione emergono durante l'implementazione degli algoritmi di ordinamento. Il più veloce programma di ordinamento per una particolare situazione può dipendere da vari fattori, come la conoscenza anticipata delle chiavi e dei dati satellite, la gerarchia della memoria (cache e memoria virtuale) del computer host e l'ambiente software. Molti di questi problemi si risolvono meglio a livello degli algoritmi, anziché "aggiustando" il codice.

Algoritmi di ordinamento

Nel Capitolo 2 abbiamo introdotto due algoritmi che ordinano n numeri reali. L'algoritmo insertion sort richiede un tempo $\Theta(n^2)$ nel caso peggiore. Tuttavia, poiché i suoi cicli interni sono compatti, è un algoritmo rapido di ordinamento sul posto per input di piccole dimensioni (ricordiamo che un algoritmo ordina ***sul posto*** soltanto se un numero costante di elementi dell'array di input sono sempre registrati all'esterno dell'array). L'algoritmo merge sort ha un tempo di esecuzione asintotico migliore, $\Theta(n \lg n)$, ma la procedura MERGE che usa non opera sul posto.

In questa parte, introdurremo altri due algoritmi che ordinano numeri reali arbitrari. Heapsort, presentato nel Capitolo 6, ordina n numeri sul posto nel tempo $O(n \lg n)$; usa un'importante struttura dati, detta heap, con la quale è possibile implementare anche una coda di priorità.

Anche quicksort, presentato nel Capitolo 7, ordina n numeri sul posto, ma il suo tempo di esecuzione nel caso peggiore è $\Theta(n^2)$. Il suo tempo di esecuzione nel caso medio, però, è $\Theta(n \lg n)$ e, in generale, è migliore di heapsort. Come insertion sort, anche quicksort ha un codice compatto, quindi il fattore costante nascosto nel suo tempo di esecuzione è piccolo. È un algoritmo popolare per ordinare grandi array di input.

Insertion sort, merge sort, heapsort e quicksort sono tutti ordinamenti per confronti: determinano l'ordinamento di un array di input confrontando gli elementi. Il Capitolo 8 inizia presentando il modello dell'albero di decisione che consente di studiare le limitazioni delle prestazioni degli ordinamenti per confronti. Applicando questo modello, dimostreremo un limite inferiore pari a $\Omega(n \lg n)$ sul tempo di esecuzione nel caso peggiore di qualsiasi ordinamento per confronti di n input, dimostrando quindi che heapsort e merge sort sono ordinamenti per confronti asintoticamente ottimali.

Il Capitolo 8 continua dimostrando che possiamo battere il limite inferiore $\Omega(n \lg n)$, se riusciamo ad acquisire informazioni sull'ordinamento dell'input con un metodo che non è basato sui confronti degli elementi. L'algoritmo counting sort, per esempio, suppone che i numeri di input appartengano all'insieme $\{1, 2, \ldots, k\}$. Utilizzando l'indicizzazione dell'array come strumento per determinare l'ordine relativo degli elementi, counting sort può ordinare n numeri nel tempo $\Theta(k + n)$. Quindi, se $k = O(n)$, counting sort viene eseguito in un tempo che è lineare nella dimensione dell'array di input. È possibile utilizzare un algoritmo correlato, radix sort, per estendere il campo di applicazione di counting sort. Se ci sono n interi da ordinare, ogni intero ha d cifre e ogni cifra appartiene all'insieme $\{1, 2, \ldots, k\}$, allora radix sort può ordinare i numeri nel tempo $\Theta(d(n + k))$. Se d è una costante e k è $O(n)$, radix sort viene eseguito in tempo lineare. Un terzo algoritmo, bucket sort, richiede la conoscenza della distribuzione probabilistica dei numeri nell'array di input. Può ordinare n numeri reali uniformemente distribuiti nell'intervallo semiaperto $[0, 1)$ nel tempo $O(n)$ nel caso medio.

Statistiche d'ordine

L'i-esima statistica d'ordine di un insieme di n numeri è l'i-esimo numero più piccolo dell'insieme. Ovviamente, è possibile selezionare l'i-esima statistica d'ordine ordinando l'input e indicizzando l'i-esimo elemento dell'output. Se non si fanno ipotesi sulla distribuzione dell'input, questo metodo viene eseguito nel tempo $\Omega(n \lg n)$, come dimostra il limite inferiore provato nel Capitolo 8.

Nel Capitolo 9 dimostreremo che è possibile trovare l'i-esimo elemento più piccolo nel tempo $O(n)$, anche quando gli elementi sono numeri reali arbitrari. Presenteremo un algoritmo con uno pseudocodice compatto che viene eseguito nel tempo $\Theta(n^2)$ nel caso peggiore, ma in tempo lineare nel caso medio. Descriveremo anche un algoritmo più complicato che viene eseguito nel tempo $O(n)$ nel caso peggiore.

Background

Sebbene molti degli argomenti trattati in questa parte non richiedano calcoli matematici difficili, tuttavia in alcuni casi sono necessarie elaborazioni matematiche sofisticate. In particolare, l'analisi del caso medio di quicksort, bucket sort e l'algoritmo delle statistiche d'ordine applicano la teoria della probabilità, che è trattata nell'Appendice C, i concetti di analisi probabilistica e gli algoritmi randomizzati descritti nel Capitolo 5. L'analisi dell'algoritmo in tempo lineare nel caso peggiore per le statistiche d'ordine richiede passaggi matematici ancora più sofisticati delle altre analisi del caso peggiore che sono svolte in questa parte del libro.

6 Heapsort

In questo capitolo presentiamo un altro algoritmo di ordinamento. Come merge sort, ma diversamente da insertion sort, il tempo di esecuzione di heapsort è $O(n \lg n)$. Come insertion sort, ma diversamente da merge sort, heapsort effettua un ordinamento sul posto: soltanto un numero costante di elementi dell'array sono memorizzati all'esterno dell'array di input in un istante qualsiasi. Quindi, heapsort combina i migliori attributi dei due algoritmi di ordinamento che abbiamo già descritto.

Heapsort introduce anche un'altra tecnica di progettazione degli algoritmi: l'uso di una struttura dati, che in questo caso è detta "heap", per gestire le informazioni durante l'esecuzione dell'algoritmo. Non soltanto la struttura heap è utile per l'heapsort, ma crea anche un'efficiente coda di priorità. La struttura heap riapparirà negli algoritmi analizzati nei capitoli successivi.

Originariamente, il termine "heap" fu coniato nel contesto dell'heapsort, ma da allora è stato utilizzato per fare riferimento ai meccanismi automatici di recupero della memoria detti "garbage collector" (spazzini), come quelli forniti dai linguaggi di programmazione Lisp e Java. La nostra struttura heap *non* è un meccanismo garbage collector e, ogni volta che faremo riferimento agli heap in questo libro, intenderemo la struttura definita in questo capitolo.

6.1 Heap

Un ***heap (binario)*** è una struttura dati composta da un array che possiamo considerare come un albero binario quasi completo (vedere il Paragrafo B.5.3), come illustra la Figura 6.1. Ogni nodo dell'albero corrisponde a un elemento dell'array che memorizza il valore del nodo. Tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo che può essere riempito da sinistra fino a un certo punto. Un array A che rappresenta un heap è un oggetto con due attributi: $lunghezza[A]$ indica il numero di elementi nell'array; $heap\text{-}size[A]$ indica il numero degli elementi dell'heap che sono registrati nell'array A. Cioè, anche se $A[1\,..\,lunghezza[A]]$ contiene numeri validi, nessun elemento dopo $A[heap\text{-}size[A]]$, dove $heap\text{-}size[A] \leq lunghezza[A]$, è un elemento dell'heap. La radice dell'albero è $A[1]$. Se i è l'indice di un nodo, gli indici di suo padre PARENT(i), del figlio sinistro LEFT(i) e del figlio destro RIGHT(i) possono essere facilmente calcolati:

PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

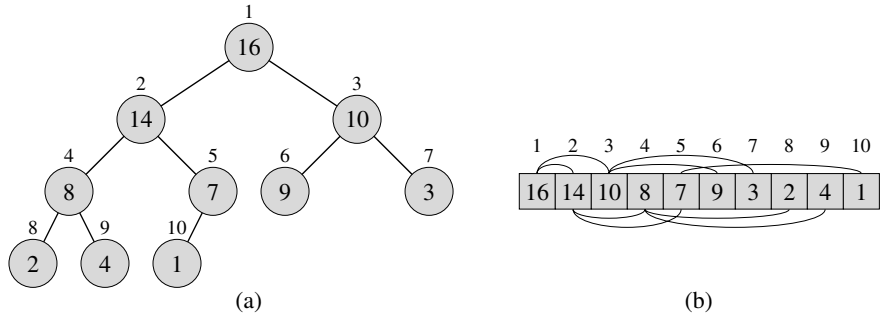


Figura 6.1 Un max-heap visto come **(a)** un albero binario e **(b)** un array. Il numero all'interno del cerchio di un nodo dell'albero è il valore registrato in quel nodo. Il numero sopra un nodo è il corrispondente indice dell'array. Sopra e sotto l'array ci sono delle linee che rappresentano le relazioni padre-figlio; i padri sono sempre a sinistra dei loro figli. L'albero ha altezza 3; il nodo con indice 4 e valore 8 ha altezza 1.

RIGHT(i)
  return $2i + 1$

Nella maggior parte dei calcolatori, la procedura LEFT può calcolare $2i$ con una sola istruzione, facendo scorrere semplicemente di una posizione a sinistra la rappresentazione binaria di i. Analogamente, la procedura RIGHT può rapidamente calcolare $2i + 1$, facendo scorrere la rappresentazione binaria di i di una posizione a sinistra e aggiungendo 1 come bit meno significativo. La procedura PARENT può calcolare $\lfloor i/2 \rfloor$ con uno scorrimento di una posizione a destra della rappresentazione di i. In una buona implementazione di heapsort, queste tre procedure sono spesso realizzate come "macro" o procedure "in linea".

Ci sono due tipi di heap binari: max-heap e min-heap. In entrambi i tipi, i valori nei nodi soddisfano una ***proprietà dell'heap***, le cui caratteristiche dipendono dal tipo di heap. In un ***max-heap***, la ***proprietà del max-heap*** è che per ogni nodo i diverso dalla radice, si ha

$$A[\text{PARENT}(i)] \geq A[i]$$

ovvero il valore di un nodo è al massimo il valore di suo padre. Quindi, l'elemento più grande di un max-heap è memorizzato nella radice e il sottoalbero di un nodo contiene valori non maggiori di quello contenuto nel nodo stesso. Un ***min-heap*** è organizzato nel modo opposto; la ***proprietà del min-heap*** è che per ogni nodo i diverso dalla radice, si ha

$$A[\text{PARENT}(i)] \leq A[i]$$

Il più piccolo elemento in un min-heap è nella radice.

Per l'algoritmo heapsort utilizzeremo i max-heap. I min-heap sono di solito utilizzati nelle code di priorità, che descriveremo nel Paragrafo 6.5. Dovremo specificare con precisione se abbiamo bisogno di un max-heap o di un min-heap per una particolare applicazione; quando le proprietà si applicano sia ai max-heap sia ai min-heap, utilizzeremo semplicemente il termine "heap".

Se osserviamo un heap nella forma di albero, definiamo ***altezza di un nodo*** il numero di archi nel cammino semplice più lungo che dal nodo scende fino a una foglia. Definiamo ***altezza di un heap*** l'altezza della sua radice. Poiché un heap di n elementi è basato su un albero binario completo, la sua altezza è $\Theta(\lg n)$ (vedere l'Esercizio 6.1-2). Vedremo che le operazioni fondamentali sugli heap vengono eseguite in un tempo che è al massimo proporzionale all'altezza dell'albero e, quindi, richiedono un tempo $O(\lg n)$. La parte restante di questo capitolo presenta alcune procedure di base e mostra come sono utilizzate in un algoritmo di ordinamento e in una coda di priorità.

- La procedura MAX-HEAPIFY, che è eseguita nel tempo $O(\lg n)$, è la chiave per conservare la proprietà del max-heap.
- La procedura BUILD-MAX-HEAP, che è eseguita in tempo lineare, genera un max-heap da un array di input non ordinato.
- La procedura HEAPSORT, che è eseguita nel tempo $O(n \lg n)$, ordina sul posto un array.
- Le quattro procedure MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY e HEAP-MAXIMUM, che sono eseguite nel tempo $O(\lg n)$, consentono a un heap di essere utilizzato come una coda di priorità.

Esercizi

6.1-1
Quali sono il numero minimo e il numero massimo di elementi in un heap di altezza h?

6.1-2
Dimostrate che un heap di n elementi ha altezza $\lfloor \lg n \rfloor$.

6.1-3
Dimostrate che in qualsiasi sottoalbero di un max-heap, la radice del sottoalbero contiene il valore più grande che si trova in qualsiasi punto del sottoalbero.

6.1-4
Dove potrebbe risiedere in un max-heap il più piccolo elemento, supponendo che tutti gli elementi siano distinti?

6.1-5
Un array ordinato è un min-heap?

6.1-6
La sequenza $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ è un max-heap?

6.1-7
Dimostrate che, nell'albero che rappresenta un heap di n elementi, le foglie sono i nodi con indici $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Conservare la proprietà dell'heap

MAX-HEAPIFY è un'importante subroutine per manipolare i max-heap. I suoi input sono un array A e un indice i dell'array. Quando viene chiamata questa subroutine, si suppone che gli alberi binari con radici in LEFT(i) e RIGHT(i) siano max-heap, ma che $A[i]$ possa essere più piccolo dei suoi figli, violando così la proprietà del max-heap. La funzione di MAX-HEAPIFY è consentire al valore $A[i]$ di "scendere" nel max-heap in modo che il sottoalbero con radice di indice i diventi un max-heap.

La Figura 6.2 illustra l'azione di MAX-HEAPIFY. A ogni passaggio, viene determinato il più grande degli elementi $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$; il suo indice viene memorizzato in $massimo$. Se $A[i]$ è più grande, allora il sottoalbero con radice nel nodo i è un max-heap e la procedura termina. Altrimenti, uno dei due figli ha l'elemento più grande e $A[i]$ viene scambiato con $A[massimo]$; in questo modo, il nodo i e i suoi figli soddisfano la proprietà del max-heap. Il nodo con indice $massimo$, però, adesso ha il valore originale $A[i]$ e, quindi, il

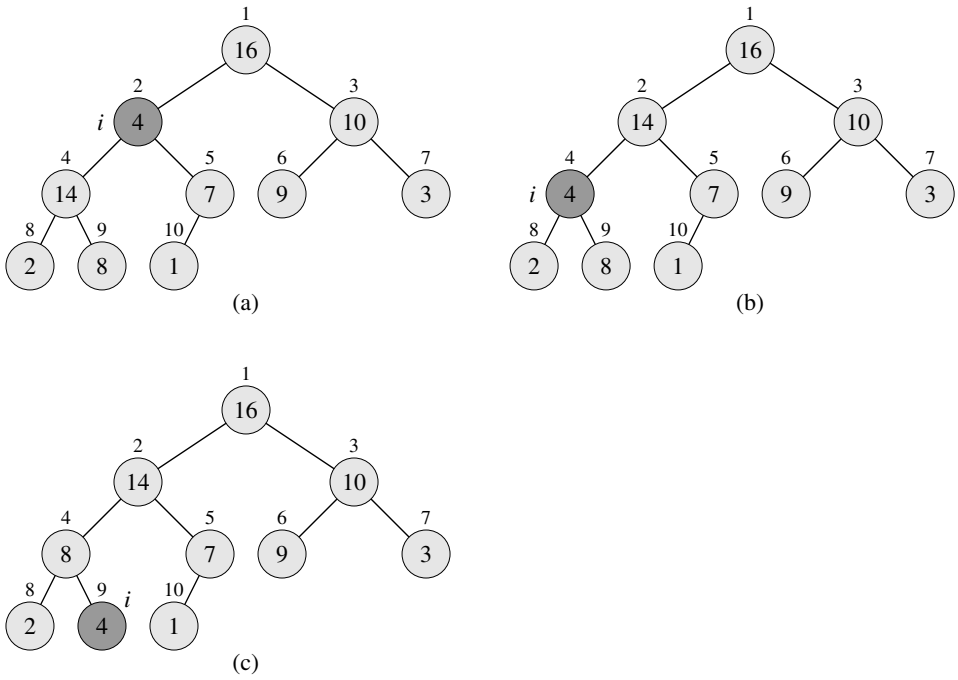


Figura 6.2 L'azione di MAX-HEAPIFY$(A, 2)$, dove $heap\text{-}size[A] = 10$. **(a)** La configurazione iniziale con $A[2]$ nel nodo $i = 2$ che viola la proprietà del max-heap, in quanto non è più grande di entrambi i figli. La proprietà del max-heap viene ripristinata nel nodo 2 in **(b)** scambiando $A[2]$ con $A[4]$; ma questo distrugge la proprietà del max-heap nel nodo 4. La chiamata ricorsiva MAX-HEAPIFY$(A, 4)$ adesso ha $i = 4$. Dopo avere scambiato $A[4]$ con $A[9]$, come illustra **(c)**, il nodo 4 è sistemato e la chiamata ricorsiva MAX-HEAPIFY$(A, 9)$ non apporta ulteriori modifiche alla struttura dati.

sottoalbero con radice in $massimo$ potrebbe violare la proprietà del max-heap. Di conseguenza, deve essere chiamata ricorsivamente la subroutine MAX-HEAPIFY per questo sottoalbero.

MAX-HEAPIFY(A, i)

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4     then massimo ← l
5     else massimo ← i
6  if r ≤ heap-size[A] and A[r] > A[massimo]
7     then massimo ← r
8  if massimo ≠ i
9     then scambia A[i] ↔ A[massimo]
10          MAX-HEAPIFY(A, massimo)
```

Il tempo di esecuzione di MAX-HEAPIFY in un sottoalbero di dimensione n con radice in un nodo i è pari al tempo $\Theta(1)$ per sistemare le relazioni fra gli elementi $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del nodo i. I sottoalberi dei figli hanno ciascuno una dimensione che non supera $2n/3$ – il caso peggiore si verifica quando l'ultima riga dell'albero è piena esattamente a metà – e il tempo di esecuzione di MAX-HEAPIFY può quindi essere descritto dalla ricorrenza

$$T(n) \leq T(2n/3) + \Theta(1)$$

La soluzione di questa ricorrenza, per il caso 2 del teorema dell'esperto (Teorema 4.1), è $T(n) = O(\lg n)$. In alternativa, possiamo indicare con $O(h)$ il tempo di esecuzione di MAX-HEAPIFY in un nodo di altezza h.

Esercizi

6.2-1

Illustrate l'azione di MAX-HEAPIFY$(A, 3)$ sull'array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$, utilizzando la Figura 6.2 come modello.

6.2-2

Iniziando dalla procedura MAX-HEAPIFY, scrivete uno pseudocodice per la procedura MIN-HEAPIFY(A, i), che svolge le corrispondenti operazioni con un min-heap. Confrontate i tempi di esecuzione delle procedure MIN-HEAPIFY e MAX-HEAPIFY.

6.2-3

Qual è l'effetto di chiamare MAX-HEAPIFY(A, i) quando l'elemento $A[i]$ è maggiore dei suoi figli?

6.2-4

Qual è l'effetto di chiamare MAX-HEAPIFY(A, i) per $i > \mathit{heap\text{-}size}[A]/2$?

6.2-5

Il codice di MAX-HEAPIFY è molto efficiente in termini di fattori costanti, tranne eventualmente per la chiamata ricorsiva nella riga 10, che potrebbe indurre qualche compilatore a generare un codice inefficiente. Scrivete una procedura MAX-HEAPIFY efficiente che usa un costrutto di controllo iterativo (un ciclo), anziché la ricorsione.

6.2-6

Dimostrate che il tempo di esecuzione nel caso peggiore di MAX-HEAPIFY su un heap di dimensione n è $\Omega(\lg n)$ (*suggerimento:* per un heap con n nodi, assegnate i valori ai nodi in modo che MAX-HEAPIFY sia chiamata ricorsivamente in ogni nodo di un cammino che scende dalla radice fino a una foglia).

6.3 Costruire un heap

Possiamo utilizzare la procedura MAX-HEAPIFY dal basso verso l'alto (bottom-up) per convertire un array $A[1\,..\,n]$, con $n = \mathit{lunghezza}[A]$, in un max-heap. Per l'Esercizio 6.1-7, tutti gli elementi nel sottoarray $A[(\lfloor n/2 \rfloor + 1)\,..\,n]$ sono foglie dell'albero e quindi ciascuno è un heap di un elemento da cui iniziare. La procedura BUILD-MAX-HEAP attraversa i restanti nodi dell'albero ed esegue MAX-HEAPIFY in ciascuno di essi.

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← lunghezza[A]
2  for i ← ⌊lunghezza[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

La Figura 6.3 illustra un esempio dell'azione di BUILD-MAX-HEAP.

Per verificare che BUILD-MAX-HEAP funziona correttamente, utilizziamo la seguente invariante di ciclo:

> All'inizio di ogni iterazione del ciclo **for**, righe 2–3, ogni nodo $i + 1$, $i + 2, \ldots, n$ è la radice di un max-heap.

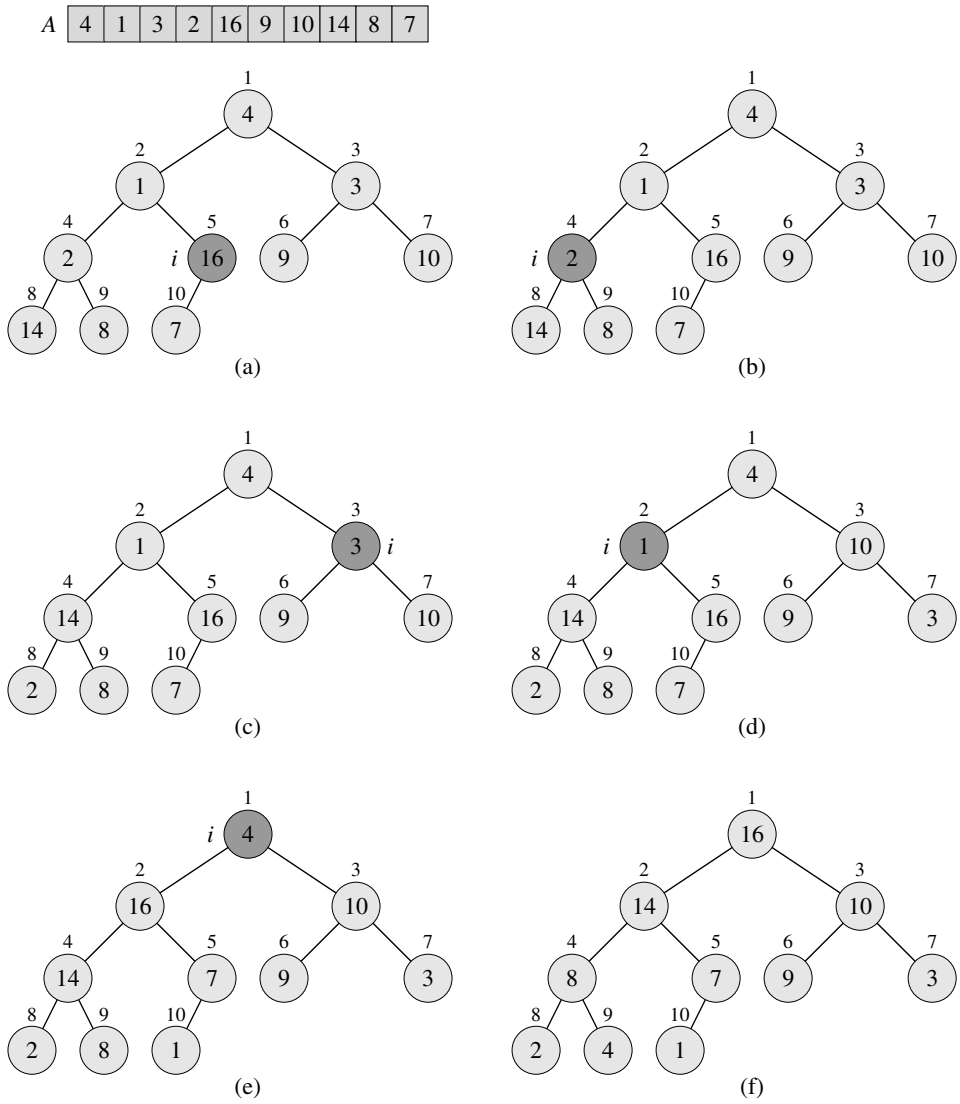


Figura 6.3 L'azione di BUILD-MAX-HEAP illustrata partendo dalla struttura dati prima della chiamata della procedura MAX-HEAPIFY nella riga 3 di BUILD-MAX-HEAP. **(a)** Un array di input di 10 elementi e l'albero binario che lo rappresenta. La figura mostra che l'indice i del ciclo fa riferimento al nodo 5 prima della chiamata della procedura MAX-HEAPIFY(A, i). **(b)** La struttura dati risultante. L'indice i del ciclo per la successiva iterazione fa riferimento al nodo 4. **(c)–(e)** Le successive iterazioni del ciclo **for** in BUILD-MAX-HEAP. Notate che ogni volta che viene chiamata la procedura MAX-HEAPIFY per un nodo, i due sottoalberi del nodo sono entrambi max-heap. **(f)** Il max-heap alla fine della procedura BUILD-MAX-HEAP.

Dobbiamo dimostrare che questa invariante è vera prima della prima iterazione del ciclo, che ogni iterazione del ciclo conserva l'invariante e che l'invariante fornisce un'utile proprietà per dimostrare la correttezza quando termina il ciclo.

Inizializzazione: prima della prima iterazione del ciclo, $i = \lfloor n/2 \rfloor$. Ogni nodo $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ è una foglia e, quindi, è la radice di un banale max-heap.

Conservazione: per verificare che ogni iterazione conserva l'invariante di ciclo, notiamo che i figli del nodo i hanno una numerazione più alta di i. Per l'invariante di ciclo, quindi, essi sono entrambi radici di max-heap. Questa è esattamente la condizione richiesta affinché la chiamata MAX-HEAPIFY(A, i) renda il nodo i la radice di un max-heap. Inoltre, la chiamata MAX-HEAPIFY preserva la proprietà che tutti i nodi $i + 1, i + 2, \ldots, n$ siano radici di max-heap. La diminuzione di i nell'aggiornamento del ciclo **for** ristabilisce l'invariante di ciclo per la successiva iterazione.

Conclusione: alla fine del ciclo, $i = 0$. Per l'invariante di ciclo, ogni nodo $1, 2, \ldots, n$ è la radice di un max-heap; in particolare, lo è il nodo 1.

Possiamo calcolare un semplice limite superiore sul tempo di esecuzione di BUILD-MAX-HEAP nel seguente modo. Ogni chiamata di MAX-HEAPIFY costa un tempo $O(\lg n)$ e ci sono $O(n)$ di queste chiamate. Quindi, il tempo di esecuzione è $O(n \lg n)$. Questo limite superiore, sebbene corretto, non è asintoticamente stretto. Possiamo ottenere un limite più stretto osservando che il tempo per eseguire MAX-HEAPIFY in un nodo varia con l'altezza del nodo nell'albero, e le altezze della maggior parte dei nodi sono piccole. L'analisi più rigorosa si basa sulle proprietà che un heap di n elementi ha un'altezza $\lfloor \lg n \rfloor$ (vedere l'Esercizio 6.1-2) e al massimo $\lceil n/2^{h+1} \rceil$ nodi di qualsiasi altezza h (vedere l'Esercizio 6.3-3).

Il tempo richiesto dalla procedura MAX-HEAPIFY quando viene chiamata per un nodo di altezza h è $O(h)$, quindi possiamo esprimere il costo totale di BUILD-MAX-HEAP come se fosse limitato dall'alto da

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

L'ultima sommatoria può essere calcolata ponendo $x = 1/2$ nella Formula (A.8):

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 \end{aligned}$$

Quindi, il tempo di esecuzione di BUILD-MAX-HEAP può essere limitato così

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) \end{aligned}$$

Dunque, possiamo costruire un max-heap da un array non ordinato in tempo lineare.

Possiamo costruire un min-heap utilizzando la procedura BUILD-MIN-HEAP, che è uguale a BUILD-MAX-HEAP, con la differenza che la chiamata di MAX-HEAPIFY nella riga 3 è sostituita dalla chiamata di MIN-HEAPIFY (vedere l'Esercizio 6.2-2). BUILD-MIN-HEAP produce un min-heap da un array lineare non ordinato in tempo lineare.

Esercizi

6.3-1

Utilizzando la Figura 6.3 come modello, illustrate l'azione di BUILD-MAX-HEAP sull'array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2

Perché l'indice i nella riga 2 del ciclo di BUILD-MAX-HEAP deve diminuire da $\lfloor lunghezza[A]/2 \rfloor$ a 1, anziché aumentare da 1 a $\lfloor lunghezza[A]/2 \rfloor$?

6.3-3

Dimostrate che ci sono al massimo $\lceil n/2^{h+1} \rceil$ nodi di altezza h in qualsiasi heap di n elementi.

6.4 L'algoritmo heapsort

L'algoritmo heapsort inizia utilizzando BUILD-MAX-HEAP per costruire un max-heap dell'array di input $A[1\,..\,n]$, dove $n = \mathit{lunghezza}[A]$. Poiché l'elemento più grande dell'array è memorizzato nella radice $A[1]$, esso può essere inserito nella sua posizione finale corretta scambiandolo con $A[n]$. Se adesso "scartiamo" il nodo n dall'heap (diminuendo $\mathit{heap\text{-}size}[A]$), notiamo che $A[1\,..\,(n-1)]$ può essere facilmente trasformato in un max-heap. I figli della radice restano max-heap, ma la nuova radice potrebbe violare la proprietà del max-heap. Per ripristinare questa proprietà, tuttavia, basta una chiamata di MAX-HEAPIFY$(A, 1)$, che lascia un max-heap in $A[1\,..\,(n-1)]$. L'algoritmo heapsort poi ripete questo processo per il max-heap di dimensione $n-1$ fino a un heap di dimensione 2 (vedere l'Esercizio 6.4-2 per una esatta invariante di ciclo).

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← lunghezza[A] downto 2
3       do scambia A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```

La Figura 6.4 illustra un esempio dell'operazione di heapsort dopo che il max-heap è stato costruito. Ogni max-heap è mostrato all'inizio di un'iterazione del ciclo **for** (righe 2–5).

La procedura HEAPSORT impiega un tempo $O(n \lg n)$, in quanto la chiamata di BUILD-MAX-HEAP impiega $O(n)$ e ciascuna delle $n-1$ chiamate di MAX-HEAPIFY impiega $O(\lg n)$.

Esercizi

6.4-1

Illustrate l'azione di HEAPSORT sull'array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$, utilizzando la Figura 6.4 come modello.

6.4-2

Dimostrate la correttezza di HEAPSORT utilizzando la seguente invariante di ciclo:

> All'inizio di ogni iterazione del ciclo **for**, righe 2–5, il sottoaray $A[1\,..\,i]$ è un max-heap che contiene gli (i) elementi più piccoli di $A[1\,..\,n]$ e il sottoarray $A[i+1\,..\,n]$ contiene gli $(n-i)$ elementi più grandi di $A[1\,..\,n]$, ordinati.

6.4-3

Qual è il tempo di esecuzione di heapsort con un array A di lunghezza n che è già ordinato in senso crescente? E se l'array è ordinato in senso decrescente?

6.4-4

Dimostrate che il tempo di esecuzione nel caso peggiore di heapsort è $\Omega(n \lg n)$.

6.4-5 $\star$

Dimostrate che, quando tutti gli elementi sono distinti, il tempo di esecuzione nel caso peggiore di heapsort è $\Omega(n \lg n)$.

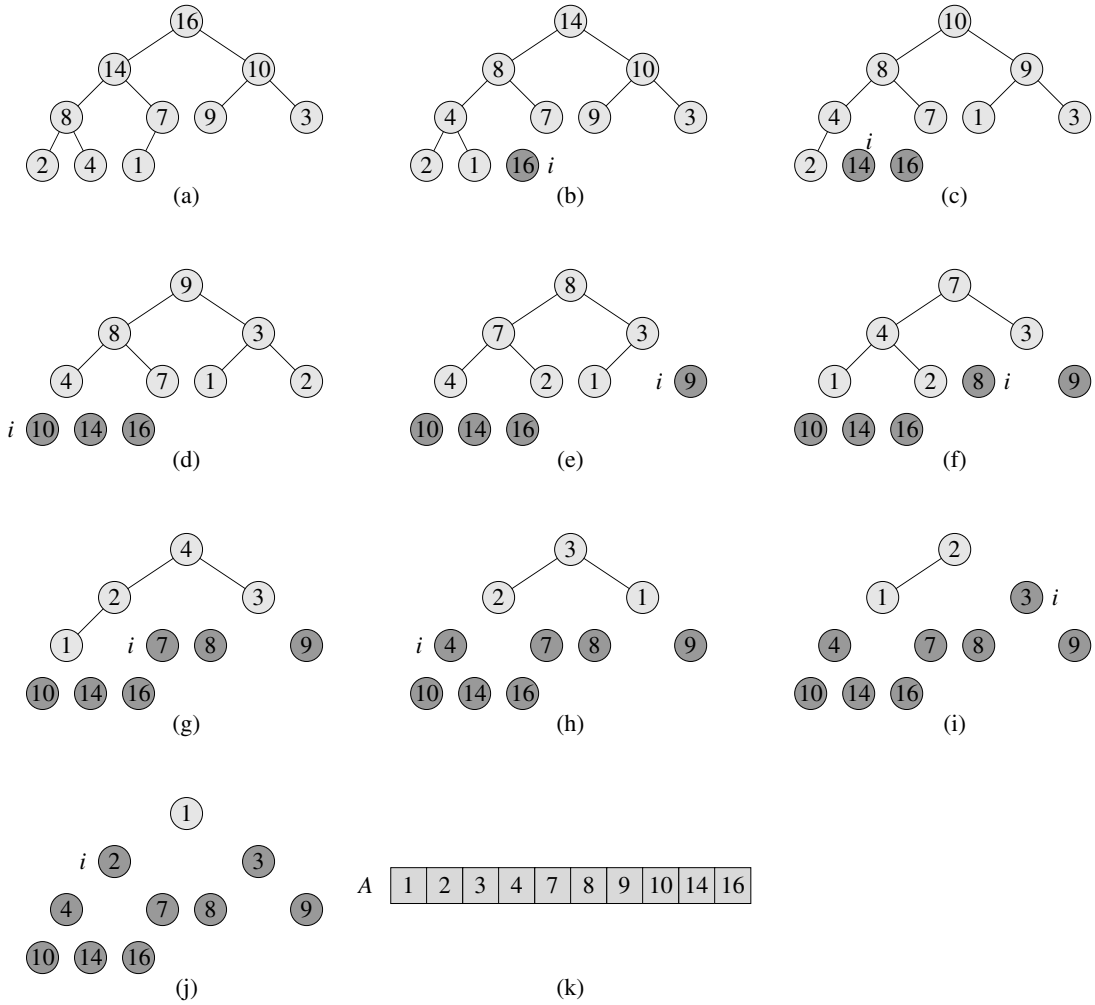


Figura 6.4 L'azione di HEAPSORT. **(a)** La struttura max-heap subito dopo essere stata costruita da BUILD-MAX-HEAP. **(b)–(j)** La struttura max-heap subito dopo una chiamata di MAX-HEAPIFY nella riga 5; è riportato il valore di i in tale istante. Soltanto i nodi più chiari restano nell'heap. **(k)** L'array A risultante è ordinato.

6.5 Code di priorità

Heapsort è un eccellente algoritmo, ma una buona implementazione di quicksort, che abbiamo presentato nel Capitolo 7, di solito batte heapsort. Nonostante ciò, la struttura dati heap ha un'enorme utilità. In questo paragrafo, presentiamo una delle applicazioni più diffuse dell'heap: un'efficiente coda di priorità. Analogamente agli heap, ci sono due tipi di code di priorità: code di max-priorità e code di min-priorità. Qui analizzeremo l'implementazione delle code di max-priorità che, a loro volta, si basano sui max-heap; l'Esercizio 6.5-3 chiede di scrivere le procedure per le code di min-priorità.

Una ***coda di priorità*** è una struttura dati che serve a mantenere un insieme S di elementi, ciascuno con un valore associato detto ***chiave***. Una ***coda di max-priorità*** supporta le seguenti operazioni.

INSERT(S, x) inserisce l'elemento x nell'insieme S. Questa operazione potrebbe essere scritta così: $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) restituisce l'elemento di S con la chiave più grande.

EXTRACT-MAX(S) elimina e restituisce l'elemento di S con la chiave più grande.

INCREASE-KEY(S, x, k) aumenta il valore della chiave dell'elemento x al nuovo valore k, che si suppone sia almeno pari al valore corrente della chiave dell'elemento x.

Un'applicazione delle code di max-priorità è quella di programmare i compiti su un computer condiviso. La coda di max-priority tiene traccia dei compiti da svolgere e delle loro relative priorità. Quando un compito è ultimato o interrotto, viene selezionato il compito con priorità più alta fra quelli in attesa mediante EXTRACT-MAX. Un nuovo compito può essere aggiunto alla coda in qualsiasi istante mediante INSERT.

In alternativa, una ***coda di min-priorità*** supporta le operazioni INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Una coda di min-priorità può essere utilizzata in un simulatore controllato da eventi. Gli elementi della coda sono gli eventi da simulare. Ogni elemento della coda è associato al tempo in cui l'evento si può verificare; questo tempo serve da chiave dell'evento. Gli eventi devono essere simulati secondo l'ordine dei loro tempi, in quanto la simulazione di un evento può causare la simulazione di altri eventi futuri. Il programma di simulazione usa EXTRACT-MIN a ogni passaggio per selezionare il successivo evento da simulare. Ogni volta che viene prodotto un nuovo evento, INSERT lo inserisce nella coda di min-priorità. Nei Capitoli 23 e 24 vedremo altre applicazioni delle code di min-priorità, mettendo in risalto l'azione di DECREASE-KEY.

Come detto, possiamo utilizzare un heap per implementate una coda di priorità. In una data applicazione, come la programmazione dei compiti o la simulazione controllata da eventi, gli elementi di una coda di priorità corrispondono agli oggetti dell'applicazione. Spesso è necessario determinare quale oggetto dell'applicazione corrisponde a un dato elemento della coda di priorità e viceversa. Pertanto, quando un heap viene utilizzato per implementare una coda di priorità, spesso occorre memorizzare un ***handle*** (aggancio) con il corrispondente oggetto dell'applicazione in ogni elemento dell'heap. L'esatta realizzazione dell'handle (un puntatore, un intero, ecc.) dipende dall'applicazione. Analogamente, occorre memorizzare un handle con il corrispondente elemento dell'heap in ogni oggetto dell'applicazione. Qui, tipicamente, l'handle è un indice dell'array. Poiché gli elementi dell'heap cambiano posizione all'interno dell'array durante le operazioni con l'heap, un'implementazione reale, dopo avere spostato un elemento dell'heap, dovrebbe aggiornare anche l'indice dell'array nel corrispondente oggetto dell'applicazione. Poiché i dettagli per accedere agli oggetti dell'applicazione dipendono molto dall'applicazione e dalla sua implementazione, non li tratteremo qui, ma ci limiteremo a notare che, in pratica, questi handle hanno bisogno di essere mantenuti correttamente.

Adesso descriviamo come implementare le operazioni di una coda di max-priorità. La procedura HEAP-MAXIMUM implementa l'operazione MAXIMUM nel tempo $\Theta(1)$.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

La procedura HEAP-EXTRACT-MAX implementa l'operazione EXTRACT-MAX. È simile al corpo del ciclo **for** (righe 3–5) della procedura HEAPSORT.

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2     then error "underflow dell'heap"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Il tempo di esecuzione di HEAP-EXTRACT-MAX è $O(\lg n)$, in quanto svolge soltanto una quantità costante di lavoro oltre al tempo $O(\lg n)$ di MAX-HEAPIFY.

La procedura HEAP-INCREASE-KEY implementa l'operazione INCREASE-KEY. L'elemento della coda di priorità la cui chiave deve essere aumentata è identificato da un indice i nell'array. Innanzitutto, la procedura aggiorna la chiave dell'elemento $A[i]$ con il suo nuovo valore. Successivamente, poiché l'aumento della chiave di $A[i]$ potrebbe violare la proprietà del max-heap, la procedura, in una maniera che ricorda il ciclo di inserzione (righe 5–7) di INSERTION-SORT nel Paragrafo 2.1, segue un percorso da questo nodo verso la radice per trovare un posto appropriato alla nuova chiave. Durante questo attraversamento, confronta ripetutamente un elemento con suo padre e scambia le loro chiavi se la chiave dell'elemento è più grande; questa operazione termina se la chiave dell'elemento è più piccola, perché in questo caso la proprietà del max-heap è soddisfatta (vedere l'Esercizio 6.5-5 per una esatta invariante di ciclo).

```
HEAP-INCREASE-KEY(A, i, chiave)
1  if chiave < A[i]
2     then error "la nuova chiave è più piccola di quella corrente"
3  A[i] ← chiave
4  while i > 1 and A[PARENT(i)] < A[i]
5        do scambia A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

La Figura 6.5 illustra un esempio dell'operazione HEAP-INCREASE-KEY. Il tempo di esecuzione di HEAP-INCREASE-KEY con un heap di n elementi è $O(\lg n)$, in quanto il cammino seguito dal nodo aggiornato nella riga 3 alla radice ha lunghezza $O(\lg n)$.

La procedura MAX-HEAP-INSERT implementa l'operazione INSERT. Prende come input la chiave del nuovo elemento da inserire nel max-heap A. La procedura prima espande il max-heap aggiungendo all'albero una nuova foglia la cui chiave è $-\infty$; poi chiama HEAP-INCREASE-KEY per impostare la chiave di questo nuovo nodo al suo valore corretto e mantenere la proprietà del max-heap.

```
MAX-HEAP-INSERT(A, chiave)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], chiave)
```

Il tempo di esecuzione della procedura MAX-HEAP-INSERT con un heap di n elementi è $O(\lg n)$. In sintesi, un heap può svolgere qualsiasi operazione con le code di priorità su un insieme di dimensione n nel tempo $O(\lg n)$.

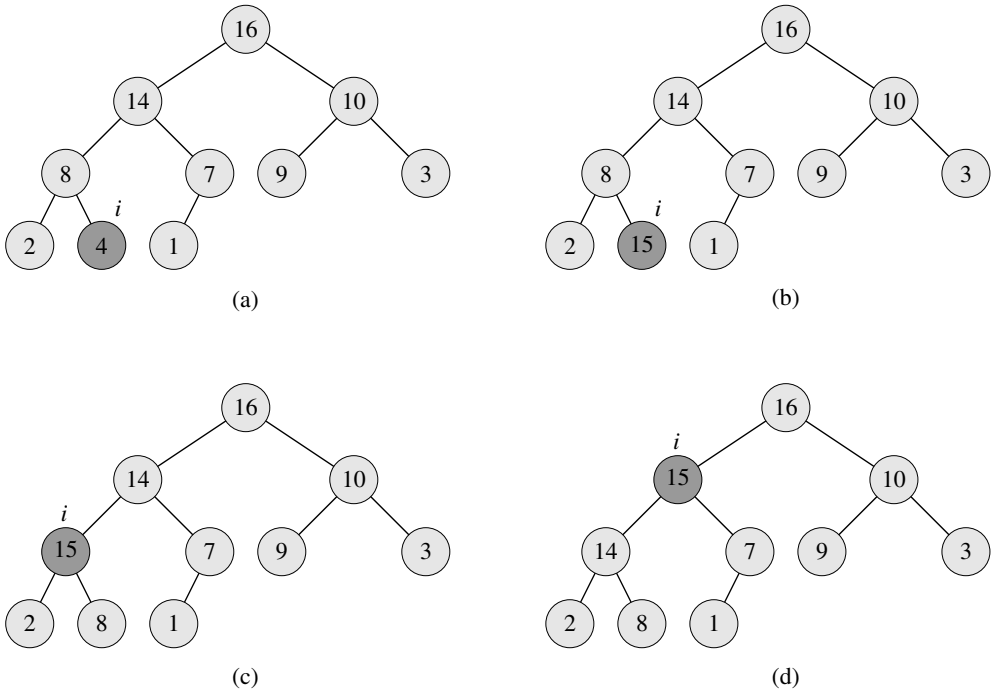


Figura 6.5 L'operazione di HEAP-INCREASE-KEY. **(a)** Il max-heap della Figura 6.4(a) con il nodo di indice i più scuro. **(b)** La chiave di questo nodo viene aumentata a 15. **(c)** Dopo un'iterazione del ciclo **while**, righe 4–6, il nodo e suo padre hanno le chiavi scambiate e l'indice i passa al padre. **(d)** Il max-heap dopo un'altra iterazione del ciclo **while**. A questo punto, $A[\text{PARENT}(i)] \geq A[i]$. La proprietà del max-heap adesso è soddisfatta è la procedura termina.

Esercizi

6.5-1

Illustrate l'operazione di HEAP-EXTRACT-MAX sull'heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2

Illustrate l'operazione di MAX-HEAP-INSERT$(A, 10)$ sull'heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$. Usate l'heap della Figura 6.5 come un modello per la chiamata di HEAP-INCREASE-KEY.

6.5-3

Scrivete lo pseudocodice per le procedure HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY e MIN-HEAP-INSERT che implementano una coda di min-priorità con un min-heap.

6.5-4

Perché dobbiamo preoccuparci di impostare la chiave del nodo inserito a $-\infty$ nella riga 2 di MAX-HEAP-INSERT quando la successiva cosa che facciamo è aumentare la sua chiave al valore desiderato?

6.5-5

Dimostrate la correttezza di HEAP-INCREASE-KEY utilizzando la seguente invariante di ciclo:

> All'inizio di ogni iterazione del ciclo **while** (righe 4–6), l'array $A[1\,..\,\mathit{heap\text{-}size}[A]]$ soddisfa la proprietà del max-heap, tranne una possibile violazione: $A[i]$ potrebbe essere più grande di $A[\text{PARENT}(i)]$.

6.5-6
Spiegate come implementare il metodo FIFO (First-In, First-Out) con una coda di priorità. Spiegate come implementare uno stack con una coda di priorità (le code e gli stack sono definiti nel Paragrafo 10.1).

6.5-7
L'azione di HEAP-DELETE(A, i) cancella l'elemento nel nodo i dall'heap A. Implementate la procedura HEAP-DELETE in modo che il suo tempo di esecuzione sia $O(\lg n)$ per un max-heap di n elementi.

6.5-8
Descrivete un algoritmo con tempo $O(n \lg k)$ per fondere k liste ordinate in un'unica lista ordinata, dove n è il numero totale di elementi di tutte le liste di input (*suggerimento:* usate un min-heap per la fusione).

6.6 Problemi

6-1 Costruire un heap mediante inserzione
La procedura BUILD-MAX-HEAP descritta nel Paragrafo 6.3 può essere implementata utilizzando ripetutamente MAX-HEAP-INSERT per inserire gli elementi nell'heap. Considerate la seguente implementazione:

```
BUILD-MAX-HEAP'(A)
1  heap-size[A] ← 1
2  for i ← 2 to lunghezza[A]
3       do MAX-HEAP-INSERT(A, A[i])
```

a. Le procedure BUILD-MAX-HEAP e BUILD-MAX-HEAP′ creano sempre lo stesso heap se vengono eseguite con lo stesso array di input? Dimostrate che lo fanno o illustrate un esempio contrario.

b. Dimostrate che, nel caso peggiore, BUILD-MAX-HEAP′ richiede un tempo $\Theta(n \lg n)$ per costruire un heap di n elementi.

6-2 Analisi di un heap d-ario
Un ***heap d-ario*** è come un heap binario, con una (possibile) eccezione che un nodo non-foglia ha d figli, anziché 2 figli.

a. Come rappresentereste un heap d-ario in un array?

b. Qual è l'altezza di un heap d-ario di n elementi espressa in funzione di n e d?

c. Realizzate un'implementazione efficiente di EXTRACT-MAX in un max-heap d-ario. Analizzate il suo tempo di esecuzione in funzione di d e n.

d. Realizzate un'implementazione efficiente di INSERT in un max-heap d-ario. Analizzate il suo tempo di esecuzione in funzione di d e n.

e. Realizzate un'implementazione efficiente di INCREASE-KEY(A, i, k), che prima imposta $A[i] \leftarrow \max(A[i], k)$ e poi aggiorna appropriatamente la struttura del max-heap. Analizzate il suo tempo di esecuzione in funzione di d e n.

6-3 I tableau di Young

Un ***tableau di Young*** è una matrice $m \times n$ nella quale gli elementi di ogni riga sono ordinati da sinistra a destra e gli elementi di ogni colonna sono ordinati dall'alto verso il basso. Alcuni elementi di un tableau di Young possono essere ∞, che noi consideriamo elementi inesistenti. Quindi, un tableau di Young può essere utilizzato per contenere $r \leq mn$ numeri finiti.

a. Disegnate un tableau di Young 4×4 che contiene gli elementi $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Se Y è un tableau di Young $m \times n$, dimostrate che Y è vuoto se $Y[1, 1] = \infty$. Dimostrate che Y è pieno (contiene mn elementi) se $Y[m, n] < \infty$.

c. Create un algoritmo per eseguire l'operazione EXTRACT-MIN in un tableau di Young $m \times n$ non vuoto nel tempo $O(m + n)$. Il vostro algoritmo dovrebbe utilizzare una subroutine ricorsiva che risolve un problema $m \times n$ risolvendo ricorsivamente un sottoproblema $(m - 1) \times n$ o $m \times (n - 1)$ (*suggerimento:* pensate a MAX-HEAPIFY). Se $T(p)$, dove $p = m + n$, è il tempo massimo di esecuzione di EXTRACT-MIN con qualsiasi tableau di Young $m \times n$, scrivete e risolvete una ricorrenza per $T(p)$ che fornisce il limite sul tempo $O(m + n)$.

d. Spiegate come inserire un nuovo elemento in un tableau di Young $m \times n$ non vuoto nel tempo $O(m + n)$.

e. Senza utilizzare alcun metodo di ordinamento come subroutine, spiegate come utilizzare un tableau di Young $n \times n$ per ordinare n^2 numeri nel tempo $O(n^3)$.

f. Indicate un algoritmo con tempo $O(m + n)$ per determinare se un particolare numero è memorizzato in un determinato tableau di Young $m \times n$.

Note

L'algoritmo heapsort è stato ideato da Williams [316], che ha descritto anche come implementare una coda di priorità con un heap. La procedura BUILD-MAX-HEAP è stata proposta da Floyd [90]. Nei Capitoli 16, 23 e 24 utilizzeremo i min-heap per implementare le code di min-priorità. Nei Capitoli 19 e 20 presenteremo anche un'implementazione con limiti migliori sul tempo per determinate operazioni.

È possibile realizzare implementazioni più veloci delle code di priorità per dati di tipo integer. Una struttura dati inventata da van Emde Boas [301] svolge le operazioni MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR e SUCCESSOR nel tempo $O(\lg \lg C)$ nel caso peggiore, a condizione che l'universo delle chiavi sia l'insieme $\{1, 2, \ldots, C\}$. Se i dati sono integer a b bit e la memoria del calcolatore è costituita da word a b bit indirizzabili, Fredman e Willard [99] hanno dimostrato come implementare MINIMUM nel tempo $O(1)$ e INSERT e EXTRACT-MIN nel tempo $O(\sqrt{\lg n})$. Thorup [299] ha migliorato il limite $O(\sqrt{\lg n})$ al tempo $O(\lg \lg n)$. Questo limite usa una quantità di spazio non vincolato in n, ma può essere implementato nello spazio lineare utilizzando l'hashing randomizzato.

Un importante caso speciale di code di priorità si verifica quando la sequenza delle operazioni EXTRACT-MIN è ***monotona***, ovvero i valori restituiti dalle successive operazioni EXTRACT-MIN sono monotonicamente crescenti nel tempo. Questo caso si presenta in molte applicazioni importanti, come l'algoritmo di Dijkstra per il problema del cammino minimo da sorgente singola, che sarà descritto nel Capitolo 24, e nella simulazione di eventi discreti.

continua

Per l'algoritmo di Dijkstra è particolarmente importante che l'operazione DECREASE-KEY sia implementata in maniera efficiente. Nel caso della sequenza monotona, se i dati sono numeri interi nell'intervallo $1, 2, \ldots, C$, Ahuja, Mehlhorn, Orlin e Tarjan [8] hanno spiegato come implementare EXTRACT-MIN e INSERT nel tempo ammortizzato $O(\lg C)$ (vedere il Capitolo 17 per maggiori informazioni sull'analisi ammortizzata) e DECREASE-KEY nel tempo $O(1)$, utilizzando una struttura dati detta *radix heap*. Il limite $O(\lg C)$ può essere migliorato a $O(\sqrt{\lg C})$ utilizzando gli heap di Fibonacci (descritti nel Capitolo 20) in combinazione con le strutture radix heap. Il limite è stato ulteriormente migliorato a $O(\lg^{1/3+\epsilon} C)$ da Cherkassky, Goldberg e Silverstein [58], che hanno combinato la struttura di bucket a più livelli di Denardo e Fox [72] con l'heap di Thorup prima citato. Raman [256] ha ulteriormente perfezionato questi risultati ottenendo un limite di $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$, per qualsiasi costante $\epsilon > 0$. Altre analisi dettagliate di questi risultati si trovano negli articoli di Raman [256] e Thorup [299].

Quicksort 7

Quicksort è un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore è $\Theta(n^2)$ con un array di input di n numeri. Nonostante questo tempo di esecuzione nel caso peggiore sia lento, quicksort spesso è la soluzione pratica migliore per effettuare un ordinamento, perché mediamente è molto efficiente: il suo tempo di esecuzione atteso è $\Theta(n \lg n)$ e i fattori costanti nascosti nella notazione $\Theta(n \lg n)$ sono molto piccoli. Inoltre ha il vantaggio di ordinare sul posto (come descritto a pagina 14) e funziona bene anche in ambienti con memoria virtuale.

Il Paragrafo 7.1 descrive l'algoritmo e un'importante subroutine utilizzata da quicksort per il partizionamento. Poiché il comportamento di quicksort è complesso, inizieremo con una discussione intuitiva delle sue operazioni nel Paragrafo 7.2 e rinvieremo l'analisi più formale alla fine del capitolo. Il Paragrafo 7.3 presenta una versione di quicksort che usa il campionamento aleatorio. Questo algoritmo ha un buon tempo di esecuzione nel caso medio e nessun particolare input provoca il comportamento nel caso peggiore dell'algoritmo. L'algoritmo randomizzato è analizzato nel Paragrafo 7.4, dove dimostriamo che viene eseguito nel tempo $\Theta(n^2)$ nel caso peggiore e, se gli elementi sono distinti, nel tempo $O(n \lg n)$ nel caso medio.

7.1 Descrizione di quicksort

Quicksort, come merge sort, è basato sul paradigma divide et impera presentato nel Paragrafo 2.3.1. Questi sono i tre passaggi del processo divide et impera per ordinare un tipico sottoarray $A[p \,..\, r]$.

Divide: partizionare l'array $A[p \,..\, r]$ in due sottoarray $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ (eventualmente vuoti) tali che ogni elemento di $A[p \,..\, q-1]$ sia minore o uguale ad $A[q]$ che, a sua volta, è minore o uguale a ogni elemento di $A[q+1 \,..\, r]$. Calcolare l'indice q come parte di questa procedura di partizionamento.

Impera: ordinare i due sottoarray $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ chiamando ricorsivamente quicksort.

Combina: poiché i sottoarray sono ordinati sul posto, non occorre alcun lavoro per combinarli: l'intero array $A[p \,..\, r]$ è ordinato.

La seguente procedura implementa quicksort.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

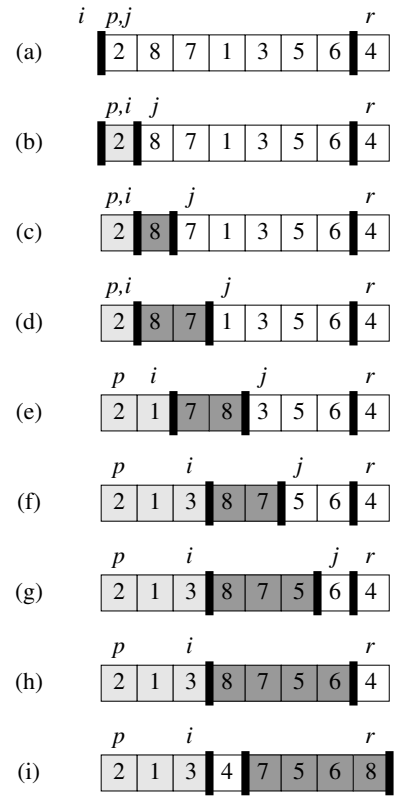


Figura 7.1 L'operazione di PARTITION su un array campione. Gli elementi dell'array su sfondo grigio chiaro sono tutti nella prima partizione con valori non maggiori di x. Gli elementi su sfondo grigio scuro sono nella seconda partizione con valori maggiori di x. Gli elementi su sfondo bianco non sono stati ancora posti in una delle prime due partizioni; l'ultimo elemento su sfondo bianco è il *pivot* o *perno*. **(a)** L'array iniziale e le impostazioni delle variabili. Nessuno degli elementi è stato posto in una delle prime due partizioni. **(b)** Il valore 2 viene "scambiato con sé stesso" e posto nella partizione dei valori più piccoli. **(c)–(d)** I valori 8 e 7 vengono inseriti nella partizione dei valori più grandi. **(e)** I valori 1 e 8 vengono scambiati e la prima partizione cresce. **(f)** I valori 3 e 7 vengono scambiati e la prima partizione cresce. **(g)–(h)** La seconda partizione acquisisce i valori 5 e 6 e il ciclo termina. **(i)** Le righe 7–8 scambiano il pivot inserendolo fra le due partizioni.

Per ordinare un intero array A, la chiamata iniziale è

QUICKSORT$(A, 1, lunghezza[A])$

Partizionare l'array

L'elemento chiave dell'algoritmo è la procedura PARTITION, che riarrangia il sottoarray $A[p\,..\,r]$ sul posto.

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  scambia A[i] ↔ A[j]
7  scambia A[i + 1] ↔ A[r]
8  return i + 1
```

La Figura 7.1 illustra l'operazione di PARTITION su un array di 8 elementi. PARTITION seleziona sempre un elemento $x = A[r]$ come ***pivot*** intorno al quale partizionare il sottoarray $A[p\,..\,r]$. Durante l'esecuzione della procedura, l'array viene suddiviso in quattro regioni (eventualmente vuote). All'inizio di ogni iterazione del ciclo **for**, righe 3–6, ogni regione soddisfa alcune proprietà, che possiamo definire come invariante di ciclo:

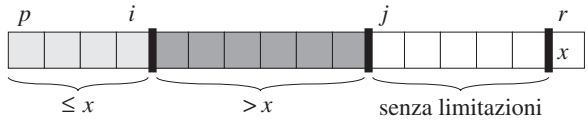


Figura 7.2 Le quattro regioni mantenute dalla procedura PARTITION in un sottoarray $A[p\,..\,r]$. I valori in $A[p\,..\,i]$ sono tutti minori o uguali a x, i valori in $A[i+1\,..\,j-1]$ sono tutti maggiori di x e $A[r] = x$. Gli elementi di $A[j\,..\,r-1]$ possono assumere qualsiasi valore.

All'inizio di ogni iterazione del ciclo, righe 3–6, per qualsiasi indice k dell'array,

1. Se $p \le k \le i$, allora $A[k] \le x$.
2. Se $i+1 \le k \le j-1$, allora $A[k] > x$.
3. Se $k = r$, allora $A[k] = x$.

La Figura 7.2 sintetizza questa struttura. Gli indici fra j e $r-1$ non rientrano in alcuno di questi tre casi e i corrispondenti valori di queste voci non hanno una particolare relazione con il pivot x.

Dobbiamo dimostrare che questa invariante di ciclo è vera prima della prima iterazione, che ogni iterazione del ciclo conserva l'invariante e che l'invariante fornisce un'utile proprietà per dimostrare la correttezza quando il ciclo termina.

Inizializzazione: prima della prima iterazione del ciclo, $i = p-1$ e $j = p$. Non ci sono valori fra p e i né fra $i+1$ e $j-1$, quindi le prime due condizioni dell'invariante di ciclo sono soddisfatte. L'assegnazione nella riga 1 soddisfa la terza condizione.

Conservazione: coma mostra la Figura 7.3, ci sono due casi da considerare, a seconda del risultato del test nella riga 4. La Figura 7.3(a) mostra che cosa accade quando $A[j] > x$; l'unica azione nel ciclo è incrementare j. Dopo l'incremento di j, la condizione 2 è soddisfatta per $A[j-1]$ e tutte le altre voci non cambiano. La Figura 7.3(b) mostra che cosa accade quando $A[j] \le x$; viene incrementato l'indice i, vengono scambiati $A[i]$ e $A[j]$ e, poi, viene incrementato l'indice j. In seguito allo scambio, adesso abbiamo $A[i] \le x$ e la condizione 1 è soddisfatta. Analogamente, abbiamo anche $A[j-1] > x$, in quanto l'elemento che è stato scambiato in $A[j-1]$ è, per l'invariante di ciclo, più grande di x.

Conclusione: alla fine del ciclo, $j = r$. Pertanto, ogni voce dell'array si trova in uno dei tre insiemi descritti dall'invariante e noi abbiamo ripartito i valori dell'array in tre insiemi: quelli minori o uguali a x, quelli maggiori di x e un insieme a un solo elemento che contiene x.

Le ultime due righe di PARTITION inseriscono il pivot al suo posto nel mezzo dell'array, scambiandolo con l'elemento più a sinistra che è maggiore di x. L'output di PARTITION adesso soddisfa le specifiche del passo *divide*.

Il tempo di esecuzione di PARTITION con il sottoarray $A[p\,..\,r]$ è $\Theta(n)$, dove $n = r - p + 1$ (vedere l'Esercizio 7.1-3).

Esercizi

7.1-1

Utilizzando la Figura 7.1 come modello, illustrate l'operazione di PARTITION sull'array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

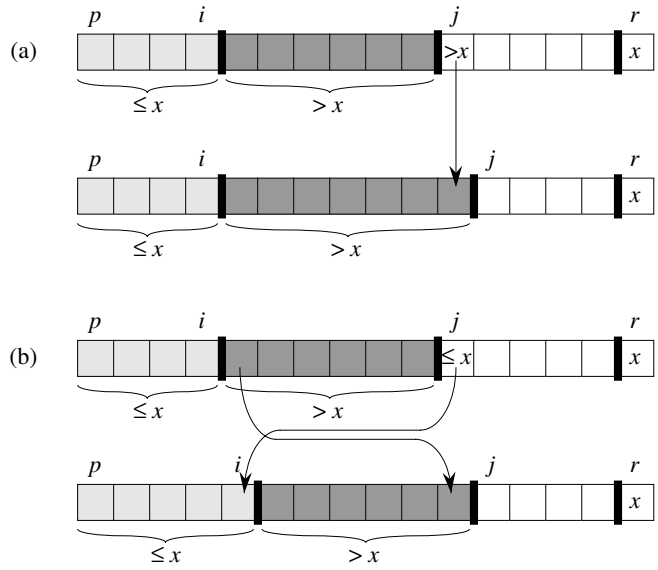


Figura 7.3 I due casi per una iterazione della procedura PARTITION. **(a)** Se $A[j] > x$, l'unica azione è incrementare j, che conserva l'invariante di ciclo. **(b)** Se $A[j] \leq x$, viene incrementato l'indice i, vengono scambiati $A[i]$ e $A[j]$; poi, viene incrementato l'indice j. Ancora una volta l'invariante di ciclo è conservata.

7.1-2
Quale valore di q restituisce PARTITION quando tutti gli elementi di $A[p\,..\,r]$ hanno lo stesso valore? Modificate PARTITION in modo che $q = \lfloor (p+r)/2 \rfloor$ quando tutti gli elementi di $A[p\,..\,r]$ hanno lo stesso valore.

7.1-3
Spiegate brevemente perché il tempo di esecuzione di PARTITION con un sottoarray di dimensione n è $\Theta(n)$.

7.1-4
Come modifichereste QUICKSORT per ordinare un array in senso non crescente?

7.2 Prestazioni di quicksort

Il tempo di esecuzione di quicksort dipende dal fatto che il partizionamento sia bilanciato o sbilanciato e questo, a sua volta, dipende da quali elementi vengono utilizzati nel partizionamento. Se il partizionamento è bilanciato, l'algoritmo viene eseguito con la stessa velocità asintotica di merge sort. Se il partizionamento è sbilanciato, invece, quicksort può essere asintoticamente lento quanto insertion sort. In questo paragrafo analizzeremo, in modo informale, le prestazioni di quicksort nel caso di partizionamento bilanciato e sbilanciato.

Partizionamento nel caso peggiore

Il comportamento nel caso peggiore di quicksort si verifica quando la routine di partizionamento produce un sottoproblema con $n-1$ elementi e uno con 0 elementi (questo è dimostrato nel Paragrafo 7.4.1). Supponiamo che questo sbilanciamento si verifichi in ogni chiamata ricorsiva. Il partizionamento costa $\Theta(n)$ in termini di tempo. Poiché per una chiamata ricorsiva su un array vuoto $T(0) = \Theta(1)$, la ricorrenza per il tempo di esecuzione può essere espressa così:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Intuitivamente, se sommiamo i costi a ogni livello della ricorsione, otteniamo una serie aritmetica (equazione (A.2)), il cui valore è $\Theta(n^2)$. In effetti, è semplice applicare il metodo di sostituzione per dimostrare che la ricorrenza $T(n) = T(n-1) + \Theta(n)$ ha la soluzione $T(n) = \Theta(n^2)$ (vedere l'Esercizio 7.2-1). Quindi, se lo sbilanciamento delle due partizioni è massimo a ogni livello ricorsivo dell'algoritmo, il tempo di esecuzione è $\Theta(n^2)$.

In definitiva, il tempo di esecuzione nel caso peggiore di quicksort non è migliore di quello di insertion sort. Inoltre, il tempo di esecuzione $\Theta(n^2)$ si ha quando l'array di input è già completamente ordinato – una situazione comune in cui insertion sort è eseguito nel tempo $O(n)$.

Partizionamento nel caso migliore

Nel caso di bilanciamento massimo, PARTITION produce due sottoproblemi, ciascuno di dimensione non maggiore di $n/2$, in quanto uno ha dimensione $\lfloor n/2 \rfloor$ e l'altro ha dimensione $\lceil n/2 \rceil - 1$. In questo caso, quicksort viene eseguito molto più velocemente. La ricorrenza per il tempo di esecuzione è

$$T(n) \leq 2T(n/2) + \Theta(n)$$

che per il caso 2 del teorema dell'esperto (Teorema 4.1) ha la soluzione $T(n) = O(n \lg n)$. Dunque, il perfetto bilanciamento dei due lati della partizione a ogni livello di ricorsione produce un algoritmo asintoticamente più veloce.

Partizionamento bilanciato

Il tempo di esecuzione nel caso medio di quicksort è molto più vicino al caso migliore che al caso peggiore, come dimostrerà l'analisi svolta nel Paragrafo 7.4. Per capire perché, dobbiamo spiegare come il bilanciamento del partizionamento influisce sulla ricorrenza che descrive il tempo di esecuzione.

Supponiamo, per esempio, che l'algoritmo di partizionamento produca sempre una ripartizione proporzionale 9-a-1, che a prima vista potrebbe sembrare molto sbilanciata. In questo caso, otteniamo la ricorrenza

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

sul tempo di esecuzione di quicksort, dove abbiamo esplicitamente incluso la costante c nascosta nel termine $\Theta(n)$. La Figura 7.4 illustra l'albero di ricorsione per questa ricorrenza. Notate che ogni livello dell'albero ha un costo cn, finché non viene raggiunta una condizione al contorno alla profondità $\log_{10} n = \Theta(\lg n)$, dopo la quale i livelli hanno al massimo un costo cn.

La ricorsione termina alla profondità $\log_{10/9} n = \Theta(\lg n)$. Il costo totale di quicksort è dunque $O(n \lg n)$. Pertanto, con una ripartizione proporzionale 9-a-1 a ogni livello di ricorsione, che intuitivamente sembra molto sbilanciata, quicksort viene eseguito nel tempo $O(n \lg n)$ – asintoticamente uguale a quello che si ha nel caso di ripartizione esattamente a metà.

In effetti, anche una ripartizione 99-a-1 determina un tempo di esecuzione pari a $O(n \lg n)$. La ragione è che qualsiasi ripartizione con proporzionalità *costante* produce un albero di ricorsione di profondità $\Theta(\lg n)$, dove il costo in ogni livello è $O(n)$. Il tempo di esecuzione è quindi $O(n \lg n)$ quando la ripartizione ha proporzionalità costante.

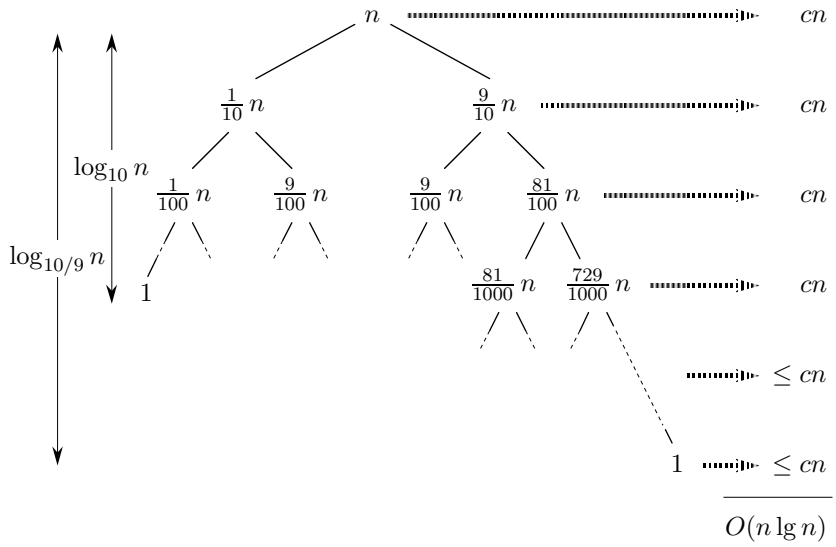


Figura 7.4 Un albero di ricorsione per QUICKSORT quando PARTITION genera sempre una ripartizione 9-a-1, determinando un tempo di esecuzione pari a $O(n \lg n)$. I nodi mostrano le dimensioni dei sottoproblemi, con i costi per livello a destra. Questi costi includono la costante c implicita nel termine $\Theta(n)$.

Alcune intuizioni sul caso medio

Per sviluppare una nozione chiara del caso medio per quicksort, dobbiamo fare un'ipotesi su quanto frequentemente prevediamo di incontrare i vari input. Il comportamento di quicksort è determinato dall'ordinamento relativo dei valori degli elementi dell'array che sono dati come input, non dai particolari valori dell'array. Analogamente a quanto fatto nell'analisi probabilistica del problema delle assunzioni (Paragrafo 5.2), ipotizzeremo per il momento che tutte le permutazioni dei numeri di input siano ugualmente probabili.

Quando eseguiamo quicksort su un array di input casuale, è poco probabile che il partizionamento avvenga sempre nello stesso modo a qualsiasi livello, come abbiamo ipotizzato nella nostra analisi informale. È logico supporre, invece, che qualche ripartizione sarà ben bilanciata e qualche altra sarà molto sbilanciata. Per esempio, l'Esercizio 7.2-6 chiede di dimostrare che circa l'80% delle volte PARTITION produce una ripartizione che è più bilanciata di 9 a 1 e che circa il 20% delle volte produce una ripartizione che meno bilanciata di 9 a 1.

Nel caso medio, PARTITION produce una combinazione di ripartizioni "buone" e "cattive". In un albero di ricorsione per l'esecuzione nel caso medio di PARTITION, le buone e le cattive ripartizioni sono distribuite a caso nell'albero. Supponiamo, tuttavia, che le ripartizioni buone e cattive si alternino nei vari livelli dell'albero e che quelle buone siano le ripartizioni nel caso migliore e quelle cattive siano le ripartizioni nel caso peggiore. La Figura 7.5(a) illustra le ripartizioni in due livelli consecutivi nell'albero di ricorsione. Nella radice dell'albero il costo del partizionamento è n e i sottoarray prodotti hanno dimensioni $n-1$ e 0: il caso peggiore. Nel livello successivo il sottoarray di dimensione $n-1$ è ripartito in due sottoarray di dimensioni $(n-1)/2-1$ e $(n-1)/2$: il caso migliore. Supponiamo che il costo della condizione al contorno sia 1 per il sottoarray di dimensione 0.

La combinazione della ripartizione cattiva seguita dalla ripartizione buona produce tre sottoarray di dimensioni 0, $(n-1)/2-1$ e $(n-1)/2$ con un costo di partizionamento combinato pari a $\Theta(n) + \Theta(n-1) = \Theta(n)$. Certamente, questo caso non è peggiore di quello della Figura 7.5(b), ovvero un unico livello di partizionamento che genera due sottoarray di dimensione $(n-1)/2$, con un costo

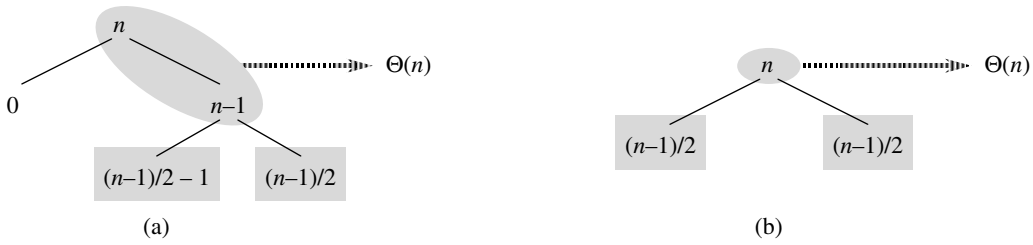


Figura 7.5 **(a)** Due livelli di un albero di ricorsione per quicksort. Il partizionamento nella radice costa n e produce una ripartizione "cattiva": due sottoarray di dimensioni 0 e $n-1$. Il partizionamento del sottoarray di dimensione $n-1$ costa $n-1$ e produce una ripartizione "buona": due sottoarray di dimensioni $(n-1)/2-1$ e $(n-1)/2$. **(b)** Un solo livello di un albero di ricorsione che è bilanciato molto bene. In entrambi i casi, il costo di partizionamento per i sottoproblemi è rappresentato da un'ellisse grigia e vale $\Theta(n)$. Inoltre i sottoproblemi che restano da risolvere nel caso (a), rappresentati da rettangoli grigi, non sono più grandi dei corrispondenti problemi che restano da risolvere nel caso (b).

di $\Theta(n)$. Quest'ultimo caso è anche bilanciato! Intuitivamente, il costo $\Theta(n-1)$ della ripartizione cattiva può essere assorbito nel costo $\Theta(n)$ della ripartizione buona, quindi la ripartizione risultante è buona. In definitiva, il tempo di esecuzione di quicksort, quando i livelli si alternano fra buone e cattive ripartizioni, è come il tempo di esecuzione nel caso in cui le ripartizioni siano soltanto buone: ancora $O(n \lg n)$, ma con una costante un po' più grande nascosta dalla notazione O. Nel Paragrafo 7.4.2 faremo un'analisi più rigorosa del caso medio di una versione randomizzata di quicksort.

Esercizi

7.2-1

Utilizzate il metodo di sostituzione per dimostrare che la ricorrenza $T(n) = T(n-1) + \Theta(n)$ ha la soluzione $T(n) = \Theta(n^2)$, come detto all'inizio del Paragrafo 7.2.

7.2-2

Qual è il tempo di esecuzione di QUICKSORT quando tutti gli elementi dell'array A hanno lo stesso valore?

7.2-3

Dimostrate che il tempo di esecuzione di QUICKSORT è $\Theta(n^2)$ quando l'array A contiene elementi distinti ed è ordinato in senso decrescente.

7.2-4

Le banche spesso registrano le transazioni in ordine cronologico, ma molte persone preferiscono ricevere l'estratto conto con gli assegni elencati in funzione dei numeri degli assegni. Le persone di solito emettono gli assegni in ordine numerico e i beneficiari di solito portano subito all'incasso gli assegni. Il problema di convertire l'ordine cronologico delle transazioni in ordine numerico degli assegni è dunque il problema di ordinare elementi di input quasi ordinati. Dimostrate che la procedura INSERTION-SORT tende a battere la procedura QUICKSORT in questo problema.

7.2-5
Supponete che le ripartizioni a qualsiasi livello di quicksort siano nella proporzione $1 - \alpha$ ad α, dove $0 < \alpha \leq 1/2$ è una costante. Dimostrate che la profondità minima di una foglia nell'albero di ricorsione è approssimativamente $-\lg n / \lg \alpha$ e la profondità massima è approssimativamente $-\lg n / \lg(1 - \alpha)$ (trascurate l'arrotondamento agli interi).

7.2-6 $\star$
Dimostrate che per qualsiasi costante $0 < \alpha \leq 1/2$, c'è la probabilità di circa $1 - 2\alpha$ che su un array di input casuale la procedura PARTITION produca una ripartizione più bilanciata della proporzione $1 - \alpha$ ad α.

7.3 Una versione randomizzata di quicksort

Nell'analisi del comportamento di quicksort nel caso medio, abbiamo fatto l'ipotesi che tutte le permutazioni dei numeri di input fossero ugualmente probabili. Nella pratica, però, non possiamo aspettarci che questo sia sempre vero (vedere l'Esercizio 7.2-4). Come detto nel Paragrafo 5.3, a volte è possibile aggiungere la casualità a un algoritmo per ottenere buone prestazioni nel caso medio con tutti gli input. Molte persone considerano la versione randomizzata di quicksort come l'algoritmo di ordinamento da scegliere nel caso di input molto estesi.

Nel Paragrafo 5.3, abbiamo randomizzato il nostro algoritmo permutando esplicitamente l'input. Potremmo fare la stessa cosa anche per quicksort; invece adotteremo un altro metodo di randomizzazione, detto ***campionamento aleatorio***, che ci consente di semplificare l'analisi. Anziché utilizzare sempre $A[r]$ come pivot, utilizzeremo un elemento scelto a caso dal sottoarray $A[p\,..\,r]$. Per fare questo, scambieremo l'elemento $A[r]$ con un elemento scelto a caso da $A[p\,..\,r]$. Questa modifica, con la quale campioniamo a caso l'intervallo $p, \ldots, r$, ci assicura che l'elemento pivot $x = A[r]$ avrà la stessa probabilità di essere uno qualsiasi degli $r - p + 1$ elementi del sottoarray. Poiché il pivot viene scelto a caso, prevediamo che la ripartizione dell'array di input potrà essere ben bilanciata in media.

Le modifiche da apportare a PARTITION e QUICKSORT sono modeste. Nella nuova procedura di partizionamento, implementiamo semplicemente lo scambio prima dell'effettivo partizionamento:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  scambia A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

Il nuovo quicksort chiama RANDOMIZED-PARTITION, anziché PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q − 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Nel prossimo paragrafo analizzeremo questo algoritmo.

Esercizi

7.3-1

Perché dobbiamo analizzare le prestazioni di un algoritmo randomizzato nel caso medio e non nel caso peggiore?

7.3-2

Durante l'esecuzione della procedura RANDOMIZED-QUICKSORT, quante chiamate del generatore di numeri casuali RANDOM vengono fatte nel caso peggiore? E nel caso migliore? Esprimete la vostra risposta nei termini della notazione Θ.

7.4 Analisi di quicksort

Il Paragrafo 7.2 ha presentato alcuni concetti intuitivi sul comportamento nel caso peggiore di quicksort e sul perché prevediamo che questo algoritmo possa essere eseguito rapidamente. In questo paragrafo analizziamo il comportamento di quicksort in modo più rigoroso. Iniziamo con l'analisi del caso peggiore, che si applica sia a QUICKSORT sia a RANDOMIZED-QUICKSORT, e concludiamo con l'analisi del caso medio di RANDOMIZED-QUICKSORT.

7.4.1 Analisi del caso peggiore

Nel Paragrafo 7.2 abbiamo visto che una ripartizione nel caso peggiore a qualsiasi livello di ricorsione in quicksort produce un tempo di esecuzione $\Theta(n^2)$ che, intuitivamente, è il tempo di esecuzione nel caso peggiore dell'algoritmo. Adesso dimostriamo questa asserzione.

Utilizzando il metodo di sostituzione (vedere il Paragrafo 4.1), possiamo dimostrare che il tempo di esecuzione di quicksort è $O(n^2)$. Sia $T(n)$ il tempo nel caso peggiore per la procedura QUICKSORT con un input di dimensione n. Otteniamo la ricorrenza

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \tag{7.1}$$

dove il parametro q varia da 0 a $n-1$, in quanto la procedura PARTITION genera due sottoproblemi con dimensione totale $n-1$. Supponiamo che la soluzione sia $T(n) \le cn^2$ per qualche costante c. Sostituendo questa soluzione nella ricorrenza (7.1), otteniamo

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

L'espressione $q^2 + (n-q-1)^2$ raggiunge il massimo nei due estremi dell'intervallo $0 \le q \le n-1$ del parametro q; infatti, la derivata seconda dell'espressione rispetto a q è positiva (vedere l'Esercizio 7.4-3). Questa osservazione ci fornisce il limite $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Riprendendo l'espressione di $T(n)$, otteniamo

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 \end{aligned}$$

perché possiamo assegnare alla costante c un valore sufficientemente grande affinché il termine $c(2n-1)$ prevalga sul termine $\Theta(n)$; quindi, $T(n) = O(n^2)$. Nel

Paragrafo 7.2 abbiamo esaminato un caso specifico in cui quicksort impiega un tempo $\Omega(n^2)$: quando il partizionamento è sbilanciato. In alternativa, l'Esercizio 7.4-1 chiede di dimostrare che la ricorrenza (7.1) ha una soluzione $T(n) = \Omega(n^2)$. Quindi, il tempo di esecuzione (nel caso peggiore) di quicksort è $\Theta(n^2)$.

7.4.2 Tempo di esecuzione atteso

Abbiamo già dato una spiegazione intuitiva sul perché il tempo di esecuzione nel caso medio di RANDOMIZED-QUICKSORT sia $O(n \lg n)$: se, in ogni livello di ricorsione, la ripartizione indotta da RANDOMIZED-PARTITION pone una frazione costante qualsiasi degli elementi in un lato della partizione, allora l'albero di ricorsione ha profondità $\Theta(\lg n)$ e in ogni livello viene svolto un lavoro $O(n)$. Anche se aggiungiamo nuovi livelli con la ripartizione più sbilanciata possibile tra questi livelli, il tempo totale resta $O(n \lg n)$. Possiamo analizzare con precisione il tempo di esecuzione atteso di RANDOMIZED-QUICKSORT, spiegando prima come opera la procedura di partizionamento e poi sfruttando questa conoscenza per ricavare un limite $O(n \lg n)$ sul tempo di esecuzione atteso (supponendo che i valori degli elementi siano distinti). Questo limite superiore sul tempo di esecuzione atteso, combinato con il limite $\Theta(n \lg n)$ nel caso migliore che abbiamo visto nel Paragrafo 7.2, fornisce un tempo di esecuzione atteso pari a $\Theta(n \lg n)$.

Tempo di esecuzione e confronti

Il tempo di esecuzione di QUICKSORT è dominato dal tempo impiegato nella procedura PARTITION. Ogni volta che viene chiamata la procedura PARTITION, viene selezionato un elemento pivot; questo elemento non sarà mai incluso nelle successive chiamate ricorsive di QUICKSORT e PARTITION. Quindi, ci possono essere al massimo n chiamate di PARTITION durante l'intera esecuzione dell'algoritmo quicksort. Una chiamata di PARTITION impiega il tempo $O(1)$ più una quantità di tempo che è proporzionale al numero di iterazioni del ciclo **for** nelle righe 3–6. Ogni iterazione di questo ciclo **for** effettua un confronto fra l'elemento pivot e un altro elemento dell'array A (riga 4). Pertanto, se contiamo il numero totale di volte che la riga 4 viene eseguita, possiamo limitare il tempo totale impiegato nel ciclo **for** durante l'intera esecuzione di QUICKSORT.

Lemma 7.1
Se X è il numero di confronti svolti nella riga 4 di PARTITION nell'intera esecuzione di QUICKSORT su un array di n elementi, allora il tempo di esecuzione di QUICKSORT è $O(n + X)$.

Dimostrazione Per la precedente discussione, ci sono n chiamate di PARTITION, ciascuna delle quali svolge una quantità costante di lavoro e poi esegue il ciclo **for** un certo numero di volte. Ogni iterazione del ciclo **for** esegue la riga 4. ■

Il nostro obiettivo, quindi, è calcolare X, il numero totale di confronti svolti in tutte le chiamate di PARTITION. Non tenteremo di analizzare quanti confronti vengono effettuati in *ogni* chiamata di PARTITION. Piuttosto, deriveremo un limite globale sul numero totale di confronti. Per farlo, dobbiamo capire quando l'algoritmo confronta due elementi dell'array e quando non lo fa. Per semplificare l'analisi, rinominiamo gli elementi dell'array A $z_1, z_2, \ldots, z_n$, dove z_i è l'i-esimo elemento più piccolo. Definiamo anche $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ l'insieme degli elementi compresi fra z_i e z_j, estremi inclusi.

Quando l'algoritmo confronta z_i e z_j? Per rispondere a questa domanda, osserviamo innanzitutto che ogni coppia di elementi viene confrontata al massimo una volta. Perché? Gli elementi sono confrontati soltanto con l'elemento pivot e, dopo che una particolare chiamata di PARTITION finisce, il pivot utilizzato in questa chiamata non viene più confrontato con nessun altro elemento.

La nostra analisi usa le variabili casuali indicatrici (vedere il Paragrafo 5.2). Definiamo

$$X_{ij} = \mathrm{I}\{z_i \text{ è confrontato con } z_j\}$$

Qui stiamo considerando se il confronto si svolge in un istante qualsiasi durante l'esecuzione dell'algoritmo, non soltanto durante un'iterazione o una chiamata di PARTITION. Poiché ogni coppia viene confrontata al massimo una volta, possiamo facilmente rappresentare il numero totale di confronti svolti dall'algoritmo in questo modo:

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

Prendendo i valori attesi da entrambi i lati e poi applicando la linearità del valore atteso e il Lemma 5.1, otteniamo

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ è confrontato con } z_j\} \end{aligned} \tag{7.2}$$

Resta da calcolare $\Pr\{z_i \text{ è confrontato con } z_j\}$. La nostra analisi suppone che ogni pivot sia scelto in modo casuale e indipendente.

È utile riflettere su quando due elementi *non* sono confrontati. Considerate come input di quicksort i numeri da 1 a 10 (in qualsiasi ordine) e supponete che il primo pivot sia 7. La prima chiamata di PARTITION separa i numeri in due insiemi: $\{1, 2, 3, 4, 5, 6\}$ e $\{8, 9, 10\}$. In questo processo, il pivot 7 viene confrontato con tutti gli altri elementi, ma nessun numero del primo insieme (per esempio, 2) è o sarà mai confrontato con qualsiasi altro numero del secondo insieme (per esempio, 9).

In generale, poiché supponiamo che i valori degli elementi siano distinti, una volta che viene scelto un pivot x con $z_i < x < z_j$, sappiamo che z_i e z_j non potranno essere confrontati in un istante successivo. Se, d'altra parte, viene scelto z_i come pivot prima di qualsiasi altro elemento di Z_{ij}, allora z_i sarà confrontato con ogni elemento di Z_{ij}, tranne sé stesso. Analogamente, se viene scelto z_j come pivot prima di qualsiasi altro elemento di Z_{ij}, allora z_j sarà confrontato con ogni elemento di Z_{ij}, tranne sé stesso. Nell'esempio in esame, vengono confrontati i valori 7 e 9, perché 7 è il primo elemento di $Z_{7,9}$ che viene scelto come pivot. I valori 2 e 9, invece, non saranno mai confrontati perché il primo elemento pivot scelto da $Z_{2,9}$ è 7. Quindi, z_i e z_j vengono confrontati se e soltanto se il primo elemento da scegliere come pivot da Z_{ij} è z_i o z_j.

Calcoliamo adesso la probabilità che si verifichi questo evento. Prima del punto in cui un elemento di Z_{ij} viene scelto come pivot, l'intero insieme Z_{ij} si trova nella stessa partizione. Pertanto, qualsiasi elemento di Z_{ij} ha la stessa probabilità di essere scelto per primo come pivot. Poiché l'insieme Z_{ij} ha $j - i + 1$ elementi e poiché i pivot vengono scelti in modo casuale e indipendente, la probabilità che qualsiasi elemento sia il primo a essere scelto come pivot è $1/(j - i + 1)$. Quindi, abbiamo

$$\begin{aligned}
\Pr\{z_i \text{ è confrontato con } z_j\} &= \Pr\{z_i \text{ o } z_j \text{ è il primo pivot scelto da } Z_{ij}\} \\
&= \Pr\{z_i \text{ è il primo pivot scelto da } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ è il primo pivot scelto da } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} \qquad (7.3)
\end{aligned}$$

La seconda riga si basa sul fatto che i due eventi si escludono a vicenda. Combinando le equazioni (7.2) e (7.3), otteniamo

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Possiamo calcolare questa sommatoria effettuando un cambio di variabili ($k = j - i$) e applicando il limite sulle serie armoniche espresso dall'equazione (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) \qquad (7.4)
\end{aligned}$$

Quindi concludiamo che, utilizzando RANDOMIZED-PARTITION, il tempo di esecuzione atteso di quicksort è $O(n \lg n)$ quando i valori degli elementi sono distinti.

Esercizi

7.4-1

Dimostrate che nella ricorrenza

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n) ,$$

$T(n) = \Omega(n^2)$.

7.4-2

Dimostrate che il tempo di esecuzione nel caso migliore di quicksort è $\Omega(n \lg n)$.

7.4-3
Dimostrate che $q^2 + (n - q - 1)^2$ raggiunge un massimo nell'intervallo $q = 0, 1, \ldots, n - 1$ quando $q = 0$ o $q = n - 1$.

7.4-4
Dimostrate che il tempo di esecuzione atteso di RANDOMIZED-QUICKSORT è $\Omega(n \lg n)$.

7.4-5
Il tempo di esecuzione di quicksort può essere migliorato in pratica sfruttando il fatto che insertion sort viene eseguito rapidamente se il suo input è "quasi" ordinato. Quando quicksort viene chiamato a operare su un sottoarray con meno di k elementi, lasciatelo terminare senza ordinare il sottoarray. Dopo che la chiamata principale di quicksort è completata, eseguite insertion sort sull'intero array per finire il processo di ordinamento. Dimostrate che questo algoritmo di ordinamento viene eseguito nel tempo atteso $O(nk + n \lg(n/k))$. Come dovrebbe essere scelto k in teoria e in pratica?

7.4-6 $\star$
Modificate la procedura PARTITION selezionando casualmente tre elementi dall'array A e partizionando rispetto alla loro mediana (il valore centrale dei tre elementi). Approssimate la probabilità di ottenere nel caso peggiore una ripartizione α-a-$(1 - \alpha)$ come una funzione di α nell'intervallo $0 < \alpha < 1$.

7.5 Problemi

7-1 Correttezza della partizione di Hoare

La versione di PARTITION presentata in questo capitolo non è l'algoritmo originale di partizionamento. La versione originale, che è stata ideata da C. A. R. Hoare, è questa:

HOARE-PARTITION(A, p, r)

```
 1  x ← A[p]
 2  i ← p − 1
 3  j ← r + 1
 4  while TRUE
 5       do repeat j ← j − 1
 6            until A[j] ≤ x
 7          repeat i ← i + 1
 8            until A[i] ≥ x
 9          if i < j
10            then scambia A[i] ↔ A[j]
11            else return j
```

a. Descrivete l'operazione di HOARE-PARTITION sull'array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, indicando i valori dell'array e i valori ausiliari dopo ogni iterazione del ciclo **while** nelle righe 4–11.

Le prossime tre domande richiedono un attento ragionamento sulla correttezza della procedura HOARE-PARTITION. Dimostrate che:

b. Gli indici i e j sono tali che non sarà possibile accedere a un elemento di A esterno al sottoarray $A[p \,..\, r]$.

c. Quando HOARE-PARTITION termina, restituisce un valore di j tale che $p \leq j < r$.

d. Ogni elemento di $A[p\,..\,j]$ è minore o uguale a qualsiasi elemento di $A[j+1\,..\,r]$ quando HOARE-PARTITION termina.

La procedura PARTITION nel Paragrafo 7.1 separa il valore del pivot (originariamente in $A[r]$) dalle due partizioni che forma. La procedura HOARE-PARTITION, d'altra parte, inserisce sempre il valore del pivot (originariamente in $A[p]$) in una delle due partizioni $A[p\,..\,j]$ e $A[j+1\,..\,r]$. Poiché $p \leq j < r$, questa ripartizione non è mai banale.

e. Riscrivete la procedura QUICKSORT per utilizzare HOARE-PARTITION.

7-2 Analisi alternativa di quicksort

Un'analisi alternativa del tempo di esecuzione dell'algoritmo quicksort randomizzato si focalizza sul tempo di esecuzione atteso di ogni singola chiamata ricorsiva di QUICKSORT, anziché sul numero di confronti effettuati.

a. Dimostrate che, dato un array di dimensione n, la probabilità che un particolare elemento sia scelto come pivot è $1/n$. Sulla base di questa dimostrazione definite le variabili casuali indicatrici $X_i = \mathrm{I}\{\text{l'}i\text{-esimo elemento più piccolo è scelto come pivot}\}$. Qual è il valore atteso $\mathrm{E}[X_i]$?

b. Sia $T(n)$ una variabile casuale che indica il tempo di esecuzione di quicksort con un array di dimensione n. Dimostrate che

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] \tag{7.5}$$

c. Dimostrate che l'equazione (7.5) può essere riscritta così:

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}[T(q)] + \Theta(n) \tag{7.6}$$

d. Dimostrate che

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \tag{7.7}$$

(*Suggerimento:* scomponete la sommatoria in due parti, una per $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$ e l'altra per $k = \lceil n/2 \rceil, \ldots, n-1$.)

e. Applicando il limite dell'equazione (7.7), dimostrate che la soluzione della ricorrenza nell'equazione (7.6) è $\mathrm{E}[T(n)] = \Theta(n \lg n)$ (*suggerimento:* dimostrate, per sostituzione, che $\mathrm{E}[T(n)] \leq an \lg n$ per valori di n sufficientemente grandi e per qualche costante positiva a).

7-3 Algoritmo Stooge-Sort

I professori Howard e Fine hanno ideato il seguente "elegante" algoritmo di ordinamento:

STOOGE-SORT(A, i, j)

```
1  if A[i] > A[j]
2     then scambia A[i] ↔ A[j]
3  if i + 1 ≥ j
4     then return
5  k ← ⌊(j − i + 1)/3⌋                ▷ Suddivisione.
6  STOOGE-SORT(A, i, j − k)           ▷ Primi due terzi.
7  STOOGE-SORT(A, i + k, j)           ▷ Ultimi due terzi.
8  STOOGE-SORT(A, i, j − k)           ▷ Ancora i primi due terzi.
```

a. Dimostrate che, se $n = lunghezza[A]$, allora la chiamata STOOGE-SORT$(A, 1, lunghezza[A])$ ordina correttamente l'array di input $A[1 \,..\, n]$.

b. Scrivete una ricorrenza e un limite asintotico stretto (notazione Θ) per il tempo di esecuzione nel caso peggiore di STOOGE-SORT.

c. Confrontate il tempo di esecuzione nel caso peggiore di STOOGE-SORT con quello di insertion sort, merge sort, heapsort e quicksort. Questi professori sono degni del ruolo che occupano?

7-4 Profondità dello stack per quicksort

L'algoritmo QUICKSORT descritto nel Paragrafo 7.1 contiene due chiamate ricorsive a sé stesso. Dopo la chiamata di PARTITION, prima viene ordinato ricorsivamente il sottoarray sinistro e poi quello destro. La seconda chiamata ricorsiva in QUICKSORT non è veramente necessaria; può essere evitata utilizzando una struttura di controllo iterativa. Questa tecnica, detta ***ricorsione in coda***, è fornita automaticamente dai buoni compilatori. Considerate la seguente versione di quicksort, che simula la ricorsione in coda.

QUICKSORT$'(A, p, r)$

```
1  while p < r
2        do ▷ Partiziona e ordina il sottoarray sinistro
3           q ← PARTITION(A, p, r)
4           QUICKSORT'(A, p, q − 1)
5           p ← q + 1
```

a. Dimostrate che QUICKSORT$'(A, 1, lunghezza[A])$ ordina correttamente gli elementi dell'array A.

I compilatori di solito eseguono le procedure ricorsive utilizzando uno ***stack*** (o pila) che contiene informazioni pertinenti a ogni chiamata ricorsiva, inclusi i valori dei parametri. Le informazioni per la chiamata più recente si trovano in cima allo stack, mentre le informazioni per la chiamata iniziale si trovano in fondo allo stack. Quando una procedura viene chiamata, le sue informazioni vengono inserite (***push***) nello stack; quando termina, le sue informazioni vengono estratte (***pop***) dallo stack. Poiché si presuppone che i parametri dell'array siano rappresentati da puntatori, le informazioni per ogni chiamata di procedura sullo stack occupano uno spazio $O(1)$ nello stack. La ***profondità dello stack*** è la quantità massima di spazio utilizzato nello stack in un istante qualsiasi durante un processo.

b. Descrivete uno scenario in cui la profondità dello stack di QUICKSORT′ è $\Theta(n)$ con un array di input di n elementi.

c. Modificate il codice di QUICKSORT′ in modo che lo stack nel caso peggiore sia $\Theta(\lg n)$. Mantenete il tempo di esecuzione atteso $O(n \lg n)$.

7-5 Partizione con la mediana-di-3

Un modo per migliorare la procedura RANDOMIZED-QUICKSORT consiste nell'effettuare il partizionamento scegliendo opportunamente (non casualmente) il pivot nel sottoarray. Un tipico approccio è il metodo della ***mediana-di-3***: scegliere come pivot la mediana (elemento centrale) di un insieme di 3 elementi selezionati a caso dal sottoarray (vedere l'Esercizio 7.4-6). Per questo problema, supponete che gli elementi nell'array di input $A[1\,..\,n]$ siano distinti e che $n \geq 3$. Indicate con $A'[1\,..\,n]$ l'array di output ordinato. Applicando il metodo della mediana-di-3 per scegliere il pivot x, definite $p_i = \Pr\{x = A'[i]\}$.

a. Scrivete una formula esatta di p_i in funzione di n e i per $i = 2, 3, \ldots, n-1$ (notate che $p_1 = p_n = 0$).

b. Di quanto deve essere aumentata la probabilità di scegliere come pivot $x = A'[\lfloor (n+1)/2 \rfloor]$, la mediana di $A[1\,..\,n]$, rispetto all'implementazione normale? Supponete che $n \to \infty$ e indicate il rapporto limite di queste probabilità.

c. Se definiamo "buona" una ripartizione in cui il pivot scelto è $x = A'[i]$, dove $n/3 \leq i \leq 2n/3$, di quanto deve essere aumentata la probabilità di ottenere una buona ripartizione rispetto all'implementazione normale? (*Suggerimento:* approssimate la sommatoria con un integrale.)

d. Dimostrate che nel tempo di esecuzione $\Omega(n \lg n)$ di quicksort il metodo della mediana-di-3 influisce soltanto sul fattore costante.

7-6 Ordinamento fuzzy degli intervalli

Considerate un problema di ordinamento in cui i numeri non sono noti con esattezza. Piuttosto, per ogni numero, conosciamo un intervallo nella retta reale cui appartiene il numero. In altre parole, conosciamo n intervalli chiusi nella forma $[a_i, b_i]$, dove $a_i \leq b_i$. L'obiettivo consiste nell'effettuare un ***ordinamento fuzzy*** di questi intervalli, cioè produrre una permutazione $\langle i_1, i_2, \ldots, i_n \rangle$ degli intervalli in modo che esista un valore $c_j \in [a_{i_j}, b_{i_j}]$ che soddisfa $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Progettate un algoritmo che effettua un ordinamento fuzzy di n intervalli. L'algoritmo deve avere la struttura generale di quicksort che ordina rapidamente gli estremi sinistri degli intervalli (i termini a_i), ma deve sfruttare la sovrapposizione degli intervalli per migliorare il tempo di esecuzione (all'aumentare della sovrapposizione degli intervalli, il problema dell'ordinamento fuzzy degli intervalli diventa sempre più facile).

b. Dimostrate che, in generale, il tempo di esecuzione atteso del vostro algoritmo è $\Theta(n \lg n)$; questo tempo, però, diventa $\Theta(n)$ quando tutti gli intervalli si sovrappongono (cioè quando esiste un valore x tale che $x \in [a_i, b_i]$ per ogni i). Il vostro algoritmo non deve essere esplicitamente verificato per questo caso speciale; piuttosto, le sue prestazioni dovrebbero migliorare naturalmente all'aumentare dell'entità della sovrapposizione degli intervalli.

Note

La procedura quicksort è stata inventata da Hoare [147]; la versione di Hoare è riportata nel Problema 7-1. La procedura PARTITION presentata nel Paragrafo 7.1 è stata ideata da N. Lomuto. L'analisi svolta nel Paragrafo 7.4 è stata sviluppata da Avrim Blum. Consultate Sedgewick [268] e Bentley [40] per informazioni dettagliate sulle tecniche di implementazione di queste procedure.

McIlroy [216] ha descritto come progettare un "killer adversary" che produce un array con il quale virtualmente qualsiasi implementazione di quicksort impiega un tempo $\Theta(n^2)$. Se l'implementazione è randomizzata, killer adversary genera l'array dopo avere esaminato le scelte casuali dell'algoritmo quicksort.

8 Ordinamento in tempo lineare

Abbiamo già introdotto parecchi algoritmi che possono ordinare n numeri nel tempo $O(n \lg n)$. Heapsort e merge sort raggiungono questo limite superiore nel caso peggiore; quicksort lo raggiunge nel caso medio. Inoltre, per ciascuno di questi algoritmi, possiamo produrre una sequenza di n numeri di input tale che l'algoritmo possa essere eseguito nel tempo $\Omega(n \lg n)$.

Questi algoritmi condividono un'interessante proprietà: *l'ordinamento che effettuano è basato soltanto su confronti fra gli elementi di input*. Questi algoritmi sono detti ***ordinamenti per confronti***. Tutti gli algoritmi di ordinamento presentati finora sono ordinamenti per confronti.

Nel Paragrafo 8.1 dimostreremo che qualsiasi ordinamento per confronti deve effettuare $\Omega(n \lg n)$ confronti nel caso peggiore per ordinare n elementi. Quindi, merge sort e heapsort sono asintoticamente ottimali e non esiste un ordinamento per confronti che è più veloce per oltre un fattore costante.

I Paragrafi 8.2, 8.3 e 8.4 esaminano tre algoritmi di ordinamento – counting sort, radix sort e bucket sort – che vengono eseguiti in tempo lineare. È inutile dire che questi algoritmi usano operazioni diverse dai confronti per effettuare l'ordinamento. Di conseguenza, il limite inferiore $\Omega(n \lg n)$ non può essere applicato a questi algoritmi.

8.1 Limiti inferiori per l'ordinamento

In un ordinamento per confronti usiamo soltanto i confronti fra gli elementi per ottenere informazioni sull'ordinamento di una sequenza di input $\langle a_1, a_2, \ldots, a_n \rangle$. Ovvero, dati due elementi a_i e a_j, svolgiamo uno dei test $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ o $a_i > a_j$ per determinare il loro ordine relativo. Non possiamo esaminare i valori degli elementi o ottenere informazioni sul loro ordine in altri modi.

In questo paragrafo supponiamo, senza perdere di generalità, che tutti gli elementi di input siano distinti. Fatta questa ipotesi, confronti della forma $a_i = a_j$ sono inutili, quindi possiamo supporre che non saranno fatti confronti di questo tipo. Inoltre notate che i confronti $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ e $a_i < a_j$ sono tutti equivalenti, perché forniscono le stesse informazioni sull'ordine relativo di a_i e a_j. Quindi, supporremo che tutti i confronti abbiano la forma $a_i \leq a_j$.

Il modello dell'albero di decisione

Gli ordinamenti per confronti possono essere visti astrattamente in termini di ***alberi di decisione***. Un albero di decisione è un albero binario completo che rappresenta i confronti fra elementi che vengono effettuati da un particolare algori-

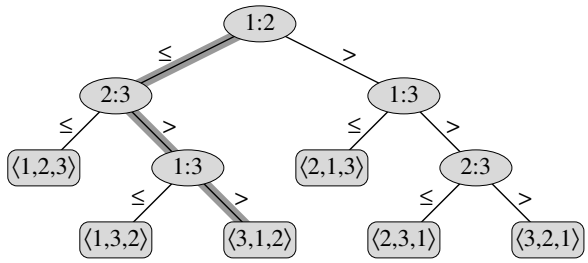


Figura 8.1 L'albero di decisione per l'algoritmo insertion sort che opera su tre elementi. Un nodo interno rappresentato da $i{:}j$ indica un confronto fra a_i e a_j. Una foglia rappresentata dalla permutazione $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indica l'ordinamento $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Il percorso ombreggiato indica le decisioni prese durante l'ordinamento della sequenza di input $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; la permutazione $\langle 3, 1, 2 \rangle$ nella foglia indica che l'ordinamento è $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. Ci sono $3! = 6$ permutazioni possibili degli elementi di input, quindi l'albero di decisione deve avere almeno 6 foglie.

tmo di ordinamento che opera su un input di una data dimensione. Il controllo, lo spostamento dei dati e tutti gli altri aspetti dell'algoritmo vengono ignorati. La Figura 8.1 illustra l'albero di decisione che corrisponde all'algoritmo insertion sort (descritto nel Paragrafo 2.1) che opera su una sequenza di input di tre elementi.

In un albero di decisione, ogni nodo interno è rappresentato da $i{:}j$ per qualche i e j nell'intervallo $1 \leq i, j \leq n$, dove n è il numero di elementi nella sequenza di input. Ogni foglia è rappresentata da una permutazione $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ (consultare il Paragrafo C.1 per informazioni sulle permutazioni). L'esecuzione dell'algoritmo di ordinamento corrisponde a tracciare un percorso dalla radice dell'albero di decisione fino a una foglia. In ogni nodo interno, viene effettuato un confronto $a_i \leq a_j$. Il sottoalbero sinistro detta i successivi confronti per $a_i \leq a_j$; il sottoalbero destro detta i successivi confronti per $a_i > a_j$.

Quando raggiunge una foglia, l'algoritmo ha stabilito l'ordinamento $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Poiché qualsiasi algoritmo di ordinamento corretto deve essere in grado di produrre ogni permutazione del suo input, una condizione necessaria affinché un ordinamento per confronti sia corretto è che ciascuna delle $n!$ permutazioni di n elementi deve apparire come una delle foglie dell'albero di decisione e che ciascuna di queste foglie deve essere raggiungibile dalla radice attraverso un percorso che corrisponde a una effettiva esecuzione dell'ordinamento per confronti (queste foglie saranno chiamate "raggiungibili"). Quindi, considereremo soltanto alberi di decisione in cui ogni permutazione si presenta come una foglia raggiungibile.

Un limite inferiore per il caso peggiore

La lunghezza del percorso più lungo dalla radice di un albero di decisione a una delle sue foglie raggiungibili rappresenta il numero di confronti che svolge il corrispondente algoritmo di ordinamento nel caso peggiore. Di conseguenza, il numero di confronti nel caso peggiore per un dato algoritmo di ordinamento per confronti è uguale all'altezza del suo albero di decisione. Un limite inferiore sulle altezze di tutti gli alberi di decisione, dove ogni permutazione si presenta come una foglia raggiungibile, è pertanto un limite inferiore sul tempo di esecuzione di qualsiasi algoritmo di ordinamento per confronti. Il seguente teorema stabilisce questo limite inferiore.

Teorema 8.1
Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \lg n)$ confronti nel caso peggiore.

Dimostrazione Per quanto detto in precedenza, è sufficiente determinare l'altezza di un albero di decisione dove ogni permutazione appare come una foglia

raggiungibile. Considerate un albero di decisione di altezza h con l foglie raggiungibili che corrisponde a un ordinamento per confronti di n elementi. Poiché ciascuna delle $n!$ permutazioni dell'input si presenta come una foglia, si ha $n! \leq l$. Dal momento che un albero binario di altezza h non ha più di 2^h foglie, si ha

$n! \leq l \leq 2^h$

Prendendo i logaritmi, questa relazione implica che

$$\begin{aligned} h &\geq \lg(n!) && \text{(perché la funzione lg è monotonicamente crescente)} \\ &= \Omega(n \lg n) && \text{(per l'equazione (3.18))} \end{aligned}$$

■

Corollario 8.2
Heapsort e merge sort sono algoritmi di ordinamento per confronti asintoticamente ottimali.

Dimostrazione I limiti superiori $O(n \lg n)$ sui tempi di esecuzione per heapsort e merge sort corrispondono al limite inferiore $\Omega(n \lg n)$ nel caso peggiore del Teorema 8.1. ■

Esercizi

8.1-1
Qual è la profondità minima possibile di una foglia in un albero di decisione per un ordinamento per confronti?

8.1-2
Determinate i limiti asintoticamente stretti su $\lg(n!)$ senza applicare l'approssimazione di Stirling; piuttosto, calcolate la sommatoria $\sum_{k=1}^{n} \lg k$ adottando le tecniche descritte nel Paragrafo A.2.

8.1-3
Dimostrate che non esiste un ordinamento per confronti il cui tempo di esecuzione è lineare per almeno metà degli $n!$ input di lunghezza n. Che cosa accade per una frazione $1/n$ degli input di lunghezza n? Che cosa accade per una frazione $1/2^n$?

8.1-4
Sia data una sequenza di n elementi da ordinare. La sequenza di input è formata da n/k sottosequenze, ciascuna di k elementi. Gli elementi di una sottosequenza sono tutti più piccoli degli elementi della successiva sottosequenza e più grandi degli elementi della precedente sottosequenza. Quindi, per ordinare l'intera sequenza di lunghezza n basta ordinare i k elementi in ciascuna delle n/k sottosequenze. Dimostrate che $\Omega(n \lg k)$ è un limite inferiore sul numero di confronti necessari a risolvere questa variante del problema di ordinamento (*suggerimento:* non è corretto combinare semplicemente i limiti inferiori delle singole sottosequenze).

8.2 Counting sort

L'algoritmo ***counting sort*** suppone che ciascuno degli n elementi di input sia un numero intero compreso nell'intervallo da 0 a k, per qualche intero k. Quando $k = O(n)$, l'ordinamento viene effettuato nel tempo $\Theta(n)$.

Il concetto che sta alla base di counting sort è determinare, per ogni elemento di input x, il numero di elementi minori di x. Questa informazione può essere

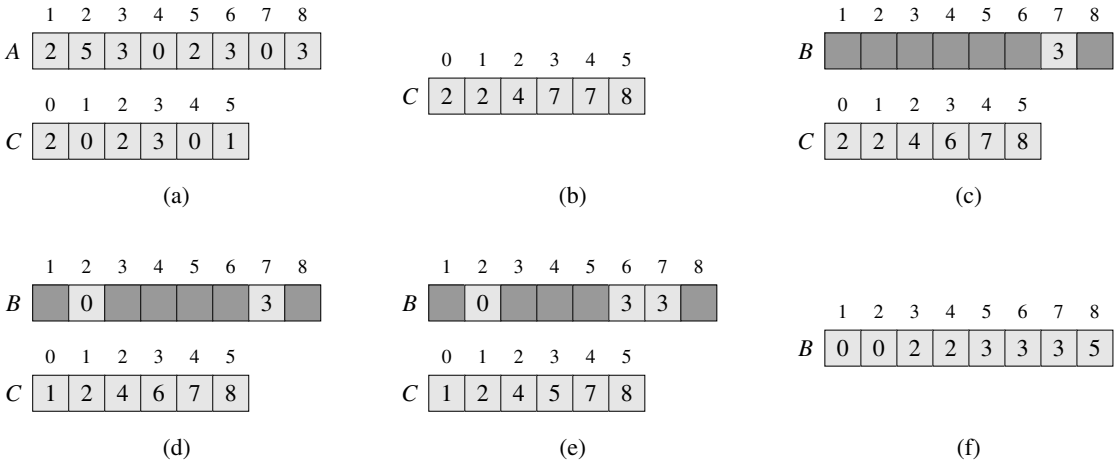


Figura 8.2 L'operazione di COUNTING-SORT su un array di input $A[1\,..\,8]$, dove ogni elemento di A è un intero non negativo non maggiore di $k = 5$. **(a)** L'array A e l'array ausiliario C dopo la riga 4. **(b)** L'array C dopo la riga 7. **(c)–(e)** L'array di output B e l'array ausiliario C, rispettivamente, dopo una, due e tre iterazioni del ciclo (righe 9–11). Le caselle di colore grigio chiaro nell'array B rappresentano gli elementi che sono stati inseriti. **(f)** L'array di output B è ordinato.

utilizzata per inserire l'elemento x direttamente nella sua posizione nell'array di output. Per esempio, se ci sono 17 elementi minori di x, allora x appartiene alla posizione di output 18. Questo schema deve essere modificato leggermente per gestire il caso in cui più elementi hanno lo stesso valore, per evitare che siano inseriti nella stessa posizione.

Nel codice di counting sort, supponiamo che l'input sia un array $A[1\,..\,n]$, quindi $\mathit{lunghezza}[A] = n$. Occorrono altri due array: l'array $B[1\,..\,n]$ contiene l'output ordinato; l'array $C[0\,..\,k]$ fornisce la memoria temporanea di lavoro.

COUNTING-SORT(A, B, k)

```
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to lunghezza[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] adesso contiene il numero di elementi uguale a i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] adesso contiene il numero di elementi minore o uguale a i.
 9  for j ← lunghezza[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

La Figura 8.2 illustra counting sort. Dopo l'inizializzazione nelle righe 1–2 del ciclo **for**, ogni elemento di input viene esaminato nelle righe 3–4 del ciclo **for**. Se il valore di un elemento di input è i, incrementiamo $C[i]$. Quindi, dopo la riga 4, $C[i]$ contiene il numero degli elementi di input uguali a i per ogni intero $i = 0, 1, \ldots, k$. Le righe 6–7 determinano, per ogni $i = 0, 1, \ldots, k$, quanti elementi di input sono minori o uguali a i, mantenendo la somma corrente dell'array C.

Infine, le righe 9–11 del ciclo **for** inseriscono l'elemento $A[j]$ nella corretta posizione ordinata dell'array di output B. Se tutti gli n elementi sono distinti, quando viene eseguita per la prima volta la riga 9, per ogni $A[j]$, il valore $C[A[j]]$ rappresenta la posizione finale corretta di $A[j]$ nell'array di output, in quanto ci sono $C[A[j]]$ elementi minori o uguali ad $A[j]$. Poiché gli elementi potrebbero non essere distinti, $C[A[j]]$ viene ridotto ogni volta che viene inserito un valore $A[j]$ nell'array B. La riduzione di $C[A[j]]$ fa sì che il successivo elemento di input con un valore uguale ad $A[j]$, se esiste, venga inserito nella posizione immediatamente prima di $A[j]$ nell'array di output.

Quanto tempo richiede counting sort? Le righe 1–2 del ciclo **for** impiegano un tempo $\Theta(k)$, le righe 3–4 del ciclo **for** impiegano un tempo $\Theta(n)$, le righe 6–7 del ciclo **for** impiegano un tempo $\Theta(k)$ e le righe 9–11 del ciclo **for** impiegano un tempo $\Theta(n)$. Quindi, il tempo totale è $\Theta(k + n)$. Di solito counting sort viene utilizzato quando $k = O(n)$, nel qual caso il tempo di esecuzione è $\Theta(n)$.

Counting sort batte il limite inferiore di $\Omega(n \lg n)$ dimostrato nel Paragrafo 8.1 perché non è un ordinamento per confronti. Infatti, il codice non effettua alcun confronto fra gli elementi di input. Piuttosto, counting sort usa i valori effettivi degli elementi come indici di un array. Il limite inferiore $\Omega(n \lg n)$ non vale se ci allontaniamo dal modello di ordinamento per confronti.

Un'importante proprietà di counting sort è la ***stabilità***: i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input. Ovvero, i pareggi fra due numeri vengono risolti applicando la seguente regola: il numero che si presenta per primo nell'array di input sarà inserito per primo nell'array di output. Normalmente, la proprietà della stabilità è importante soltanto quando i dati satellite vengono spostati insieme con gli elementi da ordinare. La stabilità di counting sort è importante per un'altra ragione: counting sort viene spesso utilizzato come subroutine di radix sort. Come vedremo nel prossimo paragrafo, la stabilità di counting sort è cruciale per la correttezza di radix sort.

Esercizi

8.2-1

Utilizzando la Figura 8.2 come modello, illustrate l'operazione di COUNTING-SORT sull'array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Dimostrate che COUNTING-SORT è stabile.

8.2-3

Supponete che la riga 9 del ciclo **for** nella procedura COUNTING-SORT sia modificata così

9 **for** $j \leftarrow 1$ **to** $lunghezza[A]$

Dimostrate che l'algoritmo opera ancora correttamente. L'algoritmo modificato è stabile?

8.2-4

Descrivete un algoritmo che, dati n numeri interi compresi nell'intervallo da 0 a k, svolga un'analisi preliminare del suo input e poi risponda nel tempo $O(1)$ a qualsiasi domanda su quanti degli n interi ricadono nell'intervallo $[a \,..\, b]$. Il vostro algoritmo dovrebbe impiegare un tempo $\Theta(n + k)$ per l'analisi preliminare.

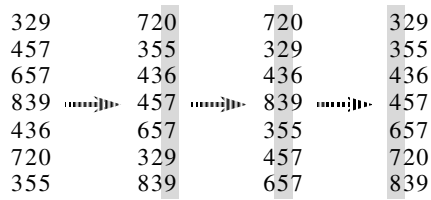


Figura 8.3 L'operazione di radix sort su una lista di sette numeri di 3 cifre. La prima colonna a sinistra è l'input. La altre colonne mostrano la lista dopo i successivi ordinamenti in funzione di posizioni con cifre significative crescenti. L'ombreggiatura indica la posizione della cifra sulla quale viene effettuato l'ordinamento per generare una lista dalla precedente.

8.3 Radix sort

Radix sort è l'algoritmo utilizzato dalle macchine per ordinare le schede perforate, che adesso si trovano soltanto nei musei di calcolatori. Le schede sono composte da 80 colonne e ogni colonna può essere perforata in una delle 12 posizioni disponibili. La macchina ordinatrice può essere meccanicamente "programmata" per esaminare una particolare colonna di ogni scheda di un mazzo e poi distribuire le singole schede in uno dei 12 contenitori in funzione della posizione perforata. Un operatore può così raccogliere le schede nei vari contenitori, in modo che le schede perforate nella prima posizione si trovino sopra quelle perforate nella seconda posizione e così via.

Per le cifre decimali, sono utilizzate soltanto 10 posizioni in ogni colonna (le altre due posizioni sono utilizzate per codificare i caratteri non numerici). Un numero di d cifre occupa un campo di d colonne. Poiché la macchina ordinatrice può esaminare una sola colonna alla volta, il problema di ordinare n schede in funzione di un numero di d cifre richiede un algoritmo di ordinamento.

Intuitivamente, vorremmo ordinare i numeri in base alla loro cifra *più significativa*, ordinare ciascuno dei contenitori risultanti in modo ricorsivo e, poi, combinare ordinatamente i mazzi delle schede. Purtroppo, poiché le schede in 9 dei 10 contenitori devono essere messe da parte per ordinare i singoli contenitori, questa procedura genera molte pile intermedie di schede di cui bisogna tenere traccia (vedere l'Esercizio 8.3-5).

Radix sort risolve il problema dell'ordinamento delle schede in una maniera contraria all'intuizione, ordinando prima le schede in base alla cifra *meno significativa*. Le schede vengono poi combinate in un unico mazzo: le schede del contenitore 0 precedono quelle del contenitore 1 che, a loro volta, precedono quelle del contenitore 2 e così via. Poi tutto il mazzo viene ordinato di nuovo in funzione della seconda cifra meno significativa e ricombinato in maniera analoga. Il processo continua finché le schede saranno ordinate in tutte le d cifre. È importante notare che a questo punto le schede sono completamente ordinate sul numero di d cifre. Quindi, occorrono soltanto d passaggi attraverso il mazzo per completare l'ordinamento. La Figura 8.3 mostra come opera radix sort con un "mazzo" di sette numeri di 3 cifre.

È essenziale che gli ordinamenti delle cifre in questo algoritmo siano stabili. L'ordinamento svolto da una macchina ordinatrice di schede è stabile, ma l'operatore deve stare attento a non cambiare l'ordine delle schede mentre escono da un contenitore, anche se tutte le schede in un contenitore hanno la stessa cifra nella colonna scelta.

In un tipico calcolatore, che è una macchina sequenziale ad accesso casuale, radix sort viene talvolta utilizzato per ordinare record di informazioni con più campi chiave. Un tipico esempio è l'ordinamento delle date in base a tre chiavi:

anno, mese e giorno. Potremmo eseguire un algoritmo di ordinamento con una funzione di confronto che opera su due date così: confronta gli anni e, nel caso di pareggio, confronta i mesi; nel caso di un altro pareggio, confronta i giorni. In alternativa, potremmo ordinare le informazioni tre volte con un ordinamento stabile: prima in base al giorno, poi in base al mese e, infine, in base all'anno.

Il codice per radix sort è semplice. La seguente procedura suppone che ogni elemento nell'array A di n elementi abbia d cifre, dove la cifra 1 è quella di ordine più basso e la cifra d è quella di ordine più alto.

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do usa un ordinamento stabile per ordinare l'array A sulla cifra i
```

Lemma 8.3
Dati n numeri di d cifre, dove ogni cifra può avere fino a k valori possibili, la procedura RADIX-SORT ordina correttamente i numeri nel tempo $\Theta(d(n+k))$, se l'ordinamento stabile utilizzato dalla procedura impiega un tempo $\Theta(n+k)$.

Dimostrazione La correttezza di radix sort si dimostra per induzione sulla colonna da ordinare (vedere l'Esercizio 8.3-3). L'analisi del tempo di esecuzione dipende dall'ordinamento stabile che viene utilizzato come algoritmo di ordinamento intermedio. Se ogni cifra si trova nell'intervallo da 0 a $k-1$ (in modo che possa assumere i k valori possibili) e k non è troppo grande, counting sort è la scelta ovvia da fare. Ogni passaggio su n numeri di d cifre richiede un tempo $\Theta(n+k)$. Poiché ci sono d passaggi, il tempo totale di radix sort è $\Theta(d(n+k))$. ■

Quando d è costante e $k = O(n)$, radix sort viene eseguito in tempo lineare. Più in generale, abbiamo una certa flessibilità sul modo in cui ripartire le singole chiavi in cifre.

Lemma 8.4
Dati n numeri di b bit e un intero positivo $r \leq b$, RADIX-SORT ordina correttamente questi numeri nel tempo $\Theta((b/r)(n+2^r))$.

Dimostrazione Per un valore $r \leq b$, consideriamo ogni chiave come se avesse $d = \lceil b/r \rceil$ cifre di r bit ciascuna. Ogni cifra è un numero intero compreso nell'intervallo da 0 a $2^r - 1$, quindi possiamo utilizzare counting sort con $k = 2^r - 1$ (per esempio, possiamo considerare una parola di 32 bit come se avesse 4 cifre di 8 bit; quindi $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ e $d = b/r = 4$). Ogni passaggio di counting sort richiede il tempo $\Theta(n+k) = \Theta(n+2^r)$; poiché ci sono d passaggi, il tempo di esecuzione totale è $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$. ■

Dati i valori di n e b, scegliamo il valore di r, con $r \leq b$, che rende minima l'espressione $(b/r)(n+2^r)$. Se $b < \lfloor \lg n \rfloor$, allora per qualsiasi valore di $r \leq b$, si ha $(n+2^r) = \Theta(n)$. Quindi, scegliendo $r = b$, si ottiene un tempo di esecuzione $(b/b)(n+2^b) = \Theta(n)$, che è asintoticamente ottimale. Se $b \geq \lfloor \lg n \rfloor$, allora scegliendo $r = \lfloor \lg n \rfloor$ si ottiene il tempo migliore a meno di un fattore costante, che possiamo verificare nel modo seguente. Scegliendo $r = \lfloor \lg n \rfloor$, si ha un tempo di esecuzione pari a $\Theta(bn/\lg n)$. Aumentando r oltre $\lfloor \lg n \rfloor$, il termine 2^r nel numeratore cresce più rapidamente del termine r nel denominatore; quindi, aumentando r oltre $\lfloor \lg n \rfloor$, si ottiene un tempo di esecuzione pari a $\Omega(bn/\lg n)$.

Se invece r diminuisse sotto $\lfloor \lg n \rfloor$, allora il termine b/r crescerebbe e il termine $n + 2^r$ resterebbe a $\Theta(n)$.

Radix sort è preferibile a un algoritmo di ordinamento basato sui confronti, come quicksort? Se $b = O(\lg n)$, come spesso accade, e scegliamo $r \approx \lg n$, allora il tempo di esecuzione di radix sort è $\Theta(n)$, che sembra migliore di $\Theta(n \lg n)$, il tempo di esecuzione nel caso medio di quicksort. Notiamo però che i fattori costanti nascosti nella notazione Θ sono differenti. Sebbene radix sort richieda meno passaggi di quicksort sulle n chiavi, tuttavia ogni passaggio di radix sort potrebbe richiedere un tempo significativamente più lungo. La scelta dell'algoritmo di ordinamento ottimale dipende dalle caratteristiche delle implementazioni, della macchina (per esempio, quicksort spesso usa le cache hardware in modo più efficiente di radix sort) e dei dati di input. Inoltre, la versione di radix sort che usa counting sort come ordinamento stabile intermedio non effettua l'ordinamento sul posto, come fanno molti degli ordinamenti per confronti con tempo $\Theta(n \lg n)$. Quindi, se lo spazio nella memoria principale è limitato, potrebbe essere preferibile un algoritmo di ordinamento sul posto come quicksort.

Esercizi

8.3-1

Utilizzando la Figura 8.3 come modello, illustrate l'operazione di RADIX-SORT con la seguente lista di parole inglesi: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2

Quali dei seguenti algoritmi di ordinamento sono stabili: insertion sort, merge sort, heapsort e quicksort? Create un semplice schema che renda stabile qualsiasi algoritmo di ordinamento. Quanto tempo e spazio in più richiede il vostro schema?

8.3-3

Utilizzate l'induzione per dimostrare che radix sort funziona correttamente. In quale punto la vostra dimostrazione richiede l'ipotesi che l'ordinamento intermedio sia stabile?

8.3-4

Dimostrate come ordinare n numeri interi compresi nell'intervallo da 0 a $n^2 - 1$ nel tempo $O(n)$.

8.3-5 $\star$

Nel primo algoritmo di ordinamento delle schede presentato in questo paragrafo, esattamente quanti passaggi sono richiesti per ordinare i numeri di d cifre nel caso peggiore? Di quante pile di schede dovrebbe tenere traccia un operatore nel caso peggiore?

8.4 Bucket sort

Il tempo di esecuzione atteso di ***bucket sort*** è lineare quando l'input è estratto da una distribuzione uniforme. Come counting sort, bucket sort è veloce perché fa un'ipotesi sull'input. Mentre counting sort suppone che l'input sia formato da interi in un piccolo intervallo, bucket sort suppone che l'input sia generato da un processo casuale che distribuisce gli elementi uniformemente nell'intervallo $[0, 1)$ (la distribuzione uniforme è definita nel Paragrafo C.2).

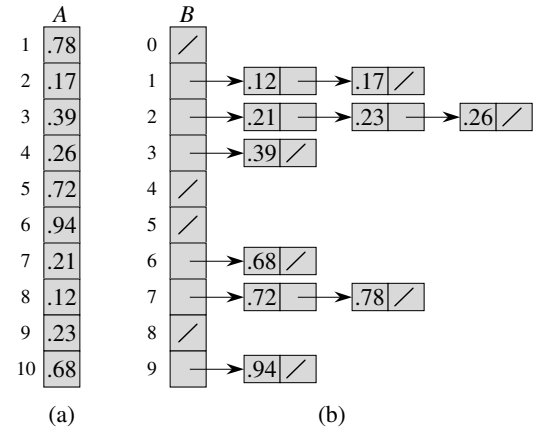


Figura 8.4
Il funzionamento di BUCKET-SORT.
(a) L'array di input $A[1\,..\,10]$.
(b) L'array $B[0\,..\,9]$ delle liste ordinate (bucket) dopo l'esecuzione della riga 5 dell'algoritmo. Il bucket i contiene i valori nell'intervallo semiaperto $[i/10, (i+1)/10)$. L'output ordinato è formato da una concatenazione ordinata delle seguenti liste: $B[0], B[1], \ldots, B[9]$.

Il concetto che sta alla base di bucket sort è quello di dividere l'intervalllo $[0, 1)$ in n sottointervalli della stessa dimensione, detti ***bucket***, e poi nel distribuire gli n numeri di input nei bucket. Poiché gli input sono uniformemente distribuiti nell'intervallo $[0, 1)$, non prevediamo che in ogni bucket ricadano molti numeri. Per produrre l'output, semplicemente ordiniamo i numeri in ogni bucket e poi esaminiamo ordinatamente i bucket, elencando gli elementi in ciascuno di essi.

Il nostro codice per bucket sort suppone che l'input sia un array A di n elementi e che ogni elemento $A[i]$ dell'array soddisfi la relazione $0 \leq A[i] < 1$. Il codice richiede un array ausiliario $B[0\,..\,n-1]$ di liste concatenate (bucket) e suppone che ci sia un meccanismo per mantenere tali liste (il Paragrafo 10.2 descrive come implementare le operazioni elementari con le liste concatenate).

BUCKET-SORT(A)
1 $n \leftarrow lunghezza[A]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** inserisci $A[i]$ nella lista $B[\lfloor nA[i] \rfloor]$
4 **for** $i \leftarrow 0$ **to** $n-1$
5 **do** ordina la lista $B[i]$ con insertion sort
6 concatena ordinatamente le liste $B[0], B[1], \ldots, B[n-1]$

La Figura 8.4 illustra l'operazione di bucket sort su un array di input di 10 numeri.

Per dimostrare che questo algoritmo funziona correttamente, consideriamo due elementi $A[i]$ e $A[j]$. Supponiamo, senza perdere di generalità, che $A[i] \leq A[j]$. Poiché $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, l'elemento $A[i]$ viene posto nello stesso bucket di $A[j]$ o in un bucket con un indice minore. Se $A[i]$ e $A[j]$ vengono posti nello stesso bucket, allora le righe 4–5 del ciclo **for** li inseriscono nell'ordine appropriato. Se $A[i]$ e $A[j]$ vengono posti in bucket differenti, allora la riga 6 li inserisce nell'ordine appropriato. Dunque, bucket sort funziona correttamente.

Per analizzare il tempo di esecuzione, notiamo che tutte le righe, tranne la 5, richiedono un tempo $O(n)$ nel caso peggiore. Resta da valutare il tempo totale richiesto dalle n chiamate di insertion sort nella riga 5.

Per analizzare il costo delle chiamate di insertion sort, indichiamo con n_i la variabile casuale che rappresenta il numero degli elementi che vengono inseriti nel bucket $B[i]$. Poiché insertion sort viene eseguito in un tempo quadratico (Paragrafo 2.2), il tempo di esecuzione di bucket sort è

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

Prendendo i valori attesi da entrambi i lati e applicando la linearità del valore atteso, si ha

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \quad \text{(per la linearità del valore atteso)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}[n_i^2]\right) \quad \text{(per l'equazione (C.21))} \qquad (8.1)
\end{aligned}$$

Noi affermiamo che

$$\mathrm{E}[n_i^2] = 2 - 1/n \qquad (8.2)$$

per $i = 0, 1, \ldots, n-1$. Non sorprende il fatto che ogni bucket i abbia lo stesso valore $\mathrm{E}[n_i^2]$, in quanto ogni valore nell'array di input A ha la stessa probabilità di ricadere in qualsiasi bucket. Per dimostrare l'equazione (8.2), definiamo le variabili casuali indicatrici

$X_{ij} = \mathrm{I}\{A[j] \text{ ricade nel bucket } i\}$

per $i = 0, 1, \ldots, n-1$ e $j = 1, 2, \ldots, n$. Quindi

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Per calcolare $\mathrm{E}[n_i^2]$, espandiamo il quadrato e raggruppiamo i termini:

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}] \qquad (8.3)
\end{aligned}$$

L'ultima riga deriva dalla linearità del valore atteso. Calcoliamo le due sommatorie separatamente. La variabile casuale indicatrice X_{ij} vale 1 con probabilità $1/n$ e 0 negli altri casi, quindi

$$\begin{aligned}
\mathrm{E}[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\
&= \frac{1}{n}
\end{aligned}$$

Quando $k \neq j$, le variabili X_{ij} e X_{ik} sono indipendenti, quindi

$$\begin{aligned} \mathrm{E}\left[X_{ij}X_{ik}\right] &= \mathrm{E}\left[X_{ij}\right]\mathrm{E}\left[X_{ik}\right] \\ &= \frac{1}{n}\cdot\frac{1}{n} \\ &= \frac{1}{n^2} \end{aligned}$$

Sostituendo questi due valori attesi nell'equazione (8.3), si ha

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n}\frac{1}{n} + \sum_{1\le j\le n}\sum_{\substack{1\le k\le n \\ k\neq j}}\frac{1}{n^2} \\ &= n\cdot\frac{1}{n} + n(n-1)\cdot\frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n} \end{aligned}$$

che dimostra l'equazione (8.2).

Utilizzando questo valore atteso nell'equazione (8.1), concludiamo che il tempo atteso di bucket sort e $\Theta(n) + n\cdot O(2-1/n) = \Theta(n)$. Quindi, l'intero algoritmo bucket sort viene eseguito nel tempo atteso lineare.

Anche se l'input non proviene da una distribuzione uniforme, bucket sort può essere ancora eseguito in tempo lineare. Finché l'input ha la proprietà che la somma dei quadrati delle dimensioni dei bucket è lineare nel numero totale degli elementi, l'equazione (8.1) ci dice che bucket sort sarà eseguito in tempo lineare.

Esercizi

8.4-1

Utilizzando la Figura 8.4 come modello, illustrate l'operazione di BUCKET-SORT sull'array $A = \langle .79,\ .13,\ .16,\ .64,\ .39,\ .20,\ .89,\ .53,\ .71,\ .42\rangle$.

8.4-2

Qual è il tempo di esecuzione nel caso peggiore dell'algoritmo bucket sort? Quale semplice modifica dell'algoritmo preserva il suo tempo di esecuzione atteso lineare e rende il suo tempo di esecuzione nel caso peggiore pari a $O(n \lg n)$?

8.4-3 $\star$

Una ***funzione di distribuzione di probabilità*** $P(x)$ di una variabile casuale X è definita da $P(x) = \Pr\{X \le x\}$. Supponete che una lista di n variabili casuali $X_1, X_2, \ldots, X_n$ venga estratta da una funzione continua di distribuzione di probabilità P che è calcolabile nel tempo $O(1)$. Spiegate come ordinare questi numeri nel tempo atteso lineare.

Problemi

8-1 Limiti inferiori sull'ordinamento per confronti nel caso medio

In questo problema dimostriamo un limite inferiore $\Omega(n \lg n)$ sul tempo di esecuzione atteso di qualsiasi ordinamento per confronti deterministico o randomizzato con n elementi distinti di input. Iniziamo a esaminare un ordinamento per

confronti deterministico A con albero di decisione T_A. Supponiamo che ogni permutazione degli input di A sia ugualmente probabile.

a. Supponiamo che ogni foglia di T_A sia etichettata con la probabilità di essere raggiunta, dato un input casuale. Dimostrate che esattamente $n!$ foglie sono etichettate con $1/n!$ e le restanti sono etichettate con 0.

b. Sia $D(T)$ la lunghezza del cammino esterno di un albero di decisione T; cioè $D(T)$ è la somma delle profondità di tutte le foglie di T. Sia T un albero di decisione con $k > 1$ foglie; siano LT e RT i sottoalberi sinistro e destro di T. Dimostrate che $D(T) = D(LT) + D(RT) + k$.

c. Sia $d(k)$ il valore minimo di $D(T)$ in tutti gli alberi di decisione T con $k > 1$ foglie. Dimostrate che $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$ (*suggerimento:* considerate un albero di decisione T con k foglie che raggiunge il minimo. Indicate con i_0 il numero di foglie in LT e con $k - i_0$ il numero di foglie in RT).

d. Dimostrate che, per un dato valore di $k > 1$ e i nell'intervallo $1 \le i \le k-1$, la funzione $i \lg i + (k-i) \lg(k-i)$ è minima per $i = k/2$. Concludete che $d(k) = \Omega(k \lg k)$.

e. Dimostrate che $D(T_A) = \Omega(n! \lg(n!))$ e concludete che il tempo atteso per ordinare n elementi è $\Omega(n \lg n)$.

Adesso considerate un ordinamento per confronti *randomizzato* B. Possiamo estendere il modello dell'albero di decisione per gestire la randomizzazione incorporando due tipi di nodi: il nodo di confronto ordinario e il nodo di "randomizzazione". Un nodo di randomizzazione modella una scelta casuale della forma $\text{RANDOM}(1, r)$ fatta dall'algoritmo B; il nodo ha r figli, ciascuno dei quali ha la stessa probabilità di essere scelto durante un'esecuzione dell'algoritmo.

f. Dimostrate che, per qualsiasi ordinamento per confronti randomizzato B, esiste un ordinamento per confronti deterministico A che, in media, non effettua più confronti di B.

8-2 Ordinamento sul posto in tempo lineare

Supponete di avere un array di n record di dati da ordinare e che la chiave di ogni record abbia il valore 0 o 1. Un algoritmo che ordina questo insieme di record potrebbe possedere alcune delle tre seguenti caratteristiche desiderabili:

1. L'algoritmo viene eseguito nel tempo $O(n)$.
2. L'algoritmo è stabile.
3. L'algoritmo ordina sul posto, utilizzando non più di una quantità costante di memoria oltre all'array originale.

a. Trovate un algoritmo che soddisfa i criteri 1 e 2.

b. Trovate un algoritmo che soddisfa i criteri 1 e 3.

c. Trovate un algoritmo che soddisfa i criteri 2 e 3.

d. Uno dei vostri algoritmi specificati nei punti (a)–(c) può essere utilizzato per ordinare n record con chiavi di b bit mediante radix sort nel tempo $O(bn)$? Spiegate come o perché no.

e. Supponete che gli n record abbiano le chiavi nell'intervallo da 1 a k. Spiegate come modificare counting sort in modo che i record possano essere ordinati sul posto nel tempo $O(n+k)$. Potete utilizzare una quantità $O(k)$ di memoria oltre all'array di input. Il vostro algoritmo è stabile? (*Suggerimento:* Come fareste con $k = 3$?)

8-3 Ordinare elementi di lunghezza variabile

a. Supponete di avere un array di interi, dove interi differenti possono avere numeri di cifre differenti, ma il numero totale di cifre su *tutti* gli interi dell'array è n. Spiegate come ordinare l'array nel tempo $O(n)$.

b. Supponete di avere un array di stringhe, dove stringhe differenti possono avere numeri differenti di caratteri, ma il numero totale di caratteri su tutte le stringhe è n. Spiegate come ordinare le stringhe nel tempo $O(n)$ (notate che l'ordine richiesto qui è quello alfabetico standard; per esempio, `a` $<$ `ab` $<$ `b`).

8-4 Il problema delle brocche d'acqua

Supponete di avere delle brocche d'acqua, n rosse e n blu, tutte di forma e dimensione diverse. Tutte le brocche rosse contengono quantità d'acqua differenti, come pure le brocche blu. Inoltre, per ogni brocca rossa c'è una brocca blu che contiene la stessa quantità d'acqua e viceversa.

È vostro compito accoppiare una brocca rossa con una blu in modo che ogni coppia contenga la stessa quantità d'acqua. Per farlo, potreste svolgere la seguente operazione: scegliete una coppia di brocche, di cui una rossa e l'altra blu, riempite la brocca rossa di acqua e, poi, versate l'acqua nella brocca blu. Questa operazione vi dirà se la brocca rossa (o quella blu) può contenere più acqua o se le due brocche hanno lo stesso volume. Supponete che questo confronto richieda un'unità di tempo. Il vostro obiettivo è trovare un algoritmo che effettua il numero mimino di confronti per accoppiare le brocche. Ricordatevi che non potete confrontare direttamente due brocche rosse o due brocche blu.

a. Descrivete un algoritmo deterministico che usa $\Theta(n^2)$ confronti per accoppiare le brocche d'acqua.

b. Dimostrate il limite inferiore $\Omega(n \lg n)$ per il numero di confronti che deve effettuare un algoritmo per risolvere questo problema.

c. Trovate un algoritmo randomizzato che abbia un numero atteso di confronti pari a $O(n \lg n)$; dimostrate che questo limite è corretto. Qual è il numero di confronti nel caso peggiore per il vostro algoritmo?

8-5 Ordinamento medio

Supponiamo che, anziché ordinare un array, sia richiesto semplicemente che gli elementi dell'array crescano in media. Più precisamente, chiamiamo ***k-ordinato*** un array A di n elementi se, per ogni $i = 1, 2, \ldots, n-k$, vale la seguente relazione:

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

a. Che cosa significa questo per un array 1-ordinato?

b. Trovate una permutazione dei numeri $1, 2, \ldots, 10$ che sia 2-ordinata, ma non ordinata.

c. Dimostrate che un array di n elementi è k-ordinato se e soltanto se $A[i] \leq A[i+k]$ per ogni $i = 1, 2, \ldots, n-k$.

d. Trovate un algoritmo che trasformi in k-ordinato un array di n elementi nel tempo $O(n \lg(n/k))$.

È anche possibile dimostrare un limite inferiore sul tempo per produrre un array k-ordinato, quando k è una costante.

e. Dimostrate che un array k-ordinato di lunghezza n può essere ordinato nel tempo $O(n \lg k)$ (*suggerimento:* usate la soluzione dell'Esercizio 6.5-8).

f. Dimostrate che quando k è una costante, occorre un tempo $\Omega(n \lg n)$ per trasformare in k-ordinato un array di n elementi (*suggerimento:* usate la soluzione del precedente punto insieme con il limite inferiore sugli ordinamenti per confronti).

8-6 Limite inferiore sulla fusione di liste ordinate

Il problema di trovare una procedura per fondere due liste ordinate si presenta frequentemente. Questa procedura viene utilizzata come una subroutine di MERGE-SORT; nel Paragrafo 2.3.1 abbiamo descritto la procedura MERGE per fondere due liste ordinate. In questo problema, dimostreremo che esiste un limite inferiore di $2n - 1$ sul numero di confronti da effettuare nel caso peggiore per fondere due liste ordinate, ciascuna contenente n elementi.

Innanzi tutto, dimostriamo un limite inferiore di $2n - o(n)$ confronti utilizzando un albero di decisione.

a. Dimostrate che, dati $2n$ numeri, ci sono $\binom{2n}{n}$ possibili modi di dividerli in due liste ordinate, ciascuna con n numeri.

b. Utilizzando un albero di decisione, dimostrate che qualsiasi algoritmo che fonde correttamente due liste ordinate usa almeno $2n - o(n)$ confronti.

Adesso dimostriamo un limite leggermente più stretto $2n - 1$.

c. Dimostrate che, se due elementi sono consecutivi nell'ordinamento e provengono da liste opposte, allora devono essere confrontati.

d. Utilizzate la risposta del precedente punto per dimostrare un limite inferiore di $2n - 1$ confronti per fondere due liste ordinate.

Note

Il modello dell'albero di decisione per studiare gli ordinamenti per confronti è stato introdotto da Ford e Johnson [94]. Knuth [185] ha trattato molte varianti del problema dell'ordinamento, incluso il limite inferiore teorico sulla complessità dell'ordinamento descritto in questo capitolo. I limiti inferiori per gli ordinamenti che usano generalizzazioni del modello dell'albero di decisione sono stati esaurientemente studiati da Ben-Or [36].

continua

Knuth attribuisce a H. H. Seward l'invenzione dell'algoritmo counting sort nel 1954 e anche l'idea di combinare counting sort con radix sort. Il processo radix sort che inizia con la cifra meno significativa sembra essere un algoritmo popolare ampiamente utilizzato dagli operatori di macchine meccaniche per ordinare le schede. Secondo Knuth, la prima pubblicazione su questo metodo è un documento del 1929 di L. J. Comrie che descrive un dispositivo per perforare le schede. Bucket sort è stato utilizzato fin dal 1956, quando l'idea di base fu proposta da E. J. Isaac e R. C. Singleton.

Munro e Raman [229] hanno creato un algoritmo di ordinamento stabile che svolge $O(n^{1+\epsilon})$ confronti nel caso peggiore, dove $0 < \epsilon \leq 1$ è una costante qualsiasi. Sebbene qualsiasi algoritmo con tempo $O(n \lg n)$ effettui meno confronti, tuttavia l'algoritmo di Munro e Raman sposta i dati soltanto $O(n)$ volte e opera sul posto.

Il caso di ordinare n interi di b bit nel tempo $o(n \lg n)$ è stato esaminato da molti ricercatori. Sono stati ottenuti diversi risultati positivi, ciascuno con ipotesi leggermente differenti sul modello di calcolo e sulle limitazioni imposte all'algoritmo. Tutti i risultati presuppongono che la memoria del calcolatore sia divisa in parole di b bit indirizzabili. Fredman e Willard [99] hanno introdotto la struttura dati detta *albero di fusione* che hanno utilizzato per ordinare n interi nel tempo $O(n \lg n / \lg \lg n)$. Successivamente, questo limite è stato migliorato a $O(n\sqrt{\lg n})$ da Andersson [16]. Questi algoritmi richiedono l'uso della moltiplicazione e varie costanti precalcolate. Andersson, Hagerup, Nilsson e Raman [17] hanno dimostrato come ordinare n interi nel tempo $O(n \lg \lg n)$ senza utilizzare la moltiplicazione, ma il loro metodo richiede uno spazio in memoria che potrebbe essere illimitato in termini di n. Utilizzando l'hashing moltiplicativo, è possibile ridurre lo spazio in memoria a $O(n)$, ma il limite $O(n \lg \lg n)$ nel caso peggiore sul tempo di esecuzione diventa un limite sul tempo atteso. Generalizzando l'albero di ricerca esponenziale di Andersson [16], Thorup [297] ha creato un algoritmo di ordinamento con tempo $O(n(\lg \lg n)^2)$ che non usa la moltiplicazione né la randomizzazione, ma lo spazio lineare. Combinando queste tecniche con alcune nuove idee, Han [137] ha migliorato il limite dell'ordinamento al tempo $O(n \lg \lg n \lg \lg \lg n)$. Sebbene questi algoritmi siano importanti scoperte teoriche, tuttavia sono tutti molto complicati e, attualmente, sembra improbabile che possano competere con gli algoritmi di ordinamento che vengono utilizzati nella programmazione pratica.

Mediane e statistiche d'ordine 9

L'i-esima ***statistica d'ordine*** di un insieme di n elementi è l'i-esimo elemento più piccolo. Per esempio, il ***minimo*** di un insieme di elementi è la prima statistica d'ordine ($i = 1$) e il ***massimo*** è l'n-esima statistica d'ordine ($i = n$). Informalmente, la ***mediana*** è il "punto di mezzo" dell'insieme. Se n è dispari, la mediana è unica, perché si verifica in corrispondenza di $i = (n + 1)/2$. Se n è pari, ci sono due mediane: una in corrispondenza di $i = n/2$ e l'altra in corrispondenza di $i = n/2 + 1$. Quindi, indipendentemente dalla parità di n, le mediane si hanno per $i = \lfloor (n + 1)/2 \rfloor$ (la ***mediana inferiore***) e per $i = \lceil (n + 1)/2 \rceil$ (la ***mediana superiore***). Per semplicità, in questo testo utilizzeremo sempre il termine "la mediana" per fare riferimento alla mediana inferiore.

Questo capitolo tratta il problema di selezionare l'i-esima statistica d'ordine da un insieme di n numeri distinti. Supponiamo, per comodità, che l'insieme contenga numeri distinti, sebbene virtualmente tutto ciò che faremo potrà essere esteso al caso in cui l'insieme contenga valori ripetuti. Il ***problema della selezione*** può essere definito formalmente in questo modo:

Input: un insieme A di n numeri (distinti) e un numero i, con $1 \leq i \leq n$.

Output: l'elemento $x \in A$ che è maggiore esattamente di altri $i - 1$ elementi di A.

Il problema della selezione può essere risolto nel tempo $O(n \lg n)$, perché possiamo ordinare i numeri utilizzando heapsort o merge sort e, poi, indirizzare semplicemente l'i-esimo elemento nell'array di output. Ci sono comunque algoritmi più veloci.

Nel Paragrafo 9.1 esamineremo il problema della selezione del minimo e del massimo in un insieme di elementi. Più interessante è il problema generale della selezione, che sarà analizzato nei due paragrafi successivi. Il Paragrafo 9.2 presenta un algoritmo pratico che raggiunge un limite $O(n)$ sul tempo di esecuzione nel caso medio, nell'ipotesi che gli elementi siano distinti. Il Paragrafo 9.3 descrive un algoritmo di interesse più teorico che raggiunge un tempo di esecuzione $O(n)$ nel caso peggiore.

9.1 Minimo e massimo

Quanti confronti sono necessari per determinare il minimo di un insieme di n elementi? Possiamo facilmente ottenere un limite superiore di $n - 1$ confronti: esaminiamo, uno alla volta, gli elementi dell'insieme e teniamo traccia dell'ultimo elemento più piccolo trovato. Nella seguente procedura supponiamo che l'insieme si trovi nell'array A, dove $lunghezza[A] = n$.

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to lunghezza[A]
3       do if min > A[i]
4            then min ← A[i]
5  return min
```

Ovviamente, anche il massimo può essere trovato con $n - 1$ confronti.

È questo quanto di meglio possiamo fare? Sì, perché possiamo ottenere un limite inferiore di $n - 1$ confronti per determinare il minimo. Immaginate qualsiasi algoritmo che determina il minimo come a un campionato fra gli elementi. Ogni confronto è una partita del campionato dove vince il più piccolo fra i due elementi. L'osservazione chiave è che ogni elemento, tranne il vincitore, deve perdere almeno una partita. Quindi, sono necessari $n - 1$ confronti per determinare il minimo; l'algoritmo MINIMUM è ottimale rispetto al numero di confronti effettuati.

Minimo e massimo simultanei

In alcune applicazioni occorre trovare il minimo e il massimo di un insieme di n elementi. Per esempio, un programma di grafica potrebbe richiedere di modificare la scala di un insieme di punti (x, y) per adattarli a uno schermo rettangolare o a un'altra unità grafica di output. Per farlo, il programma deve prima determinare il minimo e il massimo di ogni coordinata.

Non è difficile ideare un algoritmo, asintoticamente ottimale, in grado di trovare il minimo e il massimo di n elementi con $\Theta(n)$ confronti. Basta trovare il minimo e il massimo separatamente, effettuando $n - 1$ confronti per ciascuno di essi, per un totale di $2n - 2$ confronti.

In effetti, sono sufficienti al massimo $3\lfloor n/2 \rfloor$ confronti per determinare il minimo e il massimo. La strategia consiste nel conservare gli ultimi elementi minimo e massimo che sono stati trovati. Anziché confrontare ogni elemento di input con i valori correnti del minimo e del massimo, a un costo di 2 confronti per elemento, elaboriamo gli elementi in coppia. Confrontiamo due elementi di input, prima, *l'uno con l'altro* e, poi, il più piccolo dei due con il minimo corrente e il più grande dei due con il massimo corrente, con un costo di 3 confronti per ogni 2 elementi.

L'impostazione dei valori iniziali del minimo e del massimo dipende dal fatto che n sia pari o dispari. Se n è dispari, assegniamo al minimo e al massimo il valore del primo elemento e poi elaboriamo i restanti elementi in coppia. Se n è pari, effettuiamo un confronto fra i primi 2 elementi per determinare i valori iniziali del minimo e del massimo; poi elaboriamo i restanti elementi in coppia, come nel caso di n dispari.

Analizziamo il numero totale di confronti. Se n è dispari, svolgiamo $3\lfloor n/2 \rfloor$ confronti. Se n è pari, svolgiamo un confronto iniziale seguito da $3(n - 2)/2$ confronti, per un totale di $3n/2 - 2$. Quindi, in entrambi i casi, il numero totale di confronti è al massimo $3\lfloor n/2 \rfloor$.

Esercizi

9.1-1

Dimostrate che il secondo elemento più piccolo di n elementi può essere trovato con $n + \lceil \lg n \rceil - 2$ confronti nel caso peggiore (*suggerimento:* trovate anche l'elemento più piccolo).

9.1-2 $\star$

Dimostrate che sono necessari $\lceil 3n/2 \rceil - 2$ confronti nel caso peggiore per trovare il massimo e il minimo di n numeri (*suggerimento:* considerate quanti numeri sono potenzialmente massimo o minimo ed esaminate come un confronto influenza questi calcoli).

9.2 Selezione nel tempo lineare atteso

Il problema generale della selezione sembra più difficile del semplice problema di trovare un minimo. Eppure, sorprendentemente, il tempo di esecuzione asintotico per entrambi i problemi è lo stesso: $\Theta(n)$. In questo paragrafo, presentiamo un algoritmo divide et impera per il problema della selezione. L'algoritmo RANDOMIZED-SELECT è modellato sull'algoritmo quicksort descritto nel Capitolo 7. Come in quicksort, l'idea di base è partizionare ricorsivamente l'array di input. Diversamente da quicksort, che elabora ricorsivamente entrambi i lati della partizione, RANDOMIZED-SELECT opera soltanto su un lato della partizione. Questa differenza appare evidente nell'analisi: mentre quicksort ha un tempo di esecuzione atteso pari a $\Theta(n \lg n)$, il tempo di esecuzione atteso di RANDOMIZED-SELECT è $\Theta(n)$, nell'ipotesi che tutti gli elementi siano distinti.

RANDOMIZED-SELECT usa la procedura RANDOMIZED-PARTITION introdotta nel Paragrafo 7.3; quindi, come RANDOMIZED-QUICKSORT, è un algoritmo randomizzato, in quanto il suo comportamento è determinato in parte dall'output di un generatore di numeri casuali. Il seguente codice per RANDOMIZED-SELECT restituisce l'i-esimo numero più piccolo dell'array $A[p\,..\,r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i = k            ▷ il valore del pivot è la soluzione
6     then return A[q]
7  elseif i < k
8     then return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Dopo che RANDOMIZED-PARTITION viene eseguito nella riga 3 dell'algoritmo, l'array $A[p\,..\,r]$ è diviso in due sottoarray (eventualmente vuoti) $A[p\,..\,q-1]$ e $A[q+1\,..\,r]$ tali che ogni elemento di $A[p\,..\,q-1]$ è minore o uguale a $A[q]$ che, a sua volta, è minore o uguale a ogni elemento di $A[q+1\,..\,r]$. Come in quicksort, chiameremo $A[q]$ l'elemento ***pivot***. La riga 4 di RANDOMIZED-SELECT calcola il numero k di elementi nel sottoarray $A[p\,..\,q]$, ovvero il numero di elementi nel lato basso della partizione, più uno per l'elemento pivot. La riga 5 poi controlla se $A[q]$ è l'i-esimo elemento più piccolo. Se lo è, restituisce il valore di $A[q]$; altrimenti l'algoritmo determina in quale dei due sottoarray $A[p\,..\,q-1]$ e $A[q+1\,..\,r]$ si trova l'i-esimo elemento più piccolo. Se $i < k$, l'elemento desiderato si trova nel lato basso della partizione e viene selezionato ricorsivamente dal sottoarray nella riga 8. Se $i > k$, l'elemento desiderato si trova nel lato alto della partizione. Poiché conosciamo già k valori che sono più piccoli dell'i-esimo elemento più piccolo di

$A[p\,..\,r]$ – ovvero gli elementi di $A[p\,..\,q]$ – l'elemento desiderato è l'$(i-k)$-esimo elemento più piccolo di $A[q+1\,..\,r]$, che viene trovato ricorsivamente nella riga 9. Sembra che il codice consenta le chiamate ricorsive a sottoarray con 0 elementi, ma l'Esercizio 9.2-1 vi chiederà di dimostrare che questa situazione non potrà verificarsi.

Il tempo di esecuzione nel caso peggiore di RANDOMIZED-SELECT è $\Theta(n^2)$, anche per trovare il minimo, perché potremmo essere estremamente sfortunati, effettuando la partizione sempre attorno all'elemento più grande rimasto, e il partizionamento richiede un tempo $\Theta(n)$. Tuttavia, l'algoritmo funziona bene nel caso medio e, poiché è randomizzato, nessun input determina il caso peggiore nel comportamento dell'algoritmo.

Il tempo richiesto da RANDOMIZED-SELECT con un array di input $A[p\,..\,r]$ di n elementi è una variabile casuale che indichiamo con $T(n)$; otteniamo un limite superiore su $\mathrm{E}[T(n)]$ nel modo seguente. La procedura RANDOMIZED-PARTITION dà a qualsiasi elemento la stessa probabilità di essere selezionato come pivot. Quindi, per ogni k tale che $1 \le k \le n$, il sottoarray $A[p\,..\,q]$ ha k elementi (tutti minori o uguali al pivot) con probabilità $1/n$. Per $k = 1, 2, \ldots, n$, definiamo le variabili casuali indicatrici X_k dove

$$X_k = \mathrm{I}\{\text{il sottoarray } A[p\,..\,q] \text{ ha esattamente } k \text{ elementi}\}$$

Pertanto, supponendo che tutti gli elementi siano distinti, abbiamo

$$\mathrm{E}[X_k] = 1/n \tag{9.1}$$

Quando chiamiamo RANDOMIZED-SELECT e scegliamo $A[q]$ come pivot, non sappiamo, a priori, se termineremo immediatamente con la soluzione esatta o se ci sarà una ricorsione sul sottoarray $A[p\,..\,q-1]$ o sul sottoarray $A[q+1\,..\,r]$. Questa decisione dipende dalla posizione in cui si trova l'i-esimo elemento più piccolo rispetto ad $A[q]$. Supponendo che $T(n)$ sia una funzione monotonicamente crescente, possiamo limitare il tempo richiesto per la chiamata ricorsiva con il tempo richiesto per la chiamata ricorsiva sul massimo input possibile. In altre parole, supponiamo, per ottenere un limite superiore, che l'i-esimo elemento sia sempre nel lato della partizione con il maggior numero di elementi. Per una chiamata di RANDOMIZED-SELECT, la variabile casuale indicatrice X_k vale 1 per un solo valore di k e 0 per tutti gli altri valori di k. Quando $X_k = 1$, i due sottoarray sui quali potremmo effettuare la ricorsione hanno dimensioni $k-1$ e $n-k$. Quindi, abbiamo la ricorrenza

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) \end{aligned}$$

Prendendo i valori attesi, si ha

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=1}^{n} \mathrm{E}\left[X_k \cdot T(\max(k-1, n-k))\right] + O(n) && \text{(per la linearità del valore atteso)} \\
&= \sum_{k=1}^{n} \mathrm{E}\left[X_k\right] \cdot \mathrm{E}\left[T(\max(k-1, n-k))\right] + O(n) && \text{(per l'equazione (C.23))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[T(\max(k-1, n-k))\right] + O(n) && \text{(per l'equazione (9.1))}
\end{aligned}
$$

Per applicare l'equazione (C.23), contiamo sul fatto che X_k e $T(\max(k-1, n-k))$ sono variabili casuali indipendenti. L'Esercizio 9.2-2 chiede di giustificare questa asserzione.

Consideriamo l'espressione $\max(k-1, n-k)$. Abbiamo

$$
\max(k-1, n-k) = \begin{cases} k-1 & \text{se } k > \lceil n/2 \rceil \\ n-k & \text{se } k \le \lceil n/2 \rceil \end{cases}
$$

Se n è pari, ogni termine da $T(\lceil n/2 \rceil)$ fino a $T(n-1)$ appare esattamente due volte nella sommatoria e, se n è dispari, tutti questi termini appaiono due volte, mentre $T(\lfloor n/2 \rfloor)$ appare una volta soltanto. Dunque, abbiamo

$$
\mathrm{E}\left[T(n)\right] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}\left[T(k)\right] + O(n)
$$

Risolviamo la ricorrenza per sostituzione. Supponiamo che $\mathrm{E}\left[T(n)\right] \le cn$ per qualche costante c che soddisfa le condizioni iniziali della ricorrenza. Supponiamo che $T(n) = O(1)$ per n minore di qualche costante; sceglieremo questa costante successivamente. Scegliamo anche una costante a tale che la funzione descritta dal precedente termine $O(n)$ (che descrive la componente non ricorsiva del tempo di esecuzione dell'algoritmo) sia limitata dall'alto da an per ogni $n > 0$. Utilizzando questa ipotesi induttiva, otteniamo

$$
\begin{aligned}
\mathrm{E}\left[T(n)\right] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
&\le \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right)
\end{aligned}
$$

Per completare la dimostrazione, dobbiamo dimostrare che per n sufficientemente grande, quest'ultima espressione vale al massimo cn ovvero $cn/4 - c/2 - an \geq 0$. Se sommiamo $c/2$ a entrambi i lati e mettiamo in evidenza il fattore n, otteniamo $n(c/4 - a) \geq c/2$. Se scegliamo la costante c in modo che $c/4 - a > 0$, ovvero $c > 4a$, possiamo dividere entrambi i lati per $c/4 - a$, ottenendo

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}$$

In definitiva, se supponiamo che $T(n) = O(1)$ per $n < 2c/(c - 4a)$, abbiamo $\mathrm{E}[T(n)] = O(n)$. Possiamo concludere che qualsiasi statistica d'ordine, in particolare la mediana, può essere determinata mediamente in tempo lineare, nell'ipotesi che gli elementi siano distinti.

Esercizi

9.2-1

Dimostrate che nell'algoritmo RANDOMIZED-SELECT non viene mai effettuata una chiamata ricorsiva a un array di lunghezza 0.

9.2-2

Dimostrate che la variabile casuale indicatrice X_k e il valore $T(\max(k-1, n-k))$ sono indipendenti.

9.2-3

Scrivete una versione iterativa di RANDOMIZED-SELECT.

9.2-4

Supponete di utilizzare RANDOMIZED-SELECT per selezionare l'elemento minimo dell'array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Descrivete una sequenza di partizioni che determina il comportamento nel caso peggiore di RANDOMIZED-SELECT.

9.3 Selezione in tempo lineare nel caso peggiore

Esaminiamo adesso un algoritmo di selezione il cui tempo di esecuzione è $O(n)$ nel caso peggiore. Come RANDOMIZED-SELECT, anche l'algoritmo SELECT trova l'elemento desiderato partizionando ricorsivamente l'array di input. L'idea che sta alla base dell'algoritmo, tuttavia, è quella di *garantire* una buona ripartizione dell'array. SELECT usa l'algoritmo di partizionamento deterministico PARTITION, lo stesso che usa quicksort (vedere il Paragrafo 7.1), a parte una modifica che consente di accettare come parametro di input l'elemento attorno al quale effettuare il partizionamento.

L'algoritmo SELECT determina l'i-esimo elemento più piccolo di un array di input di n elementi ($n > 1$), effettuando i seguenti passi (se $n = 1$, SELECT restituisce semplicemente il suo unico elemento di input come l'i-esimo valore più piccolo).

1. Dividere gli n elementi dell'array di input in $\lfloor n/5 \rfloor$ gruppi di 5 elementi ciascuno e (al massimo) un gruppo con i restanti $n \bmod 5$ elementi.
2. Trovare la mediana di ciascuno degli $\lceil n/5 \rceil$ gruppi, effettuando prima un ordinamento per inserzione degli elementi di ogni gruppo (al massimo 5) e poi scegliendo la mediana dalla lista ordinata degli elementi di ogni gruppo.

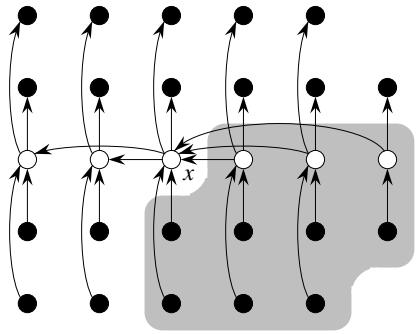


Figura 9.1 Analisi dell'algoritmo SELECT. Gli n elementi sono rappresentati da piccoli cerchi e ogni gruppo occupa una colonna. Le mediane dei gruppi sono rappresentate da cerchi bianchi; la mediana delle mediane è identificata dalla lettera x (per nostra convenzione, la mediana di un numero pari di elementi è la mediana inferiore). Le frecce vanno dagli elementi più grandi a quelli più piccoli; da questo si può notare che 3 elementi nei gruppi di 5 elementi a destra di x sono maggiori di x, e 3 elementi nei gruppi di 5 elementi a sinistra di x sono minori x. Gli elementi maggiori di x sono rappresentati su uno sfondo grigio.

3. Usare SELECT ricorsivamente per trovare la mediana x delle $\lceil n/5 \rceil$ mediane trovate nel passo 2 (se il numero di mediane è pari, allora per nostra convenzione x è la mediana inferiore).
4. Partizionare l'array di input attorno alla mediana delle mediane x utilizzando la versione modificata di PARTITION. Se k indica il numero di elementi nel lato basso della partizione più 1, allora x è il k-esimo elemento più piccolo e il lato alto della partizione contiene $n - k$ elementi.
5. Se $i = k$, restituire x; altrimenti utilizzare SELECT ricorsivamente per trovare l'i-esimo elemento più piccolo nel lato basso se $i < k$ oppure l'$(i - k)$-esimo elemento più piccolo nel lato alto se $i > k$.

Per analizzare il tempo di esecuzione di SELECT, determiniamo prima un limite inferiore sul numero di elementi che sono maggiori dell'elemento di partizionamento x. La Figura 9.1 è utile per visualizzare i conteggi. Almeno metà delle mediane trovate nel passo 2 sono maggiori o uguali alla mediana delle mediane x.[1] Quindi, almeno metà degli $\lceil n/5 \rceil$ gruppi contribuisce con 3 elementi che sono maggiori di x, tranne quel gruppo che ha meno di 5 elementi (se n non è divisibile esattamente per 5) e quel gruppo che contiene x. Se escludiamo questi due gruppi, allora il numero di elementi maggiori di x è almeno

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \;\geq\; \frac{3n}{10} - 6$$

Analogamente, il numero di elementi che sono minori di x è almeno $3n/10 - 6$. Quindi nel caso peggiore SELECT viene chiamata ricorsivamente al massimo su $7n/10 + 6$ elementi nel passo 5.

Adesso possiamo sviluppare una ricorrenza per il tempo di esecuzione $T(n)$ nel caso peggiore dell'algoritmo SELECT. I passi 1, 2 e 4 impiegano un tempo $O(n)$ (il passo 2 è formato da $O(n)$ chiamate di insertion sort su insiemi di dimensione $O(1)$). Il passo 3 impiega un tempo $T(\lceil n/5 \rceil)$; il passo 5 impiega al massimo un tempo $T(7n/10 + 6)$, supponendo che T sia monotonicamente crescente. Facciamo l'ipotesi, che a prima vista può sembrare immotivata, che qualsiasi input con meno di 140 elementi richieda un tempo $O(1)$; l'origine della costante magica 140 sarà chiarita a breve. Possiamo ottenere la seguente ricorrenza:

[1] Poiché abbiamo fatto l'ipotesi che i numeri siano distinti, tutte le mediane (tranne x) sono maggiori o minori di x.

$$T(n) \leq \begin{cases} O(1) & \text{se } n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{se } n \geq 140 \end{cases}$$

Dimostreremo che il tempo di esecuzione è lineare, applicando il metodo di sostituzione. Più specificatamente, dimostreremo che $T(n) \leq cn$ per qualche costante c opportunamente grande e per ogni $n > 0$. Iniziamo supponendo che $T(n) \leq cn$ per qualche costante c opportunamente grande e per ogni $n < 140$; questa ipotesi è valida se c è abbastanza grande. Scegliamo anche una costante a tale che la funzione descritta dal precedente termine $O(n)$ (che descrive la componente non ricorsiva del tempo di esecuzione dell'algoritmo) sia limitata dall'alto da an per ogni $n > 0$. Sostituendo questa ipotesi induttiva nel lato destro della ricorrenza, si ottiene

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

che vale al massimo cn, se

$$-cn/10 + 7c + an \leq 0 \tag{9.2}$$

La disequazione (9.2) è equivalente alla disequazione $c \geq 10a(n/(n-70))$ se $n > 70$. Poiché supponiamo che $n \geq 140$, allora $n/(n-70) \leq 2$; quindi scegliendo $c \geq 20a$ è soddisfatta la disequazione (9.2) (notate che la costante 140 non ha nulla di speciale; potremmo sostituirla con un altro numero intero strettamente maggiore di 70 e poi scegliere opportunamente c). Il tempo di esecuzione nel caso peggiore di SELECT è dunque lineare.

Come in un ordinamento per confronti (vedere il Paragrafo 8.1), SELECT e RANDOMIZED-SELECT determinano le informazioni sull'ordine relativo degli elementi esclusivamente tramite i confronti degli elementi. Ricordiamo dal Capitolo 8 che l'ordinamento richiede un tempo $\Omega(n \lg n)$ nel modello dei confronti, anche nel caso medio (vedere il Problema 8-1). Gli algoritmi di ordinamento in tempo lineare descritti nel Capitolo 8 fanno delle ipotesi sull'input. Gli algoritmi di selezione in tempo lineare presentati in questo capitolo, invece, non richiedono alcuna ipotesi sull'input. Essi non sono soggetti al limite inferiore $\Omega(n \lg n)$ perché riescono a risolvere il problema della selezione senza ordinare gli elementi.

Quindi, il tempo di esecuzione è lineare perché questi algoritmi non effettuano ordinamenti; il comportamento in tempo lineare non è un risultato delle ipotesi sull'input, come nel caso degli algoritmi di ordinamento del Capitolo 8. L'ordinamento richiede un tempo $\Omega(n \lg n)$ nel modello dei confronti, anche nel caso medio (vedere il Problema 8-1), pertanto il metodo di ordinamento e indicizzazione presentato nell'introduzione di questo capitolo è asintoticamente inefficiente.

Esercizi

9.3-1

Nell'algoritmo SELECT gli elementi di input sono suddivisi in gruppi di 5. L'algoritmo potrà operare in tempo lineare se gli elementi sono suddivisi in gruppi di 7? Dimostrate che SELECT non viene eseguito in tempo lineare se vengono utilizzati gruppi di 3 elementi.

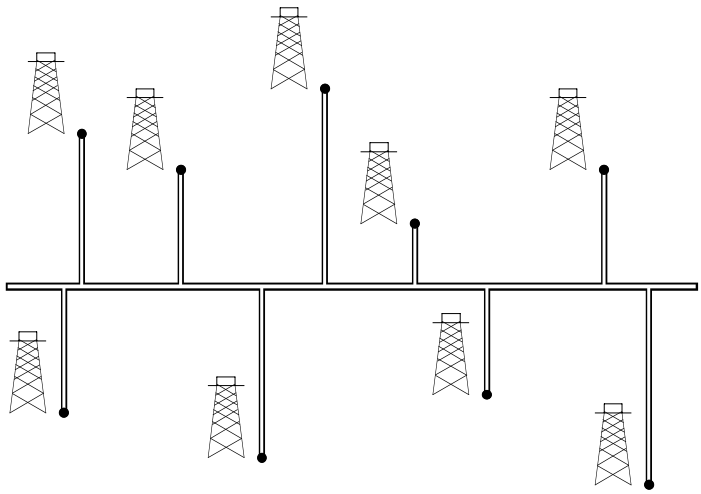


Figura 9.2 Il professor Olay deve determinare la posizione della condotta orizzontale che rende minima la lunghezza totale delle condotte verticali dai pozzi petroliferi.

9.3-2
Analizzate SELECT per dimostrare che, se $n \geq 140$, allora almeno $\lceil n/4 \rceil$ elementi sono maggiori della mediana delle mediane x e almeno $\lceil n/4 \rceil$ elementi sono minori di x.

9.3-3
Spiegate come quicksort possa essere eseguito nel tempo $O(n \lg n)$ nel caso peggiore, supponendo che tutti gli elementi siano distinti.

9.3-4 $\star$
Supponete che un algoritmo usi soltanto i confronti per trovare l'i-esimo elemento più piccolo in un insieme di n elementi. Dimostrate inoltre che l'algoritmo può trovare $i - 1$ elementi più piccoli e $n - i$ elementi più grandi senza svolgere confronti aggiuntivi.

9.3-5
Supponete di avere una subroutine "black-box" (scatola nera) che trova la mediana in tempo lineare nel caso peggiore. Create un semplice algoritmo in tempo lineare che risolve il problema della selezione per un'arbitraria statistica d'ordine.

9.3-6
I k-esimi ***quantili*** di un insieme di n elementi sono le $k - 1$ statistiche d'ordine che dividono l'insieme ordinato in k insiemi della stessa dimensione (a meno di 1). Create un algoritmo con tempo $O(n \lg k)$ che elenca i k-esimi quantili di un insieme.

9.3-7
Descrivete un algoritmo con tempo $O(n)$ che, dato un insieme S di n numeri distinti e un intero positivo $k \leq n$, trova k numeri in S che sono più vicini alla mediana di S.

9.3-8
Siano $X[1 \,..\, n]$ e $Y[1 \,..\, n]$ due array, ciascuno contenente n numeri già ordinati. Scrivete un algoritmo con tempo $O(\lg n)$ per trovare la mediana di tutti i $2n$ elementi degli array X e Y.

9.3-9

Il professor Olay è un consulente di una compagnia petrolifera, che sta progettando un importante oleodotto che attraversa da est a ovest una vasta area con n pozzi petroliferi. Da ogni pozzo parte una condotta che deve essere collegata direttamente alla condotta principale lungo il percorso minimo (da nord o sud), come illustra la Figura 9.2. Se sono note le coordinate x e y dei pozzi, in che modo il professor Olay può scegliere la posizione ottimale della condotta principale (quella che rende minima la lunghezza totale delle condotte verticali che collegano i pozzi alla condotta principale)? Dimostrate che la posizione ottimale può essere trovata in tempo lineare.

Problemi

9-1 Trovare in ordine i k numeri più grandi

Dato un insieme di n numeri, trovare in ordine i k numeri più grandi utilizzando un algoritmo basato sui confronti. Trovate l'algoritmo che implementa ciascuno dei seguenti metodi con il miglior tempo di esecuzione asintotico nel caso peggiore; analizzate i tempi di esecuzione in funzione di n e k.

a. Ordinare i numeri ed elencare i k numeri più grandi.

b. Costruire una coda di max-priorità dai numeri e chiamare k volte la procedura EXTRACT-MAX.

c. Utilizzare un algoritmo per statistiche d'ordine che trova il k-esimo numero più grande, crea le partizioni attorno a questo numero e ordina i k numeri più grandi.

9-2 Mediana ponderata

Per n elementi distinti $x_1, x_2, \ldots, x_n$ con pesi positivi $w_1, w_2, \ldots, w_n$ tali che $\sum_{i=1}^{n} w_i = 1$, la ***mediana (inferiore) ponderata*** è l'elemento x_k che soddisfa le relazioni

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

e

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}$$

a. Dimostrate che la mediana di $x_1, x_2, \ldots, x_n$ è la mediana ponderata di x_i con pesi $w_i = 1/n$ per $i = 1, 2, \ldots, n$.

b. Come può essere calcolata la mediana ponderata di n elementi nel tempo $O(n \lg n)$ nel caso peggiore utilizzando un algoritmo di ordinamento?

c. Spiegate come calcolare la mediana ponderata nel tempo $\Theta(n)$ nel caso peggiore utilizzando un algoritmo in tempo lineare come SELECT, che è descritto nel Paragrafo 9.3.

Il ***problema della posizione dell'ufficio postale*** è definito nel seguente modo. Dati n punti $p_1, p_2, \ldots, p_n$ con i pesi associati $w_1, w_2, \ldots, w_n$, trovare un punto p (non necessariamente appartenente a uno dei punti di input) che rende minima la sommatoria $\sum_{i=1}^{n} w_i \, d(p, p_i)$, dove $d(a, b)$ è la distanza tra i punti a e b.

d. Dimostrate che la mediana ponderata è una soluzione ottima per il problema monodimensionale della posizione dell'ufficio postale, in cui i punti sono semplicemente numeri reali e la distanza tra i punti a e b è $d(a, b) = |a - b|$.

e. Trovate la soluzione ottima per il problema bidimensionale della posizione dell'ufficio postale, in cui i punti sono coppie di coordinate (x, y) e la distanza tra i punti $a = (x_1, y_1)$ e $b = (x_2, y_2)$ è la ***distanza di Manhattan*** data da $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3 Piccole statistiche d'ordine

Abbiamo dimostrato che il numero di confronti $T(n)$ nel caso peggiore utilizzato da SELECT per selezionare l'i-esima statistica d'ordine da n numeri soddisfa la relazione $T(n) = \Theta(n)$, ma la costante nascosta dalla notazione Θ è piuttosto grande. Se il valore di i è piccolo rispetto a n, possiamo implementare una procedura differente che usa SELECT come subroutine, ma effettua un minor numero di confronti nel caso peggiore.

a. Descrivete un algoritmo che usa $U_i(n)$ confronti per trovare l'i-esimo elemento più piccolo di n elementi, dove

$$U_i(n) = \begin{cases} T(n) & \text{se } i \geq n/2 \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{negli altri casi} \end{cases}$$

(*Suggerimento:* iniziate con $\lfloor n/2 \rfloor$ confronti di coppie disgiunte; poi eseguite una ricorsione sull'insieme che contiene l'elemento più piccolo di ogni coppia.)

b. Dimostrate che, se $i < n/2$, allora $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Dimostrate che, se i è una costante minore di $n/2$, allora $U_i(n) = n + O(\lg n)$.

d. Dimostrate che, se $i = n/k$ per $k \geq 2$, allora $U_i(n) = n + O(T(2n/k) \lg k)$.

Note

L'algoritmo che trova la mediana in tempo lineare nel caso peggiore è stato sviluppato da Blum, Floyd, Pratt, Rivest e Tarjan [43]. La versione veloce nel caso medio è stata ideata da Hoare [146]. Floyd e Rivest [92] hanno sviluppato una versione migliore nel caso medio che crea le partizioni attorno a un elemento selezionato ricorsivamente da un piccolo campione di elementi.

Non è ancora noto con esattezza il numero di confronti che sono necessari per determinare la mediana. Bent e John [38] hanno trovato un limite inferiore di $2n$ confronti per trovare la mediana. Schönhage, Paterson e Pippenger [265] hanno trovato un limite superiore di $3n$ confronti. Dor e Zwick [79] hanno migliorato entrambi questi limiti; il loro limite superiore è leggermente più piccolo di $2,95n$ e quello inferiore è leggermente più grande di $2n$. Paterson [239] descrive questi risultati e altri lavori correlati.

III Strutture dati

Introduzione

Gli insiemi sono fondamentali per l'informatica e la matematica. Mentre gli insiemi matematici sono immutabili, gli insiemi manipolati dagli algoritmi possono crescere, ridursi o cambiare nel tempo. Questi insiemi sono detti ***dinamici***. I prossimi cinque capitoli presentano alcune tecniche di base per rappresentare e manipolare gli insiemi dinamici finiti.

Gli algoritmi possono richiedere vari tipi di operazioni da svolgere sugli insiemi. Per esempio, molti algoritmi richiedono soltanto la capacità di inserire e cancellare degli elementi da un insieme e di verificare l'appartenenza di un elemento a un insieme. Un insieme dinamico che supporta queste operazioni è detto ***dizionario***. Altri algoritmi richiedono operazioni più complicate. Per esempio, le code di min-priorità, che sono state introdotte nel Capitolo 6 nell'ambito della struttura heap, supportano le operazioni per inserire un elemento in un insieme e per estrarre l'elemento più piccolo da un insieme. Il modo migliore di implementare un insieme dinamico dipende dalle operazioni che devono essere supportate.

Gli elementi di un insieme dinamico

In una tipica implementazione di un insieme dinamico, ogni elemento è rappresentato da un oggetto i cui campi possono essere esaminati e manipolati se c'è un puntatore all'oggetto (il Paragrafo 10.3 descrive l'implementazione di oggetti e puntatori negli ambienti di programmazione che non li prevedono come tipi di dati di base). Per alcuni tipi di insiemi dinamici si suppone che uno dei campi dell'oggetto sia un campo ***chiave*** di identificazione. Se le chiavi sono tutte diverse, possiamo pensare all'insieme dinamico come a un insieme di valori chiave. L'oggetto può contenere ***dati satelliti***, che vengono spostati in altri campi dell'oggetto, senza essere utilizzati in altro modo dall'implementazione dell'insieme. L'oggetto può anche includere campi che vengono manipolati dalle operazioni svolte sull'insieme; questi campi possono contenere dati o puntatori ad altri oggetti dell'insieme.

Alcuni insiemi dinamici presuppongono che le chiavi siano estratte da un insieme totalmente ordinato, come i numeri reali o l'insieme di tutte le parole secondo il consueto ordine alfabetico (un insieme totalmente ordinato soddisfa la proprietà della tricotomia, definita a pagina 42). L'ordinamento totale ci consente di definire

l'elemento minimo dell'insieme, per esempio, o di parlare del prossimo elemento più grande di un determinato elemento dell'insieme.

Operazioni sugli insiemi dinamici

Le operazioni su un insieme dinamico possono essere raggruppate in due categorie: le ***interrogazioni*** o ***query*** che restituiscono semplicemente informazioni sull'insieme; le ***operazioni di modifica*** che cambiano l'insieme. Elenchiamo qui di seguito una serie di operazioni tipiche. Un'applicazione reale di solito richiede l'implementazione di un numero limitato di queste operazioni.

SEARCH(S, k)
: Una query che, dato un insieme S e un valore chiave k, restituisce un puntatore x a un elemento di S tale che $chiave[x] = k$ oppure NIL se un elemento così non appartiene a S.

INSERT(S, x)
: Un'operazione di modifica che inserisce nell'insieme S l'elemento puntato da x. Di solito, si suppone che qualsiasi campo dell'elemento x richiesto dall'implementazione dell'insieme sia stato già inizializzato.

DELETE(S, x)
: Un'operazione di modifica che, dato un puntatore x a un elemento dell'insieme S, rimuove x da S (notate che questa operazione usa un puntatore a un elemento x, non un valore chiave).

MINIMUM(S)
: Una query su un insieme totalmente ordinato S che restituisce un puntatore all'elemento di S con la chiave più piccola.

MAXIMUM(S)
: Una query su un insieme totalmente ordinato S che restituisce un puntatore all'elemento di S con la chiave più grande.

SUCCESSOR(S, x)
: Una query che, dato un elemento x la cui chiave appartiene a un insieme totalmente ordinato S, restituisce un puntatore al prossimo elemento più grande di S oppure NIL se x è l'elemento massimo.

PREDECESSOR(S, x)
: Una query che, dato un elemento x la cui chiave appartiene a un insieme totalmente ordinato S, restituisce un puntatore al prossimo elemento più piccolo di S oppure NIL se x è l'elemento minimo.

Le query SUCCESSOR e PREDECESSOR vengono spesso estese a insiemi con chiavi non distinte. Per un insieme con n chiavi, è lecito supporre che una chiamata di MINIMUM seguita da $n - 1$ chiamate di SUCCESSOR enumeri ordinatamente gli elementi dell'insieme.

Il tempo impiegato per eseguire un'operazione su un insieme, di solito, è misurato in funzione della dimensione dell'insieme, che viene specificata come uno degli argomenti dell'operazione. Per esempio, il Capitolo 13 descrive una struttura dati che è in grado di svolgere una qualsiasi delle operazioni precedentemente elencate su un insieme di dimensione n nel tempo $O(\lg n)$.

Sintesi della Parte III

I Capitoli 10–14 descrivono diverse strutture dati che possono essere utilizzate per implementare insiemi dinamici; molte di queste strutture saranno utilizzate successivamente per costruire algoritmi efficienti che risolvono vari tipi di problemi. Un'altra importante struttura dati – l'heap – è stata già presentata nel Capitolo 6.

Il Capitolo 10 illustra i concetti essenziali per operare con semplici strutture dati, come stack, code, liste concatenate e alberi con radice. In questo capitolo spiegheremo anche come implementare oggetti e puntatori in quegli ambienti di programmazione che non li includono fra i tipi di dati di base. Molti di questi argomenti dovrebbero essere familiari a chiunque abbia svolto un corso introduttivo di programmazione.

Il Capitolo 11 presenta le tabelle hash, che supportano le operazioni INSERT, DELETE e SEARCH sui dizionari. Nel caso peggiore, l'hashing richiede un tempo $\Theta(n)$ per svolgere un'operazione SEARCH, ma il tempo atteso per le operazioni con le tabelle hash è $O(1)$. L'analisi dell'hashing è basata sulla teoria della probabilità, ma la maggior parte del capitolo non richiede la conoscenza di questa materia.

Gli alberi binari di ricerca, che sono trattati nel Capitolo 12, supportano tutte le operazioni sugli insiemi dinamici precedentemente elencate. Nel caso peggiore, ogni operazione richiede un tempo $\Theta(n)$ su un albero con n elementi, ma su un albero binario di ricerca costruito in modo casuale, il tempo atteso per ogni operazione è $O(\lg n)$. Gli alberi binari di ricerca sono la base di molte altre strutture dati.

Gli alberi RB (Red-Black), una variante degli alberi binari di ricerca, sono presentati nel Capitolo 13. Diversamente dai normali alberi binari di ricerca, gli alberi RB garantiscono buone prestazioni: le operazioni richiedono un tempo $O(\lg n)$ nel caso peggiore. Un albero RB è un albero di ricerca bilanciato; il Capitolo 18 presenta un altro tipo di albero di ricerca bilanciato, detto albero B. Sebbene i meccanismi degli alberi RB siano alquanto complicati, tuttavia nel capitolo potete racimolare gran parte delle informazioni sulle sue proprietà, senza bisogno di studiare dettagliatamente tali meccanismi. Nonostante questo, sarà molto istruttivo analizzare attentamente il codice.

Nel Capitolo 14 spieghiamo come estendere gli alberi RB per supportare operazioni diverse da quelle di base precedentemente elencate. La prima estensione ci consente di mantenere dinamicamente le statistiche d'ordine di un insieme di chiavi; la seconda estensione ci consente di mantenere gli intervalli dei numeri reali.

10 Strutture dati elementari

In questo capitolo esamineremo la rappresentazione di insiemi dinamici mediante semplici strutture dati che usano i puntatori. Sebbene molte strutture dati complesse possano essere modellate utilizzando i puntatori, presenteremo soltanto le strutture più semplici: stack (o pile), code, liste concatenate e alberi con radice. Presenteremo anche un metodo per rappresentare oggetti e puntatori mediante gli array.

10.1 Stack e code

Gli stack e le code sono insiemi dinamici dove l'elemento che viene rimosso dall'operazione DELETE è predeterminato. In uno ***stack*** l'elemento cancellato dall'insieme è quello inserito per ultimo: lo stack implementa lo schema ***LIFO*** (Last-In, First-Out). Analogamente, in una ***coda*** l'elemento cancellato è sempre quello che è rimasto nell'insieme per più tempo: la coda implementa lo schema ***FIFO*** (First-In, First-Out). Ci sono vari modi efficienti per implementare gli stack e le code in un calcolatore. In questo paragrafo spiegheremo come utilizzare un semplice array per implementare entrambe le strutture dati.

Stack

L'operazione INSERT su uno stack è detta PUSH, mentre l'operazione DELETE, che non ha un elemento come argomento, è detta POP. Questi nomi inglesi sono allusioni alle pile (*stack*) reali, come le pile di piatti caricate a molla che sono utilizzate nelle tavole calde. L'ordine in cui i piatti vengono rimossi (*pop*) dalla pila è inverso a quello in cui sono stati inseriti (*push*), in quanto è accessibile soltanto il piatto che è in cima alla pila.

Come illustra la Figura 10.1, possiamo implementare uno stack di n elementi al massimo con un array $S[1 \,..\, n]$. L'array ha un attributo $top[S]$ che è l'indice dell'elemento inserito più di recente. Lo stack è formato dagli elementi $S[1 \,..\, top[S]]$, dove $S[1]$ è l'elemento in fondo allo stack e $S[top[S]]$ è l'elemento in cima.

Se $top[S] = 0$, lo stack non contiene elementi ed è ***vuoto***. L'operazione STACK-EMPTY verifica se uno stack è vuoto. Se si tenta di estrarre un elemento (operazione POP) da uno stack vuoto, si ha un ***underflow*** dello stack, che di norma è un errore. Se $top[S]$ supera n, si ha un ***overflow*** dello stack (nel nostro pseudocodice non ci preoccuperemo dell'overflow dello stack).

Le operazioni sullo stack possono essere implementate con poche righe di codice.

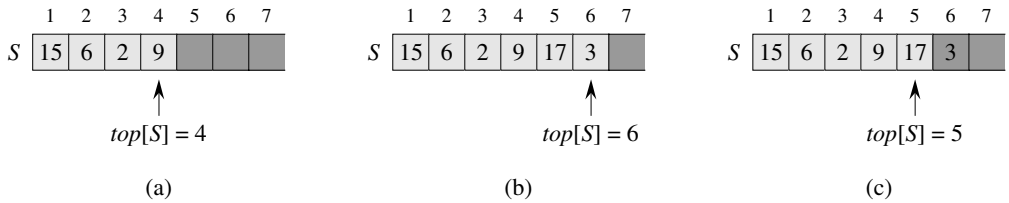


Figura 10.1 Implementazione di uno stack S con un array. Gli elementi dello stack appaiono soltanto nelle posizioni con sfondo grigio chiaro. **(a)** Lo stack S ha 4 elementi. L'elemento in cima allo stack è 9. **(b)** Lo stack S dopo le chiamate PUSH$(S, 17)$ e PUSH$(S, 3)$. **(c)** Lo stack S dopo la chiamata POP(S) ha ceduto l'elemento 3, che è l'elemento inserito più di recente. Sebbene l'elemento 3 appaia ancora nell'array, non appartiene più allo stack; l'elemento in cima allo stack è l'elemento 17.

```
STACK-EMPTY(S)
1  if top[S] = 0
2     then return TRUE
3     else return FALSE
```

```
PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"
3     else top[S] ← top[S] − 1
4          return S[top[S] + 1]
```

La Figura 10.1 illustra gli effetti delle operazioni di modifica PUSH e POP. Ciascuna delle tre operazioni sullo stack richiede un tempo $O(1)$.

Code

Chiameremo ENQUEUE l'operazione INSERT su una coda. Chiameremo DEQUEUE l'operazione DELETE; come l'operazione POP su uno stack, anche DEQUEUE non richiede un elemento come argomento. La proprietà FIFO di una coda fa sì che essa funzioni come le file delle persone che a volte si formano davanti agli sportelli degli uffici.

La coda ha un inizio (*head*) e una fine (*tail*). Quando un elemento viene inserito nella coda, prende posto alla fine della coda, esattamente come l'ultima persona che arriva si mette in fondo alla fila. L'elemento rimosso è sempre quello che si trova all'inizio della coda, come la persona all'inizio della fila, che è quella rimasta in attesa più a lungo delle altre (per fortuna, non dobbiamo preoccuparci degli elementi che non rispettano la fila).

La Figura 10.2 illustra un modo di implementare una coda di $n - 1$ elementi al massimo, utilizzando un array $Q[1 \,..\, n]$. L'attributo $head[Q]$ indica (o punta) l'inizio della coda. L'attributo $tail[Q]$ indica la prossima posizione in cui l'ultimo

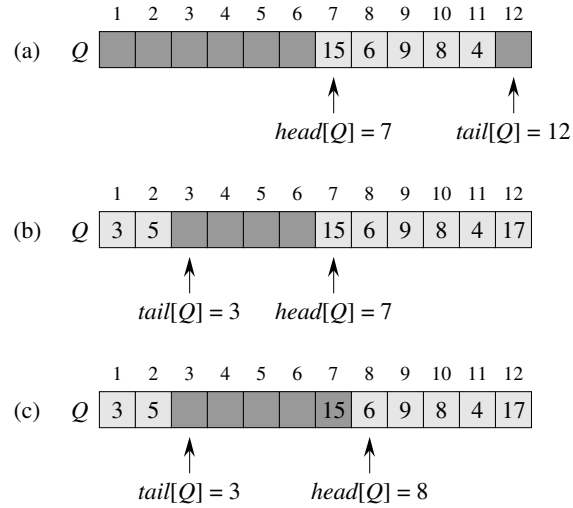


Figura 10.2 Una coda implementata con un array $Q[1\,..\,12]$. Gli elementi della coda appaiono soltanto nelle posizioni con sfondo grigio chiaro. **(a)** La coda ha 5 elementi nelle posizioni $Q[7\,..\,11]$. **(b)** La configurazione della coda dopo le chiamate ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$ e ENQUEUE$(Q, 5)$. **(c)** La configurazione della coda dopo che la chiamata DEQUEUE(Q) ha rimosso il valore chiave 15, che si trovava all'inizio della coda. Il nuovo valore chiave dell'inizio della coda adesso è 6.

elemento che arriva sarà inserito nella coda. Gli elementi della coda occupano le posizioni $head[Q], head[Q]+1, \ldots, tail[Q]-1$, poi "vanno a capo" nel senso che la posizione 1 segue immediatamente la posizione n secondo un ordine circolare. Se $head[Q] = tail[Q]$, la coda è vuota. All'inizio $head[Q] = tail[Q] = 1$. Se la coda è vuota, il tentativo di rimuovere un elemento dalla coda provoca un underflow. Se $head[Q] = tail[Q] + 1$, la coda è piena e il tentativo di inserire un elemento provoca un overflow.

Nelle nostre procedure ENQUEUE e DEQUEUE abbiamo omesso i controlli degli errori di underflow e overflow. L'Esercizio 10.1-4 vi chiederà di aggiungere il codice che controlla queste due condizioni di errore.

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = lunghezza[Q]
3     then tail[Q] ← 1
4     else tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = lunghezza[Q]
3     then head[Q] ← 1
4     else head[Q] ← head[Q] + 1
5  return x
```

La Figura 10.2 illustra gli effetti delle operazioni ENQUEUE e DEQUEUE. Ogni operazione richiede un tempo $O(1)$.

Esercizi

10.1-1

Utilizzando La Figura 10.1 come modello, illustrate il risultato di ogni operazione nella sequenza PUSH$(S, 4)$, PUSH$(S, 1)$, PUSH$(S, 3)$, POP(S), PUSH$(S, 8)$ e POP(S) su uno stack S inizialmente vuoto memorizzato nell'array $S[1\,..\,6]$.

10.1-2
Spiegate come implementare due stack in un array $A[1\,..\,n]$ in modo da non provocare overflow in nessuno dei due stack, a meno che il numero totale di elementi nei due stack non sia n. Le operazioni PUSH e POP dovrebbero essere eseguite nel tempo $O(1)$.

10.1-3
Utilizzando la Figura 10.2 come modello, illustrate il risultato di ogni operazione nella sequenza ENQUEUE$(Q, 4)$, ENQUEUE$(Q, 1)$, ENQUEUE$(Q, 3)$, DEQUEUE(Q), ENQUEUE$(Q, 8)$ e DEQUEUE(Q) su una coda Q inizialmente vuota memorizzata nell'array $Q[1\,..\,6]$.

10.1-4
Riscrivete ENQUEUE e DEQUEUE per rilevare le condizioni di underflow e overflow di una coda.

10.1-5
Mentre uno stack consente l'inserimento e la cancellazione di elementi in una sola estremità e una coda consente l'inserimento in una estremità e la cancellazione nell'altra estremità, una ***coda doppia*** o ***deque*** (double-ended queue) permette di inserire e cancellare elementi in entrambe le estremità. Scrivete quattro procedure con tempo $O(1)$ per inserire e cancellare elementi in entrambe le estremità di una coda doppia costruita da un array.

10.1-6
Spiegate come implementare una coda utilizzando due stack. Analizzate il tempo di esecuzione delle operazioni sulla coda.

10.1-7
Spiegate come implementare uno stack utilizzando due code. Analizzate il tempo di esecuzione delle operazioni sullo stack.

10.2 Liste concatenate

Una ***lista concatenata*** è una struttura dati i cui oggetti sono disposti in ordine lineare. Diversamente da un array in cui l'ordine lineare è determinato dagli indici dell'array, l'ordine in una lista concatenata è determinato da un puntatore in ogni oggetto. Le liste concatenate sono una rappresentazione semplice e flessibile degli insiemi dinamici; supportano (anche se in modo non necessariamente efficiente) tutte le operazioni elencate a pagina 166.

Come illustra la Figura 10.3, ogni elemento di una ***lista doppiamente concatenata*** L è un oggetto con un campo chiave key e altri due campi puntatori: $next$ e $prev$. L'oggetto può anche contenere altri dati satelliti. Dato un elemento x nella lista, $next[x]$ punta al suo successore nella lista concatenata, mentre $prev[x]$ punta al suo predecessore. Se $prev[x] = \text{NIL}$, l'elemento x non ha un predecessore e quindi è il primo elemento della lista, detto ***testa*** o ***head*** della lista. Se $next[x] = \text{NIL}$, l'elemento x non ha un successore e quindi è l'ultimo elemento della lista, che è detto ***coda*** o ***tail*** della lista. Un attributo $head[L]$ punta al primo elemento della lista. Se $head[L] = \text{NIL}$, la lista è vuota.

Una lista può avere varie forme: può essere singolarmente o doppiamente concatenata, ordinata oppure no, circolare oppure no. Se una lista è ***singolarmente concatenata***, omettiamo il puntatore $prev$ in ogni elemento. Se una lista è ***ordinata***,

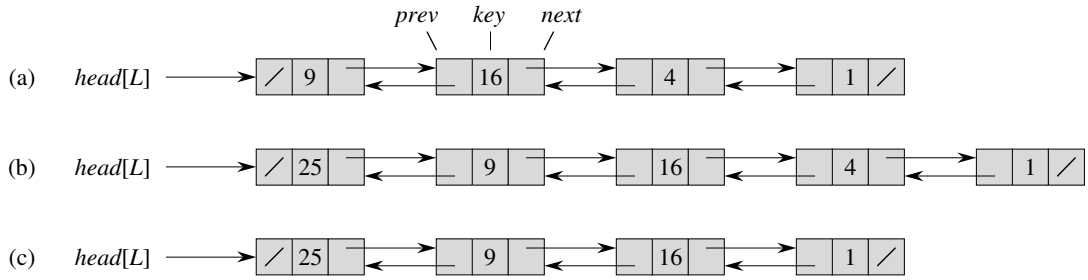


Figura 10.3 **(a)** Una lista doppiamente concatenata L che rappresenta l'insieme dinamico $\{1, 4, 9, 16\}$. Ogni elemento della lista è un oggetto con campi per la chiave e i puntatori (rappresentati da frecce) agli oggetti precedente e successivo. Il campo $next$ della coda e il campo $prev$ della testa sono NIL (questo valore speciale è indicato da una barra inclinata). L'attributo $head[L]$ punta al primo elemento della lista. **(b)** Dopo l'esecuzione di LIST-INSERT(L, x), dove $key[x] = 25$, la lista concatenata ha un nuovo oggetto con chiave 25 come nuova testa. Questo nuovo oggetto punta alla vecchia testa con chiave 9. **(c)** Il risultato della successiva chiamata LIST-DELETE(L, x), dove x punta all'oggetto con chiave 4.

l'ordine lineare della lista corrisponde all'ordine lineare delle chiavi memorizzate negli elementi della lista; l'elemento minimo è la testa della lista e l'elemento massimo è la coda della lista. Una lista può essere ***non ordinata***; gli elementi di questa lista possono presentarsi in qualsiasi ordine. In una lista ***circolare***, il puntatore $prev$ della testa della lista punta alla coda e il puntatore $next$ della coda della lista punta alla testa. Pertanto, la lista può essere vista come un anello di elementi. Nella parte restante di questo paragrafo faremo l'ipotesi che le liste con le quali operiamo siano non ordinate e doppiamente concatenate.

Ricerca in una lista concatenata

La procedura LIST-SEARCH(L, k) trova il primo elemento con la chiave k nella lista L mediante una semplice ricerca lineare, restituendo un puntatore a questo elemento. Se nessun oggetto con la chiave k è presente nella lista, allora viene restituito il valore NIL. Per la lista concatenata nella Figura 10.3(a), la chiamata LIST-SEARCH$(L, 4)$ restituisce un puntatore al terzo elemento e la chiamata LIST-SEARCH$(L, 7)$ restituisce NIL.

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3       do x ← next[x]
4  return x
```

Per effettuare una ricerca in una lista di n oggetti, la procedura LIST-SEARCH impiega il tempo $\Theta(n)$ nel caso peggiore, in quanto potrebbe essere necessario esaminare l'intera lista.

Inserire un elemento in una lista concatenata

Dato un elemento x il cui campo key sia stato già impostato, la procedura LIST-INSERT inserisce x davanti alla lista concatenata, come illustra la Figura 10.3(b).

LIST-INSERT(L, x)
1 $next[x] \leftarrow head[L]$
2 **if** $head[L] \neq$ NIL
3 **then** $prev[head[L]] \leftarrow x$
4 $head[L] \leftarrow x$
5 $prev[x] \leftarrow$ NIL

Il tempo di esecuzione di LIST-INSERT con una lista di n elementi è $O(1)$.

Cancellare un elemento da una lista concatenata

La procedura LIST-DELETE rimuove un elemento x da una lista concatenata L. Deve ricevere un puntatore a x; poi elimina x dalla lista aggiornando i puntatori. Per cancellare un elemento con una data chiave, dobbiamo prima chiamare LIST-SEARCH per ottenere un puntatore all'elemento.

LIST-DELETE(L, x)
1 **if** $prev[x] \neq$ NIL
2 **then** $next[prev[x]] \leftarrow next[x]$
3 **else** $head[L] \leftarrow next[x]$
4 **if** $next[x] \neq$ NIL
5 **then** $prev[next[x]] \leftarrow prev[x]$

La Figura 10.3(c) mostra come viene cancellato un elemento da una lista concatenata. LIST-DELETE viene eseguita nel tempo $O(1)$, ma se vogliamo eliminare un elemento con una data chiave, occorre un tempo $\Theta(n)$ nel caso peggiore, perché dobbiamo prima chiamare LIST-SEARCH.

Sentinelle

Il codice di LIST-DELETE sarebbe più semplice se potessimo ignorare le condizioni al contorno nella testa e nella coda della lista.

LIST-DELETE$'(L, x)$
1 $next[prev[x]] \leftarrow next[x]$
2 $prev[next[x]] \leftarrow prev[x]$

Una ***sentinella*** è un oggetto fittizio che ci consente di semplificare le condizioni al contorno. Per esempio, supponiamo di fornire alla lista L un oggetto $nil[L]$ che rappresenta NIL, ma ha tutti i campi degli altri elementi della lista. Ogni volta che troviamo un riferimento a NIL nel codice della lista, sostituiamolo con un riferimento alla sentinella $nil[L]$. Come illustra la Figura 10.4, questo trasforma una normale lista doppiamente concatenata in una ***lista circolare doppiamente concatenata con una sentinella***, dove la sentinella $nil[L]$ è posta fra la testa e la coda; il campo $next[nil[L]]$ punta alla testa della lista, mentre $prev[nil[L]]$ punta alla coda. Analogamente, il campo $next$ della coda e il campo $prev$ della testa puntano entrambi a $nil[L]$. Poiché $next[nil[L]]$ punta alla testa, possiamo eliminare del tutto l'attributo $head[L]$, sostituendo i suoi riferimenti con i riferimenti a $next[nil[L]]$. Una lista vuota è formata soltanto dalla sentinella, in quanto $next[nil[L]]$ e $prev[nil[L]]$ possono essere impostati entrambi a $nil[L]$.

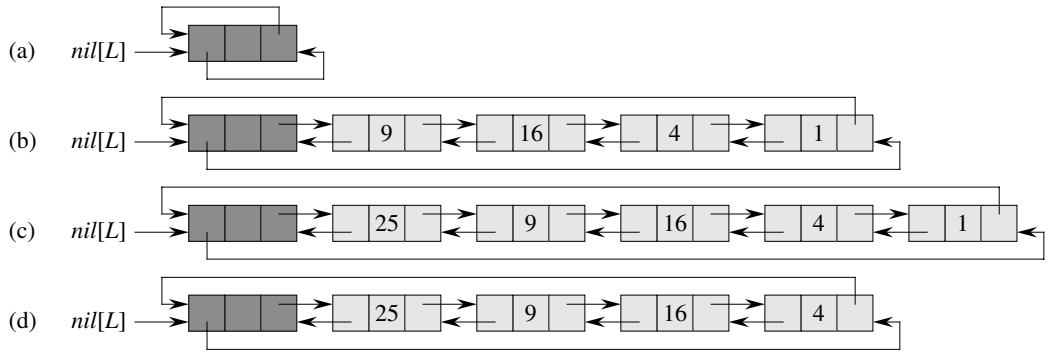


Figura 10.4 Una lista circolare doppiamente concatenata con una sentinella. La sentinella $nil[L]$ è posta fra la testa e la coda. L'attributo $head[L]$ non è più richiesto, in quanto possiamo accedere alla testa della lista tramite $next[nil[L]]$. **(a)** Una lista vuota. **(b)** La lista concatenata della Figura 10.3(a) con la chiave 9 in testa e la chiave 1 in coda. **(c)** La lista dopo l'esecuzione di LIST-INSERT$'(L, x)$, dove $key[x] = 25$. Il nuovo oggetto diventa la testa della lista. **(d)** La lista dopo l'eliminazione dell'oggetto con chiave 1. La nuova coda è l'oggetto con chiave 4.

Il codice di LIST-SEARCH è uguale al precedente, a parte la modifica dei riferimenti a NIL e $head[L]$.

LIST-SEARCH$'(L, k)$

```
1  x ← next[nil[L]]
2  while x ≠ nil[L] and key[x] ≠ k
3        do x ← next[x]
4  return x
```

Utilizziamo la procedura di due righe LIST-DELETE$'$ per cancellare un elemento dalla lista. Utilizziamo la seguente procedura per inserire un elemento nella lista.

LIST-INSERT$'(L, x)$

```
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

La Figura 10.4 illustra gli effetti di LIST-INSERT$'$ e LIST-DELETE$'$ su una lista campione.

Le sentinelle raramente abbassano i limiti asintotici sui tempi delle operazioni con le strutture dati, ma possono ridurre i fattori costanti. Il vantaggio derivante dall'uso delle sentinelle all'interno dei cicli, di solito, riguarda la chiarezza del codice, più che la velocità; il codice della lista concatenata, per esempio, si semplifica se si usano le sentinelle, ma si risparmia soltanto il tempo $O(1)$ nelle procedure LIST-INSERT$'$ e LIST-DELETE$'$. In altre situazione, tuttavia, l'uso delle sentinelle aiuta a compattare il codice in un ciclo, riducendo così il coefficiente, diciamo, di n o n^2 nel tempo di esecuzione.

Le sentinelle non dovrebbero essere utilizzate in modo indiscriminato. Se ci sono molte liste piccole, lo spazio extra in memoria utilizzato dalle loro sentinelle potrebbe essere un notevole spreco di memoria. In questo libro utilizziamo le sentinelle soltanto se semplificano davvero il codice.

Esercizi

10.2-1
L'operazione INSERT per gli insiemi dinamici può essere implementata per una lista singolarmente concatenata nel tempo $O(1)$? E l'operazione DELETE?

10.2-2
Implementate uno stack utilizzando una lista singolarmente concatenata L. Le operazioni PUSH e POP dovrebbero richiedere ancora il tempo $O(1)$.

10.2-3
Implementate una coda utilizzando una lista singolarmente concatenata L. Le operazioni ENQUEUE e DEQUEUE dovrebbero richiedere ancora il tempo $O(1)$.

10.2-4
Come detto in precedenza, ogni iterazione del ciclo nella procedura LIST-SEARCH$'$ richiede due test: uno per $x \neq nil[L]$ e l'altro per $key[x] \neq k$. Spiegate come eliminare il test per $x \neq nil[L]$ in ogni iterazione.

10.2-5
Implementate le operazioni per i dizionari INSERT, DELETE e SEARCH utilizzando liste circolari singolarmente concatenate. Quali sono i tempi di esecuzione delle vostre procedure?

10.2-6
L'operazione UNION per gli insiemi dinamici richiede due insiemi disgiunti S_1 e S_2 come input e restituisce un insieme $S = S_1 \cup S_2$ che è formato da tutti gli elementi di S_1 e S_2. Gli insiemi S_1 e S_2, di solito, vengono distrutti dall'operazione. Spiegate come realizzare UNION nel tempo $O(1)$ utilizzando una lista appropriata come struttura dati.

10.2-7
Scrivete una procedura non ricorsiva con tempo $\Theta(n)$ che inverte una lista singolarmente concatenata di n elementi. La procedura dovrebbe utilizzare non più di una quantità costante di memoria oltre a quella richiesta per la lista stessa.

10.2-8 $\star$
Spiegate come implementare una lista doppiamente concatenata utilizzando un solo puntatore $np[x]$ per ogni elemento, anziché i due puntatori $next$ e $prev$. Supponete che tutti i valori del puntatore siano interpretati come interi di k bit e definite $np[x]$ come $np[x] = next[x]$ XOR $prev[x]$, l'operatore "or esclusivo" di k bit fra $next[x]$ e $prev[x]$ (il valore NIL è rappresentato da 0). Controllate di avere descritto tutte le informazioni che sono necessarie per accedere alla testa della lista. Spiegate come implementare le procedure SEARCH, INSERT e DELETE per tale lista. Spiegate inoltre come invertire la lista nel tempo $O(1)$.

10.3 Implementare puntatori e oggetti

Come possiamo implementare puntatori e oggetti in quei linguaggi, come il Fortran, che non li prevedono? In questo paragrafo descriveremo due tecniche per implementare alcune strutture dati concatenate senza utilizzare un esplicito tipo di dato puntatore. Sintetizzeremo oggetti e puntatori tramite array e indici di array.

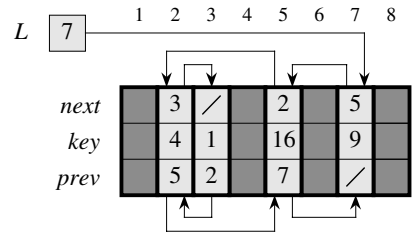


Figura 10.5 La lista della Figura 10.3(a) rappresentata dagli array key, $next$ e $prev$. Ogni segmento verticale degli array rappresenta un singolo oggetto. I puntatori memorizzati corrispondono agli indici degli array indicati in alto; le frecce mostrano come interpretarli. Le posizioni degli oggetti su sfondo grigio chiaro contengono gli elementi della lista. La variabile L contiene l'indice della testa della lista.

Rappresentazione di oggetti con più array

È possibile rappresentare una collezione di oggetti che hanno gli stessi campi utilizzando un array per ogni campo. Per esempio, la Figura 10.5 illustra come possiamo implementare la lista concatenata della Figura 10.3(a) con tre array. L'array key contiene i valori delle chiavi che si trovano correntemente nell'insieme dinamico; i puntatori sono memorizzati negli array $next$ e $prev$. Per un dato indice x dell'array, $key[x]$, $next[x]$ e $prev[x]$ rappresentano un oggetto della lista concatenata. Questo significa che un puntatore x è semplicemente un indice comune agli array key, $next$ e $prev$.

Nella Figura 10.3(a) l'oggetto con chiave 4 segue l'oggetto con chiave 16 nella lista concatenata. Nella Figura 10.5 la chiave 4 appare in $key[2]$ e la chiave 16 appare in $key[5]$, quindi abbiamo $next[5] = 2$ e $prev[2] = 5$. Sebbene la costante NIL appaia nel campo $next$ della coda e nel campo $prev$ della testa, di solito, utilizziamo un intero (come 0 o -1) che non può essere affatto un indice di array. Una variabile L contiene l'indice della testa della lista.

Nel nostro pseudocodice abbiamo utilizzato le parentesi quadre per denotare sia l'indice di un array sia il campo (attributo) di un oggetto. In entrambi i casi, il significato di $key[x]$, $next[x]$ e $prev[x]$ è coerente con l'implementazione pratica.

Rappresentazione di oggetti con un solo array

Le parole nella memoria di un calcolatore, tipicamente, sono indirizzate da numeri interi compresi fra 0 e $M - 1$, dove M è un intero opportunamente grande. In molti linguaggi di programmazione un oggetto occupa un insieme contiguo di locazioni nella memoria del calcolatore. Un puntatore è semplicemente l'indirizzo della prima locazione di memoria dell'oggetto; le altre locazioni di memoria all'interno dell'oggetto possono essere indicizzate aggiungendo un offset al puntatore.

Per implementare gli oggetti, possiamo adottare la stessa strategia degli ambienti di programmazione che non prevedono un esplicito tipo di dato puntatore. Per esempio, la Figura 10.6 mostra come un singolo array A possa essere utilizzato per memorizzare la lista concatenata delle Figure 10.3(a) e 10.5. Un oggetto occupa un sottoarray contiguo $A[j \,..\, k]$. Ogni campo dell'oggetto corrisponde a un offset nell'intervallo fra 0 e $k - j$; l'indice j è il puntatore dell'oggetto. Nella Figura 10.6 gli offset che corrispondono a key, $next$ e $prev$ sono, rispettivamente, 0, 1 e 2. Per leggere il valore di $prev[i]$, dato un puntatore i, aggiungiamo il valore i del puntatore all'offset 2, ottenendo $A[i + 2]$.

La rappresentazione con un solo array è flessibile perché consente agli oggetti di lunghezza differente di essere memorizzati nello stesso array. Il problema di gestire tale collezione eterogenea di oggetti è più difficile del problema di gestire una collezione omogenea, dove tutti gli oggetti hanno gli stessi campi. Poiché la

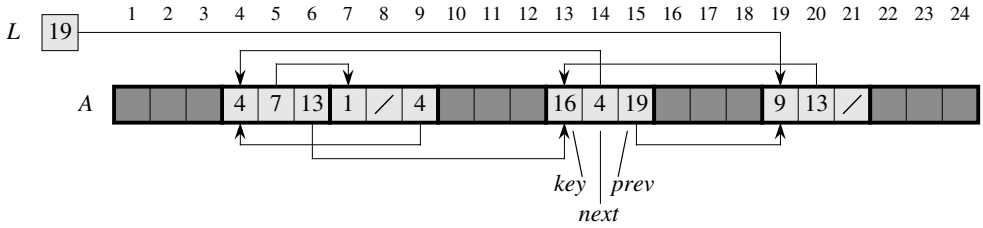


Figura 10.6 La lista concatenata delle Figure 10.3(a) e 10.5 rappresentata con un singolo array A. Ogni elemento della lista è un oggetto che occupa un sottoarray contiguo di lunghezza 3 all'interno dell'array. I tre campi key, $next$ e $prev$ corrispondono, rispettivamente, agli offset 0, 1 e 2. Il puntatore di un oggetto è l'indice del primo elemento dell'oggetto. Gli oggetti che contengono gli elementi della lista sono illustrati con uno sfondo grigio chiaro; le frecce indicano l'ordinamento della lista.

maggior parte delle strutture dati che esamineremo sono composte da elementi omogenei, sarà sufficiente per i nostri scopi utilizzare la rappresentazione degli oggetti con più array.

Allocare e liberare gli oggetti

Per inserire una chiave in un insieme dinamico rappresentato da una lista doppiamente concatenata, dobbiamo allocare un puntatore a un oggetto correntemente inutilizzato nella rappresentazione della lista concatenata. Quindi, è utile gestire lo spazio in memoria degli oggetti che non sono correntemente utilizzati nella rappresentazione della lista concatenata, in modo da potere allocare nuovi oggetti. In alcuni sistemi, esistono dei meccanismi automatici, detti ***garbage collector*** (spazzini della memoria), che hanno la responsabilità di determinare quali oggetti sono inutilizzati. Molte applicazioni, però, sono sufficientemente semplici da potere svolgere il compito di restituire un oggetto inutilizzato a un gestore della memoria. Esaminiamo adesso il problema di allocare e liberare (o deallocare) gli oggetti omogenei utilizzando l'esempio di una lista doppiamente concatenata rappresentata da più array.

Supponiamo che gli array nella rappresentazione con più array abbiano lunghezza m e che in qualche istante l'insieme dinamico contenga $n \leq m$ elementi. Allora n oggetti rappresentano gli elementi che si trovano correntemente nell'insieme dinamico e i restanti $m - n$ oggetti sono ***liberi***; gli oggetti liberi possono essere utilizzati per rappresentare gli elementi da inserire in futuro nell'insieme dinamico.

Manteniamo gli oggetti liberi in una lista singolarmente concatenata, che chiameremo ***free list***. Questa lista usa soltanto l'array $next$, che contiene i puntatori $next$ all'interno della lista. La testa della free list è mantenuta nella variabile globale $free$. Quando l'insieme dinamico rappresentato dalla lista concatenata L non è vuoto, la free list può essere intrecciata con la lista L, come illustra la Figura 10.7. Notate che ogni oggetto nella rappresentazione può trovarsi nella lista L o nella free list, ma non in entrambe le liste.

La free list è uno stack: il prossimo oggetto allocato è l'ultimo che è stato liberato. Possiamo adattare le operazioni su stack PUSH e POP per implementare, rispettivamente, le procedure per allocare e liberare gli oggetti. Supponiamo che la variabile globale $free$ utilizzata nelle seguenti procedure punti al primo elemento della free list.

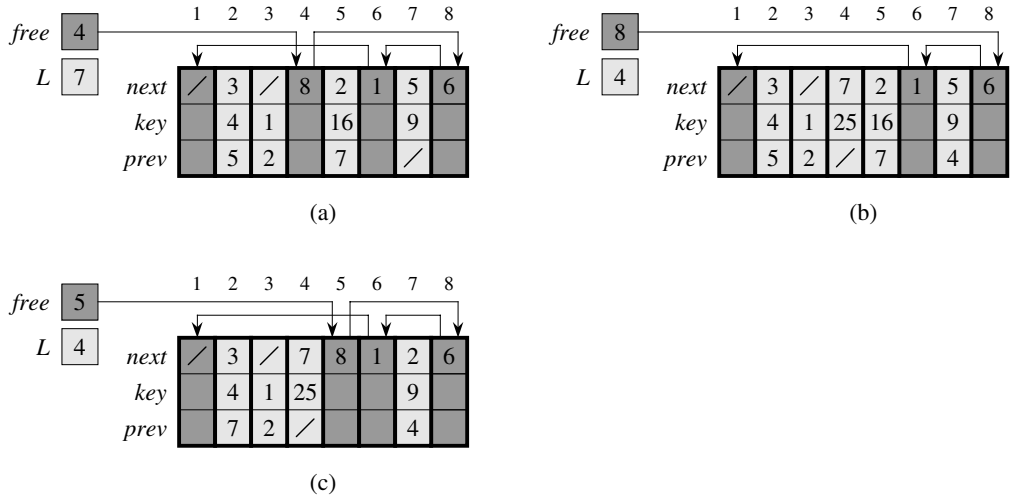


Figura 10.7 L'effetto delle procedure ALLOCATE-OBJECT e FREE-OBJECT. **(a)** La lista della Figura 10.5 (su sfondo chiaro) e una free list (su sfondo scuro). Le frecce indicano la struttura della free list. **(b)** Il risultato della chiamata ALLOCATE-OBJECT() (che restituisce l'indice 4), dell'impostazione di $key[4]$ a 25 e della chiamata LIST-INSERT$(L, 4)$. La nuova testa della free list è l'oggetto 8, che prima era $next[4]$ nella free list. **(c)** Dopo l'esecuzione di LIST-DELETE$(L, 5)$, viene chiamata la procedura FREE-OBJECT(5). L'oggetto 5 diventa la nuova testa della free list e l'oggetto 8 lo segue nella free list.

```
ALLOCATE-OBJECT()
1  if free = NIL
2     then error "Spazio esaurito!"
3     else x ← free
4          free ← next[x]
5          return x
```

```
FREE-OBJECT(x)
1  next[x] ← free
2  free ← x
```

La free list inizialmente contiene tutti gli n oggetti non allocati. Quando lo spazio nella free list è esaurito, la procedura ALLOCATE-OBJECT segnala un errore. È tipico utilizzare un'unica free list a servizio di più liste concatenate. La Figura 10.8 illustra due liste concatenate e una free list intrecciate tramite gli array key, $next$ e $prev$.

Le due procedure vengono eseguite nel tempo $O(1)$; questo le rende molto praticabili. Possono essere modificate per operare con qualsiasi collezione omogenea di oggetti, utilizzando uno dei campi dell'oggetto come campo $next$ nella free list.

Esercizi

10.3-1

Disegnate uno schema della sequenza $\langle 13, 4, 8, 19, 5, 11 \rangle$ memorizzata come una lista doppiamente concatenata utilizzando la rappresentazione con più array. Fate lo stesso per la rappresentazione con un solo array.

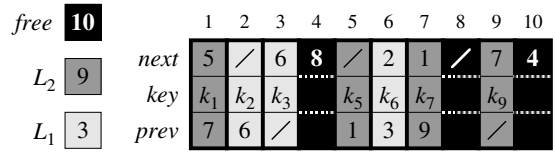


Figura 10.8 Due liste concatenate, L_1 (su sfondo chiaro) e L_2 (su sfondo scuro), intrecciate con una free list (su sfondo più scuro).

10.3-2
Scrivete le procedure ALLOCATE-OBJECT e FREE-OBJECT per una collezione omogenea di oggetti implementata da una rappresentazione con un solo array.

10.3-3
Perché non occorre impostare o resettare i campi $prev$ degli oggetti nell'implementazione delle procedure ALLOCATE-OBJECT e FREE-OBJECT?

10.3-4
Spesso è preferibile mantenere compatta la memorizzazione di tutti gli elementi di una lista doppiamente concatenata, utilizzando, per esempio, le prime m locazioni degli indici nella rappresentazione con più array (è questo il caso di un ambiente di calcolo con memoria virtuale paginata). Spiegate come implementare le procedure ALLOCATE-OBJECT e FREE-OBJECT in modo che la rappresentazione sia compatta. Supponete che non ci siano puntatori a elementi della lista concatenata all'esterno della lista stessa (*suggerimento:* usate l'implementazione di uno stack mediante array).

10.3-5
Sia L una lista doppiamente concatenata di lunghezza m memorizzata negli array key, $prev$ e $next$ di lunghezza n. Supponete che questi array siano gestiti dalle procedure ALLOCATE-OBJECT e FREE-OBJECT che mantengono una free list F doppiamente concatenata. Supponete inoltre che m degli n elementi siano nella lista L e $n - m$ elementi siano nella free list. Scrivete una procedura COMPACTIFY-LIST(L, F) che, date la lista L e la free list F, sposta gli elementi in L in modo che occupino le posizioni dell'array $1, 2, \ldots, m$ e regola la free list F in modo che resti corretta, occupando le posizioni dell'array $m+1, m+2, \ldots, n$. Il tempo di esecuzione della vostra procedura dovrebbe essere $\Theta(m)$; la procedura dovrebbe utilizzare soltanto una quantità costante di spazio extra. Dimostrate la correttezza della vostra procedura.

10.4 Rappresentazione di alberi con radice

I metodi per rappresentare le liste che abbiamo descritto nel precedente paragrafo si estendono a qualsiasi struttura dati omogenea. In questo paragrafo, analizzeremo specificatamente il problema di rappresentare gli alberi con radice mediante strutture dati concatenate. Esamineremo prima gli alberi binari e poi un metodo per gli alberi con radice i cui nodi possono avere un numero arbitrario di figli.

Rappresentiamo i singoli nodi di un albero con un oggetto. Analogamente alle liste concatenate, supponiamo che ogni nodo contenga un campo key. I restanti campi che ci interessano sono puntatori ad altri nodi, che variano a seconda del tipo di albero.

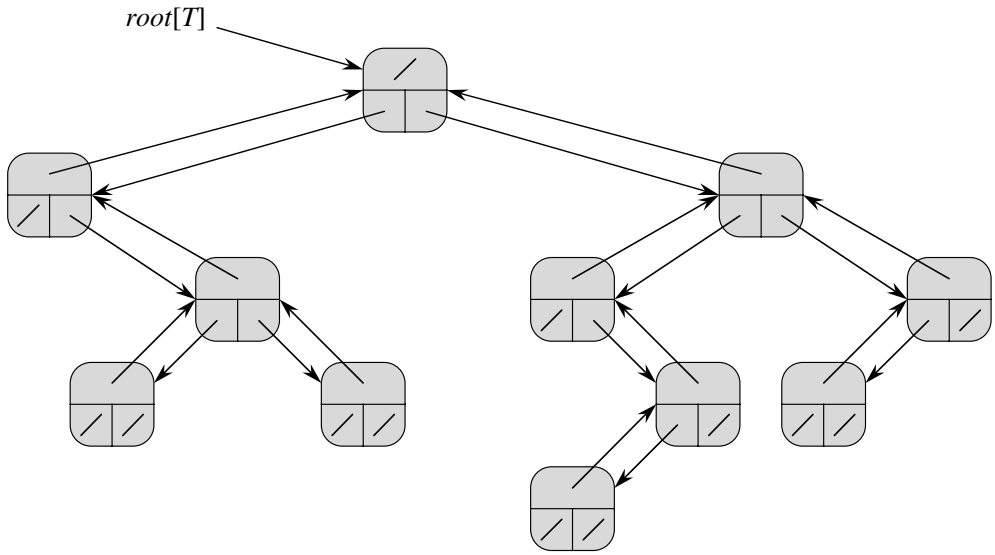


Figura 10.9 La rappresentazione di un albero binario T. Ogni nodo x ha i campi $p[x]$ (in alto), $left[x]$ (in basso a sinistra) e $right[x]$ (in basso a destra). I campi key non sono indicati.

Alberi binari

Come illustra la Figura 10.9, utilizziamo i campi p, $left$ e $right$ per memorizzare i puntatori al padre, al figlio sinistro e al figlio destro di ogni nodo in un albero binario T. Se $p[x] = \text{NIL}$, allora x è la radice. Se il nodo x non ha un figlio sinistro, allora $left[x] = \text{NIL}$; lo stesso vale per il figlio destro. L'attributo $root[T]$ punta alla radice dell'intero albero T. Se $root[T] = \text{NIL}$, allora l'albero è vuoto.

Alberi con ramificazione senza limite

Lo schema per rappresentare un albero binario può essere esteso a qualsiasi classe di alberi in cui il numero di figli di ogni nodo sia al massimo una qualche costante k: sostituiamo i campi $left$ e $right$ con $child_1, child_2, \ldots, child_k$. Questo schema non funziona più se il numero di figli di un nodo è illimitato, perché non sappiamo in anticipo quanti campi (gli array nella rappresentazione con più array) allocare. Inoltre, anche nel caso in cui il numero di figli k fosse limitato da una grande costante, ma la maggior parte dei nodi avesse un piccolo numero di figli, potremmo sprecare una notevole quantità di memoria.

Fortunatamente, esiste uno schema geniale che usa gli alberi binari per rappresentare gli alberi con un numero arbitrario di figli. Questo schema ha il vantaggio di utilizzare soltanto uno spazio $O(n)$ per qualsiasi albero con radice di n nodi. La ***rappresentazione figlio-sinistro fratello-destro*** è illustrata nella Figura 10.10. Come prima, ogni nodo contiene un puntatore p al padre e $root[T]$ punta alla radice dell'albero T. Anziché avere un puntatore a ciascuno dei suoi figli, però, ogni nodo x ha soltanto due puntatori:

1. $left\text{-}child[x]$ punta al figlio più a sinistra del nodo x.
2. $right\text{-}sibling[x]$ punta al fratello di x immediatamente a destra.

Se il nodo x non ha figli, allora $left\text{-}child[x] = \text{NIL}$. Se il nodo x è il figlio più a destra di suo padre, allora $right\text{-}sibling[x] = \text{NIL}$.

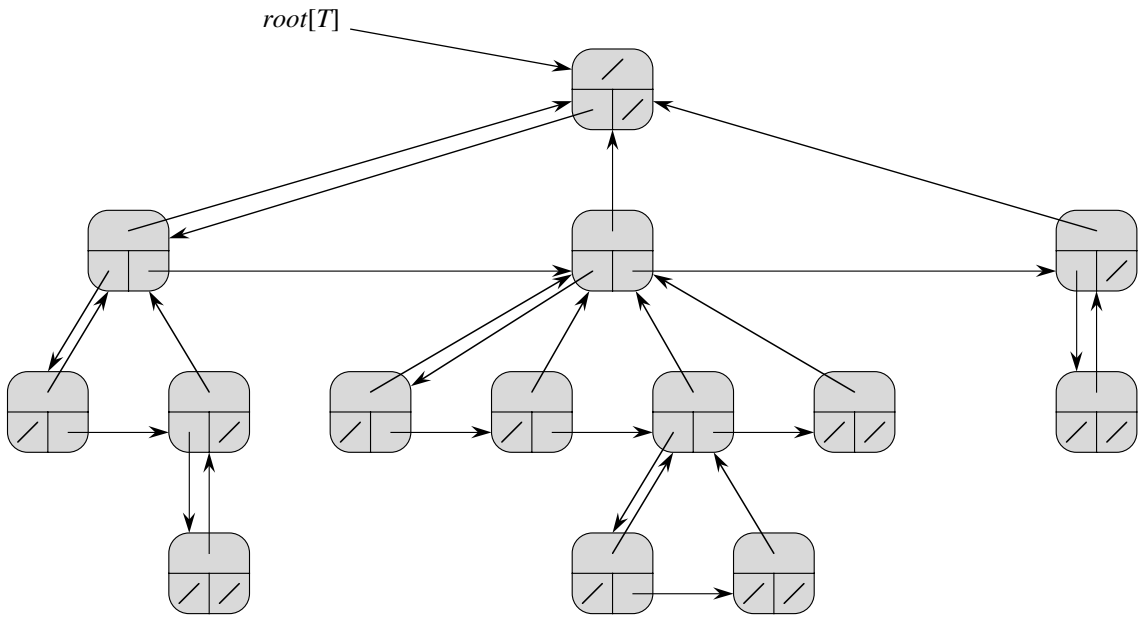


Figura 10.10 La rappresentazione figlio-sinistro, fratello-destro di un albero T. Ogni nodo x ha i campi $p[x]$ (in alto), $left\text{-}child[x]$ (in basso a sinistra) e $right\text{-}sibling[x]$ (in basso a destra). I campi key non sono indicati.

Altre rappresentazioni di alberi

Gli alberi con radice, a volte, vengono rappresentati in altri modi. Nel Capitolo 6, per esempio, abbiamo rappresentato un heap (che si basa su un albero binario completo) mediante un solo array e un indice. Gli alberi presentati nel Capitolo 21 sono attraversati soltanto in direzione della radice, quindi sono presenti soltanto i puntatori al padre; mancano i puntatori ai figli. Esistono molti altri schemi. Lo schema migliore dipende dall'applicazione.

Esercizi

10.4-1

Disegnate l'albero binario con radice di indice 6 che è rappresentato dai seguenti campi.

indice	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2

Scrivete una procedura ricorsiva con tempo $O(n)$ che, dato un albero binario di n nodi, visualizza la chiave di ogni nodo dell'albero.

10.4-3
Scrivete una procedura non ricorsiva con tempo $O(n)$ che, dato un albero binario di n nodi, visualizza la chiave di ogni nodo dell'albero. Utilizzate uno stack come struttura dati ausiliaria.

10.4-4
Scrivete una procedura con tempo $O(n)$ che visualizza tutte le chiavi di un albero arbitrario con radice di n nodi, dove l'albero è memorizzato utilizzando la rappresentazione figlio-sinistro fratello-destro.

10.4-5 $\star$
Scrivete una procedura non ricorsiva con tempo $O(n)$ che, dato un albero binario di n nodi, visualizza la chiave di ogni nodo. La procedura non deve utilizzare più di una quantità costante di memoria all'esterno dell'albero stesso e non deve modificare l'albero, neanche solo temporaneamente, durante l'esecuzione.

10.4-6 $\star$
La rappresentazione figlio-sinistro fratello-destro di un albero arbitrario con radice usa tre puntatori in ogni nodo: $left\text{-}child$, $right\text{-}sibling$ e $parent$. Partendo da un nodo qualsiasi, il padre di questo nodo può essere raggiunto e identificato in un tempo costante e tutti i suoi figli possono essere raggiunti e identificati in un tempo lineare nel numero dei figli. Spiegate come utilizzare soltanto due puntatori e un valore booleano in ogni nodo in modo che il padre di un nodo o tutti i suoi figli possano essere raggiunti e identificati in un tempo lineare nel numero dei figli.

Problemi

10-1 Confronti fra liste

Per ciascuno dei quattro tipi di liste della seguente tabella, qual è il tempo di esecuzione asintotico nel caso peggiore per ciascuna delle operazioni su un insieme dinamico?

	Non ordinata, singolarmente concatenata	Ordinata, singolarmente concatenata	Non ordinata, doppiamente concatenata	Ordinata, doppiamente concatenata
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2 Heap fondibili con liste concatenate

Un ***heap fondibile*** (mergeable heap) supporta le seguenti operazioni: MAKE-HEAP (che crea un heap fondibile vuoto), INSERT, MINIMUM, EXTRACT-MIN e

UNION.[1] Spiegate come implementare gli heap fondibili utilizzando le liste concatenate in ciascuno dei seguenti casi. Provate a rendere ogni operazione più efficiente possibile. Analizzate il tempo di esecuzione di ogni operazione in funzione della dimensione degli insiemi dinamici utilizzati.

a. Le liste sono ordinate.

b. Le liste non sono ordinate.

c. Le liste non sono ordinate e gli insiemi dinamici da fondere sono disgiunti.

10-3 Ricerca in una lista ordinata compatta

L'Esercizio 10.3-4 chiedeva come mantenere compatta una lista di n elementi nelle prime n posizioni di un array. Supponete che tutte le chiavi siano distinte e che la lista compatta sia anche ordinata, ovvero $key[i] < key[next[i]]$ per ogni $i = 1, 2, \ldots, n$ tale che $next[i] \neq$ NIL. Sotto questi ipotesi, dimostrate che il seguente algoritmo randomizzato può essere utilizzato per effettuare delle ricerche nella lista nel tempo atteso $O(\sqrt{n})$.

```
COMPACT-LIST-SEARCH(L, n, k)
 1  i ← head[L]
 2  while i ≠ NIL and key[i] < k
 3        do j ← RANDOM(1, n)
 4           if key[i] < key[j] and key[j] ≤ k
 5             then i ← j
 6                  if key[i] = k
 7                    then return i
 8           i ← next[i]
 9  if i = NIL or key[i] > k
10     then return NIL
11     else  return i
```

Se ignoriamo le righe 3–7 della procedura, abbiamo un normale algoritmo di ricerca in una lista concatenata ordinata, dove l'indice i punta, a turno, a ogni posizione della lista. La ricerca termina quando l'indice i "va oltre" la fine della lista o quando $key[i] \geq k$. Nel secondo caso, se $key[i] = k$, chiaramente abbiamo trovato una chiave con il valore k. Se, invece, $key[i] > k$, allora non troveremo mai una chiave con il valore k e, quindi, bisogna interrompere la ricerca.

Le righe 3–7 tentano di saltare su una posizione j scelta a caso. Tale salto è vantaggioso se $key[j]$ è maggiore di $key[i]$ e non maggiore di k; nel qual caso, j indica una posizione nella lista che i dovrà raggiungere durante una normale ricerca. Poiché la lista è compatta, sappiamo che qualsiasi scelta di j tra 1 e n indica qualche oggetto della lista, non un elemento della free list.

---

[1] Poiché, secondo la nostra definizione, un heap fondibile supporta le operazioni MINIMUM e EXTRACT-MIN, possiamo fare riferimento a questo heap anche con il termine ***min-heap fondibile***. D'altra parte, se un heap fondibile supportasse le operazioni MAXIMUM e EXTRACT-MAX, potremmo chiamarlo ***max-heap fondibile***.

Anziché analizzare direttamente le prestazioni di COMPACT-LIST-SEARCH, esamineremo un algoritmo correlato, COMPACT-LIST-SEARCH$'$, che esegue due cicli distinti. Questo algoritmo richiede un parametro addizionale t, che determina un limite superiore sul numero di iterazioni del primo ciclo.

```
COMPACT-LIST-SEARCH'(L, n, k, t)
 1  i ← head[L]
 2  for q ← 1 to t
 3       do j ← RANDOM(1, n)
 4          if key[i] < key[j] and key[j] ≤ k
 5            then i ← j
 6                 if key[i] = k
 7                   then return i
 8  while i ≠ NIL and key[i] < k
 9       do i ← next[i]
10  if i = NIL or key[i] > k
11     then return NIL
12     else  return i
```

Per confrontare l'esecuzione degli algoritmi COMPACT-LIST-SEARCH(L, n, k) e COMPACT-LIST-SEARCH$'(L, n, k, t)$, supponete che la sequenza degli interi restituita dalle chiamate di RANDOM$(1, n)$ sia la stessa per entrambi gli algoritmi.

a. Supponete che COMPACT-LIST-SEARCH(L, n, k) richieda t iterazioni del ciclo **while** (righe 2–8). Dimostrate che COMPACT-LIST-SEARCH$'(L, n, k, t)$ fornisce lo stesso risultato e che il numero totale di iterazioni dei due cicli **for** e **while** all'interno di COMPACT-LIST-SEARCH$'$ è almeno t.

Nella chiamata COMPACT-LIST-SEARCH$'(L, n, k, t)$, sia X_t la variabile casuale che descrive la distanza nella lista concatenata (attraverso la catena dei puntatori $next$) dalla posizione i alla chiave desiderata k, dopo t iterazioni del ciclo **for** (righe 2–7).

b. Verificate che il tempo di esecuzione atteso di COMPACT-LIST-SEARCH$'(L, n, k, t)$ può essere espresso da $O(t + \mathrm{E}[X_t])$.

c. Dimostrate la relazione $\mathrm{E}[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$ (*suggerimento:* applicate l'equazione (C.24)).

d. Dimostrate che $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$.

e. Verificate che $\mathrm{E}[X_t] \leq n/(t+1)$.

f. Dimostrate che COMPACT-LIST-SEARCH$'(L, n, k, t)$ viene eseguito nel tempo atteso $O(t + n/t)$.

g. Concludete che il tempo di esecuzione atteso di COMPACT-LIST-SEARCH è $O(\sqrt{n})$.

h. Perché bisogna supporre che tutte le chiavi siano distinte in COMPACT-LIST-SEARCH? Dimostrate che i salti casuali non necessariamente aiutano asintoticamente quando la lista contiene chiavi ripetute.

Note

Aho, Hopcroft e Ullman [6] e Knuth [182] sono eccellenti testi di riferimento per le strutture dati elementari. Molti altri testi trattano sia le strutture dati di base sia la loro implementazione in un particolare linguaggio di programmazione. Fra questi libri di testo citiamo Goodrich e Tamassia [128], Main [209], Shaffer [273] e Weiss [310, 312, 313]. Gonnet [126] ha raccolto i dati sperimentali sulle prestazioni di molte operazioni con le strutture dati.

L'origine degli stack e delle code come strutture dati dell'informatica non è chiara, in quanto le corrispondenti nozioni esistevano già nella matematica e nella pratica aziendale basata su carta prima dell'introduzione dei calcolatori digitali. Knuth [182] cita A. M. Turing in merito allo sviluppo degli stack per il concatenamento delle subroutine nel 1947.

Anche le strutture dati basate sui puntatori sembrano essere un'invenzione popolare. Secondo Knuth, sembra che i puntatori siano stati utilizzati nei primi calcolatori con memorie a tamburo. Il linguaggio A-1 sviluppato da G. M. Hopper nel 1951 rappresentava le formule algebriche come alberi binari. Knuth attribuisce al linguaggio IPL-II, sviluppato nel 1956 da A. Newell, J. C. Shaw e H. A. Simon, il riconoscimento dell'importanza e della diffusione dei puntatori. Il loro linguaggio IPL-III, sviluppato nel 1957, includeva esplicite operazioni con gli stack.

11 Hashing

Molte applicazioni richiedono un insieme dinamico che supporta soltanto le operazioni di dizionario INSERT, SEARCH e DELETE. Per esempio, il compilatore di un linguaggio di programmazione mantiene una tabella di simboli, nella quale le chiavi degli elementi sono stringhe di caratteri arbitrarie che corrispondono a identificatori nel linguaggio.

Una tabella hash è una struttura dati efficace per implementare i dizionari. Sebbene la ricerca di un elemento in una tabella hash possa richiedere lo stesso tempo per ricercare un elemento in una lista concatenata – il tempo $\Theta(n)$ nel caso peggiore – in pratica l'hashing ha ottime prestazioni. Sotto ipotesi ragionevoli, il tempo atteso per cercare un elemento in una tabella hash è $O(1)$.

Una tabella hash è una generalizzazione della nozione più semplice di array ordinario. L'indirizzamento diretto in un array ordinario rende efficiente l'uso delle nostre capacità di esaminare una posizione arbitraria in un array nel tempo $O(1)$. Il Paragrafo 11.1 descrive dettagliatamente l'indirizzamento diretto; questo tipo di indirizzamento è applicabile quando possiamo permetterci di allocare un array che ha una posizione per ogni chiave possibile.

Quando il numero di chiavi effettivamente memorizzate è piccolo rispetto al numero totale di chiavi possibili, le tabelle hash diventano un'alternativa efficiente per indirizzare direttamente un array, in quanto una tabella hash tipicamente usa un array di dimensione proporzionale al numero di chiavi effettivamente memorizzate. Anziché utilizzare una chiave direttamente come un indice dell'array, l'indice viene *calcolato* dalla chiave.

Il Paragrafo 11.2 presenta i concetti principali delle tabelle hash, mettendo in particolare evidenza il *concatenamento* come un metodo per gestire le *collisioni* (si verifica una collisione quando più chiavi vengono associate a uno stesso indice di array). Il Paragrafo 11.3 spiega come calcolare gli indici di un array dalle chiavi utilizzando le funzioni hash. Presenteremo e analizzeremo alcune variazioni sul tema di base. Il Paragrafo 11.4 descrive un altro metodo per gestire le collisioni: l'*indirizzamento aperto*. Il concetto essenziale è che l'hashing è una tecnica estremamente pratica ed efficace: le operazioni fondamentali sui dizionari in media richiedono soltanto un tempo $O(1)$. Il Paragrafo 11.5 descrive l'*hashing perfetto* che permette di effettuare ricerche nel tempo $O(1)$ nel *caso peggiore*, quando l'insieme delle chiavi da memorizzare è statico (cioè, quando l'insieme delle chiavi non cambia più, una volta che è stato memorizzato).

11.1 Tabelle a indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo. Supponiamo che un'applicazione

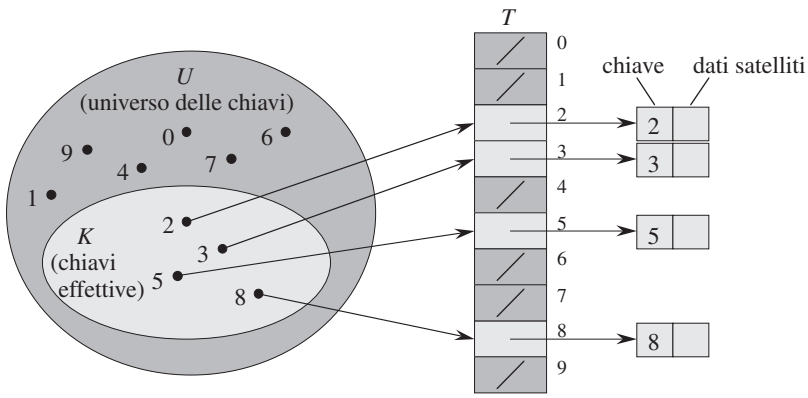


Figura 11.1
Implementazione di un insieme dinamico tramite la tabella a indirizzamento diretto T. Ogni chiave nell'universo $U = \{0, 1, \ldots, 9\}$ corrisponde a un indice della tabella. L'insieme $K = \{2, 3, 5, 8\}$ delle chiavi effettive determina gli slot nella tabella che contengono i puntatori agli elementi. Gli altri slot (su sfondo più scuro) contengono la costante NIL.

abbia bisogno di un insieme dinamico in cui ogni elemento ha una chiave estratta dall'universo $U = \{0, 1, \ldots, m-1\}$, dove m non è troppo grande. Supponiamo inoltre che due elementi non possano avere la stessa chiave.

Per rappresentare l'insieme dinamico, utilizziamo un array o ***tabella a indirizzamento diretto***, che indicheremo con $T[0\,..\,m-1]$, dove ogni posizione o ***slot*** corrisponde a una chiave nell'universo U. La Figura 11.1 illustra il metodo; lo slot k punta a un elemento dell'insieme con chiave k. Se l'insieme non contiene l'elemento con chiave k, allora $T[k] = \text{NIL}$.

Le operazioni di dizionario sono semplici da implementare.

DIRECT-ADDRESS-SEARCH(T, k)
  return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
  $T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
  $T[key[x]] \leftarrow \text{NIL}$

Ciascuna di queste operazioni è veloce: basta il tempo $O(1)$.

Per alcune applicazioni, gli elementi dell'insieme dinamico possono essere memorizzati nella stessa tabella a indirizzamento diretto. Ovvero, anziché memorizzare la chiave e i dati satelliti di un elemento in un oggetto esterno alla tabella a indirizzamento diretto, con un puntatore da uno slot della tabella all'oggetto, possiamo memorizzare l'oggetto nello slot stesso, risparmiando spazio. Inoltre, spesso non è necessario memorizzare la chiave dell'oggetto, in quanto se abbiamo l'indice di un oggetto nella tabella, abbiamo la sua chiave. Se le chiavi non sono memorizzate, però, dobbiamo disporre di qualche metodo per sapere se uno slot è vuoto.

Esercizi

11.1-1

Supponete che un insieme dinamico S sia rappresentato da una tabella a indirizzamento diretto T di lunghezza m. Descrivete una procedura che trova l'elemento massimo di S. Qual è la prestazione della vostra procedura nel caso peggiore?

11.1-2
Un ***vettore di bit*** è semplicemente un array che contiene i bit 0 e 1. Un vettore di bit di lunghezza m occupa meno spazio di un array con m puntatori. Descrivete come utilizzare un vettore di bit per rappresentare un insieme dinamico di elementi distinti senza dati satelliti. Le operazioni di dizionario dovrebbero essere eseguite nel tempo $O(1)$.

11.1-3
Spiegate come implementare una tabella a indirizzamento diretto in cui le chiavi degli elementi memorizzati non hanno bisogno di essere distinte e gli elementi possono avere dati satelliti. Tutte e tre le operazioni di dizionario (INSERT, DELETE e SEARCH) dovrebbero essere eseguite nel tempo $O(1)$ (non dimenticate che DELETE richiede come argomento un puntatore a un oggetto da cancellare, non una chiave).

11.1-4 $\star$
Provate a implementare un dizionario utilizzando l'indirizzamento diretto su un array di enorme dimensione. All'inizio, le voci dell'array possono contenere dati non significativi; non sarebbe sensato inizializzare l'intero array, considerando la sua dimensione. Descrivete uno schema per implementare un dizionario a indirizzamento diretto su un array di tale dimensione. Ogni oggetto memorizzato dovrebbe occupare lo spazio $O(1)$; le operazioni SEARCH, INSERT e DELETE dovrebbero impiegare ciascuna un tempo $O(1)$; l'inizializzazione della struttura dati dovrebbe richiedere un tempo $O(1)$ (*suggerimento:* per capire se una voce dell'array è valida oppure no, usate uno stack aggiuntivo di dimensione pari al numero di chiavi effettivamente memorizzate nel dizionario).

11.2 Tabelle hash

La difficoltà dell'indirizzamento diretto è ovvia: se l'universo delle chiavi U è esteso, memorizzare una tabella T di dimensione $|U|$ può essere impraticabile, o perfino impossibile, considerando la memoria disponibile in un tipico calcolatore. Inoltre, l'insieme K delle chiavi *effettivamente memorizzate* può essere così piccolo rispetto a U che la maggior parte dello spazio allocato per la tabella T sarebbe sprecato.

Quando l'insieme K delle chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le chiavi possibili, una tabella hash richiede molto meno spazio di una tabella a indirizzamento diretto. Specificatamente, lo spazio richiesto può essere ridotto a $\Theta(|K|)$, senza perdere il vantaggio di ricercare un elemento nella tabella hash nel tempo $O(1)$. L'unico problema è che questo limite vale per il *tempo medio*, mentre nell'indirizzamento diretto vale per il *tempo nel caso peggiore*.

Con l'indirizzamento diretto, un elemento con chiave k è memorizzato nello slot k. Con l'hashing, questo elemento è memorizzato nello slot $h(k)$; cioè, utilizziamo una ***funzione hash*** h per calcolare lo slot dalla chiave k. Qui h associa l'universo U delle chiavi agli slot di una ***tabella hash*** $T[0\,..\,m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\}$$

Diciamo che un elemento con chiave k corrisponde allo slot $h(k)$ o anche che $h(k)$ è il ***valore hash*** della chiave k. La Figura 11.2 illustra il concetto di base.

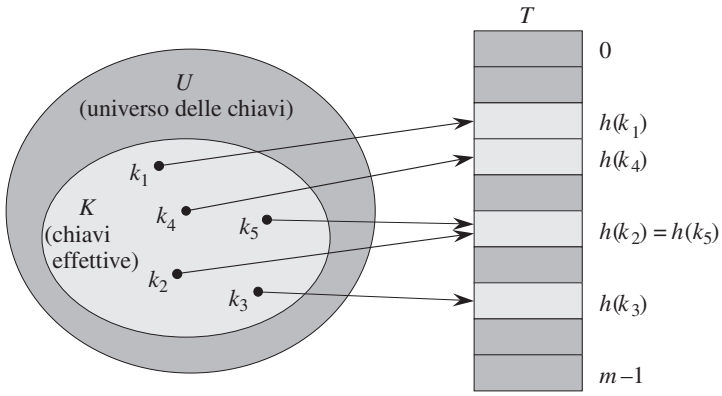


Figura 11.2 Usare una funzione hash h per associare le chiavi agli slot della tabella hash. Le chiavi k_2 e k_5 sono associate allo stesso slot, quindi sono in collisione.

Il compito della funzione hash è ridurre l'intervallo degli indici dell'array che devono essere gestiti. Anziché $|U|$ valori, abbiamo bisogno di gestire soltanto m valori; corrispondentemente, si riduce lo spazio richiesto in memoria.

C'è un problema: due chiavi possono essere associate allo stesso slot. Questo evento si chiama ***collisione***. Fortunatamente, ci sono delle tecniche efficaci per risolvere i conflitti creati dalle collisioni.

Ovviamente, la soluzione ideale sarebbe quella di evitare qualsiasi collisione. Potremmo tentare di raggiungere questo obiettivo scegliendo un'opportuna funzione hash h. Un'idea potrebbe essere quella di fare apparire "casuale" la funzione h, evitando così le collisioni o almeno riducendo al minimo il loro numero. Il significato letterale del termine "hash" (polpettone fatto con avanzi di carne e verdure tritati) evoca l'immagine di rimescolamenti e spezzettamenti casuali, che rendono bene l'idea ispiratrice di questo approccio (naturalmente, una funzione hash deve essere deterministica, nel senso che un dato input k dovrebbe produrre sempre lo stesso output $h(k)$). Tuttavia, poiché $|U| > m$, ci devono essere almeno due chiavi che hanno lo stesso valore hash; evitare completamente le collisioni è quindi impossibile. In definitiva, anche se una funzione hash ben progettata e apparentemente "casuale" può ridurre al minimo il numero di collisioni, occorre comunque un metodo per risolvere le collisioni che si verificano.

Il resto di questo paragrafo descrive la tecnica più semplice per risolvere le collisioni: il concatenamento. Il Paragrafo 11.4 presenta un metodo alternativo per risolvere le collisioni: l'indirizzamento aperto.

Risoluzione delle collisioni mediante concatenamento

Nel ***concatenamento*** poniamo tutti gli elementi che sono associati allo stesso slot in una lista concatenata, come illustra la Figura 11.3. Lo slot j contiene un puntatore alla testa della lista di tutti gli elementi memorizzati che corrispondono a j; se tali elementi non esistono, lo slot j contiene la costante NIL. Le operazioni di dizionario su una tabella hash T sono facili da implementare quando le collisioni sono risolte con il concatenamento.

CHAINED-HASH-INSERT(T, x)

inserisce x nella testa della lista $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)

ricerca un elemento con chiave k nella lista $T[h(k)]$

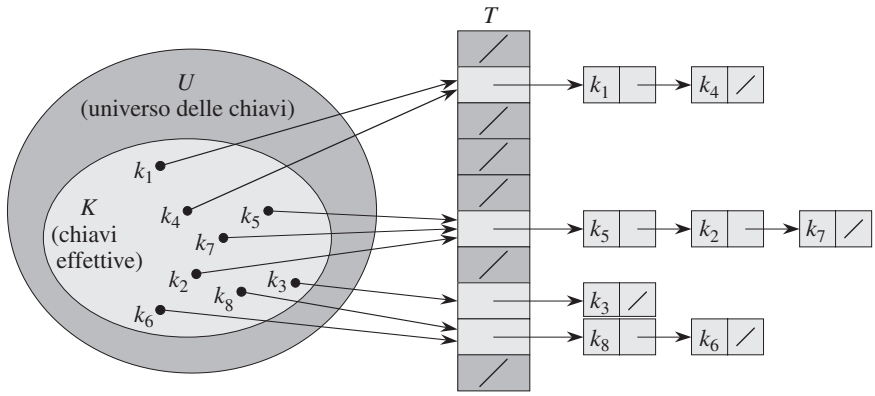


Figura 11.3 Risoluzione delle collisioni mediante concatenamento. Ogni slot $T[j]$ della tabella hash contiene una lista concatenata di tutte le chiavi il cui valore hash è j. Per esempio, $h(k_1) = h(k_4)$ e $h(k_5) = h(k_2) = h(k_7)$.

CHAINED-HASH-DELETE(T, x)
cancella x dalla lista $T[h(key[x])]$

Il tempo di esecuzione nel caso peggiore per l'inserimento è $O(1)$. La procedura di inserimento è veloce, in parte perché suppone che l'elemento x da inserire non sia presente nella tabella; se necessario, questa ipotesi può essere verificata (con un costo aggiuntivo) svolgendo una ricerca, prima di effettuare l'inserimento. Per la ricerca, il tempo di esecuzione nel caso peggiore è proporzionale alla lunghezza della lista; in seguito analizzeremo questa operazione più dettagliatamente. La cancellazione di un elemento x può essere realizzata nel tempo $O(1)$ se le liste sono doppiamente concatenate.

Notate che CHAINED-HASH-DELETE prende come input un elemento x, non la sua chiave k, quindi non occorre cercare prima x. Se le liste fossero singolarmente concatenate, non sarebbe di grande aiuto prendere come input l'elemento x al posto della chiave k. Dobbiamo comunque trovare x nella lista $T[h(key[x])]$, in modo che il collegamento $next$ del predecessore di x possa essere appropriatamente impostato per rimuovere x. In questo caso, la cancellazione e la ricerca avrebbero essenzialmente lo stesso tempo di esecuzione.

Analisi dell'hashing con concatenamento

Come sono le prestazioni dell'hashing con il concatenamento? In particolare, quanto tempo occorre per ricercare un elemento con una data chiave?

Data una tabella hash T con m slot dove sono memorizzati n elementi, definiamo ***fattore di carico*** α della tabella T il rapporto n/m, ossia il numero medio di elementi memorizzati in una catena. La nostra analisi sarà fatta in funzione di α, che può essere minore, uguale o maggiore di 1.

Il comportamento nel caso peggiore dell'hashing con concatenamento è pessimo: tutte le n chiavi sono associate allo stesso slot, creando una lista di lunghezza n. Il tempo di esecuzione della ricerca è quindi $\Theta(n)$ più il tempo per calcolare la funzione hash – non migliore del tempo richiesto nel caso in cui avessimo utilizzato una lista concatenata per tutti gli elementi. Chiaramente, le tabelle hash non sono utilizzate per le loro prestazioni nel caso peggiore (tuttavia, l'hashing perfetto, descritto nel Paragrafo 11.5, presenta buone prestazioni nel caso peggiore quando l'insieme delle chiavi è statico).

Le prestazioni medie dell'hashing dipendono, in media, dal modo in cui la funzione hash h distribuisce l'insieme delle chiavi da memorizzare tra gli m slot. Il Paragrafo 11.3 tratta questi problemi; per adesso supponiamo che qualsiasi elemento abbia la stessa probabilità di essere associato a uno qualsiasi degli m slot, indipendentemente dallo slot cui sarà associato qualsiasi altro elemento. Questa ipotesi è alla base dell'***hashing uniforme semplice***.

Per $j = 0, 1, \ldots, m-1$, indicando con n_j la lunghezza della lista $T[j]$, avremo

$$n = n_0 + n_1 + \cdots + n_{m-1} \tag{11.1}$$

e il valore medio di n_j sarà $\mathrm{E}[n_j] = \alpha = n/m$.

Supponiamo che il valore hash $h(k)$ possa essere calcolato nel tempo $O(1)$, in modo che il tempo richiesto per cercare un elemento con chiave k dipenda linearmente dalla lunghezza $n_{h(k)}$ della lista $T[h(k)]$. Mettendo da parte il tempo $O(1)$ richiesto per calcolare la funzione hash e accedere allo slot $h(k)$, consideriamo il numero atteso di elementi esaminati dall'algoritmo di ricerca, ovvero il numero di elementi nella lista $T[h(k)]$ che vengono controllati per vedere se le loro chiavi sono uguali a k. Considereremo due casi. Nel primo caso, la ricerca non ha successo: nessun elemento nella tabella ha la chiave k. Nel secondo caso, la ricerca ha successo e viene trovato un elemento con chiave k.

Teorema 11.1
In una tabella hash le cui collisioni sono risolte con il concatenamento, una ricerca senza successo richiede un tempo atteso $\Theta(1+\alpha)$, nell'ipotesi di hashing uniforme semplice.

Dimostrazione Nell'ipotesi di hashing uniforme semplice, qualsiasi chiave k non ancora memorizzata nella tabella ha la stessa probabilità di essere associata a uno qualsiasi degli m slot. Il tempo atteso per ricercare senza successo una chiave k è il tempo atteso per svolgere le ricerche fino alla fine della lista $T[h(k)]$, che ha una lunghezza attesa pari a $\mathrm{E}[n_{h(k)}] = \alpha$. Quindi, il numero atteso di elementi esaminato in una ricerca senza successo è α e il tempo totale richiesto (incluso quello per calcolare $h(k)$) è $\Theta(1 + \alpha)$. ∎

Il caso di ricerca con successo è un po' differente, perché ogni lista non ha la stessa probabilità di essere oggetto delle ricerche. Piuttosto, la probabilità che una lista sia oggetto delle ricerche è proporzionale al numero di elementi che contiene. Nonostante questo, il tempo atteso è ancora $\Theta(1 + \alpha)$.

Teorema 11.2
In una tabella hash le cui collisioni sono risolte con il concatenamento, una ricerca con successo richiede, in media, un tempo $\Theta(1 + \alpha)$, nell'ipotesi di hashing uniforme semplice.

Dimostrazione Supponiamo che l'elemento da ricercare abbia la stessa probabilità di essere uno qualsiasi degli n elementi memorizzati nella tabella. Il numero di elementi esaminati durante una ricerca con successo di un elemento x è uno in più del numero di elementi che si trovano prima di x nella lista di x. Gli elementi che precedono x nella lista sono stati inseriti tutti dopo di x, perché i nuovi elementi vengono posti all'inizio della lista. Per trovare il numero atteso di elementi esaminati, prendiamo la media, sugli n elementi x nella tabella, di 1 più il numero atteso di elementi aggiunti alla lista di x dopo che x è stato aggiunto alla lista.

Indichiamo con x_i l'i-esimo elemento inserito nella tabella, per $i = 1, 2, \ldots, n$, e sia $k_i = key[x_i]$. Per le chiavi k_i e k_j, definiamo la variabile casuale indicatrice $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$. Nell'ipotesi di hashing uniforme semplice, abbiamo $\Pr\{h(k_i) = h(k_j)\} = 1/m$ e, quindi, per il Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/m$. Dunque, il numero atteso di elementi esaminati in una ricerca con successo è

$$
\begin{aligned}
&\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&\quad= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \quad \text{(per la linearità del valore atteso)} \\
&\quad= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&\quad= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\quad= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&\quad= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(per l'equazione (A.1))} \\
&\quad= 1+\frac{n-1}{2m} \\
&\quad= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

In conclusione, il tempo totale richiesto per una ricerca con successo (incluso il tempo per calcolare la funzione hash) è $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

Qual è il significato di questa analisi? Se il numero di slot della tabella hash è almeno proporzionale al numero di elementi della tabella, abbiamo $n = O(m)$ e, di conseguenza, $\alpha = n/m = O(m)/m = O(1)$. Pertanto, la ricerca richiede in media un tempo costante. Poiché l'inserimento richiede il tempo $O(1)$ nel caso peggiore e la cancellazione richiede il tempo $O(1)$ nel caso peggiore quando le liste sono doppiamente concatenate, tutte le operazioni di dizionario possono essere svolte, in media, nel tempo $O(1)$.

Esercizi

11.2-1

Supponete di usare una funzione hash h per associare n chiavi distinte a un array T di lunghezza m. Nell'ipotesi di hashing uniforme semplice, qual è il numero atteso di collisioni? Più precisamente, qual è la cardinalità attesa di $\{\{k, l\} : k \neq l \text{ e } h(k) = h(l)\}$?

11.2-2

Provate a inserire le chiavi $5, 28, 19, 15, 20, 33, 12, 17, 10$ in una tabella hash, risolvendo le collisioni mediante il concatenamento. Supponete che la tabella abbia 9 slot e che la funzione hash sia $h(k) = k \bmod 9$.

11.2-3
Il professor Marley ritiene che si possa ottenere un notevole miglioramento delle prestazioni modificando lo schema del concatenamento in modo che ogni lista sia mantenuta ordinata. Come influisce la modifica del professore sul tempo di esecuzione delle ricerche con successo, delle ricerche senza successo, degli inserimenti e delle cancellazioni?

11.2-4
Indicate come allocare e liberare lo spazio per gli elementi all'interno della stessa tabella hash concatenando tutti gli slot inutilizzati in una free list. Supponete che uno slot possa contenere un flag, un elemento e un puntatore oppure un flag e due puntatori. Tutte le operazioni del dizionario e della free list dovrebbero essere eseguite nel tempo atteso $O(1)$. La free list deve essere doppiamente concatenata o è sufficiente che sia singolarmente concatenata?

11.2-5
Dimostrate che, se $|U| > nm$, esiste un sottoinsieme di U di dimensione n formato da chiavi che sono associate tutte allo stesso slot e quindi il tempo di esecuzione nel caso peggiore per l'hashing con concatenamento è $\Theta(n)$.

11.3 Funzioni hash

In questo paragrafo, tratteremo alcuni problemi che riguardano il progetto di buone funzioni hash; poi presenteremo tre schemi per la loro realizzazione. Due degli schemi, l'hashing per divisione e l'hashing per moltiplicazione, sono euristici nella natura, mentre il terzo schema, l'hashing universale, usa la randomizzazione per fornire buone prestazioni dimostrabili.

Caratteristiche di una buona funzione hash

Una buona funzione hash soddisfa (approssimativamente) l'ipotesi dell'hashing uniforme semplice: ogni chiave ha la stessa probabilità di essere associata a uno qualsiasi degli m slot, indipendentemente dallo slot cui sarà associata qualsiasi altra chiave.

Purtroppo, di solito non è possibile verificare questa condizione, in quanto raramente è nota la distribuzione delle probabilità secondo la quale vengono estratte le chiavi, e le chiavi potrebbero non essere estratte in maniera indipendente. A volte tale distribuzione è nota. Per esempio, se le chiavi sono numeri reali casuali k distribuiti in modo indipendente e uniforme nell'intervallo $0 \leq k < 1$, la funzione hash

$$h(k) = \lfloor km \rfloor$$

soddisfa la condizione dell'hashing uniforme semplice.

Nella pratica, spesso è possibile utilizzare le tecniche euristiche per realizzare funzioni hash con buone prestazioni. Le informazioni qualitative sulla distribuzione delle chiavi possono essere utili in questa fase di progettazione. Per esempio, considerate la tabella dei simboli di un compilatore, in cui le chiavi sono stringhe di caratteri che rappresentano gli identificatori all'interno di un programma. Simboli strettamente correlati, come `pt` e `pts`, si trovano spesso nello stesso programma. Una buona funzione hash dovrebbe ridurre al minimo la probabilità che tali varianti siano associate allo stesso slot.

Un buon approccio consiste nel derivare il valore hash in modo che sia indipendente da qualsiasi schema che possa esistere nei dati. Per esempio, il "metodo della divisione" (descritto nel Paragrafo 11.3.1) calcola il valore hash come il resto della divisione fra la chiave e un determinato numero primo. Questo metodo spesso fornisce buoni risultati, purché il numero primo sia scelto in modo da non essere correlato a nessuno schema nella distribuzione delle chiavi.

Infine, notiamo che alcune applicazioni delle funzioni hash potrebbero richiedere proprietà più vincolanti di quelle richieste dall'hashing uniforme semplice. Per esempio, potremmo richiedere che le chiavi che sono "vicine" in qualche maniera forniscano valori hash che siano distanti (questa proprietà è particolarmente desiderabile quando si usa la scansione lineare, che è definita nel Paragrafo 11.4). L'hashing universale, descritto nel Paragrafo 11.3.3, spesso offre le proprietà richieste.

Interpretare le chiavi come numeri naturali

La maggior parte delle funzioni hash suppone che l'universo delle chiavi sia l'insieme dei numeri naturali $\mathbf{N} = \{0, 1, 2, \ldots\}$. Quindi, se le chiavi non sono numeri naturali, occorre un metodo per interpretarli come tali. Per esempio, una stringa di caratteri può essere interpretata come un numero intero espresso nella notazione di un'opportuna base. Quindi, l'identificatore `pt` può essere interpretato come la coppia di interi $(112, 116)$ nella rappresentazione decimale, in quanto `p` $= 112$ e `t` $= 116$ nel set dei caratteri ASCII; poi, esprimendo questa coppia di valori nella notazione con base 128, `pt` diventa $(112 \cdot 128) + 116 = 14452$. Di solito è semplice in un'applicazione trovare un metodo per interpretare ogni chiave come un numero naturale (eventualmente grande). Nei prossimi paragrafi supporremo che le chiavi siano numeri naturali.

11.3.1 Il metodo della divisione

Quando si applica il ***metodo della divisione*** per creare una funzione hash, una chiave k viene associata a uno degli m slot prendendo il resto della divisione fra k e m; cioè la funzione hash è

$$h(k) = k \bmod m$$

Per esempio, se la tabella hash ha dimensione $m = 12$ e la chiave è $k = 100$, allora $h(k) = 4$. Questo metodo è molto veloce perché richiede una sola operazione di divisione.

Quando utilizziamo il metodo della divisione, di solito, evitiamo certi valori di m. Per esempio, m non dovrebbe essere una potenza di 2, perché se $m = 2^p$, allora $h(k)$ rappresenta proprio i p bit meno significativi di k. A meno che non sia noto che tutti gli schemi dei p bit di ordine inferiore abbiano la stessa probabilità, è meglio rendere la funzione hash dipendente da tutti i bit della chiave. Come vi sarà chiesto di dimostrare nell'Esercizio 11.3-3, scegliere $m = 2^p - 1$, quando k è una stringa di caratteri interpretata nella base 2^p, potrebbe essere una cattiva soluzione, perché la permutazione dei caratteri di k non cambia il suo valore hash.

Un numero primo non troppo vicino a una potenza esatta di 2 è spesso una buona scelta per m. Per esempio, supponiamo di allocare una tabella hash (risolvendo le collisioni mediante concatenamento) per contenere circa $n = 2000$ stringhe di caratteri, dove ogni carattere ha 8 bit. Poiché non ci dispiace esaminare in media 3

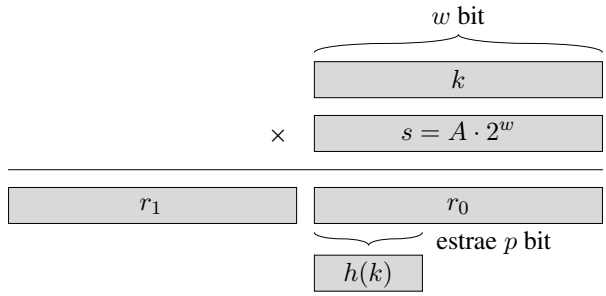


Figura 11.4 Il metodo della moltiplicazione per creare una funzione hash. Il valore della chiave k, rappresentato con w bit, viene moltiplicato per il valore $s = A \cdot 2^w$, anch'esso rappresentato con w bit. I p bit più significativi della metà meno significativa del prodotto formano il valore hash $h(k)$ desiderato.

elementi in una ricerca senza successo, allochiamo una tabella hash di dimensione $m = 701$. Abbiamo scelto 701 perché è un numero primo vicino a $2000/3$, ma non a una potenza qualsiasi di 2. Trattando ogni chiave k come un numero intero, la nostra funzione hash diventa

$$h(k) = k \bmod 701$$

11.3.2 Il metodo della moltiplicazione

Il ***metodo della moltiplicazione*** per creare funzioni hash si svolge in due passaggi. Prima moltiplichiamo la chiave k per una costante A nell'intervallo $0 < A < 1$ ed estraiamo la parte frazionaria di kA. Poi moltiplichiamo questo valore per m e prendiamo la parte intera inferiore del risultato. In sintesi, la funzione hash è

$$h(k) = \lfloor m\,(k\,A \bmod 1)\rfloor$$

dove "$k\,A \bmod 1$" rappresenta la parte frazionaria di kA, cioè $kA - \lfloor kA \rfloor$.

Un vantaggio del metodo della moltiplicazione è che il valore di m non è critico. Tipicamente, lo scegliamo come una potenza di 2 ($m = 2^p$ per qualche intero p) in modo poi possiamo implementare facilmente la funzione hash nella maggior parte dei calcolatori nel modo seguente.

Supponiamo che la dimensione della parola della macchina sia w bit e che k entri in una sola parola. Limitiamo A a una frazione della forma $s/2^w$, dove s è un intero nell'intervallo $0 < s < 2^w$. Facendo riferimento alla Figura 11.4, moltiplichiamo prima k per l'intero di w bit $s = A \cdot 2^w$. Il risultato è un valore di $2w$ bit $r_1 2^w + r_0$, dove r_1 è la parola di ordine superiore del prodotto e r_0 è la parola di ordine inferiore del prodotto. Il valore hash desiderato di p bit è formato dai p bit più significativi di r_0.

Sebbene questo metodo funzioni con qualsiasi valore della costante A, tuttavia con qualche valore funziona meglio che con altri. La scelta ottimale dipende dalle caratteristiche dei dati da sottoporre all'hashing. Knuth [185] ritiene che

$$A \approx (\sqrt{5} - 1)/2 = 0,6180339887\ldots \tag{11.2}$$

abbia la probabilità di funzionare ragionevolmente bene.

Come esempio, supponiamo di avere $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ e $w = 32$. Adattando il valore proposto da Knuth, scegliamo A come frazione della forma $s/2^{32}$ che è più vicina a $(\sqrt{5} - 1)/2$, cosicché $A = 2654435769/2^{32}$. Quindi $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$ e così $r_1 = 76300$ e $r_0 = 17612864$. I 14 bit più significativi di r_0 forniscono il valore $h(k) = 67$.

⋆ 11.3.3 Hashing universale

Se un avversario sleale scegliesse le chiavi da elaborare con qualche funzione hash costante, potrebbe scegliere n chiavi che si associano tutte allo stesso slot, generando un tempo medio di ricerca pari a $\Theta(n)$. Qualsiasi funzione hash costante è vulnerabile a un comportamento così inefficiente nel caso peggiore. L'unico sistema efficace per migliorare la situazione è scegliere *casualmente* la funzione hash in modo che sia *indipendente* dalle chiavi che devono essere effettivamente memorizzate. Questo approccio, detto ***hashing universale***, può permettere di ottenere buone prestazioni in media, indipendentemente da quali chiavi sceglie l'avversario.

Il concetto fondamentale dell'hashing universale consiste nel selezionare in modo casuale la funzione hash da una classe di funzioni accuratamente progettate, all'inizio dell'esecuzione dell'algoritmo. Come nel caso di quicksort, la randomizzazione garantisce che nessun input possa provocare il comportamento nel caso peggiore dell'algoritmo. A causa della randomizzazione, l'algoritmo può comportarsi in modo differente ogni volta che viene eseguito, anche con lo stesso input, garantendo buone prestazioni nel caso medio con qualsiasi input.

Riprendendo l'esempio della tabella dei simboli di un compilatore, scopriamo che la scelta degli identificatori effettuata dal programmatore adesso non può provocare prestazioni di hashing sistematicamente scadenti. Le prestazioni scadenti si verificano soltanto quando il compilatore sceglie una funzione hash casuale che provoca un'associazione inefficiente dell'insieme degli identificatori, ma la probabilità che si verifichi questo caso è piccola ed è la stessa per qualsiasi insieme di identificatori della stessa dimensione.

Sia $\mathcal{H}$ una collezione finita di funzioni hash che associano un dato universo U di chiavi all'intervallo $\{0, 1, \ldots, m-1\}$. Tale collezione è detta ***universale*** se, per ogni coppia di chiavi distinte $k, l \in U$, il numero di funzioni hash $h \in \mathcal{H}$ per le quali $h(k) = h(l)$ è al massimo $|\mathcal{H}|/m$. In altre parole, con una funzione hash scelta a caso da $\mathcal{H}$, la probabilità di una collisione fra due chiavi distinte k e l non è maggiore della probabilità $1/m$ di una collisione nel caso in cui $h(k)$ e $h(l)$ fossero scelte in modo casuale e indipendente dall'insieme $\{0, 1, \ldots, m-1\}$.

Il seguente teorema dimostra che una classe universale di funzioni hash garantisce un buon comportamento nel caso medio. Ricordiamo che n_i indica la lunghezza della lista $T[i]$.

Teorema 11.3
Supponiamo che una funzione hash h sia scelta da una collezione universale di funzioni hash e sia utilizzata per associare n chiavi a una tabella T di dimensione m, applicando il concatenamento per risolvere le collisioni. Se la chiave k non è nella tabella, allora la lunghezza attesa $\mathrm{E}[n_{h(k)}]$ della lista dove viene associata la chiave k è al massimo α. Se la chiave k è nella tabella, allora la lunghezza attesa $\mathrm{E}[n_{h(k)}]$ della lista che contiene la chiave k è al massimo $1 + \alpha$.

Dimostrazione Notiamo che i valori attesi qui riguardano la scelta della funzione hash e non dipendono da nessuna ipotesi sulla distribuzione delle chiavi. Per ogni coppia k e l di chiavi distinte, definiamo la variabile casuale indicatrice $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Poiché per definizione, una singola coppia di chiavi collide con probabilità al massimo $1/m$, abbiamo $\Pr\{h(k) = h(l)\} \leq 1/m$ e, quindi, il Lemma 5.1 implica che $\mathrm{E}[X_{kl}] \leq 1/m$.

Adesso definiamo, per ogni chiave k, la variabile casuale Y_k che è uguale al numero di chiavi diverse da k che sono associate allo stesso slot di k, quindi

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}$$

Ne consegue che

$$\begin{aligned} \mathrm{E}[Y_k] &= \mathrm{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} \mathrm{E}[X_{kl}] \quad \text{(per la linearità del valore atteso)} \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m} \end{aligned}$$

Il resto della dimostrazione varia a seconda che la chiave k si trovi oppure no nella tabella T.

- Se $k \notin T$, allora $n_{h(k)} = Y_k$ e $|\{l : l \in T \text{ and } l \neq k\}| = n$. Quindi $\mathrm{E}[n_{h(k)}] = \mathrm{E}[Y_k] \leq n/m = \alpha$.
- Se $k \in T$, allora poiché la chiave k appare nella lista $T[h(k)]$ e il numero indicato da Y_k non include la chiave k, abbiamo $n_{h(k)} = Y_k + 1$ e $|\{l : l \in T \text{ e } l \neq k\}| = n-1$. Quindi $\mathrm{E}[n_{h(k)}] = \mathrm{E}[Y_k]+1 \leq (n-1)/m+1 = 1 + \alpha - 1/m < 1 + \alpha$. ■

Il seguente corollario dice che l'hashing universale offre il vantaggio desiderato: adesso è impossibile per un avversario scegliere una sequenza di operazioni che forza il tempo di esecuzione nel caso peggiore. Randomizzando intelligentemente la scelta della funzione hash durante l'esecuzione, abbiamo la garanzia che ogni sequenza di operazioni possa essere gestita con un buon tempo di esecuzione atteso.

Corollario 11.4
Utilizzando l'hashing universale e la risoluzione delle collisioni mediante concatenamento in una tabella con m slot, occorre il tempo atteso $\Theta(n)$ per gestire qualsiasi sequenza di n operazioni INSERT, SEARCH e DELETE che contiene $O(m)$ operazioni INSERT.

Dimostrazione Poiché il numero di inserimenti è $O(m)$, abbiamo $n = O(m)$ e quindi $\alpha = O(1)$. Le operazioni INSERT e DELETE richiedono un tempo costante e, per il Teorema 11.3, il tempo atteso per ogni operazione SEARCH è $O(1)$. Per la linearità del valore atteso, quindi, il tempo atteso dell'intera sequenza di operazioni è $O(n)$. Poiché ogni operazione richiede il tempo $\Omega(1)$, è dimostrato il limite $\Theta(n)$. ■

Progettare una classe universale di funzioni hash

È abbastanza semplice progettare una classe universale di funzioni hash, come dimostreremo con l'aiuto di un po' di teoria dei numeri. Se non avete dimestichezza con la teoria dei numeri, consultate il Capitolo 31. Iniziamo a scegliere

un numero primo p sufficientemente grande in modo che ogni possibile chiave k sia compresa nell'intervallo da 0 a $p-1$, estremi inclusivi. Indichiamo con $\mathbf{Z}_p$ l'insieme $\{0, 1, \ldots, p-1\}$ e con $\mathbf{Z}_p^*$ l'insieme $\{1, 2, \ldots, p-1\}$. Essendo p un numero primo, possiamo risolvere le equazioni modulo p con i metodi descritti nel Capitolo 31. Poiché supponiamo che la dimensione dell'universo delle chiavi sia maggiore del numero di numero di slot nella tabella hash, abbiamo $p > m$.

Adesso definiamo la funzione hash $h_{a,b}$ per ogni $a \in \mathbf{Z}_p^*$ e ogni $b \in \mathbf{Z}_p$ utilizzando una trasformazione lineare seguita da riduzioni modulo p e poi modulo m:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m \tag{11.3}$$

Per esempio, con $p = 17$ e $m = 6$, abbiamo $h_{3,4}(8) = 5$. La famiglia di tutte queste funzioni hash è

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^* \text{ e } b \in \mathbf{Z}_p\} \tag{11.4}$$

Ogni funzione hash $h_{a,b}$ associa $\mathbf{Z}_p$ a $\mathbf{Z}_m$. Questa classe di funzioni hash ha la bella proprietà che la dimensione m dell'intervallo di output è arbitraria – non necessariamente un numero primo – una caratteristica che utilizzeremo nel Paragrafo 11.5. Poiché le scelte possibili per a sono $p-1$ e le scelte possibili per b sono p, ci sono $p(p-1)$ funzioni hash in $\mathcal{H}_{p,m}$.

Teorema 11.5
La classe $\mathcal{H}_{p,m}$ delle funzioni hash definita dalle equazioni (11.3) e (11.4) è universale.

Dimostrazione Considerate due chiavi distinte k e l dell'insieme $\mathbf{Z}_p$, con $k \neq l$. Per una data funzione hash $h_{a,b}$ poniamo

$$\begin{aligned} r &= (ak + b) \bmod p \\ s &= (al + b) \bmod p \end{aligned}$$

Notiamo prima che $r \neq s$. Perché? Osserviamo che

$$r - s \equiv a(k - l) \pmod p$$

Ne consegue che $r \neq s$ perchè p è un numero primo e a e $(k-l)$ sono entrambi non nulli modulo p; quindi anche il loro prodotto deve essere non nullo modulo p per il Teorema 31.6. Pertanto, durante il calcolo di qualsiasi $h_{a,b}$ in $\mathcal{H}_{p,m}$, input distinti k e l vengono associati a valori distinti r e s modulo p; non ci sono collisioni al "livello mod p". Inoltre, ciascuna delle $p(p-1)$ possibili scelte per la coppia (a, b) con $a \neq 0$ genera una coppia (r, s) *differente* con $r \neq s$, in quanto possiamo ricavare a e b, noti i valori r e s:

$$\begin{aligned} a &= ((r - s)((k - l)^{-1} \bmod p)) \bmod p \\ b &= (r - ak) \bmod p \end{aligned}$$

Dove $((k-l)^{-1} \bmod p)$ indica l'unico inverso per la moltiplicazione, modulo p, di $k - l$. Poiché ci sono soltanto $p(p-1)$ coppie possibili (r, s) con $r \neq s$, c'è una corrispondenza uno-a-uno fra le coppie (a, b) con $a \neq 0$ e le coppie (r, s) con $r \neq s$. Quindi, per qualsiasi coppia di input k e l, se scegliamo (a, b) uniformemente a caso da $\mathbf{Z}_p^* \times \mathbf{Z}_p$, la coppia risultante (r, s) ha la stessa probabilità di essere qualsiasi coppia di valori distinti modulo p.

Ne consegue che la probabilità che le chiavi distinte k e l collidano è uguale alla probabilità che $r \equiv s \pmod{m}$ quando r e s sono scelte a caso come valori distinti modulo p. Per un dato valore di r, dei restanti $p-1$ valori possibili di s, il numero di valori s tali che $s \neq r$ e $s \equiv r \pmod{m}$ è al massimo

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p+m-1)/m) - 1 \quad \text{(per la disequazione (3.6))} \\ &= (p-1)/m \end{aligned}$$

La probabilità che s collida con r quando viene ridotta modulo m è al massimo $((p-1)/m)/(p-1) = 1/m$. Di conseguenza, per qualsiasi coppia di valori distinti $k, l \in \mathbf{Z}_p$, si ha

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m$$

Quindi $\mathcal{H}_{p,m}$ è davvero universale. ■

Esercizi

11.3-1

Supponete di effettuare una ricerca in una lista concatenata di lunghezza n, dove ogni elemento contiene una chiave k e un valore hash $h(k)$. Ogni chiave è una lunga stringa di caratteri. Come potreste trarre vantaggio dai valori hash quando ricercate nella lista un elemento con una data chiave?

11.3-2

Supponete che una stringa di r caratteri venga associata da una funzione hash a m slot, trattandola come se fosse un numero con base 128 e applicando il metodo della divisione. Il numero m è facilmente rappresentato come una parola di 32 bit nel calcolatore, ma la stringa di r caratteri, trattata come un numero con base 128, richiede molte parole. Come può essere applicato il metodo della divisione per calcolare il valore hash della stringa di caratteri senza utilizzare più di un numero costante di parole nello spazio di memoria esterno alla stringa stessa?

11.3-3

Considerate una variante del metodo della divisione in cui $h(k) = k \bmod m$, dove $m = 2^p - 1$ e k è una stringa di caratteri interpretata come un numero con base 2^p. Dimostrate che, se la stringa x può essere derivata dalla stringa y permutando i suoi caratteri, allora le stringhe x e y sono associate allo stesso valore. Indicate un'applicazione nella cui funzione hash non sarebbe desiderabile questa proprietà.

11.3-4

Considerate una tabella hash di dimensione $m = 1000$ e una corrispondente funzione hash $h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor$ per $A = (\sqrt{5} - 1)/2$. Calcolate gli slot cui saranno associate le chiavi 61, 62, 63, 64 e 65.

11.3-5 ⋆

Una famiglia $\mathcal{H}$ di funzioni hash che associano un insieme finito U a un insieme finito B si definisce ***ϵ-universale*** se, per ogni coppia di elementi distinti k e l di U, si ha

$$\Pr\{h(k) = h(l)\} \leq \epsilon$$

Dove la probabilità è calcolata sulle estrazioni casuali della funzione hash h dalla famiglia $\mathcal{H}$. Dimostrate che una famiglia ϵ-universale di funzioni hash deve avere

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}$$

11.3-6 $\star$

Indichiamo con U l'insieme delle n-tuple (una n-tupla è una serie di n valori) estratte da $\mathbf{Z}_p$; inoltre, sia $B = \mathbf{Z}_p$, dove p è un numero primo. Definiamo la funzione hash $h_b : U \to B$ per $b \in \mathbf{Z}_p$ su una n-tupla di input $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ da U in questo modo:

$$h_b(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

Sia $\mathcal{H} = \{h_b : b \in \mathbf{Z}_p\}$. Dimostrate che $\mathcal{H}$ è $((n-1)/p)$-universale secondo la definizione di ϵ-universale data nell'Esercizio 11.3-5 (*suggerimento:* vedere l'Esercizio 31.4-4).

11.4 Indirizzamento aperto

Nell'***indirizzamento aperto***, tutti gli elementi sono memorizzati nella stessa tabella hash; ovvero ogni voce della tabella contiene un elemento dell'insieme dinamico o la costante NIL. Quando cerchiamo un elemento, esaminiamo sistematicamente gli slot della tabella finché non troviamo l'elemento desiderato o finché non ci accorgiamo che l'elemento non si trova nella tabella.

Diversamente dal concatenamento, non ci sono liste né elementi memorizzati all'esterno della tabella. Quindi, nell'indirizzamento aperto, la tabella hash può "riempirsi" al punto tale che non possono essere effettuati altri inserimenti; il fattore di carico α non supera mai 1. Ovviamente, potremmo memorizzare le liste per il concatenamento all'interno della tabella hash, negli slot altrimenti inutilizzati della tabella hash (vedere l'Esercizio 11.2-4), ma il vantaggio dell'indirizzamento aperto sta nel fatto che esclude completamente i puntatori. Anziché seguire i puntatori, *calcoliamo* la sequenza degli slot da esaminare. La memoria extra liberata per non avere memorizzato i puntatori offre alla tabella hash un gran numero di slot, a parità di memoria occupata, consentendo potenzialmente di ridurre il numero di collisioni e di accelerare le operazioni di ricerca.

Per effettuare un inserimento mediante l'indirizzamento aperto, esaminiamo in successione le posizioni della tabella hash (***scansione*** o ***sondaggio***), finché non troviamo uno slot vuoto in cui inserire la chiave. Anziché seguire sempre lo stesso ordine $0, 1, \ldots, m-1$ (che richiede un tempo di ricerca $\Theta(n)$), la sequenza delle posizioni esaminate durante una scansione *dipende dalla chiave da inserire*. Per determinare quali slot esaminare, estendiamo la funzione hash in modo da includere il numero di scansioni (a partire da 0) come secondo input. Quindi, la funzione hash diventa

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

Con l'indirizzamento aperto si richiede che, per ogni chiave k, la ***sequenza di scansione***

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

sia una permutazione di $\langle 0, 1, \ldots, m-1 \rangle$, in modo che ogni posizione della tabella hash venga considerata come slot per una nuova chiave mentre la tabella si riempie.

Nel seguente pseudocodice supponiamo che gli elementi della tabella hash T siano chiavi senza dati satelliti; la chiave k è identica all'elemento che contiene la chiave k. Ogni slot contiene una chiave o la costante NIL (se lo slot è vuoto).

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = NIL
4            then T[j] ← k
5                 return j
6            else  i ← i + 1
7    until i = m
8  error "overflow della tabella hash"
```

L'algoritmo che ricerca la chiave k esamina la stessa sequenza di slot che ha esaminato l'algoritmo di inserimento quando ha inserito la chiave k. Quindi, la ricerca può terminare (senza successo) quando trova uno slot vuoto, perché la chiave k sarebbe stata inserita lì e non dopo nella sua sequenza di scansione (questo ragionamento presuppone che le chiavi non vengano cancellate dalla tabella hash). La procedura HASH-SEARCH prende come input una tabella hash T e una chiave k; restituisce j se lo slot j contiene la chiave k oppure NIL se la chiave k non si trova nella tabella T.

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4            then return j
5          i ← i + 1
6    until T[j] = NIL o i = m
7  return NIL
```

La cancellazione da una tabella hash a indirizzamento aperto è un'operazione difficile. Quando cancelliamo una chiave dallo slot i, non possiamo semplicemente marcare questo slot come vuoto inserendovi la costante NIL. Così facendo, potrebbe essere impossibile ritrovare qualsiasi chiave k durante il cui inserimento abbiamo esaminato lo slot i e lo abbiamo trovato occupato. Una soluzione consiste nel marcare lo slot registrandovi il valore speciale DELETED, anziché NIL. Poi dovremo modificare la procedura HASH-INSERT per trattare tale slot come se fosse vuoto, in modo da potere inserire una nuova chiave. Nessuna modifica è richiesta per HASH-SEARCH, perché questa procedura ignora i valori DELETED durante la ricerca. Notate che, quando si usa il valore speciale DELETED, i tempi di ricerca non dipendono più dal fattore di carico α; per questo motivo, il concatenamento viene selezionato più frequentemente come tecnica di risoluzione delle collisioni quando le chiavi devono essere cancellate.

Nella nostra analisi facciamo l'ipotesi di ***hashing uniforme***: supponiamo che ogni chiave abbia la stessa probabilità di avere come sequenza di scansione una delle $m!$ permutazioni di $\langle 0, 1, \ldots, m-1 \rangle$. L'hashing uniforme estende il concetto di hashing uniforme semplice definito precedentemente al caso in cui la funzione

hash produce, non un singolo numero, ma un'intera sequenza di scansione. Poiché è difficile implementare il vero hashing uniforme, in pratica si usano delle approssimazioni accettabili (come il doppio hashing, definito più avanti).

Tre tecniche vengono comunemente utilizzate per calcolare le sequenze di scansione richieste dall'indirizzamento aperto: scansione lineare, scansione quadratica e doppio hashing. Tutte e tre le tecniche garantiscono che $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ sia una permutazione di $\langle 0,1,\ldots,m-1\rangle$ per ogni chiave k. Tuttavia, nessuna di queste tecniche soddisfa l'ipotesi di hashing uniforme, in quanto nessuna di esse è in grado di generare più di m^2 sequenze di scansione differenti (anziché $m!$, come richiede l'hashing uniforme). Il doppio hashing ha il maggior numero di sequenze di scansione e, come si può prevedere, sembra che dia i risultati migliori.

Scansione lineare

Data una funzione hash ordinaria $h' : U \to \{0,1,\ldots,m-1\}$, che chiameremo ***funzione hash ausiliaria***, il metodo della ***scansione lineare*** usa la funzione hash

$$h(k,i) = (h'(k) + i) \bmod m$$

per $i = 0,1,\ldots,m-1$. Data la chiave k, il primo slot esaminato è $T[h'(k)]$, che è lo slot dato dalla funzione hash ausiliaria; il secondo slot esaminato è $T[h'(k)+1]$ e, così via, fino allo slot $T[m-1]$. Poi, la scansione riprende dagli slot $T[0], T[1], \ldots$ fino a $T[h'(k)-1]$. Poiché la scansione iniziale determina l'intera sequenza delle scansioni, ci sono soltanto m sequenze di scansione distinte.

La scansione lineare è facile da implementare, ma presenta un problema noto come ***clustering primario***: si formano lunghe file di slot occupati, che aumentano il tempo medio di ricerca. Le file (cluster) si formano perché uno slot vuoto preceduto da i slot pieni ha la probabilità $(i+1)/m$ di essere il prossimo a essere occupato. Le lunghe file di slot occupati tendono a diventare più lunghe e il tempo di ricerca medio aumenta.

Scansione quadratica

La ***scansione quadratica*** usa una funzione hash della forma

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \tag{11.5}$$

dove h' è una funzione hash ausiliaria, c_1 e $c_2 \neq 0$ sono costanti ausiliarie e $i = 0,1,\ldots,m-1$. La posizione iniziale esaminata è $T[h'(k)]$; le posizioni successivamente esaminate sono distanziate da quantità che dipendono in modo quadratico dal numero di scansione i. Questa tecnica funziona molto meglio della scansione lineare, ma per fare pieno uso della tabella hash, i valori di c_1, c_2 e m sono vincolati. Il Problema 11-3 illustra un modo per selezionare questi parametri. Inoltre, se due chiavi hanno la stessa posizione iniziale di scansione, allora le loro sequenze di scansione sono identiche, perché $h(k_1,0) = h(k_2,0)$ implica $h(k_1,i) = h(k_2,i)$. Questa proprietà porta a una forma più lieve di clustering, che chiameremo ***clustering secondario***. Come nella scansione lineare, la scansione iniziale determina l'intera sequenza, quindi vengono utilizzate soltanto m sequenze di scansione distinte.

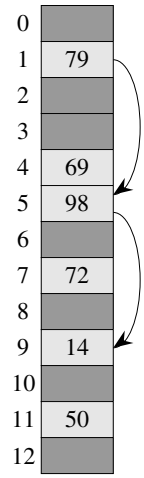


Figura 11.5
Inserimento con doppio hashing. La tabella hash ha dimensione 13 con $h_1(k) = k \bmod 13$ e $h_2(k) = 1 + (k \bmod 11)$. Poiché $14 \equiv 1 \pmod{13}$ e $14 \equiv 3 \pmod{11}$, la chiave 14 viene inserita nello slot vuoto 9, dopo che gli slot 1 e 5 sono stati esaminati e trovati occupati.

Doppio hashing

Il doppio hashing è uno dei metodi migliori disponibili per l'indirizzamento aperto, perché le permutazioni prodotte hanno molte delle caratteristiche delle permutazioni scelte a caso. Il ***doppio hashing*** usa una funzione hash della forma

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

dove h_1 e h_2 sono funzioni hash ausiliarie. La scansione inizia dalla posizione $T[h_1(k)]$; le successive posizioni di scansione sono distanziate dalle precedenti posizioni di una quantità $h_2(k)$, modulo m. Quindi, diversamente dal caso della scansione lineare o quadratica, la sequenza di scansione qui dipende in due modi dalla chiave k, perché potrebbero variare la posizione iniziale di scansione e/o la distanza fra due posizioni successive di scansione. La Figura 11.5 illustra un esempio di inserimento con doppio hashing.

Il valore $h_2(k)$ deve essere relativamente primo con la dimensione m della tabella hash per l'intera tabella hash in cui effettuare le ricerche (vedere l'Esercizio 11.4-3). Un modo pratico per garantire questa condizione è scegliere m come una potenza di 2 e definire h_2 in modo che produca sempre un numero dispari. Un altro modo è scegliere m come un numero primo e definire h_2 in modo che generi sempre un numero intero positivo minore di m. Per esempio, potremmo scegliere m come un numero primo e porre

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod m') \end{aligned}$$

dove m' deve essere scelto un po' più piccolo di m (come $m - 1$). Per esempio, se $k = 123456$, $m = 701$ e $m' = 700$, abbiamo $h_1(k) = 80$ e $h_2(k) = 257$, quindi la scansione inizia dalla posizione 80 e si ripete ogni 257 slot (modulo m) finché non sarà trovata la chiave o non saranno esaminati tutti gli slot.

Il doppio hashing è migliore delle scansioni lineari e quadratiche in quanto usa $\Theta(m^2)$ sequenze di scansione, anziché $\Theta(m)$, perché ogni possibile coppia $(h_1(k), h_2(k))$ produce una distinta sequenza di scansione. Di conseguenza, le prestazioni del doppio hashing sembrano molto prossime a quelle dello schema "ideale" dell'hashing uniforme.

Analisi dell'hashing a indirizzamento aperto

La nostra analisi dell'indirizzamento aperto, come quella del concatenamento, è espressa in termini del fattore di carico $\alpha = n/m$ della tabella hash, al tendere di n e m a infinito. Ovviamente, con l'indirizzamento aperto, abbiamo al massimo un elemento per slot, quindi $n \leq m$, e questo implica che $\alpha \leq 1$.

Supponiamo che venga applicato l'hashing uniforme. In questo schema teorico, la sequenza di scansione $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ utilizzata per inserire o ricercare una chiave k ha la stessa probabilità di essere una permutazione di $\langle 0, 1, \ldots, m-1\rangle$. Ovviamente, una data chiave è associata a un'unica sequenza di scansione costante; questo significa che, considerando la distribuzione delle probabilità nello spazio delle chiavi e l'operazione della funzione hash sulle chiavi, ogni possibile sequenza di scansione è equamente probabile.

Adesso analizziamo il numero atteso di scansioni dell'hashing con indirizzamento aperto nell'ipotesi di hashing uniforme, iniziando dall'analisi del numero di scansioni fatte in una ricerca senza successo.

Teorema 11.6
Nell'ipotesi di hashing uniforme, data una tabella hash a indirizzamento aperto con un fattore di carico $\alpha = n/m < 1$, il numero atteso di scansioni in una ricerca senza successo è al massimo $1/(1-\alpha)$.

Dimostrazione In una ricerca senza successo, ogni scansione, tranne l'ultima, accede a uno slot occupato che non contiene la chiave desiderata e l'ultimo slot esaminato è vuoto. Definiamo la variabile casuale X come il numero di scansioni fatte in una ricerca senza successo; definiamo inoltre A_i (con $i = 1, 2, \ldots$) come l'evento in cui l'i-esima scansione trova uno slot occupato. Allora l'evento $\{X \geq i\}$ è l'intersezione degli eventi $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. Limiteremo $\Pr\{X \geq i\}$ limitando $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. Per l'Esercizio C.2-6, si ha

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = {} & \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ & \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}\end{aligned}$$

Poiché ci sono n elementi e m slot, $\Pr\{A_1\} = n/m$. Per $j > 1$, la probabilità che la j-esima scansione trovi uno slot occupato, dato che le prime $j-1$ scansioni hanno trovato slot occupati, è $(n-j+1)/(m-j+1)$. Questa probabilità deriva dal fatto che dovremmo trovare uno dei restanti $(n-(j-1))$ elementi in uno degli $(m-(j-1))$ slot non ancora esaminati e, per l'ipotesi di hashing uniforme, la probabilità è il rapporto di queste quantità. Osservando che $n < m$ implica che $(n-j)/(m-j) \leq n/m$ per ogni j tale che $0 \leq j < m$, allora per ogni i tale che $1 \leq i \leq m$, si ha

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}\end{aligned}$$

Adesso utilizziamo l'equazione (C.24) per limitare il numero atteso di scansioni:

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

$$\begin{aligned} &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \end{aligned}$$

■

Il precedente limite $1 + \alpha + \alpha^2 + \alpha^3 + \cdots$ ha un'interpretazione intuitiva. Una scansione è sempre effettuata. Con una probabilità approssimativamente pari ad α, la prima scansione trova uno slot occupato, quindi occorre effettuare una seconda scansione. Con una probabilità approssimativamente pari ad α^2, i primi due slot sono occupati, quindi occorre effettuare una terza scansione e così via.

Se α è una costante, il Teorema 11.6 indica che una ricerca senza successo viene eseguita nel tempo $O(1)$. Per esempio, se la tabella hash è piena a metà, il numero medio di scansioni in una ricerca senza successo è al massimo $1/(1 - 0,5) = 2$. Se è piena al 90%, il numero medio di scansioni è al massimo $1/(1 - 0,9) = 10$.

Il Teorema 11.6 fornisce le prestazioni della procedura HASH-INSERT in maniera quasi immediata.

Corollario 11.7
L'inserimento di un elemento in una tabella hash a indirizzamento aperto con un fattore di carico α richiede in media non più di $1/(1 - \alpha)$ scansioni, nell'ipotesi di hashing uniforme.

Dimostrazione Un elemento viene inserito soltanto se c'è spazio nella tabella e, quindi, $\alpha < 1$. L'inserimento di una chiave richiede una ricerca senza successo seguita dalla sistemazione della chiave nel primo slot vuoto che viene trovato. Quindi, il numero atteso di scansioni è al massimo $1/(1 - \alpha)$. ■

Il calcolo del numero atteso di scansioni per una ricerca con successo richiede un po' più di lavoro.

Teorema 11.8
Data una tabella hash a indirizzamento aperto con un fattore di carico $\alpha < 1$, il numero atteso di scansioni in una ricerca con successo è al massimo

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

supponendo che l'hashing sia uniforme e che ogni chiave nella tabella abbia la stessa probabilità di essere cercata.

Dimostrazione La ricerca di una chiave k segue la stessa sequenza di scansione che è stata seguita quando è stato inserito l'elemento con chiave k. Per il Corollario 11.7, se k era la $(i+1)$-esima chiave inserita nella tabella hash, il numero atteso di scansioni fatte in una ricerca di k è al massimo $1/(1 - i/m) = m/(m - i)$. Calcolando la media su tutte le n chiavi della tabella hash, si ottiene il numero medio di scansioni durante una ricerca con successo:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} \quad = \quad \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \quad =$$

$$= \frac{1}{\alpha}(H_m - H_{m-n})$$

Il termine $H_i = \sum_{j=1}^{i} 1/j$ rappresenta l'i-esimo numero armonico (definito dall'equazione (A.7)). Applicando la tecnica per limitare una sommatoria con un integrale (descritta nel Paragrafo A.2), otteniamo la seguente espressione come limite del numero atteso di scansioni in una ricerca con successo:

$$\begin{aligned}
\frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} 1/k \\
&\leq \frac{1}{\alpha}\int_{m-n}^{m} (1/x)\,dx \quad \text{(per la disequazione (A.12))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}
\end{aligned}$$

■

Se la tabella hash è piena a metà, il numero atteso delle scansioni in una ricerca con successo è minore di $1,387$. Se la tabella hash è piena al 90%, il numero atteso di scansioni è minore di $2,559$.

Esercizi

11.4-1

Supponete di inserire le chiavi $10, 22, 31, 4, 15, 28, 17, 88, 59$ in una tabella hash di lunghezza $m = 11$ utilizzando l'indirizzamento aperto con la funzione hash ausiliaria $h'(k) = k \bmod m$. Illustrate il risultato dell'inserimento di queste chiavi utilizzando la scansione lineare, la scansione quadratica con $c_1 = 1$ e $c_2 = 3$ e il doppio hashing con $h_2(k) = 1 + (k \bmod (m-1))$.

11.4-2

Scrivete uno pseudocodice per HASH-DELETE come indicato nel testo e modificate HASH-INSERT per gestire il valore speciale DELETED.

11.4-3 ⋆

Supponete di applicare la tecnica del doppio hashing per risolvere le collisioni, ovvero utilizzate la funzione hash $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. Dimostrate che, se m e $h_2(k)$ hanno il massimo comune divisore $d \geq 1$ per qualche chiave k, allora una ricerca senza successo della chiave k esamina la quota $1/d$ della tabella hash, prima di restituire lo slot $h_1(k)$. Pertanto, se $d = 1$, e quindi m e $h_2(k)$ sono numeri relativamente primi, la ricerca può interessare l'intera tabella hash (*suggerimento:* consultare il Capitolo 31).

11.4-4

Considerate una tabella hash a indirizzamento aperto con hashing uniforme. Calcolate i limiti superiori sul numero atteso di scansioni in una ricerca senza successo e sul numero atteso di scansioni in una ricerca con successo quando il fattore di carico è $3/4$ e quando è $7/8$.

11.4-5 ⋆

Considerate una tabella hash a indirizzamento aperto con un fattore di carico α. Trovate il valore non nullo α per cui il numero atteso di scansioni in una ricerca

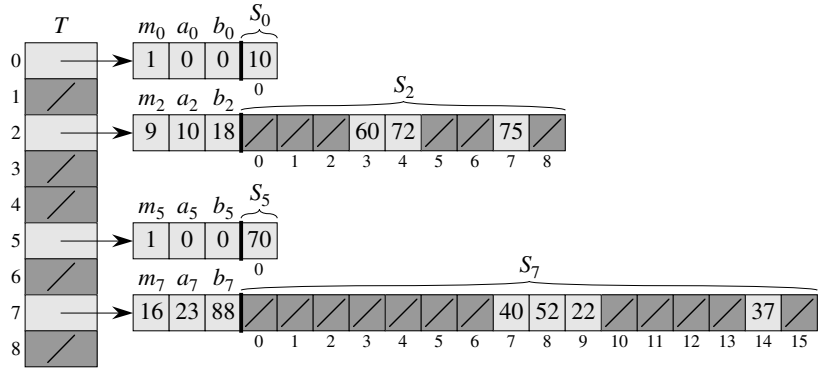


Figura 11.6
Applicazione dell'hashing perfetto per memorizzare l'insieme $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. La funzione hash esterna è $h(k) = ((ak + b) \bmod p) \bmod m$, dove $a = 3$, $b = 42$, $p = 101$ e $m = 9$. Per esempio, $h(75) = 2$, quindi la chiave 75 viene associata allo slot 2 della tabella T. Una tabella hash secondaria S_j memorizza tutte le chiavi che vengono associate allo slot j. La dimensione della tabella hash S_j è m_j e la funzione hash associata è $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Poiché $h_2(75) = 7$, la chiave 75 viene memorizzata nello slot 7 della tabella hash secondaria S_2. Non ci sono collisioni in nessuna delle tabelle hash secondarie, quindi la ricerca richiede un tempo costante nel caso peggiore.

senza successo è pari a due volte il numero atteso di scansioni in una ricerca con successo. Utilizzate i limiti superiori dati dai Teoremi 11.6 e 11.8 per questi numeri attesi di scansioni.

⋆ 11.5 Hashing perfetto

Sebbene sia utilizzato molto spesso per le sue ottime prestazioni attese, l'hashing può essere utilizzato per ottenere prestazioni eccellenti nel *caso peggiore* quando l'insieme delle chiavi è ***statico***: una volta che le chiavi sono memorizzate nella tabella, l'insieme delle chiavi non cambia più. Alcune applicazioni hanno insiemi di chiavi statici per natura: considerate l'insieme delle parole riservate di un linguaggio di programmazione oppure l'insieme dei nomi dei file registrati in un CD-ROM. Chiamiamo ***hashing perfetto*** una tecnica di hashing secondo la quale il numero di accessi in memoria richiesti per svolgere una ricerca è $O(1)$ nel caso peggiore.

Il concetto di base per creare uno schema di hashing perfetto è semplice. Utilizziamo uno schema di hashing a due livelli, con un hashing universale in ciascun livello. La Figura 11.6 illustra questo approccio.

Il primo livello è essenzialmente lo stesso dell'hashing con concatenamento: le n chiavi sono associate a m slot utilizzando una funzione hash h accuratamente selezionata da una famiglia di funzioni hash universali.

Tuttavia, anziché creare una lista di chiavi che si associate allo slot j, utilizziamo una piccola ***tabella hash secondaria*** S_j con una funzione hash h_j associata. Scegliendo le funzioni hash h_j accuratamente, possiamo garantire che non ci siano collisioni al livello secondario.

Per garantire che non ci siano collisioni al livello secondario, però, la dimensione m_j della tabella hash S_j dovrà avere il quadrato del numero n_j delle chiavi che si associano allo slot j. Sebbene possa sembrare probabile che tale dipendenza quadratica di m_j da n_j aumenti eccessivamente le esigenze complessive di spazio in memoria, tuttavia dimostreremo che, scegliendo opportunamente la funzione hash al primo livello, la quantità totale attesa di spazio utilizzato resterà $O(n)$.

Utilizziamo le funzioni hash scelte dalle classi universali delle funzioni hash descritte nel Paragrafo 11.3.3. La funzione hash del primo livello è scelta dalla classe $\mathcal{H}_{p,m}$ dove, come nel Paragrafo 11.3.3, p è un numero primo maggiore del valore di qualsiasi chiave. Quelle chiavi che sono associate allo slot j vengono

riassociate in una tabella hash secondaria S_j di dimensione m_j, utilizzando una funzione hash h_j scelta dalla classe $\mathcal{H}_{p,m_j}$.[1]

Procederemo in due passaggi. In primo luogo, determineremo come garantire che le tabelle secondarie non abbiano collisioni. In secondo luogo, dimostreremo che la quantità attesa di memoria complessivamente utilizzata – per la tabella hash primaria e per tutte le tabelle hash secondarie – è $O(n)$.

Teorema 11.9
Se memorizziamo n chiavi in una tabella hash di dimensione $m = n^2$ utilizzando una funzione hash h scelta a caso da una classe universale di funzioni hash, la probabilità che si verifichi una collisione è minore di $1/2$.

Dimostrazione Ci sono $\binom{n}{2}$ coppie di chiavi che possono collidere; ogni coppia collide con probabilità $1/m$, se h è scelta a caso da una famiglia universale $\mathcal{H}$ di funzioni hash. Sia X una variabile casuale che conta il numero di collisioni. Se $m = n^2$, il numero atteso di collisioni è

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2 \end{aligned}$$

Notate che questa analisi è simile a quella del paradosso del compleanno (Paragrafo 5.4.1). Applicando la disequazione di Markov (C.29), $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, con $t = 1$, si conclude la dimostrazione. ■

Dalla situazione descritta nel Teorema 11.9, con $m = n^2$, deriva che una funzione hash h scelta a caso da $\mathcal{H}$ è più probabile che non abbia *nessuna* collisione. Dato l'insieme K con n chiavi da associare alla tabella hash (ricordiamo che K è un insieme statico), è quindi facile trovare una funzione hash h esente da collisioni con pochi tentativi casuali.

Quando n è grande, però, una tabella hash di dimensione $m = n^2$ è eccessivamente grande. Pertanto, adottiamo un metodo di hashing a due livelli e utilizziamo l'approccio del Teorema 11.9 soltanto per associare le voci all'interno di ogni slot. Per associare le chiavi agli $m = n$ slot viene utilizzata una funzione hash h esterna (primo livello). Poi, se n_j chiavi vengono associate allo slot j, viene utilizzata una tabella hash secondaria S_j di dimensione $m_j = n_j^2$ per effettuare una ricerca senza collisioni in un tempo costante.

Adesso passiamo al problema di garantire che la memoria complessivamente utilizzata sia $O(n)$. Poiché la dimensione m_j della j-esima tabella hash secondaria cresce con il quadrato del numero n_j delle chiavi memorizzate, c'è il rischio che lo spazio totale occupato in memoria sia eccessivo.

Se la dimensione della tabella del primo livello è $m = n$, allora la quantità di memoria utilizzata è $O(n)$ per la tabella hash primaria, per lo spazio delle dimensioni m_j delle tabelle hash secondarie e per lo spazio dei parametri a_j e b_j che

[1] Quando $n_j = m_j = 1$, non occorre necessariamente una funzione hash per lo slot j; quando scegliamo una funzione hash $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ per tale slot, utilizziamo semplicemente $a = b = 0$.

definiscono le funzioni hash secondarie h_j estratte dalla classe $\mathcal{H}_{p,m_j}$ descritta nel Paragrafo 11.3.3 (tranne quando $n_j = 1$ e usiamo $a = b = 0$). Il seguente teorema e il successivo corollario forniscono un limite alla dimensione combinata attesa di tutte le tabelle hash secondarie. Un secondo corollario limita la probabilità che la dimensione combinata di tutte le tabelle hash secondarie sia superlineare.

Teorema 11.10
Se memorizziamo n chiavi in una tabella hash di dimensione $m = n$ utilizzando una funzione hash h scelta a caso da una classe universale di funzioni hash, si ha

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

dove n_j è il numero delle chiavi che sono associate allo slot j.

Dimostrazione Iniziamo con la seguente identità che vale per qualsiasi intero a non negativo:

$$a^2 = a + 2\binom{a}{2} \tag{11.6}$$

Abbiamo

$$\begin{aligned}
&\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \\
&\quad = \mathrm{E}\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(per l'equazione (11.6))} \\
&\quad = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(per la linearità del valore atteso)} \\
&\quad = \mathrm{E}[n] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(per l'equazione (11.1))} \\
&\quad = n + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(perché } n \text{ non è una variabile casuale)}
\end{aligned}$$

Per calcolare la sommatoria $\sum_{j=0}^{m-1} \binom{n_j}{2}$, osserviamo che essa è pari al numero totale di collisioni. Per la proprietà dell'hashing universale, il valore atteso di questa sommatoria è al massimo

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}$$

in quanto $m = n$. Quindi

$$\begin{aligned}
\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &\leq n + 2\,\frac{n-1}{2} \\
&= 2n - 1 \\
&< 2n
\end{aligned}$$

■

Corollario 11.11
Se memorizziamo n chiavi in una tabella hash di dimensione $m = n$ utilizzando una funzione hash h scelta a caso da una classe universale di funzioni hash e impostiamo la dimensione di ogni tabella hash secondaria a $m_j = n_j^2$ per $j = 0, 1, \ldots, m-1$, allora la quantità attesa di memoria richiesta per tutte le tabelle hash secondarie in uno schema di hashing perfetto è minore di $2n$.

Dimostrazione Poiché $m_j = n_j^2$ per $j = 0, 1, \ldots, m-1$, dal Teorema 11.10 otteniamo

$$\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n \tag{11.7}$$

che completa la dimostrazione. ■

Corollario 11.12
Se memorizziamo n chiavi in una tabella hash di dimensione $m = n$ utilizzando una funzione hash h scelta a caso da una classe di funzioni hash e impostiamo la dimensione di ogni tabella hash secondaria a $m_j = n_j^2$ per $j = 0, 1, \ldots, m-1$, la probabilità che la memoria totale utilizzata per le tabelle hash secondarie sia uguale o maggiore di $4n$ è minore di $1/2$.

Dimostrazione Applichiamo la disequazione di Markov (C.29), $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$, alla disequazione (11.7) con $X = \sum_{j=0}^{m-1} m_j$ e $t = 4n$:

$$\begin{aligned}
\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} &\leq \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\
&< \frac{2n}{4n} \\
&= 1/2
\end{aligned}$$

■

Dal Corollario 11.12 osserviamo che, provando poche funzioni hash scelte a caso da una famiglia universale, se ne ottiene rapidamente una che usa una quantità di memoria accettabile.

Esercizi

11.5-1 ⋆
Supponete di inserire n chiavi in una tabella hash di dimensione m utilizzando l'indirizzamento aperto e l'hashing uniforme. Indicate con $p(n, m)$ la probabilità che non si verifichino collisioni. Dimostrate che $p(n, m) \leq e^{-n(n-1)/2m}$ (*suggerimento:* esaminate l'equazione (3.11)). Dimostrate che, quando n supera $\sqrt{m}$, la probabilità di evitare le collisioni tende rapidamente a zero.

Problemi

11-1 Limite sul numero di scansioni nell'hashing
Una tabella hash di dimensione m è utilizzata per memorizzare n elementi, con $n \leq m/2$. Per risolvere le collisioni viene utilizzato l'indirizzamento aperto.

a. Nell'ipotesi di hashing uniforme, dimostrate che per $i = 1, 2, \ldots, n$, la probabilità che l'i-esimo inserimento richieda più di k scansioni è al massimo 2^{-k}.

b. Dimostrate che per $i = 1, 2, \ldots, n$, la probabilità che l'i-esimo inserimento richieda più di $2 \lg n$ scansioni è al massimo $1/n^2$.

Indicate con la variabile casuale X_i il numero di scansioni richieste dall'i-esimo inserimento. Avete dimostrato nel punto (b) che $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Indicate con la variabile casuale $X = \max_{1 \leq i \leq n} X_i$ il numero massimo di scansioni richieste da uno qualsiasi degli n inserimenti.

c. Dimostrate che $\Pr\{X > 2 \lg n\} \leq 1/n$.

d. Dimostrate che la lunghezza attesa $\mathrm{E}[X]$ della sequenza di scansione più lunga è $O(\lg n)$.

11-2 Limite sulla dimensione degli slot nel concatenamento

Supponete di avere una tabella hash con n slot e di risolvere le collisioni con il concatenamento; supponete inoltre che n chiavi siano inserite nella tabella. Ogni chiave ha la stessa probabilità di essere associata a ciascuno slot. Indicate con M il numero massimo di chiavi in qualsiasi slot dopo che tutte le chiavi sono state inserite. Il vostro compito è dimostrare un limite superiore $O(\lg n / \lg \lg n)$ su $\mathrm{E}[M]$, il valore atteso di M.

a. Dimostrate che la probabilità Q_k che k chiavi siano associate a un particolare slot è data da

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$$

b. Sia P_k la probabilità che $M = k$, cioè la probabilità che lo slot con la maggior parte delle chiavi contenga k chiavi. Dimostrate che $P_k \leq nQ_k$.

c. Applicate la formula di approssimazione di Stirling (equazione (3.17)) per dimostrare che $Q_k < e^k / k^k$.

d. Dimostrate che esiste una costante $c > 1$ tale che $Q_{k_0} < 1/n^3$ per $k_0 = c \lg n / \lg \lg n$. Concludete che $P_k < 1/n^2$ per $k \geq k_0 = c \lg n / \lg \lg n$.

e. Dimostrate che

$$\mathrm{E}[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}$$

Concludete che $\mathrm{E}[M] = O(\lg n / \lg \lg n)$.

11-3 Scansione quadratica

Supponete di avere una chiave k da cercare in una tabella hash con le posizioni $0, 1, \ldots, m-1$; supponete inoltre di avere una funzione hash h che associa lo spazio delle chiavi all'insieme $\{0, 1, \ldots, m-1\}$. Lo schema di ricerca è il seguente.

1. Calcolate il valore $i \leftarrow h(k)$ e impostate $j \leftarrow 0$.
2. Cercate la chiave desiderata k nella posizione i. Se la trovate o se questa posizione è vuota, terminate la ricerca.
3. Impostate $j \leftarrow (j+1) \bmod m$ e $i \leftarrow (i+j) \bmod m$; ritornate al punto 2.

Supponete che m sia una potenza di 2.

a. Dimostrate che questo schema è un'istanza dello schema generale della "scansione quadratica" mettendo in evidenza le costanti appropriate c_1 e c_2 per l'equazione (11.5).

b. Dimostrate che questo algoritmo esamina tutte le posizioni della tabella nel caso peggiore.

11-4 Hashing k-universale e autenticazione

Sia $\mathcal{H}$ una classe di funzioni hash in cui ogni funzione $h \in \mathcal{H}$ associa l'universo U delle chiavi a $\{0, 1, \ldots, m-1\}$. Diciamo che $\mathcal{H}$ è ***k-universale*** se, per ogni sequenza costante di k chiavi distinte $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)} \rangle$ e per qualsiasi h scelta a caso da $\mathcal{H}$, la sequenza $\langle h(x^{(1)}), h(x^{(2)}), \ldots, h(x^{(k)}) \rangle$ ha la stessa probabilità di essere una delle m^k sequenze di lunghezza k con elementi estratti da $\{0, 1, \ldots, m-1\}$.

a. Dimostrate che, se la famiglia $\mathcal{H}$ delle funzioni hash è 2-universale, allora è universale.

b. Supponete che l'universo U sia l'insieme di n-tuple di valori estratti da $\mathbf{Z}_p = \{0, 1, \ldots, p-1\}$, dove p è un numero primo. Considerate un elemento $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. Per qualsiasi n-tupla $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$, definite la funzione hash h_a in questo modo

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p$$

e sia $\mathcal{H} = \{h_a\}$. Dimostrate che $\mathcal{H}$ è universale, ma non 2-universale. (*Suggerimento:* trovate una chiave per la quale tutte le funzioni hash in $\mathcal{H}$ producono lo stesso valore.)

c. Modificate leggermente $\mathcal{H}$ rispetto al punto (b): per qualsiasi $a \in U$ e per qualsiasi $b \in \mathbf{Z}_p$, definite

$$h'_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

e $\mathcal{H}' = \{h'_{a,b}\}$. Dimostrate che $\mathcal{H}'$ è 2-universale. (*Suggerimento:* considerate costanti $x \in U$ e $y \in U$, con $x_i \neq y_i$ per qualche i. Che cosa accade a $h'_{a,b}(x)$ e $h'_{a,b}(y)$ quando a_i e b vanno oltre $\mathbf{Z}_p$?)

d. Alice e Bob hanno concordato in segreto di utilizzare una funzione hash h che appartiene a una famiglia $\mathcal{H}$ 2-universale di funzioni hash. Ogni $h \in \mathcal{H}$ associa un universo di chiavi U a $\mathbf{Z}_p$, dove p è un numero primo. Successivamente,

Alice invia un messaggio m a Bob tramite Internet, con $m \in U$. Alice autentica questo messaggio inviando anche un tag di autenticazione $t = h(m)$; Bob verifica che la coppia (m, t) che ha ricevuto soddisfa davvero $t = h(m)$. Supponete che un avversario intercetti (m, t) durante la trasmissione e tenti di imbrogliare Bob sostituendo la coppia (m, t) con una coppia differente (m', t'). Dimostrate che la probabilità che l'avversario riesca a ingannare Bob facendogli accettare la coppia (m', t') è al massimo $1/p$, indipendentemente da quanta potenza di calcolo abbia a disposizione l'avversario e indipendentemente dal fatto che l'avversario conosca la famiglia $\mathcal{H}$ delle funzioni hash utilizzate.

Note

Knuth [185] e Gonnet [126] sono eccellenti testi di riferimento per l'analisi degli algoritmi di hashing. Knuth attribuisce a H. P. Luhn (1953) l'invenzione delle tabelle hash e del metodo di concatenamento per risolvere le collisioni. All'incirca nello stesso periodo, G. M. Amdahl ideò l'indirizzamento aperto.

Nel 1979, Carter e Wegman svilupparono la nozione di classi universali delle funzioni hash [52]. Fredman, Komlós e Szemerédi [96] progettarono lo schema dell'hashing perfetto per gli insiemi statici presentati nel Paragrafo 11.5. Dietzfelbinger e altri autori [73] hanno esteso questo metodo agli insiemi dinamici, riuscendo a gestire inserimenti e cancellazioni nel tempo atteso ammortizzato pari a $O(1)$.

12 Alberi binari di ricerca

Gli alberi di ricerca sono strutture dati che supportano molte operazioni sugli insiemi dinamici, fra le quali SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE. Quindi, un albero di ricerca può essere utilizzato sia come dizionario sia come coda di priorità. Le operazioni di base su un albero binario di ricerca richiedono un tempo proporzionale all'altezza dell'albero. Per un albero binario completo con n nodi, tali operazioni sono eseguite nel tempo $\Theta(\lg n)$ nel caso peggiore. Se, invece, l'albero è una catena lineare di n nodi, le stesse operazioni richiedono un tempo $\Theta(n)$ nel caso peggiore. Come vedremo nel Paragrafo 12.4, l'altezza attesa di un albero binario di ricerca costruito in modo casuale è $O(\lg n)$, quindi le operazioni elementari degli insiemi dinamici svolte su questo tipo di albero richiedono in media il tempo $\Theta(\lg n)$. In pratica, non possiamo sempre garantire che gli alberi binari di ricerca possano essere costruiti in modo casuale, tuttavia esistono delle varianti di alberi binari di ricerca che assicurano buone prestazioni nel caso peggiore. Il Capitolo 13 presenta una di queste varianti: gli alberi RB (Red-Black), che hanno altezza $O(\lg n)$. Il Capitolo 18 introduce gli alberi B, che sono particolarmente adatti a mantenere i database nella memoria secondaria (disco) con accesso casuale.

Dopo avere presentato le proprietà fondamentali degli alberi binari di ricerca, i paragrafi successivi descriveranno come attraversare un albero binario di ricerca per visualizzarne ordinatamente i valori, come ricercare un valore in un albero binario di ricerca, come trovare l'elemento minimo o massimo, come trovare il predecessore o il successore di un elemento e come inserire o cancellare un elemento da un albero binario di ricerca. Le proprietà matematiche fondamentali degli alberi sono descritte nell'Appendice B.

12.1 Che cos'è un albero binario di ricerca?

Un albero binario di ricerca è organizzato, come suggerisce il nome, in un albero binario (Figura 12.1), che può essere rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto. Oltre a un campo chiave (key) e ai dati satelliti, ogni nodo dell'albero contiene i campi $left$, $right$ e p che puntano ai nodi che corrispondono, rispettivamente, al figlio sinistro, al figlio destro e al padre del nodo. Se manca un figlio o il padre, i corrispondenti campi contengono il valore NIL. Il nodo radice (*root*) è l'unico nodo nell'albero il cui campo padre (p) è NIL.

Le chiavi in un albero binario di ricerca sono sempre memorizzate in modo da soddisfare la ***proprietà degli alberi binari di ricerca***:

> Sia x un nodo in un albero binario di ricerca. Se y è un nodo nel sottoalbero sinistro di x, allora $key[y] \leq key[x]$. Se y è un nodo nel sottoalbero destro di x, allora $key[x] \leq key[y]$.

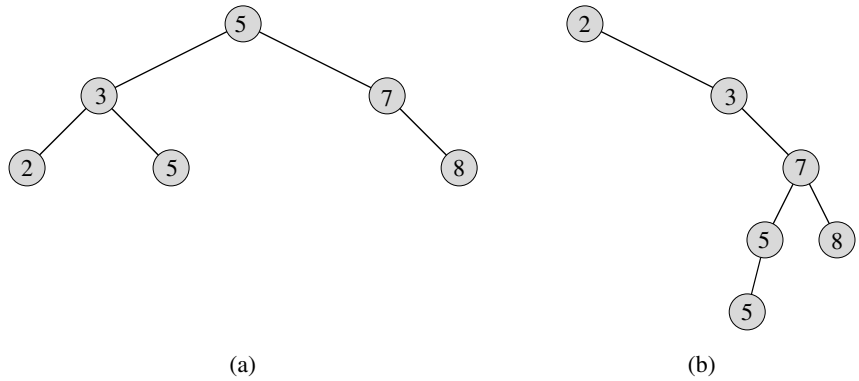


Figura 12.1 Alberi binari di ricerca. Per qualsiasi nodo x, le chiavi nel sottoalbero sinistro di x sono al massimo $key[x]$ e le chiavi nel sottoalbero destro di x sono almeno $key[x]$. Alberi binari di ricerca differenti possono rappresentare lo stesso insieme di valori. Il tempo di esecuzione nel caso peggiore per la maggior parte delle operazioni di ricerca in un albero è proporzionale all'altezza dell'albero. **(a)** Un albero binario di ricerca con 6 nodi e altezza 2. **(b)** Un albero binario di ricerca meno efficiente di altezza 4 che contiene le stesse chiavi.

Così, nella Figura 12.1(a), la chiave della radice è 5, le chiavi 2, 3 e 5 nel suo sottoalbero sinistro non sono maggiori 5, e le chiavi 7 e 8 nel suo sottoalbero destro non sono minori 5. La stessa proprietà vale per qualsiasi nodo dell'albero. Per esempio, la chiave 3 nella Figura 12.1(a) non è minore della chiave 2 nel suo sottoalbero sinistro e non è maggiore della chiave 5 nel suo sottoalbero destro.

La proprietà degli alberi binari di ricerca consente di visualizzare ordinatamente tutte le chiavi di un albero binario di ricerca con un semplice algoritmo ricorsivo di ***attraversamento simmetrico di un albero*** (inorder). Questo algoritmo è così chiamato perché la chiave della radice di un sottoalbero viene visualizzata fra i valori del suo sottoalbero sinistro e quelli del suo sottoalbero destro. Analogamente, un algoritmo di ***attraversamento anticipato di un albero*** (preorder) visualizza la radice prima dei valori dei suoi sottoalberi e un algoritmo di ***attraversamento posticipato di un albero*** (postorder) visualizza la radice dopo i valori dei suoi sottoalberi. Per utilizzare la seguente procedura per visualizzare tutti gli elementi di un albero binario di ricerca T, la chiamata da fare è INORDER-TREE-WALK$(root[T])$.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          visualizza key[x]
4          INORDER-TREE-WALK(right[x])
```

Per esempio, l'attraversamento simmetrico visualizza le chiavi in ciascuno dei due alberi binari di ricerca illustrati nella Figura 12.1 nel seguente ordine: $2, 3, 5, 5, 7, 8$. La correttezza dell'algoritmo si ricava direttamente per induzione dalla proprietà degli alberi binari di ricerca.

Occorre un tempo $\Theta(n)$ per attraversare un albero binario di ricerca di n nodi, perché, dopo la chiamata iniziale, la procedura viene chiamata ricorsivamente esattamente due volte per ogni nodo dell'albero – una volta per il figlio sinistro e una volta per il figlio destro. Il seguente teorema fornisce una prova più formale che occorre un tempo lineare per un attraversamento simmetrico di un albero.

Teorema 12.1

Se x è la radice di un sottoalbero di n nodi, la chiamata INORDER-TREE-WALK(x) richiede il tempo $\Theta(n)$.

Dimostrazione Sia $T(n)$ il tempo richiesto dalla procedura INORDER-TREE-WALK quando viene chiamata per la radice di un sottoalbero di n nodi. INORDER-TREE-WALK richiede una piccola quantità costante di tempo con un sottoalbero vuoto (per il test $x \neq$ NIL), quindi $T(0) = c$ per qualche costante positiva c.

Per $n > 0$, supponete che INORDER-TREE-WALK sia chiamata per un nodo x il cui sottoalbero sinistro ha k nodi e il cui sottoalbero destro ha $n - k - 1$ nodi. Il tempo per eseguire INORDER-TREE-WALK(x) è $T(n) = T(k) + T(n - k - 1) + d$ per qualche costante positiva d che tiene conto del tempo per eseguire INORDER-TREE-WALK(x), escludendo il tempo impiegato nelle chiamate ricorsive.

Applichiamo il metodo di sostituzione per provare che $T(n) = \Theta(n)$ dimostrando che $T(n) = (c+d)n+c$. Per $n = 0$, abbiamo $(c+d) \cdot 0+c = c = T(0)$. Per $n > 0$, abbiamo

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c \end{aligned}$$

che completa la dimostrazione. ■

Esercizi

12.1-1

Disegnate gli alberi binari di ricerca di altezza $2, 3, 4, 5$ e 6 per l'insieme delle chiavi $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1-2

Qual è la differenza fra la proprietà degli alberi binari di ricerca e la proprietà del min-heap (vedere pagina 107)? È possibile utilizzare la proprietà del min-heap per visualizzare ordinatamente le chiavi di un albero di n nodi nel tempo $O(n)$? Spiegate come o perché no.

12.1-3

Scrivete un algoritmo non ricorsivo che svolge un attraversamento simmetrico di un albero (*suggerimento:* c'è una soluzione semplice che usa uno stack come struttura dati ausiliaria e una soluzione elegante, ma più complicata, che non usa lo stack, ma suppone di verificare l'uguaglianza di due puntatori).

12.1-4

Scrivete gli algoritmi ricorsivi che svolgono gli attraversamenti anticipato e posticipato di un albero di n nodi nel tempo $\Theta(n)$.

12.1-5

Dimostrate che, poiché l'ordinamento per confronti di n elementi richiede il tempo $\Omega(n \lg n)$ nel caso peggiore, qualsiasi algoritmo basato sui confronti che viene utilizzato per costruire un albero binario di ricerca da una lista arbitraria di n elementi richiede il tempo $\Omega(n \lg n)$ nel caso peggiore.

12.2 Interrogazione di un albero binario di ricerca

Una tipica operazione svolta su albero binario di ricerca è quella di cercare una chiave memorizzata nell'albero. Oltre all'operazione SEARCH, gli alberi binari

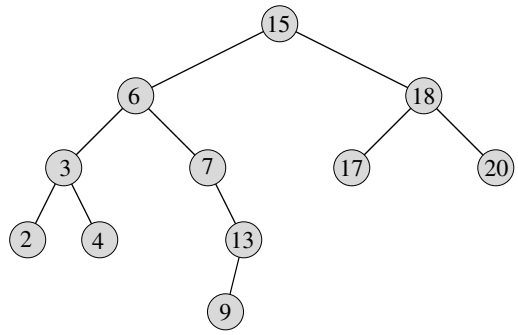


Figura 12.2
Interrogazioni in un albero binario di ricerca. Per cercare la chiave 13 nell'albero, seguiamo il percorso $15 \to 6 \to 7 \to 13$ dalla radice. La chiave minima è 2, che può essere trovata seguendo i puntatori *left* dalla radice. La chiave massima 20 si trova seguendo i puntatori *right* dalla radice. Il successore del nodo con la chiave 15 è il nodo con la chiave 17, perché è la chiave minima nel sottoalbero destro di 15. Il nodo con la chiave 13 non ha un sottoalbero destro, quindi il suo successore è il nonno più prossimo il cui figlio sinistro è anche nonno. In questo caso, il nodo con la chiave 15 è il suo successore.

di ricerca supportano interrogazioni (query) quali MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR. In questo paragrafo, esamineremo queste operazioni e dimostreremo che ciascuna di esse può essere eseguita nel tempo $O(h)$ in un albero binario di altezza h.

Ricerca

Utilizziamo la seguente procedura per cercare un nodo con una data chiave in un albero binario di ricerca. Dato un puntatore alla radice dell'albero e una chiave k, TREE-SEARCH restituisce un puntatore a un nodo con chiave k, se esiste, altrimenti restituisce il valore NIL.

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else  return TREE-SEARCH(right[x], k)
```

La procedura inizia la sua ricerca dalla radice e segue un percorso verso il basso lungo l'albero, come illustra la Figura 12.2. Per ogni nodo x che incontra, confronta la chiave k con $key[x]$. Se le due chiavi sono uguali, la ricerca termina. Se k è minore di $key[x]$, la ricerca continua nel sottoalbero sinistro di x, in quanto la proprietà degli alberi binari di ricerca implica che k non può essere memorizzata nel sottoalbero destro. Simmetricamente, se k è maggiore di $key[x]$, la ricerca continua nel sottoalbero destro. I nodi incontrati durante la ricorsione formano un percorso verso il basso dalla radice dell'albero, quindi il tempo di esecuzione di TREE-SEARCH è $O(h)$, dove h è l'altezza dell'albero.

La stessa procedura può essere scritta in modo iterativo "srotolando" la ricorsione in un ciclo **while**. Questa versione è più efficiente nella maggior parte dei calcolatori.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

Minimo e massimo

Un elemento di un albero binario di ricerca la cui chiave è un minimo può sempre essere trovato seguendo, a partire dalla radice, i puntatori $left$ dei figli a sinistra, fino a quando non viene incontrato un valore NIL, come illustra la Figura 12.2. La seguente procedura restituisce un puntatore all'elemento minimo nel sottoalbero con radice in un nodo x.

TREE-MINIMUM(x)
1 **while** $left[x] \neq$ NIL
2 **do** $x \leftarrow left[x]$
3 **return** x

La proprietà degli alberi binari di ricerca garantisce la correttezza della procedura TREE-MINIMUM. Se un nodo x non ha un sottoalbero sinistro, allora poiché ogni chiave nel sottoalbero destro di x è almeno grande quanto $key[x]$, la chiave minima nel sottoalbero con radice in x è $key[x]$. Se il nodo x ha un sottoalbero sinistro, allora poiché nessuna chiave nel sottoalbero destro è minore di $key[x]$ e ogni chiave nel sottoalbero sinistro non è maggiore di $key[x]$, la chiave minima nel sottoalbero con radice in x può essere trovata nel sottoalbero con radice in $left[x]$.

Lo pseudocodice per TREE-MAXIMUM è simmetrico.

TREE-MAXIMUM(x)
1 **while** $right[x] \neq$ NIL
2 **do** $x \leftarrow right[x]$
3 **return** x

Entrambe queste procedure vengono eseguite nel tempo $O(h)$ in un albero di altezza h perché, come in TREE-SEARCH, la sequenza dei nodi incontrati forma un percorso che scende dalla radice.

Successore e predecessore

Dato un nodo in un albero binario di ricerca, a volte è importante trovare il suo successore nell'ordine stabilito da un attraversamento simmetrico. Se tutte le chiavi sono distinte, il successore di un nodo x è il nodo con la più piccola chiave che è maggiore di $key[x]$. La struttura di un albero binario di ricerca ci consente di determinare il successore di un nodo senza mai confrontare le chiavi. La seguente procedura restituisce il successore di un nodo x in un albero binario di ricerca, se esiste, oppure NIL se x ha la chiave massima nell'albero.

TREE-SUCCESSOR(x)
1 **if** $right[x] \neq$ NIL
2 **then return** TREE-MINIMUM$(right[x])$
3 $y \leftarrow p[x]$
4 **while** $y \neq$ NIL and $x = right[y]$
5 **do** $x \leftarrow y$
6 $y \leftarrow p[y]$
7 **return** y

Il codice della procedura TREE-SUCCESSOR prevede due casi. Se il sottoalbero destro del nodo x non è vuoto, allora il successore di x è proprio il nodo

più a sinistra nel sottoalbero destro, che viene trovato nella riga 2 chiamando TREE-MINIMUM$(right[x])$. Per esempio, il successore del nodo con la chiave 15 nella Figura 12.2 è il nodo con la chiave 17.

D'altra parte, come chiede di dimostrare l'Esercizio 12.2-6, se il sottoalbero destro del nodo x è vuoto e x ha un successore y, allora y è il nonno più prossimo di x il cui figlio sinistro è anche nonno di x. Nella Figura 12.2, il successore del nodo con la chiave 13 è il nodo con la chiave 15. Per trovare y, semplicemente risaliamo l'albero partendo da x, finché incontriamo un nodo che è il figlio sinistro di suo padre; questa operazione è svolta dalle righe 3–7 di TREE-SUCCESSOR.

Il tempo di esecuzione di TREE-SUCCESSOR in un albero di altezza h è $O(h)$, perché seguiamo un percorso che sale o uno che scende. Anche la procedura TREE-PREDECESSOR, che è simmetrica a TREE-SUCCESSOR, viene eseguita nel tempo $O(h)$.

Anche se le chiavi non sono distinte, definiamo successore e predecessore di qualsiasi nodo x quel nodo che viene restituito, rispettivamente, dalle chiamate di TREE-SUCCESSOR(x) e TREE-PREDECESSOR(x).

In sintesi, abbiamo dimostrato il seguente teorema.

Teorema 12.2
Le operazioni sugli insiemi dinamici SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR possono essere svolte nel tempo $O(h)$ in un albero binario di ricerca di altezza h. ■

Esercizi

12.2-1
Supponete che un albero binario di ricerca abbia i numeri compresi fra 1 e 1000 e vogliate cercare il numero 363. Quale delle seguenti sequenze *non* può essere la sequenza dei nodi esaminata?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2
Scrivete le versioni ricorsive di TREE-MINIMUM e TREE-MAXIMUM.

12.2-3
Scrivete la procedura TREE-PREDECESSOR.

12.2-4
Il professor Bunyan crede di avere scoperto un'importante proprietà degli alberi binari di ricerca. Supponete che la ricerca della chiave k in un albero binario di ricerca termini in una foglia. Considerate tre insiemi: A, le chiavi a sinistra del percorso di ricerca; B, le chiavi lungo il percorso di ricerca; C, le chiavi a destra del percorso di ricerca. Il professor Bunyan sostiene che tre chiavi qualsiasi $a \in A$, $b \in B$ e $c \in C$ devono soddisfare la relazione $a \leq b \leq c$. Indicate un piccolo esempio contrario alla tesi del professore.

12.2-5
Dimostrate che se un nodo in un albero binario di ricerca ha due figli, allora il suo successore non ha un figlio sinistro e il suo predecessore non ha un figlio destro.

12.2-6
Considerate un albero binario di ricerca T le cui chiavi sono distinte. Dimostrate che se il sottoalbero destro di un nodo x in T è vuoto e x ha un successore y, allora y è il nonno più prossimo di x il cui figlio sinistro è anche un nonno di x (ricordiamo che ogni nodo è nonno di sé stesso).

12.2-7
L'attraversamento simmetrico di un albero binario di ricerca di n nodi può essere implementato trovando l'elemento minimo nell'albero con la procedura TREE-MINIMUM e, poi, effettuando $n - 1$ chiamate di TREE-SUCCESSOR. Dimostrate che questo algoritmo viene eseguito nel tempo $\Theta(n)$.

12.2-8
Dimostrate che, in un albero binario di ricerca di altezza h, k successive chiamate di TREE-SUCCESSOR richiedono il tempo $O(k+h)$, indipendentemente dal nodo di partenza.

12.2-9
Siano T un albero binario di ricerca con chiavi distinte, x un nodo foglia e y suo padre. Dimostrate che $key[y]$ è la più piccola chiave di T che è maggiore di $key[x]$ o la più grande chiave di T che è minore di $key[x]$.

12.3 Inserimento e cancellazione

Le operazioni di inserimento e cancellazione modificano l'insieme dinamico rappresentato da un albero binario di ricerca. La struttura dati deve essere modificata per riflettere questa modifica, ma in modo tale che la proprietà degli alberi binari di ricerca resti valida. Come vedremo, modificare l'albero per inserire un nuovo elemento è relativamente semplice, ma gestire le cancellazioni è molto più complicato.

Inserimento

Per inserire un nuovo valore v in un albero binario di ricerca T, utilizziamo TREE-INSERT. Questa procedura riceve un nodo z per il quale $key[z] = v$, $left[z] =$ NIL e $right[z] =$ NIL; modifica T e qualche campo di z in modo che z sia inserito in una posizione appropriata nell'albero.

La Figura 12.3 illustra il funzionamento di TREE-INSERT. Esattamente come le procedure TREE-SEARCH e ITERATIVE-TREE-SEARCH, TREE-INSERT inizia dalla radice dell'albero e segue un percorso verso il basso. Il puntatore x segue il percorso e il puntatore y è mantenuto come padre di x. Dopo l'inizializzazione, le righe 3–7 del ciclo **while** spostano questi due puntatori verso il basso, andando a sinistra o a destra a seconda dell'esito del confronto fra $key[z]$ e $key[x]$, finché x non viene impostato al valore NIL. Questo valore occupa la posizione dove vogliamo inserire l'elemento di input z. Le righe 8–13 impostano i puntatori che determinano l'inserimento di z.

Analogamente alle altre operazioni elementari con gli alberi di ricerca, la procedura TREE-INSERT viene eseguita nel tempo $O(h)$ in un albero di altezza h.

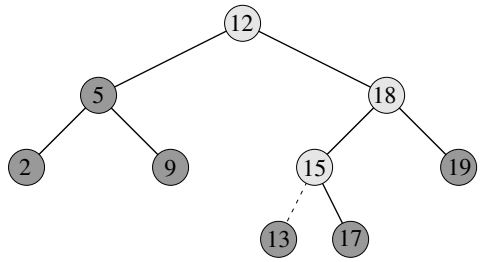


Figura 12.3 L'elemento con la chiave 13 viene inserito in un albero binario di ricerca. I nodi più chiari indicano il percorso dalla radice verso la posizione dove l'elemento viene inserito. La linea tratteggiata indica il collegamento che è stato aggiunto per inserire l'elemento.

TREE-INSERT(T, z)

```
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ L'albero T era vuoto
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

Cancellazione

La procedura per cancellare un nodo z da un albero binario di ricerca richiede come argomento un puntatore a z. La procedura considera i tre casi illustrati nella Figura 12.4. Se z non ha figli, modifichiamo suo padre $p[z]$ per sostituire z con NIL come suo figlio. Se il nodo ha un solo figlio, rimuoviamo z creando un nuovo collegamento tra suo figlio e suo padre. Infine, se il nodo z ha due figli, creiamo un nuovo collegamento per rimuovere il suo successore y che non ha un figlio sinistro (vedere l'Esercizio 12.2-5), poi sostituiamo la chiave e i dati satelliti di z con la chiave e i dati satelliti di y.

Il codice di TREE-DELETE implementa questi tre casi in modo leggermente diverso.

Nelle righe 1–3, l'algoritmo determina un nodo y da rimuovere con un collegamento. Il nodo y è il nodo di input z (se z ha al massimo un figlio) o il successore di z (se z ha due figli). Poi, le righe 4–6 impostano x al figlio non-NIL di y oppure a NIL se y non ha figli. Il nodo y è rimosso con un collegamento nelle righe 7–13 modificando i puntatori in $p[y]$ e x. L'eliminazione di y con un collegamento è complicata dal fatto di dovere gestire opportunamente le condizioni al contorno, che si verificano quando $x =$ NIL o quando y è la radice. Infine, nelle righe 14–16, se il successore di z era il nodo rimosso con un collegamento, la chiave e i dati satelliti di y vengono spostati in z, sovrapponendosi alla chiave e ai dati satelliti preesistenti. Il nodo y viene restituito nella riga 17, quindi la procedura chiamante può riciclarlo tramite la free list. La procedura viene eseguita nel tempo $O(h)$ in un albero di altezza h.

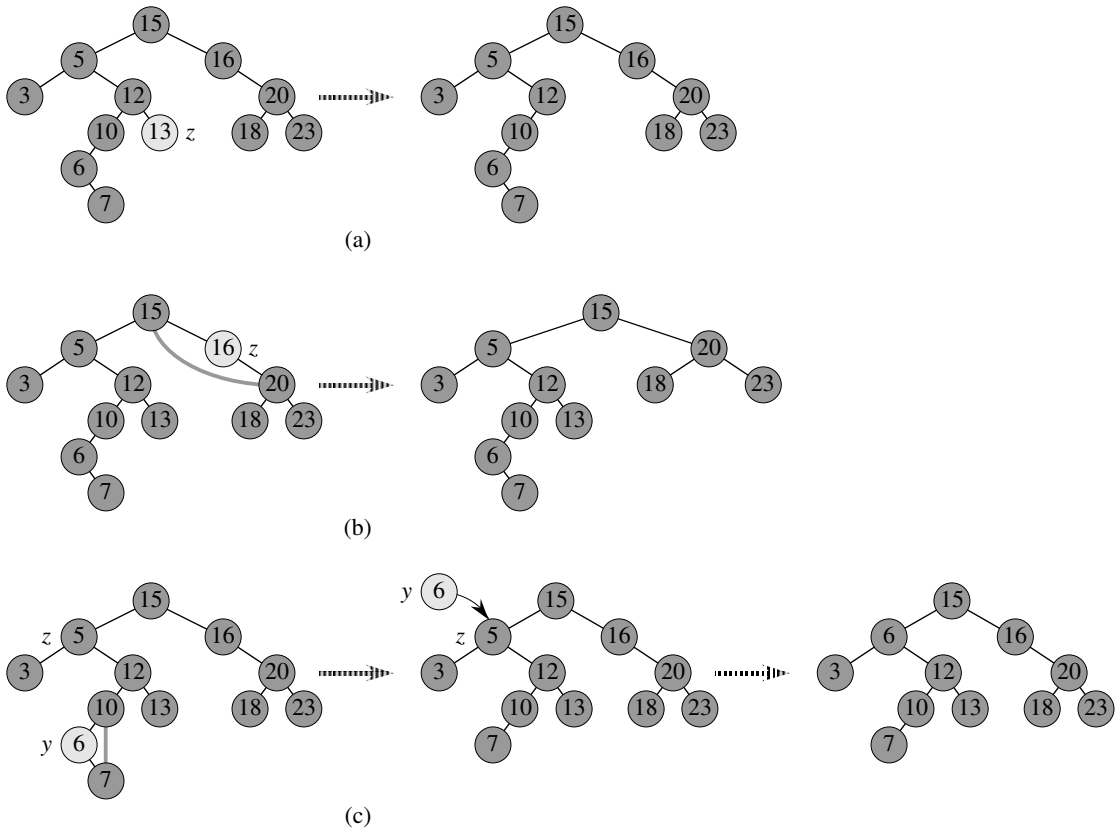


Figura 12.4 Cancellare un nodo z da un albero binario di ricerca. Il nodo che viene effettivamente rimosso dipende da quanti figli ha z; questo nodo è illustrato con uno sfondo più chiaro. **(a)** Se z non ha figli, viene semplicemente rimosso. **(b)** Se z ha un solo figlio, rimuoviamo z con un collegamento fra suo figlio e suo padre. **(c)** Se z ha due figli, rimuoviamo con un collegamento il suo successore y, che ha al massimo un figlio e, poi, sostituiamo la chiave e i dati satelliti di z con la chiave e i dati satelliti di y.

TREE-DELETE(T, z)

```
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copia i dati satelliti di y in z
17  return y
```

In sintesi, abbiamo dimostrato il seguente teorema.

Teorema 12.3
Le operazioni sugli insiemi dinamici INSERT e DELETE possono essere svolte nel tempo $O(h)$ in un albero binario di ricerca di altezza h. ■

Esercizi

12.3-1
Scrivete una versione ricorsiva della procedura TREE-INSERT.

12.3-2
Supponete che un albero binario di ricerca sia costruito inserendo ripetutamente alcuni valori distinti nell'albero. Dimostrate che il numero di nodi esaminati durante la ricerca di un valore è pari a uno più il numero di nodi esaminati quando il valore fu inserito per la prima volta nell'albero.

12.3-3
Possiamo ordinare un dato insieme di n numeri costruendo prima un albero binario di ricerca che contiene questi numeri (usando ripetutamente TREE-INSERT per inserire i numeri uno alla volta) e poi visualizzando i numeri mediante un attraversamento simmetrico dell'albero. Quali sono i tempi di esecuzione nel caso peggiore e nel caso migliore per questo algoritmo di ordinamento?

12.3-4
Supponete che un'altra struttura dati contenga un puntatore a un nodo y di un albero binario di ricerca. Supponete inoltre che il predecessore z di y sia cancellato dall'albero dalla procedura TREE-DELETE. Quale problema potrebbe nascere? Come dovrebbe essere modificata la procedura TREE-DELETE per risolvere questo problema?

12.3-5
L'operazione di cancellazione è "commutativa" nel senso che cancellare prima x e poi y da un albero binario di ricerca equivale a cancellare prima y e poi x? Spiegate perché sì oppure indicare un esempio contrario.

12.3-6
Utilizzando la procedura TREE-DELETE, se il nodo z ha due figli, è possibile rimuovere con un collegamento il suo predecessore, anziché il suo successore. Alcuni ritengono che, con una buona strategia che offre la stessa priorità al predecessore e al successore, è possibile ottenere prestazioni empiriche migliori. Come dovrebbe essere modificata la procedura TREE-DELETE per implementare tale strategia?

⋆ 12.4 Alberi binari di ricerca costruiti in modo casuale

Abbiamo visto che tutte le operazioni elementari su un albero binario di ricerca vengono eseguite nel tempo $O(h)$, dove h è l'altezza dell'albero. Tuttavia, questa altezza varia mentre gli elementi vengono inseriti o cancellati. Per esempio, se gli elementi vengono inseriti in ordine strettamente crescente, l'albero sarà una catena di altezza $n - 1$. D'altra parte, l'Esercizio B.5-4 dimostra che $h \geq \lfloor \lg n \rfloor$.

Analogamente a quicksort, possiamo dimostrare che il comportamento nel caso medio è molto più vicino al caso migliore del caso peggiore.

Purtroppo, sappiamo poco sull'altezza media di un albero binario di ricerca quando utilizziamo sia l'inserimento sia la cancellazione per crearlo. Se l'albero è creato soltanto con gli inserimenti, l'analisi diventa più trattabile. Definiamo quindi un ***albero binario di ricerca costruito in modo casuale*** con n chiavi come quello che si ottiene inserendo in ordine casuale le chiavi in un albero inizialmente vuoto, dove ciascuna delle $n!$ permutazioni delle chiavi di input ha la stessa probabilità (l'Esercizio 12.4-3 chiede di dimostrare che questo concetto è diverso dal supporre che ogni albero binario di ricerca con n chiavi abbia la stessa probabilità). In questo paragrafo, dimostreremo che l'altezza attesa di un albero binario di ricerca costruito in modo casuale con n chiavi è $O(\lg n)$. Si suppone che tutte le chiavi siano distinte.

Iniziamo definendo tre variabili casuali che aiutano a misurare l'altezza di un albero binario di ricerca costruito in modo casuale. Indichiamo con X_n l'altezza di un albero binario di ricerca costruito in modo casuale con n chiavi e definiamo l'***altezza esponenziale*** $Y_n = 2^{X_n}$. Quando costruiamo un albero binario di ricerca con n chiavi, scegliamo una chiave per la radice e indichiamo con R_n la variabile casuale che contiene il rango di questa chiave all'interno dell'insieme di n chiavi. Il valore di R_n ha la stessa probabilità di essere un elemento qualsiasi dell'insieme $\{1, 2, \ldots, n\}$. Se $R_n = i$, allora il sottoalbero sinistro della radice è un albero binario di ricerca costruito in modo casuale con $i - 1$ chiavi e il sottoalbero destro è un albero binario di ricerca costruito in modo casuale con $n - i$ chiavi. Poiché l'altezza di un albero binario è uno più la maggiore delle altezze dei due sottoalberi della radice, l'altezza esponenziale di un albero binario è due volte la più grande delle altezze esponenziali dei due sottoalberi della radice. Sapendo che $R_n = i$, abbiamo

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$$

Come casi base, abbiamo $Y_1 = 1$, perché l'altezza esponenziale di un albero con un nodo è $2^0 = 1$ e, per comodità, definiamo $Y_0 = 0$.

Poi definiamo le variabili casuali indicatrici $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, dove

$$Z_{n,i} = \mathrm{I}\{R_n = i\}$$

Poiché R_n ha la stessa probabilità di essere un elemento qualsiasi di $\{1, 2, \ldots, n\}$, abbiamo che $\Pr\{R_n = i\} = 1/n$ per $i = 1, 2, \ldots, n$, e quindi, per il Lemma 5.1,

$$\mathrm{E}[Z_{n,i}] = 1/n \tag{12.1}$$

per $i = 1, 2, \ldots, n$. Poiché un solo valore di $Z_{n,i}$ è 1 e tutti gli altri sono 0, abbiamo anche

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)$$

Dimostreremo che $\mathrm{E}[Y_n]$ è polinomiale in n, e questo in definitiva implica che $\mathrm{E}[X_n] = O(\lg n)$.

La variabile casuale indicatrice $Z_{n,i} = \mathrm{I}\{R_n = i\}$ è indipendente dai valori di Y_{i-1} e Y_{n-i}. Avendo scelto $R_n = i$, il sottoalbero sinistro, la cui altezza esponenziale è Y_{i-1}, è costruito in modo casuale con le $i - 1$ chiavi i cui ranghi sono minori di i. Questo sottoalbero è proprio come qualsiasi albero binario di ricerca

costruito in modo casuale con $i-1$ chiavi. Diversamente dal numero di chiavi che contiene, la struttura di questo sottoalbero non è affatto influenzata dalla scelta di $R_n = i$; quindi le variabili casuali Y_{i-1} e $Z_{n,i}$ sono indipendenti. Analogamente, il sottoalbero destro, la cui altezza esponenziale è Y_{n-i}, è costruito in modo casuale con le $n-i$ chiavi i cui ranghi sono maggiori di i. La sua struttura è indipendente dal valore di R_n, e quindi le variabili casuali Y_{n-i} e $Z_{n,i}$ sono indipendenti. Quindi

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[Z_{n,i}\left(2 \cdot \max(Y_{i-1}, Y_{n-i})\right)\right] && \text{(per la linearità del valore atteso)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && \text{(per l'indipendenza)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}\left[2 \cdot \max(Y_{i-1}, Y_{n-i})\right] && \text{(per l'equazione (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}\left[\max(Y_{i-1}, Y_{n-i})\right] && \text{(per l'equazione (C.21))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(per l'Esercizio C.3-4)}
\end{aligned}$$

Ogni termine $\mathrm{E}[Y_0], \mathrm{E}[Y_1], \ldots, \mathrm{E}[Y_{n-1}]$ appare due volte nell'ultima sommatoria, una volta come $\mathrm{E}[Y_{i-1}]$ e una volta come $\mathrm{E}[Y_{n-i}]$; quindi abbiamo la ricorrenza

$$\mathrm{E}[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i] \tag{12.2}$$

Applicando il metodo di sostituzione, dimostreremo che per qualsiasi intero positivo n, la ricorrenza (12.2) ha la soluzione

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}$$

Per farlo, utilizzeremo l'identità

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4} \tag{12.3}$$

L'Esercizio 12.4-1 chiede di dimostrare questa identità. Per i casi base, verifichiamo che i limiti

$$0 = Y_0 = \mathrm{E}[Y_0] \le \frac{1}{4}\binom{3}{3} = \frac{1}{4} \quad \text{e} \quad 1 = Y_1 = \mathrm{E}[Y_1] \le \frac{1}{4}\binom{1+3}{3} = 1$$

sono validi. Per la sostituzione, si ha

$$
\begin{aligned}
\mathrm{E}[Y_n] &\leq \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\
&\leq \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(per l'ipotesi induttiva)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(per l'equazione (12.3))} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3}
\end{aligned}
$$

Abbiamo limitato $\mathrm{E}[Y_n]$, ma il nostro obiettivo finale è limitare $\mathrm{E}[X_n]$. Come chiede di dimostrare l'Esercizio 12.4-4, la funzione $f(x) = 2^x$ è convessa (vedere pagina 939). Di conseguenza, possiamo applicare la disequazione di Jensen (C.25)

$$2^{\mathrm{E}[X_n]} \leq \mathrm{E}[2^{X_n}] = \mathrm{E}[Y_n]$$

per ottenere che

$$
\begin{aligned}
2^{\mathrm{E}[X_n]} &\leq \frac{1}{4}\binom{n+3}{3} \\
&= \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3+6n^2+11n+6}{24}
\end{aligned}
$$

Prendendo i logaritmi da entrambi i lati, si ha $\mathrm{E}[X_n] = O(\lg n)$. Quindi, abbiamo dimostrato il seguente teorema.

Teorema 12.4
L'altezza attesa di un albero binario di ricerca costruito in modo casuale con n chiavi è $O(\lg n)$. ■

Esercizi

12.4-1
Dimostrate l'equazione (12.3).

12.4-2
Descrivete un albero binario di ricerca di n nodi, dove la profondità media di un nodo nell'albero è $\Theta(\lg n)$, mentre l'altezza dell'albero è $\omega(\lg n)$. Calcolate un limite superiore asintotico sull'altezza di un albero binario di ricerca di n nodi in cui la profondità media di un nodo è $\Theta(\lg n)$.

12.4-3
Dimostrate che il concetto di albero binario di ricerca scelto in modo casuale con n chiavi, dove ogni albero binario di ricerca con n chiavi ha la stessa probabilità di essere scelto, è diverso dal concetto di albero binario di ricerca costruito in modo casuale che abbiamo descritto in questo paragrafo (*suggerimento:* elencate le possibilità quando $n = 3$).

12.4-4
Dimostrate che la funzione $f(x) = 2^x$ è convessa.

12.4-5 $\star$
Considerate la procedura RANDOMIZED-QUICKSORT che opera su una sequenza di n numeri distinti di input. Dimostrate che per qualsiasi costante $k > 0$, per tutte le $n!$ permutazioni dell'input, tranne $O(1/n^k)$ permutazioni, il tempo di esecuzione è $O(n \lg n)$.

Problemi

12-1 Alberi binari di ricerca con chiavi uguali

Le chiavi uguali sono un problema per l'implementazione degli alberi binari di ricerca.

a. Quali sono le prestazioni asintotiche della procedura TREE-INSERT quando viene utilizzata per inserire n elementi con chiavi identiche in un albero binario di ricerca inizialmente vuoto?

Vi proponiamo di migliorare TREE-INSERT verificando prima della riga 5 se è valida la relazione $key[z] = key[x]$ e verificando prima della riga 11 se è valida la relazione $key[z] = key[y]$. Se le relazioni sono valide, implementate una delle seguenti strategie. Per ogni strategia, trovate le prestazioni asintotiche dell'inserimento di n elementi con chiavi identiche in un albero binario di ricerca inizialmente vuoto (le strategie sono descritte per la riga 5 che confronta le chiavi di z e x; sostituite x con y per le strategie relative alla riga 11).

b. Mantenete un flag booleano $b[x]$ nel nodo x e impostate x a $left[x]$ o $right[x]$ a seconda del valore di $b[x]$, che si alterna fra FALSE e TRUE ogni volta che il nodo x viene visitato durante l'inserimento di un nodo con la stessa chiave di x.

c. Mantenete una lista di nodi con chiavi uguali nel nodo x e inserite z nella lista.

d. Impostate in modo casuale x a $left[x]$ o $right[x]$ (indicate le prestazioni nel caso peggiore e ricavate in modo informale le prestazioni nel caso medio).

12-2 Radix tree

Date due stringhe $a = a_0a_1 \ldots a_p$ e $b = b_0b_1 \ldots b_q$, dove a_i e b_j appartengono a un set ordinato di caratteri, diciamo che la stringa a è ***lessicograficamente minore*** della stringa b se è soddisfatta una delle seguenti condizioni:

1. Esiste un numero intero j, con $0 \leq j \leq \min(p, q)$, tale che $a_i = b_i$ per ogni $i = 0, 1, \ldots, j - 1$ e $a_j < b_j$.
2. $p < q$ e $a_i = b_i$ per ogni $i = 0, 1, \ldots, p$.

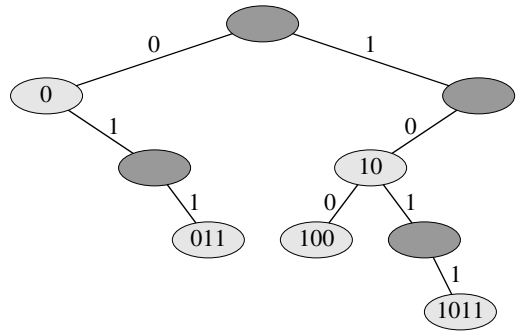


Figura 12.5 Un radix tree con le stringhe di bit 1011, 10, 011, 100 e 0. La chiave di un nodo qualsiasi può essere determinata seguendo un percorso che va dalla radice al nodo. Non c'è bisogno, quindi, di memorizzare le chiavi nei nodi; le chiavi sono indicate soltanto per scopi illustrativi. I nodi hanno uno sfondo più scuro se le corrispondenti chiavi non si trovano nell'albero; tali nodi sono presenti soltanto per stabilire un percorso con gli altri nodi.

Per esempio, se a e b sono stringhe di bit, allora $10100 < 10110$ per la regola 1 (ponendo $j = 3$) e $10100 < 101000$ per la regola 2. Questo è simile all'ordinamento utilizzato nei dizionari della lingua inglese.

La struttura dati ***radix tree*** illustrata nella Figura 12.5 contiene le stringhe di bit 1011, 10, 011, 100 e 0. Durante la ricerca di una chiave $a = a_0 a_1 \ldots a_p$, si va a sinistra in un nodo di profondità i se $a_i = 0$, a destra se $a_i = 1$. Sia S un insieme di stringhe binarie distinte, la cui somma delle lunghezze è n. Dimostrate come utilizzare un radix tree per ordinare lessicograficamente le stringhe di S nel tempo $\Theta(n)$. Per l'esempio illustrato nella Figura 12.5, l'output dell'ordinamento dovrebbe essere la sequenza 0, 011, 10, 100, 1011.

12-3 Profondità media di un nodo in un albero binario di ricerca costruito in modo casuale

In questo problema dimostreremo che la profondità media di un nodo in un albero binario di ricerca costruito in modo casuale con n nodi è $O(\lg n)$. Sebbene questo risultato sia meno stretto di quello del Teorema 12.4, la tecnica adottata rivela una sorprendente somiglianza fra la costruzione di un albero binario di ricerca e l'esecuzione della procedura RANDOMIZED-QUICKSORT, descritta nel Paragrafo 7.3. Definiamo ***lunghezza totale del percorso*** $P(T)$ di un albero binario T la somma, per tutti i nodi x di T, della profondità del nodo x, che indichiamo con $d(x, T)$.

a. Dimostrate che la profondità media di un nodo in T è

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T)$$

Quindi, si vuole dimostrare che il valore atteso di $P(T)$ è $O(n \lg n)$.

b. Indicate con T_L e T_R, rispettivamente, i sottoalberi sinistro e destro dell'albero T. Dimostrate che, se T ha n nodi, allora

$$P(T) = P(T_L) + P(T_R) + n - 1$$

c. Indicate con $P(n)$ la lunghezza totale media del percorso di un albero binario di ricerca costruito in modo casuale con n nodi. Dimostrate che

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

d. Dimostrate che è possibile riscrivere $P(n)$ in questo modo

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

e. Ricordando l'analisi alternativa della versione randomizzata di quicksort data nel Problema 7-2, concludete che $P(n) = O(n \lg n)$.

A ogni chiamata ricorsiva di quicksort, scegliete un elemento pivot casuale per partizionare l'insieme degli elementi da ordinare. Ogni nodo di un albero binario di ricerca partiziona l'insieme degli elementi che ricadono nel sottoalbero con radice in quel nodo.

f. Descrivete un'implementazione di quicksort dove i confronti per ordinare un insieme di elementi sono esattamente gli stessi confronti per inserire gli elementi in un albero binario di ricerca (l'ordine in cui avvengono i confronti può variare, ma devono essere fatti gli stessi confronti).

12-4 Numero di alberi binari differenti

Indichiamo con b_n il numero di alberi binari differenti con n nodi. In questo problema, troverete una formula per b_n e anche una stima asintotica.

a. Dimostrate che $b_0 = 1$ e che, per $n \geq 1$, si ha

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$

b. Facendo riferimento al Problema 4-5 per la definizione di una funzione generatrice, sia $B(x)$ la funzione generatrice

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

Dimostrate che $B(x) = xB(x)^2 + 1$, e quindi un modo per esprimere $B(x)$ in forma chiusa è

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1 - 4x}\right)$$

Lo sviluppo in ***serie di Taylor*** della funzione $f(x)$ intorno al punto $x = a$ è dato da

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x - a)^k$$

dove $f^{(k)}(x)$ è la derivata di ordine k di f nel punto x.

c. Dimostrate che

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(l'n-esimo ***numero catalano***) utilizzando lo sviluppo in serie di Taylor di $\sqrt{1 - 4x}$ intorno al punto $x = 0$. (Se preferite, anziché utilizzare lo sviluppo in serie di Taylor, potete utilizzare la generalizzazione dello sviluppo in serie binomiale (C.4) a esponenti non interi n, dove per qualsiasi numero reale n e qualsiasi intero k, il coefficiente binomiale $\binom{n}{k}$ è interpretato come $n(n-1)\cdots(n-k+1)/k!$ se $k \geq 0$, 0 negli altri casi.)

d. Dimostrate che

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n))$$

Note

Knuth [185] descrive le forme elementari dell'albero binario di ricerca e numerose varianti. Pare che gli alberi binari di ricerca siano stati scoperti in maniera indipendente da un certo numero di persone alla fine degli anni cinquanta. I *radix tree* sono spesso chiamati *trie* (questo termine deriva dalle lettere centrali della parola "retrieval"). Anche questi alberi sono trattati da Knuth [185].

Il Paragrafo 15.5 dimostra come costruire un albero binario di ricerca ottimale quando le frequenze di ricerca sono note prima della costruzione dell'albero. Ovvero, date le frequenze di ricerca per ogni chiave e le frequenze di ricerca per i valori che ricadono fra le chiavi nell'albero, costruiamo un albero binario di ricerca per il quale un insieme di ricerche che segue queste frequenze esamina il numero minimo di nodi.

Aslam [23] è l'autore della dimostrazione (presentata nel Paragrafo 12.4) che limita l'altezza attesa di un albero binario di ricerca costruito in modo casuale. Martínez e Roura [211] descrivono gli algoritmi randomizzati per inserire e cancellare gli elementi da un albero binario di ricerca; il risultato di entrambe le operazioni è un albero binario di ricerca casuale. Tuttavia, la loro definizione di albero binario di ricerca casuale differisce leggermente da quella di albero binario di ricerca costruito in modo casuale che abbiamo presentato in questo capitolo.

Alberi red-black 13

Nel Capitolo 12 si è visto che un albero binario di ricerca di altezza h può implementare qualsiasi operazione elementare sugli insiemi dinamici – come SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT e DELETE – nel tempo $O(h)$. Queste operazioni sono, quindi, veloci se l'altezza dell'albero di ricerca è piccola; ma se l'altezza è grande, le loro prestazioni potrebbero non essere migliori di quelle di una lista concatenata. Gli alberi red-black sono uno dei vari schemi di alberi di ricerca che vengono "bilanciati" per garantire che le operazioni elementari sugli insiemi dinamici richiedano un tempo $O(\lg n)$ nel caso peggiore.

13.1 Proprietà degli alberi red-black

Un ***albero red-black*** è un albero binario di ricerca con un bit aggiuntivo di memoria per ogni nodo: il ***colore*** del nodo, che può essere ROSSO (red) o NERO (black). Assegnando dei vincoli al modo in cui i nodi possono essere colorati lungo qualsiasi percorso che va dalla radice a una foglia, gli alberi red-black garantiscono che nessuno di tali percorsi sia più di due volte più lungo di qualsiasi altro, quindi l'albero è approssimativamente ***bilanciato***.

Ogni nodo dell'albero adesso contiene i campi $color$, key, $left$, $right$ e p. Se manca un figlio o il padre di un nodo, il corrispondente campo puntatore del nodo contiene il valore NIL. Tratteremo questi valori NIL come puntatori a nodi esterni (foglie) dell'albero binario di ricerca e i nodi normali che contengono le chiavi come nodi interni dell'albero.

Un albero binario di ricerca è un albero red-black se soddisfa le seguenti ***proprietà red-black***:

1. Ogni nodo è rosso o nero.
2. La radice è nera.
3. Ogni foglia (NIL) è nera.
4. Se un nodo è rosso, allora entrambi i suoi figli sono neri.
5. Per ogni nodo, tutti i percorsi che vanno dal nodo alle foglie discendenti contengono lo stesso numero di nodi neri.

La Figura 13.1(a) illustra un esempio di albero red-black.

Per semplificare la gestione delle condizioni al contorno nel codice di un albero red-black, utilizzeremo una sola sentinella per rappresentare NIL (vedere pagina 173). Per un albero red-black T, la sentinella $nil[T]$ è un oggetto con gli stessi campi di un nodo ordinario dell'albero; il suo campo $color$ è NERO e gli altri campi – p, $left$, $right$ e key – possono essere impostati a valori arbitrari. Come

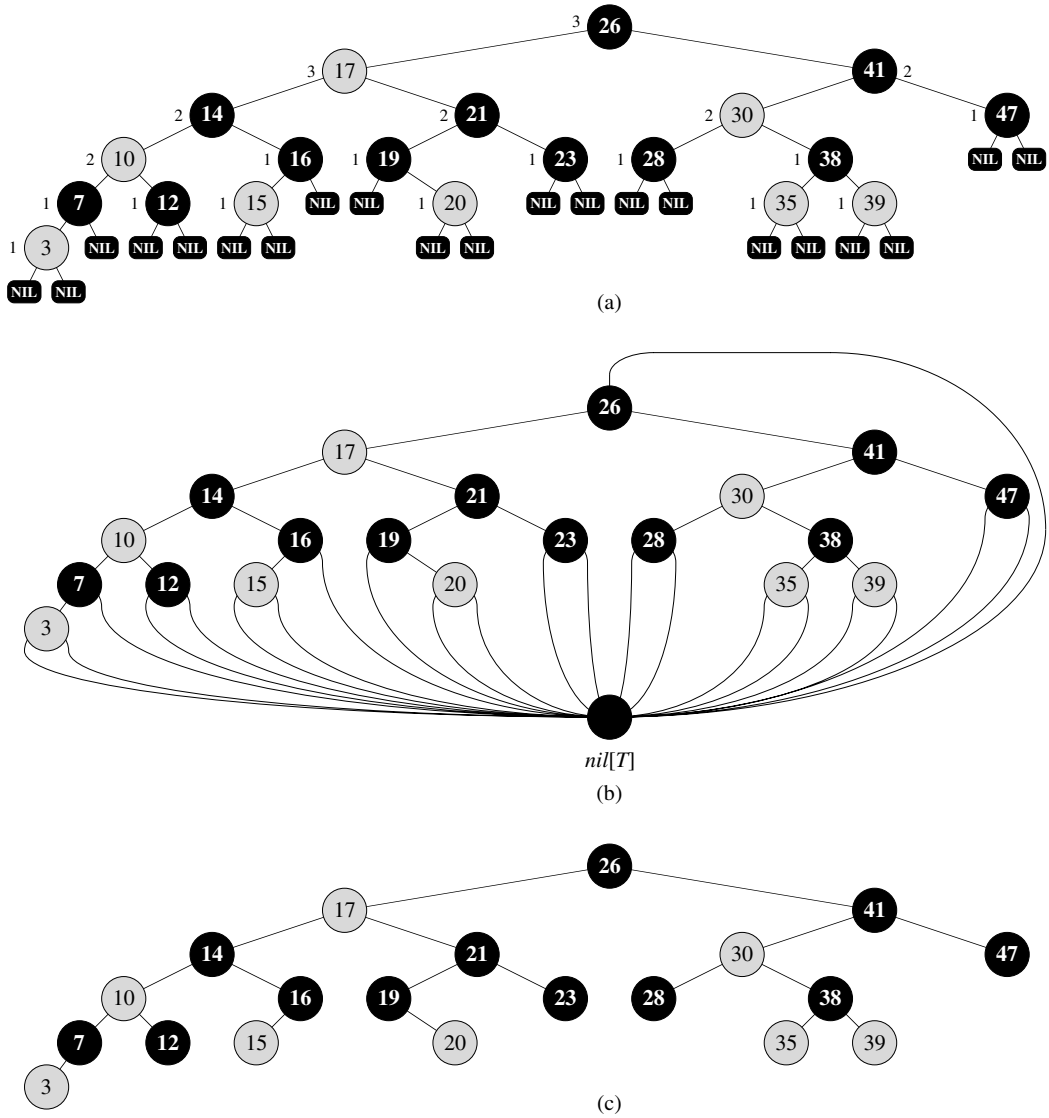


Figura 13.1 Un albero red-black con i nodi neri su sfondo nero e i nodi rossi su sfondo grigio. Ogni nodo di un albero red-black è rosso o nero; i figli di un nodo rosso sono entrambi neri; qualsiasi percorso semplice che va da un nodo a una foglia discendente contiene lo stesso numero di nodi neri. **(a)** Ogni foglia, rappresentata da NIL, è nera. Ogni nodo non-NIL è marcato con la sua altezza nera; i nodi NIL hanno l'altezza nera pari a 0. **(b)** Lo stesso albero red-black dove tutti i nodi NIL sono stati sostituiti dall'unica sentinella $nil[T]$, che è sempre nera (le altezze nere non sono indicate). La sentinella è anche padre della radice. **(c)** Lo stesso albero red-black senza le foglie e il padre della radice. Utilizzeremo questo schema di rappresentazione nella parte restante di questo capitolo.

illustra la Figura 13.1(b), tutti i puntatori a NIL sono sostituiti con puntatori alla sentinella $nil[T]$.

Utilizziamo la sentinella in modo che possiamo trattare un figlio NIL di un nodo x come un nodo ordinario il cui padre è x. Avremmo potuto aggiungere un nodo sentinella distinto per ogni NIL nell'albero, in modo che il padre di ogni nodo NIL fosse ben definito; tuttavia questo approccio avrebbe sprecato dello spazio in memoria. Piuttosto, utilizziamo l'unica sentinella $nil[T]$ per rappresentare tutti i nodi NIL – tutte le foglie e il padre della radice. I valori dei campi p, $left$, $right$

e key della sentinella sono immateriali, sebbene possiamo impostarli durante il corso di una procedura per comodità.

In generale, concentriamo la nostra attenzione ai nodi interni di un albero red-black, perché contengono i valori delle chiavi (key). Nella parte successiva di questo capitolo, ometteremo le foglie nella rappresentazione degli alberi red-black, come illustra la Figura 13.1(c).

Definiamo ***altezza nera*** di un nodo x, indicata da $\mathrm{bh}(x)$, il numero di nodi neri lungo un percorso che inizia dal nodo x (ma non lo include) e finisce in una foglia. Per la proprietà 5, il concetto di altezza nera è ben definito, in quanto tutti i percorsi che scendono dal nodo hanno lo stesso numero di nodi neri. Per definizione, l'altezza nera di un albero red-black è l'altezza nera della sua radice.

Il seguente lemma dimostra perché gli alberi red-black creano buoni alberi di ricerca.

Lemma 13.1
L'altezza massima di un albero red-black con n nodi interni è $2\lg(n+1)$.

Dimostrazione Iniziamo dimostrando che il sottoalbero con radice in un nodo x qualsiasi contiene almeno $2^{\mathrm{bh}(x)} - 1$ nodi interni. Proveremo questa asserzione per induzione sull'altezza di x. Se l'altezza di x è 0, allora x deve essere una foglia ($nil[T]$) e il sottoalbero con radice in x contiene realmente almeno $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ nodi interni. Per il passo induttivo, consideriamo un nodo x che ha un'altezza positiva ed è un nodo interno con due figli. Ogni figlio ha un'altezza nera pari a $\mathrm{bh}(x)$ o $\mathrm{bh}(x) - 1$, a seconda se il suo colore è, rispettivamente, rosso o nero. Poiché l'altezza di un figlio di x è minore dell'altezza dello stesso x, possiamo applicare l'ipotesi induttiva per concludere che ogni figlio ha almeno $2^{\mathrm{bh}(x)-1} - 1$ nodi interni. Quindi, il sottoalbero con radice in x contiene almeno $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ nodi interni; questo dimostra l'asserzione.

Per completare la dimostrazione del lemma, indichiamo con h l'altezza dell'albero. Per la proprietà 4, almeno metà dei nodi in qualsiasi percorso semplice dalla radice a una foglia (esclusa la radice) deve essere nera. Di conseguenza, l'altezza nera della radice deve essere almeno $h/2$; quindi, abbiamo

$$n \geq 2^{h/2} - 1$$

Spostando 1 nel lato sinistro e prendendo i logaritmi di entrambi i lati, otteniamo $\lg(n+1) \geq h/2$ o $h \leq 2\lg(n+1)$. ■

Una immediata conseguenza di questo lemma è che le operazioni sugli insiemi dinamici SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR possono essere implementate nel tempo $O(\lg n)$ negli alberi red-black, perché possono essere eseguite nel tempo $O(h)$ in un albero di ricerca di altezza h (come dimostrato nel Capitolo 12) e qualsiasi albero red-black di n nodi è un albero di ricerca di altezza $O(\lg n)$ (ovviamente, i riferimenti a NIL nei algoritmi del Capitolo 12 dovranno essere sostituiti con $nil[T]$.) Sebbene gli algoritmi TREE-INSERT e TREE-DELETE del Capitolo 12 siano eseguiti nel tempo $O(\lg n)$ quando l'input è un albero red-black, tuttavia essi non supportano direttamente le operazioni sugli insiemi dinamici INSERT e DELETE, in quanto non garantiscono che l'albero binario di ricerca modificato sarà un albero red-black. Tuttavia, nei Paragrafi 13.3

e 13.4 vedremo che queste due operazioni possono essere effettivamente eseguite nel tempo $O(\lg n)$.

Esercizi

13.1-1

Seguendo lo schema della Figura 13.1(a), disegnate l'albero binario di ricerca completo di altezza 3 con le chiavi $\{1, 2, \ldots, 15\}$. Aggiungete le foglie NIL e colorate i nodi in tre modi diversi in modo che le altezze nere degli alberi red-black risultanti siano 2, 3 e 4.

13.1-2

Disegnate l'albero red-black che si ottiene dopo la chiamata di TREE-INSERT con la chiave 36 per l'albero illustrato nella Figura 13.1. Se il nodo inserito è rosso, l'albero risultante è un albero red-black? Che cosa cambia se il nodo inserito è nero?

13.1-3

Definiamo ***albero red-black rilassato*** un albero binario di ricerca che soddisfa le proprietà red-black 1, 3, 4 e 5. In altre parole, la radice può essere rossa o nera. Considerate un albero red-black rilassato T la cui radice è rossa. Se coloriamo di nero la radice di T, ma non apportiamo altre modifiche a T, l'albero risultante è un albero red-black?

13.1-4

Supponete che ogni nodo rosso in un albero red-black venga "assorbito" dal suo padre nero, in modo che i figli del nodo rosso diventino figli del padre nero (ignorate ciò che accade alla chiavi). Quali sono i possibili gradi di un nodo nero dopo che tutti i suoi figli rossi sono stati assorbiti? Che cosa potete dire sulle profondità delle foglie dell'albero risultante?

13.1-5

Dimostrate che, in un albero red-black, il percorso semplice più lungo che va da un nodo x a una foglia discendente ha un'altezza al massimo doppia di quella del percorso semplice più breve dal nodo x a una foglia discendente.

13.1-6

Qual è il numero massimo di nodi interni in un albero red-black di altezza nera k? Qual è il numero minimo?

13.1-7

Descrivete un albero red-black di n chiavi che realizza il massimo rapporto fra nodi interni rossi e nodi interni neri. Qual è questo rapporto? Quale albero ha il più piccolo rapporto e qual è il suo valore?

13.2 Rotazioni

Le operazioni di ricerca TREE-INSERT e TREE-DELETE, quando vengono eseguite in un albero red-black di n chiavi, richiedono un tempo $O(\lg n)$. Poiché queste operazioni modificano l'albero, il risultato potrebbe violare le proprietà red-black elencate nel Paragrafo 13.1. Per ripristinare queste proprietà, dobbiamo modificare i colori di qualche nodo dell'albero e anche la struttura dei puntatori.

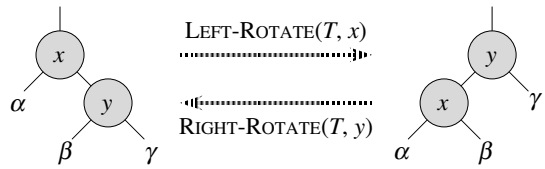


Figura 13.2 Le rotazioni in un albero binario di ricerca. L'operazione LEFT-ROTATE(T, x) trasforma la configurazione dei due nodi a sinistra nella configurazione a destra cambiando un numero costante di puntatori. La configurazione a destra può essere trasformata nella configurazione a sinistra con l'operazione inversa RIGHT-ROTATE(T, y). Le lettere α, β e γ rappresentano sottoalberi arbitrari. Una rotazione preserva la proprietà degli alberi binari di ricerca: le chiavi in α precedono $key[x]$, che precede le chiavi in β, che precedono $key[y]$, che precede le chiavi in γ.

La struttura dei puntatori viene modificata tramite una ***rotazione***: un'operazione locale in un albero di ricerca che preserva la proprietà degli alberi binari di ricerca. La Figura 13.2 illustra i due tipi di rotazioni: rotazione sinistra e rotazione destra. Quando eseguiamo una rotazione sinistra in un nodo x, supponiamo che il suo figlio destro y non sia $nil[T]$; x può essere qualsiasi nodo il cui figlio destro non è $nil[T]$. La rotazione sinistra "fa perno" sul collegamento tra x e y; il nodo y diventa la nuova radice del sottoalbero, con x come figlio sinistro di y e il figlio sinistro di y come figlio destro di x.

Lo pseudocodice per LEFT-ROTATE suppone che $right[x] \neq nil[T]$ e che il padre della radice sia $nil[T]$.

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Imposta y.
 2  right[x] ← left[y]      ▷ Ruota il sottoalbero sinistro di y
                              nel sottoalbero destro di x.
 3  if left[y] ≠ nil[T]
 4     then p[left[y]] ← x
 5  p[y] ← p[x]             ▷ Collega il padre di x a y.
 6  if p[x] = nil[T]
 7     then radice[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x             ▷ Pone x a sinistra di y.
12  p[x] ← y
```

La Figura 13.3 illustra il funzionamento di LEFT-ROTATE. Il codice per RIGHT-ROTATE è simmetrico. Entrambe le procedure LEFT-ROTATE e RIGHT-ROTATE vengono eseguite nel tempo $O(1)$. Soltanto i puntatori vengono modificati da una rotazione; tutti gli altri campi di un nodo non cambiano.

Esercizi

13.2-1

Scrivete lo pseudocodice per RIGHT-ROTATE.

13.2-2

Dimostrate che in ogni albero binario di ricerca di n nodi ci sono esattamente $n - 1$ rotazioni possibili.

13.2-3

Siano a, b e c dei nodi arbitrari, rispettivamente, nei sottoalberi α, β e γ dell'albero a sinistra nella Figura 13.2. Come cambiano le profondità di a, b e c quando viene effettuata una rotazione sinistra del nodo x?

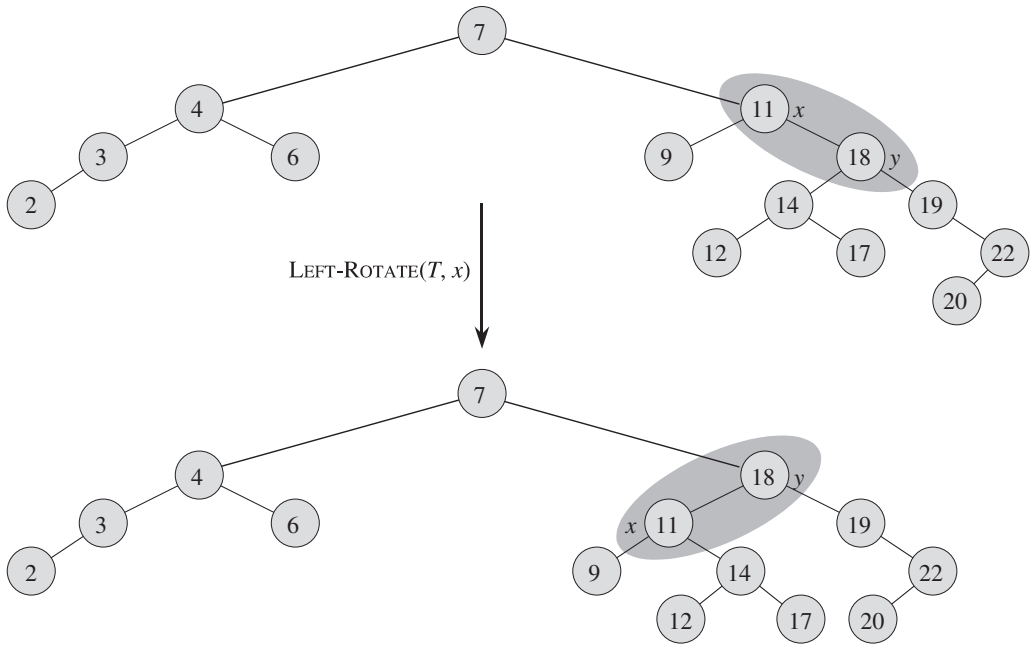


Figura 13.3 Esempio di come la procedura LEFT-ROTATE(T, x) modifica un albero binario di ricerca. Gli attraversamenti simmetrici dell'albero di input e dell'albero modificato producono la stessa lista dei valori delle chiavi.

13.2-4
Dimostrate che un albero binario di ricerca di n nodi può essere trasformato in un altro albero binario di ricerca di n nodi effettuando $O(n)$ rotazioni (*suggerimento:* prima dimostrate che bastano al massimo $n - 1$ rotazioni destre per trasformare l'albero in una catena verso destra).

13.2-5 $\star$
Diciamo che un albero binario di ricerca T_1 può essere ***convertito a destra*** in un albero binario di ricerca T_2 se è possibile ottenere T_2 da T_1 attraverso una serie di chiamate di RIGHT-ROTATE. Indicate un esempio di due alberi T_1 e T_2 tali che T_1 non può essere convertito a destra in T_2. Poi dimostrate che se un albero T_1 può essere convertito a destra in T_2, allora può essere convertito a destra con $O(n^2)$ chiamate di RIGHT-ROTATE.

13.3 Inserimento

L'inserimento di un nodo in un albero red-black di n nodi può essere effettuato nel tempo $O(\lg n)$. Utilizziamo una versione leggermente modificata della procedura TREE-INSERT (Paragrafo 12.3) per inserire un nodo z nell'albero T come se fosse un normale albero binario di ricerca e poi coloriamo z di rosso. Per garantire che le proprietà red-black siano preservate, chiamiamo una procedura ausiliaria RB-INSERT-FIXUP per ricolorare i nodi ed effettuare le rotazioni. La chiamata RB-INSERT(T, z) inserisce il nodo z, il cui campo key si suppone sia stato già riempito, nell'albero red-black T.

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← radice[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then radice[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← ROSSO
17  RB-INSERT-FIXUP(T, z)
```

Ci sono quattro differenze fra le procedure TREE-INSERT e RB-INSERT. In primo luogo, tutte le istanze di NIL in TREE-INSERT sono sostituite con $nil[T]$. In secondo luogo, impostiamo $left[z]$ e $right[z]$ a $nil[T]$ nelle righe 14–15 di RB-INSERT, per mantenere la struttura appropriata dell'albero. In terzo luogo, coloriamo z di rosso nella riga 16. In quarto luogo, poiché colorare z di rosso può causare una violazione di una delle proprietà red-black, chiamiamo RB-INSERT-FIXUP(T, z) nella riga 17 di RB-INSERT per ripristinare le proprietà red-black.

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = ROSSO
 2        do if p[z] = left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y] = ROSSO
 5                      then color[p[z]] ← NERO                    ▷ Caso 1
 6                           color[y] ← NERO                       ▷ Caso 1
 7                           color[p[p[z]]] ← ROSSO                ▷ Caso 1
 8                           z ← p[p[z]]                           ▷ Caso 1
 9                      else  if z = right[p[z]]
10                               then z ← p[z]                     ▷ Caso 2
11                                    LEFT-ROTATE(T, z)            ▷ Caso 2
12                           color[p[z]] ← NERO                    ▷ Caso 3
13                           color[p[p[z]]] ← ROSSO                ▷ Caso 3
14                           RIGHT-ROTATE(T, p[p[z]])              ▷ Caso 3
15              else  (come la clausola then
                           con "right" e "left" scambiati)
16  color[radice[T]] ← NERO
```

Per capire come opera RB-INSERT-FIXUP, dovremo scindere l'analisi del codice in tre fasi principali. Nella prima fase, determineremo quali violazioni delle pro-

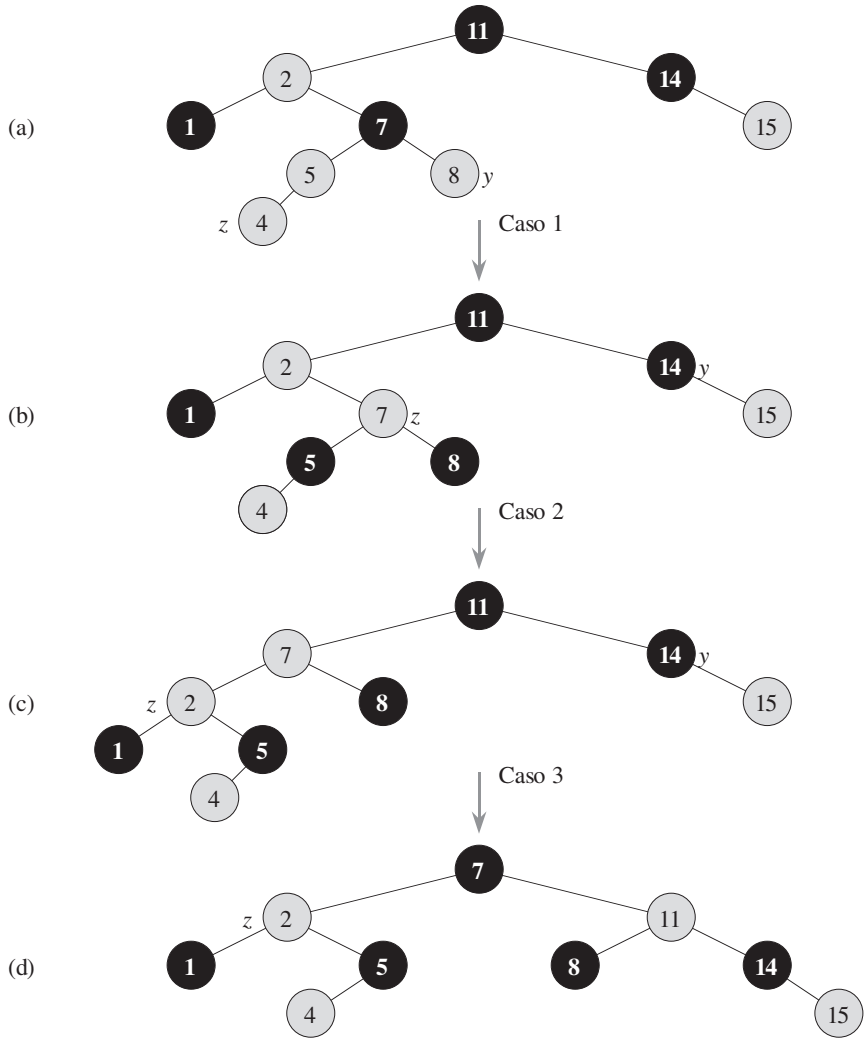


Figura 13.4
Il funzionamento di RB-INSERT-FIXUP. **(a)** Un nodo z dopo l'inserimento. Poiché z e suo padre $p[z]$ sono entrambi rossi, si verifica una violazione della proprietà 4. Poiché lo zio y di z è rosso, si può applicare il caso 1 del codice. I nodi sono ricolorati e il puntatore z si sposta verso l'alto nell'albero, ottenendo l'albero illustrato in **(b)**. Ancora una volta, z e suo padre sono entrambi rossi, ma lo zio y di z è nero. Poiché z è il figlio destro di $p[z]$, si può applicare il caso 2. Viene effettuata una rotazione sinistra; l'albero risultante è illustrato in **(c)**. Adesso z è il figlio sinistro di suo padre e si può applicare il caso 3. Una rotazione destra genera l'albero illustrato in **(d)**, che è un valido albero red-black.

prietà red-black sono introdotte da RB-INSERT quando il nodo z viene inserito e colorato di rosso. Nella seconda fase, esamineremo l'obiettivo globale del ciclo **while** (righe 1–15). Infine, analizzeremo i tre casi[1] in cui è suddiviso il ciclo **while** e vedremo come raggiungono l'obiettivo. La Figura 13.4 illustra come opera RB-INSERT-FIXUP in un albero red-black campione.

Quali proprietà red-black possono essere violate prima della chiamata di RB-INSERT-FIXUP? La proprietà 1 certamente continua a essere valida, come la proprietà 3, in quanto entrambi i figli del nuovo nodo rosso che è stato inserito sono la sentinella $nil[T]$. Anche la proprietà 5, secondo la quale il numero di nodi neri è lo stesso in ogni percorso da un dato nodo, è soddisfatta perché il nodo z sostituisce la sentinella (nera), e il nodo z è rosso con figli sentinella. Quindi, le uniche proprietà che potrebbero essere violate sono la proprietà 2, che richiede che la radice sia nera, e la proprietà 4, che dice che un nodo rosso non può avere un figlio rosso. Entrambe le possibili violazioni si verificano se z è rosso. La proprietà 2 è

[1] Il caso 2 ricade nel caso 3, quindi questi due casi non si escludono a vicenda.

violata se z è la radice; la proprietà 4 è violata se il padre di z è rosso. La Figura 13.4(a) illustra una violazione della proprietà 4 dopo l'inserimento del nodo z.

Le righe 1–15 del ciclo **while** conservano la seguente invariante di tre parti:

All'inizio di ogni iterazione del ciclo:

a. Il nodo z è rosso.
b. Se $p[z]$ è la radice, allora $p[z]$ è nero.
c. Se ci sono violazioni delle proprietà red-black, al massimo ce n'è una e riguarda la proprietà 2 o la proprietà 4. Se c'è una violazione della proprietà 2, questa si verifica perché il nodo z è la radice ed è rosso. Se c'è una violazione della proprietà 4, essa si verifica perché z e $p[z]$ sono entrambi rossi.

La parte (c), che tratta le violazioni delle proprietà red-black, è più importante delle parti (a) e (b) per dimostrare che RB-INSERT-FIXUP ripristina le proprietà red-black. Utilizzeremo le parti (a) e (b) per spiegare, cammin facendo, le situazioni del codice. Poiché siamo interessati al nodo z e ai nodi vicini, è utile sapere dalla parte (a) che z è rosso. Utilizzeremo la parte (b) per dimostrare che il nodo $p[p[z]]$ esiste, quando faremo riferimento ad esso nelle righe 2, 3, 7, 8, 13 e 14.

Ricordiamo che bisogna dimostrare che un'invariante di ciclo è vera prima della prima iterazione del ciclo, che ogni iterazione conserva l'invariante e che l'invariante fornisce un'utile proprietà alla conclusione del ciclo.

Cominciamo dai temi dell'inizializzazione e della conclusione. Poi, quando esamineremo più dettagliatamente come funziona il corpo del ciclo, dimostreremo che il ciclo conserva l'invariante a ogni iterazione. Dimostreremo inoltre che ci sono due possibili risultati per ogni iterazione del ciclo: il puntatore z si sposta verso l'alto nell'albero oppure vengono effettuate delle rotazioni e il ciclo termina.

Inizializzazione: prima della prima iterazione del ciclo, partiamo da un albero red-black senza violazioni dove abbiamo inserito un nodo rosso z. Dimostriamo che tutte le parti dell'invariante sono vere nell'istante in cui viene chiamata la procedura RB-INSERT-FIXUP:

a. Quando viene chiamata RB-INSERT-FIXUP, z è il nodo rosso che è stato aggiunto.
b. Se $p[z]$ è la radice, allora $p[z]$ è inizialmente nero e non cambia colore prima della chiamata di RB-INSERT-FIXUP.
c. Abbiamo già visto che le proprietà 1, 3 e 5 sono valide quando viene chiamata la procedura RB-INSERT-FIXUP.

   Se c'è una violazione della proprietà 2, allora la radice rossa deve essere il nodo z appena inserito, che è l'unico nodo interno nell'albero. Poiché il padre ed entrambi i figli di z sono la sentinella, che è nera, non c'è violazione della proprietà 4. Quindi, questa violazione della proprietà 2 è l'unica violazione delle proprietà red-black nell'intero albero.

   Se c'è una violazione della proprietà 4, allora poiché i figli del nodo z sono sentinelle nere e l'albero non aveva altre violazioni prima dell'inserimento di z, la violazione deve essere attribuita al fatto che z e $p[z]$ sono entrambi rossi. Non ci sono altre violazioni delle proprietà red-black.

Conclusione: quando il ciclo termina, ciò accade perché $p[z]$ è nero (se z è la radice, allora $p[z]$ è la sentinella $nil[T]$, che è nera). Quindi, non c'è violazione della proprietà 4 alla conclusione del ciclo. Per l'invariante di ciclo, l'unica proprietà che potrebbe essere violata è la proprietà 2. La riga 16 ripristina anche questa proprietà, cosicché quando RB-INSERT-FIXUP termina, tutte le proprietà red-black sono valide.

Conservazione: in effetti, ci sarebbero sei casi da considerare nel ciclo **while**, ma tre di essi sono simmetrici agli altri tre, a seconda che il padre $p[z]$ di z sia un figlio sinistro o un figlio destro del nonno $p[p[z]]$ di z; ciò è determinato nella riga 2. Abbiamo riportato il codice soltanto per la situazione in cui $p[z]$ è un figlio sinistro. Il nodo $p[p[z]]$ esiste, in quanto per la parte (b) dell'invariante di ciclo, se $p[z]$ è la radice, allora $p[z]$ è nero. Poiché entriamo in una iterazione del ciclo soltanto se $p[z]$ è rosso, sappiamo che $p[z]$ non può essere la radice. Quindi, $p[p[z]]$ esiste.

Il caso 1 si distingue dai casi 2 e 3 per il colore del fratello del padre di z (lo "zio" di z). La riga 3 fa sì che y punti allo zio $right[p[p[z]]]$ di z; un test viene fatto nella riga 4. Se y è rosso, allora viene applicato il caso 1, altrimenti il controllo passa ai casi 2 e 3. In tutti e tre i casi, il nonno $p[p[z]]$ di z è nero, in quanto il padre $p[z]$ è rosso, quindi la proprietà 4 è violata soltanto fra z e $p[z]$.

Caso 1: lo zio y di z è rosso

La Figura 13.5 illustra la situazione per il caso 1 (righe 5–8). Questo caso viene eseguito quando $p[z]$ e y sono entrambi rossi. Poiché $p[p[z]]$ è nero, possiamo colorare di nero $p[z]$ e y, risolvendo così il problema che z e $p[z]$ sono entrambi rossi; coloriamo di rosso $p[p[z]]$ per conservare la proprietà 5. Poi ripetiamo il ciclo **while** con $p[p[z]]$ come il nuovo nodo z. Il puntatore z si sposta di due livelli in alto nell'albero. Adesso dimostriamo che il caso 1 conserva l'invariante di ciclo all'inizio della successiva iterazione. Utilizziamo z per indicare il nodo z nell'iterazione corrente e $z' = p[p[z]]$ per indicare il nodo z nel test della riga 1 prima della successiva iterazione.

a. Poiché questa iterazione colora di rosso $p[p[z]]$, il nodo z' è rosso all'inizio della successiva iterazione.
b. Il nodo $p[z']$ è $p[p[p[z]]]$ in questa iterazione e il colore di questo nodo non cambia. Se questo nodo è la radice, il suo colore era nero prima di questa iterazione e resta nero all'inizio della successiva iterazione.
c. Abbiamo già dimostrato che il caso 1 conserva la proprietà 5 e, chiaramente, non introduce una violazione delle proprietà 1 e 3.

   Se il nodo z' è la radice all'inizio della successiva iterazione, allora il caso 1 ha corretto l'unica violazione della proprietà 4 in questa iterazione. Poiché z' è rosso ed è la radice, la proprietà 2 diventa l'unica a essere violata e questa violazione è dovuta a z'.

   Se il nodo z' non è la radice all'inizio della successiva iterazione, allora il caso 1 non ha creato una violazione della proprietà 2. Il caso 1 ha corretto l'unica violazione della proprietà 4 che esisteva all'inizio di questa iterazione. Poi ha colorato di rosso z' e ha lasciato solo $p[z']$. Se $p[z']$ era nero, non c'è violazione della proprietà 4. Se $p[z']$ era rosso, la colorazione di rosso di z' ha creato una violazione della proprietà 4 fra z' e $p[z']$.

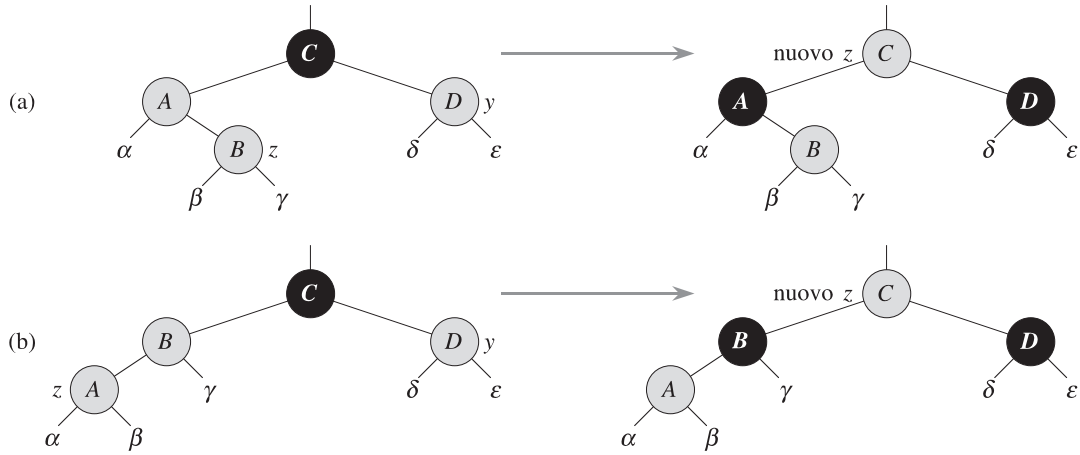


Figura 13.5 Il caso 1 della procedura RB-INSERT. La proprietà 4 è violata, in quanto z e suo padre $p[z]$ sono entrambi rossi. La stessa azione viene svolta se **(a)** z è un figlio destro o **(b)** z è un figlio sinistro. I sottoalberi α, β, γ, δ e ε hanno la radice nera e la stessa altezza nera. Il codice per il caso 1 cambia i colori di qualche nodo, preservando la proprietà 5: tutti i percorsi che scendono da un nodo a una foglia hanno lo stesso numero di nodi neri. Il ciclo **while** continua con il nonno $p[p[z]]$ di z come il nuovo nodo z. Qualsiasi violazione della proprietà 4 adesso può verificarsi soltanto fra il nuovo nodo z, che è rosso, e suo padre, se anche questo nodo è rosso.

Caso 2: lo zio y di z è nero e z è un figlio destro
Caso 3: lo zio y di z è nero e z è un figlio sinistro

Nei casi 2 e 3, il colore dello zio y di z è nero. I due casi si distinguono a seconda che z sia un figlio destro o sinistro di $p[z]$. Le righe 10–11 costituiscono il caso 2, che è illustrato nella Figura 13.6 insieme con il caso 3. Nel caso 2, il nodo z è un figlio destro di suo padre. Effettuiamo immediatamente una rotazione sinistra per trasformare la situazione nel caso 3 (righe 12–14), in cui il nodo z è un figlio sinistro. Poiché z e $p[z]$ sono entrambi rossi, la rotazione non influisce né sull'altezza nera dei nodi né sulla proprietà 5. Sia che entriamo nel caso 3 direttamente o tramite il caso 2, lo zio y di z è nero, perché altrimenti avremmo eseguito il caso 1. In aggiunta, il nodo $p[p[z]]$ esiste, perché abbiamo dimostrato che questo nodo esisteva quando sono state eseguite le righe 2 e 3; inoltre, dopo avere spostato z di un livello in alto nella riga 10 e poi di un livello in basso nella riga 11, l'identità di $p[p[z]]$ resta invariata. Nel caso 3, cambiamo qualche colore ed effettuiamo una rotazione destra per preservare la proprietà 5; dopodiché, dal momento che non abbiamo più due nodi rossi in una riga, abbiamo finito. Il corpo del ciclo **while** non viene eseguito un'altra volta, in quanto $p[z]$ ora è nero.

Adesso dimostriamo che i casi 2 e 3 conservano l'invariante di ciclo (come appena dimostrato, $p[z]$ sarà nero prima del successivo test nella riga 1 e il corpo del ciclo non sarà eseguito di nuovo).

a. Il caso 2 fa sì che z punti a $p[z]$, che è rosso. Non ci sono altre modifiche di z o del suo colore nei casi 2 e 3.
b. Il caso 3 colora di nero $p[z]$, in modo che $p[z]$ sia nero, se è la radice all'inizio della successiva iterazione.
c. Come nel caso 1, le proprietà 1, 3 e 5 si conservano nei casi 2 e 3.

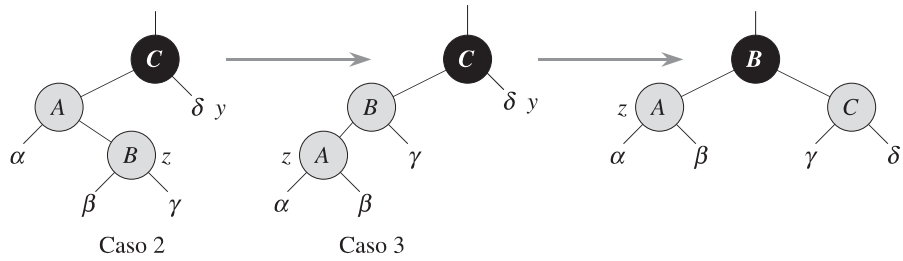


Figura 13.6 I casi 2 e 3 della procedura RB-INSERT. Come nel caso 1, la proprietà 4 è violata nel caso 2 e nel caso 3 perché z e suo padre $p[z]$ sono entrambi rossi. I sottoalberi α, β, γ e δ hanno una radice nera (α, β e γ per la proprietà 4 e δ perché altrimenti saremmo nel caso 1) e la stessa altezza nera. Il caso 2 è trasformato nel caso 3 da una rotazione sinistra, che preserva la proprietà 5: ogni percorso che scende da un nodo a una foglia ha lo stesso numero di nodi neri. Il caso 3 cambia qualche colore ed effettua una rotazione destra per preservare anche la proprietà 5. Il ciclo **while** termina, perché la proprietà 4 è soddisfatta: non ci sono più due nodi rossi in una riga.

Poiché il nodo z non è la radice nei casi 2 e 3, sappiamo che non c'è violazione della proprietà 2. I casi 2 e 3 non introducono una violazione della proprietà 2, perché l'unico nodo che è colorato rosso diventa un figlio di un nodo nero per la rotazione effettuata nel caso 3.

I casi 2 e 3 correggono l'unica violazione della proprietà 4 e non introducono un'altra violazione.

Avendo dimostrato che ogni iterazione del ciclo conserva l'invariante, abbiamo verificato che la procedura RB-INSERT-FIXUP ripristina correttamente le proprietà red-black.

Analisi

Qual è il tempo di esecuzione di RB-INSERT? Poiché l'altezza di un albero red-black di n nodi è $O(\lg n)$, le righe 1–16 di RB-INSERT richiedono il tempo $O(\lg n)$. Nella procedura RB-INSERT-FIXUP, il ciclo **while** viene ripetuto soltanto se viene eseguito il caso 1; poi il puntatore z si sposta di due livelli in alto nell'albero. Il numero totale di volte che può essere eseguito il ciclo **while** è quindi $O(\lg n)$. Di conseguenza, RB-INSERT richiede un tempo totale pari a $O(\lg n)$. È interessante notare che il ciclo **while** non effettua mai più di due rotazioni, perché termina se viene eseguito il caso 2 o il caso 3.

Esercizi

13.3-1

Nella riga 16 di RB-INSERT, coloriamo di rosso il nodo z appena inserito. Notate che se avessimo scelto di colorare di nero il nodo z, allora la proprietà 4 di un albero red-black non sarebbe stata violata. Perché non abbiamo scelto di colorare di nero il nodo z?

13.3-2

Illustrate gli alberi red-black che si ottengono dopo avere inserito in successione le chiavi $41, 38, 31, 12, 19, 8$ in un albero red-black inizialmente vuoto.

13.3-3

Supponete che l'altezza nera di ciascuno dei sottoalberi $\alpha, \beta, \gamma, \delta, \varepsilon$ nelle Figure 13.5 e 13.6 sia k. Etichettate i nodi in ogni figura con la loro altezza nera per verificare che la proprietà 5 è preservata dalla trasformazione indicata.

13.3-4

Il professor Teach teme che RB-INSERT-FIXUP possa impostare $color[nil[T]]$ a ROSSO, nel qual caso il test nella riga 1 non farebbe terminare il ciclo quando z è la radice. Dimostrate che il timore del professore è infondato verificando che RB-INSERT-FIXUP non imposta mai $color[nil[T]]$ a ROSSO.

13.3-5
Considerate un albero red-black che viene formato inserendo n nodi mediante la procedura RB-INSERT. Dimostrate che, se $n > 1$, l'albero ha almeno un nodo rosso.

13.3-6
Spiegate come implementare una procedura RB-INSERT efficiente se la rappresentazione degli alberi red-black non include lo spazio per i puntatori ai nodi padre.

13.4 Cancellazione

Analogamente ad altre operazioni elementari su un albero red-black di n nodi, la cancellazione di un nodo richiede un tempo $O(\lg n)$. La rimozione di un nodo da un albero red-black è un'operazione soltanto un po' più complicata dell'inserimento di un nodo.

La procedura RB-DELETE è una modifica secondaria della procedura TREE-DELETE (descritta nel Paragrafo 12.3); dopo avere rimosso un nodo, chiama la procedura ausiliaria RB-DELETE-FIXUP che cambia i colori ed effettua le rotazioni per ripristinare le proprietà red-black.

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] o right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then radice[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copia i dati satelliti di y in z
16  if color[y] = NERO
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

Ci sono tre differenze fra le procedure TREE-DELETE e RB-DELETE. In primo luogo, tutti i riferimenti a NIL in TREE-DELETE sono sostituiti con i riferimenti alla sentinella $nil[T]$ in RB-DELETE. In secondo luogo, è stato tolto il test per sapere se x è NIL nella riga 7 di TREE-DELETE e l'assegnazione $p[x] \leftarrow p[y]$ avviene in modo incondizionato nella riga 7 di RB-DELETE. Quindi, se x è la sentinella $nil[T]$, il suo puntatore p punta al padre del nodo y cancellato. In terzo luogo, se y è nero, viene effettuata una chiamata di RB-DELETE-FIXUP nelle righe 16–17. Se y è rosso, le proprietà red-black sono ancora valide quando y viene rimosso, per le seguenti ragioni:

- le altezze nere nell'albero non sono cambiate
- non sono stati creati nodi rossi adiacenti
- la radice resta nera perché, se il nodo y fosse stato rosso, non sarebbe potuto essere la radice.

Il nodo x che viene passato a RB-DELETE-FIXUP è uno di questi due nodi: 1) il nodo che era l'unico figlio di y prima della rimozione di y, se y aveva un figlio che non era la sentinella $nil[T]$; 2) la sentinella $nil[T]$, se y non aveva figli. In quest'ultimo caso, l'assegnazione incondizionata nella riga 7 garantisce che il padre di x adesso sia il nodo che in precedenza era il padre di y, indipendentemente dal fatto che x sia un nodo interno con una chiave o la sentinella $nil[T]$.

Adesso possiamo esaminare come la procedura RB-DELETE-FIXUP ripristina le proprietà red-black nell'albero di ricerca.

RB-DELETE-FIXUP(T, x)

```
 1  while x ≠ radice[T] e color[x] = NERO
 2      do if x = left[p[x]]
 3            then w ← right[p[x]]
 4                 if color[w] = ROSSO
 5                   then color[w] ← NERO                          ▷ Caso 1
 6                        color[p[x]] ← ROSSO                      ▷ Caso 1
 7                        LEFT-ROTATE(T, p[x])                     ▷ Caso 1
 8                        w ← right[p[x]]                          ▷ Caso 1
 9                 if color[left[w]] = NERO e color[right[w]] = NERO
10                   then color[w] ← ROSSO                         ▷ Caso 2
11                        x ← p[x]                                 ▷ Caso 2
12                   else if color[right[w]] = NERO
13                          then color[left[w]] ← NERO             ▷ Caso 3
14                               color[w] ← ROSSO                  ▷ Caso 3
15                               RIGHT-ROTATE(T, w)                ▷ Caso 3
16                               w ← right[p[x]]                   ▷ Caso 3
17                        color[w] ← color[p[x]]                   ▷ Caso 4
18                        color[p[x]] ← NERO                       ▷ Caso 4
19                        color[right[w]] ← NERO                   ▷ Caso 4
20                        LEFT-ROTATE(T, p[x])                     ▷ Caso 4
21                        x ← radice[T]                            ▷ Caso 4
22            else (come la clausola then con "right" e "left" scambiati)
23  color[x] ← NERO
```

Se il nodo y rimosso nella procedura RB-DELETE è nero, potrebbero verificarsi tre problemi. In primo luogo, se y era la radice e un figlio rosso di y diventa la nuova radice, abbiamo violato la proprietà 2. In secondo luogo, se entrambi i nodi x e $p[y]$ (che adesso è anche $p[x]$) erano rossi, allora abbiamo violato la proprietà 4. In terzo luogo, in seguito alla rimozione di y, qualsiasi percorso che in precedenza conteneva y, adesso ha un nodo nero in meno; quindi, la proprietà 5 è violata da qualsiasi antenato di y nell'albero. Possiamo correggere questo problema dicendo che il nodo x ha un nodo nero "extra". Ovvero, se aggiungiamo 1 al conteggio dei nodi neri in qualsiasi percorso che contiene x, allora con questa interpretazione la proprietà 5 è valida.

Quando eliminiamo il nodo nero y, "imponiamo" il suo colore a suo figlio. Il problema è che adesso il nodo x non è né rosso né nero, violando così la proprietà 1. Piuttosto, il nodo x è "doppiamente nero" o "rosso e nero" e contribuisce, rispettivamente, con 2 o 1 al conteggio dei nodi neri nei percorsi che contengono x. L'attributo $color$ di x sarà ancora ROSSO (se x è rosso e nero) o NERO (se x è doppiamente nero). In altre parole, il nero extra in un nodo si riflette sul puntamento di x al nodo, non sull'attributo $color$.

La procedura RB-DELETE-FIXUP ripristina le proprietà 1, 2 e 4. Gli esercizi 13.4-1 e 13.4-2 chiedono di dimostrare che la procedura ripristina le proprietà 2 e 4; quindi nella parte restante di questo paragrafo, concentreremo la nostra analisi sulla proprietà 1. L'obiettivo del ciclo **while** (righe 1–22) è spostare il nero extra in alto nell'albero finché

1. x punta a un nodo rosso e nero, nel qual caso coloriamo (singolarmente) di nero x nella riga 23,
2. x punta alla radice, nel qual caso il nero extra può essere semplicemente "rimosso" oppure
3. vengono effettuate opportune rotazioni e ricolorazioni.

All'interno del ciclo **while**, x punta sempre a un nodo doppiamente nero che non è la radice. Determiniamo nella riga 2 se x è un figlio sinistro o un figlio destro di suo padre $p[x]$. (Abbiamo riportato il codice per la situazione in cui x è un figlio sinistro; la situazione in cui x è un figlio destro – riga 22 – è simmetrica.) Manteniamo un puntatore w al fratello di x. Poiché il nodo x è doppiamente nero, il nodo w non può essere $nil[T]$, altrimenti il numero di nodi neri nel percorso da $p[x]$ alla foglia w (singolarmente nera) sarebbe più piccolo del numero di nodi neri nel percorso da $p[x]$ a x.

I quattro casi[2] del codice sono illustrati nella Figura 13.7. Prima di esaminare in dettaglio i singoli casi, analizziamo più in generale come possiamo verificare che la trasformazione in ciascuno di questi casi preserva la proprietà 5. Il concetto base è che in ciascun caso il numero di nodi neri (incluso il nero extra di x) dalla radice (inclusa) del sottoalbero illustrato fino a ciascuno dei sottoalberi $\alpha, \beta, \ldots, \zeta$ è preservato dalla trasformazione. Quindi, se la proprietà 5 è valida prima della trasformazione, continua ad essere valida anche dopo. Per esempio, nella Figura 13.7(a), che illustra il caso 1, il numero di nodi neri dalla radice al sottoalbero α o β è 3, prima e dopo la trasformazione (ricordiamo che il nodo x aggiunge un nero extra). Analogamente, il numero di nodi neri dalla radice a uno dei sottoalberi γ, δ, ε, e ζ è 2, prima e dopo la trasformazione. Nella Figura 13.7(b), il conteggio deve includere il valore c dell'attributo $color$ della radice del sottoalbero illustrato, che può essere ROSSO o NERO. Se definiamo $\mathrm{count}(\text{ROSSO}) = 0$ e $\mathrm{count}(\text{NERO}) = 1$, allora il numero di nodi neri dalla radice ad α è $2 + \mathrm{count}(c)$, prima e dopo la trasformazione. In questo caso, dopo la trasformazione, il nuovo nodo x ha l'attributo $color$ impostato a c, ma questo nodo, in effetti, è rosso e nero (se $c =$ ROSSO) o doppiamente nero (se $c =$ NERO). Gli altri casi possono essere verificati in maniera analoga (vedere l'Esercizio 13.4-5).

[2]Come nella procedura RB-INSERT-FIXUP, i casi di RB-DELETE-FIXUP non si escludono a vicenda.

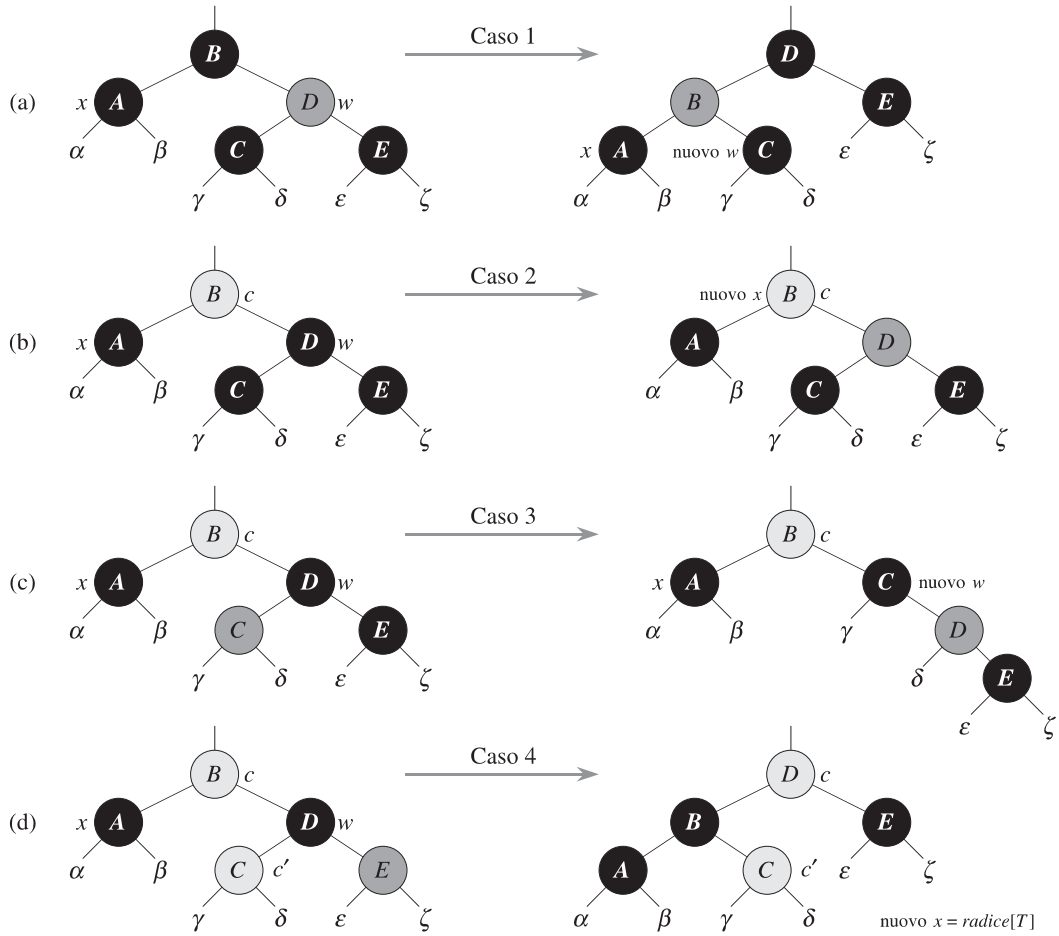


Figura 13.7 I casi nel ciclo **while** della procedura RB-DELETE-FIXUP. I nodi su sfondo nero hanno l'attributo *color* impostato a NERO, quelli su sfondo grigio scuro hanno l'attributo *color* impostato a ROSSO e quelli su sfondo grigio chiaro hanno l'attributo *color* rappresentato da c o c', che può essere ROSSO o NERO. Le lettere $\alpha, \beta, \ldots, \zeta$ rappresentano sottoalberi arbitrari. In ogni caso, la configurazione a sinistra è trasformata nella configurazione a destra cambiando qualche colore e/o effettuando una rotazione. Qualsiasi nodo puntato da x ha un nero extra e può essere doppiamente nero o rosso e nero. L'unico caso che provoca la ripetizione del ciclo è il caso 2. **(a)** Il caso 1 è trasformato nel caso 2, 3 o 4 scambiando i colori dei nodi B e D ed effettuando una rotazione sinistra. **(b)** Nel caso 2, il nero extra rappresentato dal puntatore x viene spostato in alto nell'albero colorando di rosso il nodo D e puntando x al nodo B. Se entriamo nel caso 2 attraverso il caso 1, il ciclo **while** termina perché il nuovo nodo x è rosso e nero, pertanto il valore c del suo attributo *color* è ROSSO. **(c)** Il caso 3 è trasformato nel caso 4 scambiando i colori dei nodi C e D ed effettuando una rotazione destra. **(d)** Nel caso 4, il nero extra rappresentato da x può essere rimosso cambiando qualche colore ed effettuando una rotazione sinistra (senza violare le proprietà red-black); poi il ciclo termina.

Caso 1: il fratello w di x è rosso

Il caso 1 (righe 5–8 della procedura RB-DELETE-FIXUP e Figura 13.7(a)) si verifica quando il nodo w, il fratello del nodo x, è rosso. Poiché w deve avere i figli neri, possiamo scambiare i colori di w e $p[x]$ e poi effettuare una rotazione sinistra di $p[x]$, senza violare nessuna delle proprietà red-black. Il nuovo fratello di x, che era uno dei figli di w prima della rotazione, adesso è nero, e quindi abbiamo trasformato il caso 1 nel caso 2, 3 o 4.

I casi 2, 3 e 4 si verificano quando il nodo w è nero; si distinguono per i colori dei figli di w.

Caso 2: il fratello w di x è nero ed entrambi i figli di w sono neri
Nel caso 2 (righe 10–11 della procedura RB-DELETE-FIXUP e Figura 13.7(b)), entrambi i figli di w sono neri. Poiché anche w è nero, togliamo un nero sia da x sia da w, lasciando x con un solo nero e w rosso. Per compensare la rimozione di un nero da x e w, aggiungiamo un nero extra a $p[x]$, che originariamente era rosso o nero. Per farlo, ripetiamo il ciclo **while** con $p[x]$ come il nuovo nodo x. Notate che, se entriamo nel caso 2 attraverso il caso 1, il nuovo nodo x è rosso e nero, perché l'originale $p[x]$ era rosso. Quindi, il valore c dell'attributo $color$ del nuovo nodo x è ROSSO; il ciclo termina quando verifica la condizione del ciclo. Il nuovo nodo x viene poi colorato (singolarmente) di nero nella riga 23.

Caso 3: il fratello w di x è nero, il figlio sinistro di w è rosso e il figlio destro di w è nero
Il caso 3 (righe 13–16 e Figura 13.7(c)) si verifica quando w è nero, suo figlio sinistro è rosso e suo figlio destro è nero. Possiamo scambiare i colori di w e di sui figlio sinistro $left[w]$ e poi effettuare una rotazione destra di w, senza violare nessuna delle proprietà red-black. Il nuovo fratello w di x adesso è un nodo nero con un figlio destro rosso, quindi abbiamo trasformato il caso 3 nel caso 4.

Caso 4: il fratello w di x è nero e il figlio destro di w è rosso
Il caso 4 (righe 17–21 e Figura 13.7(d)) si verifica quando il fratello w del nodo x è nero e il figlio destro di w è rosso. Cambiando qualche colore ed effettuando una rotazione sinistra di $p[x]$, possiamo rimuovere il nero extra da x, rendendolo singolarmente nero, senza violare nessuna delle proprietà red-black. L'impostazione di x come radice determina la conclusione del ciclo **while** quando viene verificata la condizione del ciclo.

Analisi

Qual è il tempo di esecuzione di RB-DELETE? Poiché l'altezza di un albero red-black di n nodi è $O(\lg n)$, il costo totale della procedura senza la chiamata di RB-DELETE-FIXUP richiede il tempo $O(\lg n)$. All'interno di RB-DELETE-FIXUP, ciascuno dei casi 1, 3 e 4 termina dopo avere effettuato un numero costante di ricolorazioni e al massimo tre rotazioni. Il caso 2 è l'unico caso in cui il ciclo **while** può essere ripetuto; quindi il puntatore x si sposta verso l'alto nell'albero al massimo $O(\lg n)$ volte e non viene effettuata alcuna rotazione. Dunque, la procedura RB-DELETE-FIXUP richiede il tempo $O(\lg n)$ ed effettua al massimo tre rotazioni; in definitiva, anche il tempo totale di RB-DELETE è $O(\lg n)$.

Esercizi

13.4-1
Dimostrate che, dopo l'esecuzione della procedura RB-DELETE-FIXUP, la radice dell'albero deve essere nera.

13.4-2
Dimostrate che, se nella procedura RB-DELETE x e $p[y]$ sono entrambi rossi, allora la proprietà 4 è ripristinata dalla chiamata RB-DELETE-FIXUP(T, x).

13.4-3
Nell'Esercizio 13.3-2 avete trovato l'albero red-black che si ottiene inserendo in successione le chiavi $41, 38, 31, 12, 19, 8$ in un albero inizialmente vuoto. Adesso trovate gli alberi red-black che si ottengono cancellando in successione le chiavi $8, 12, 19, 31, 38, 41$.

13.4-4
In quali righe del codice di RB-DELETE-FIXUP potremmo esaminare o modificare la sentinella $nil[T]$?

13.4-5
Per ciascun caso della Figura 13.7 calcolate il numero di nodi neri dalla radice del sottoalbero illustrato fino a ciascuno dei sottoalberi $\alpha, \beta, \ldots, \zeta$; verificate che ogni conteggio ottenuto non cambia dopo la trasformazione. Quando un nodo ha l'attributo $color$ pari a c o c', usate la notazione $\mathrm{count}(c)$ o $\mathrm{count}(c')$ simbolicamente nei vostri conteggi.

13.4-6
I professori Skelton e Baron temono che, all'inizio del caso 1 della procedura RB-DELETE-FIXUP, il nodo $p[x]$ possa non essere nero. Se i professori avessero ragione, allora le righe 5–6 sarebbero sbagliate. Dimostrate che $p[x]$ è nero all'inizio del caso 1 e, quindi, il timore dei professori è infondato.

13.4-7
Supponete che un nodo x venga inserito in un albero red-black con la procedura RB-INSERT e, poi, immediatamente cancellato con la procedura RB-DELETE. L'albero red-black risultante è lo stesso di quello iniziale? Spiegate la vostra risposta.

Problemi

13-1 Insiemi dinamici persistenti
Durante l'esecuzione di un algoritmo, a volte potrebbe essere necessario conservare le versioni passate di un insieme dinamico mentre viene aggiornato; un insieme così è detto ***persistente***. Un modo per implementare un insieme persistente consiste nel copiare l'intero insieme ogni volta che viene modificato, ma questo approccio può rallentare un programma e anche consumare molto spazio in memoria. In alcuni casi, è possibile fare di meglio.

Considerate un insieme persistente S con le operazioni INSERT, DELETE e SEARCH, che implementiamo utilizzando gli alberi binari di ricerca come illustra la Figura 13.8(a). Manteniamo una radice distinta per ogni versione dell'insieme. Per inserire la chiave 5 nell'insieme, creiamo un nuovo nodo con la chiave 5. Questo nodo diventa il figlio sinistro di un nuovo nodo con la chiave 7, perché non possiamo modificare il nodo esistente con la chiave 7. Analogamente, il nuovo nodo con la chiave 7 diventa il figlio sinistro di un nuovo nodo con la chiave 8 il cui figlio destro è il nodo esistente con la chiave 10. Il nuovo nodo con la chiave 8 diventa, a sua volta, il figlio destro di una nuova radice r' con la chiave 4 il cui figlio sinistro è il nodo esistente con la chiave 3. Quindi, copiamo soltanto una parte dell'albero e condividiamo alcuni nodi con l'albero originale, come illustra la Figura 13.8(b).

Supponiamo che ogni nodo dell'albero abbia i campi key, $left$ e $right$, ma non il campo p del padre del nodo (vedere anche l'Esercizio 13.3-6).

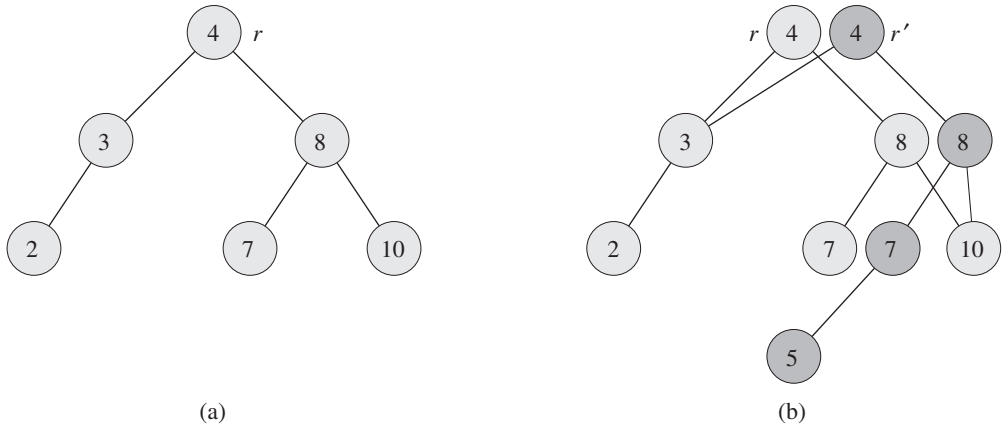


Figura 13.8 **(a)** Un albero binario di ricerca con le chiavi $2, 3, 4, 7, 8, 10$. **(b)** L'albero binario di ricerca persistente che si ottiene inserendo la chiave 5. La versione più recente dell'insieme è formata dai nodi raggiungibili dalla radice r', mentre la precedente versione è formata dai nodi raggiungibili da r. I nodi su sfondo più scuro vengono aggiunti quando viene inserita la chiave 5.

a. Per un generico albero binario di ricerca persistente, identificate i nodi che devono essere modificati per inserire una chiave k o cancellare un nodo y.

b. Scrivete una procedura PERSISTENT-TREE-INSERT che, dato un albero persistente T e una chiave k da inserire, restituisce un nuovo albero persistente T' che è il risultato dell'inserimento di k in T.

c. Se l'altezza dell'albero binario di ricerca persistente T è h, quanto spazio e tempo richiede la vostra implementazione di PERSISTENT-TREE-INSERT? (Lo spazio richiesto è proporzionale al numero dei nuovi nodi allocati.)

d. Supponete di avere incluso il campo p (padre) in ogni nodo. In questo caso, PERSISTENT-TREE-INSERT dovrà svolgere delle operazioni di copia addizionali. Dimostrate che adesso PERSISTENT-TREE-INSERT richiede un tempo e uno spazio pari a $\Omega(n)$, dove n è il numero di nodi dell'albero.

e. Spiegate come utilizzare gli alberi red-black per garantire che lo spazio e il tempo di esecuzione nel caso peggiore siano pari a $O(\lg n)$ per l'inserimento o la cancellazione.

13-2 Unione di alberi red-black

L'operazione di ***unione*** richiede due insiemi dinamici S_1 e S_2 e un elemento x tale che per ogni $x_1 \in S_1$ e $x_2 \in S_2$, si ha $key[x_1] \leq key[x] \leq key[x_2]$. Il risultato è un insieme $S = S_1 \cup \{x\} \cup S_2$. In questo problema descriveremo come implementare l'operazione di unione con gli alberi red-black.

a. Dato un albero red-black T, memorizzate la sua altezza nera nel campo $bh[T]$. Dimostrate che questo campo può essere mantenuto da RB-INSERT e RB-DELETE senza richiedere uno spazio extra nei nodi dell'albero e senza aumentare i tempi di esecuzione asintotici. Dimostrate che discendendo lungo l'albero T, è possibile determinare l'altezza nera di ogni nodo nel tempo $O(1)$ per ogni nodo visitato.

Implementate l'operazione RB-JOIN(T_1, x, T_2), che distrugge T_1 e T_2 e restituisce un albero red-black $T = T_1 \cup \{x\} \cup T_2$. Sia n il numero totale dei nodi in T_1 e T_2.

b. Supponete che $bh[T_1] \geq bh[T_2]$. Descrivete un algoritmo con tempo $O(\lg n)$ che trova un nodo nero y in T_1 con la chiave più grande fra quei nodi la cui altezza nera è $bh[T_2]$.

c. Sia T_y il sottoalbero con radice nel nodo y. Spiegate come $T_y \cup \{x\} \cup T_2$ può sostituire T_y nel tempo $O(1)$ senza distruggere la proprietà degli alberi binari di ricerca.

d. Di che colore dovrebbe essere colorato x per preservare le proprietà red-black 1, 3 e 5? Spiegate come le proprietà 2 e 4 possano essere applicate nel tempo $O(\lg n)$.

e. Dimostrate che l'ipotesi del punto (b) non riduce la generalità del problema. Descrivete la situazione simmetrica che si verifica quando $bh[T_1] \leq bh[T_2]$.

f. Dimostrate che il tempo di esecuzione di RB-JOIN è $O(\lg n)$.

13-3 Alberi AVL

Un ***albero AVL*** è un albero binario di ricerca che è ***bilanciato in altezza***: per ogni nodo x, le altezze dei sottoalberi sinistro e destro di x differiscono al massimo di 1. Per implementare un albero AVL, bisogna mantenere un campo extra in ogni nodo: $h[x]$ è l'altezza del nodo x. Come per qualsiasi albero binario di ricerca T, supponiamo che $radice[T]$ punti al nodo radice.

a. Dimostrate che un albero AVL di n nodi ha un'altezza $O(\lg n)$ (*suggerimento:* dimostrate che in albero AVL di altezza h, ci sono almeno F_h nodi, dove F_h è l'h-esimo numero di Fibonacci).

b. Per inserire un nodo in un albero AVL, il nodo viene posto prima nella posizione appropriata nell'ordine di un albero binario di ricerca. Dopo questo inserimento, l'albero potrebbe non essere più bilanciato in altezza. Specificatamente, le altezze dei figli di qualche nodo potrebbero differire di 2. Descrivete una procedura BALANCE(x) che prende un sottoalbero con radice in x i cui figli sono bilanciati in altezza e hanno altezze che differiscono al massimo di 2 (cioè $|h[right[x]] - h[left[x]]| \leq 2$) e modifica il sottoalbero per bilanciarlo in altezza (*suggerimento:* usate le rotazioni).

c. Utilizzando la parte (b), descrivete una procedura ricorsiva AVL-INSERT(x, z) che prende un nodo x all'interno di un albero AVL e un nodo appena creato z (il cui campo key sia già stato riempito) e aggiunge z al sottoalbero con radice in x, mantenendo la proprietà che x è la radice di un albero AVL. Analogamente alla procedura TREE-INSERT descritta nel Paragrafo 12.3, supponete che il campo $key[z]$ sia stato già riempito e che $left[z] = \text{NIL}$ e $right[z] = \text{NIL}$; supponete inoltre che $h[z] = 0$. Quindi, per inserire il nodo z nell'albero AVL T, chiamate AVL-INSERT$(radice[T], z)$.

d. Dimostrate che la procedura AVL-INSERT, eseguita in un albero AVL di n nodi, impiega il tempo $O(\lg n)$ e svolge $O(1)$ rotazioni.

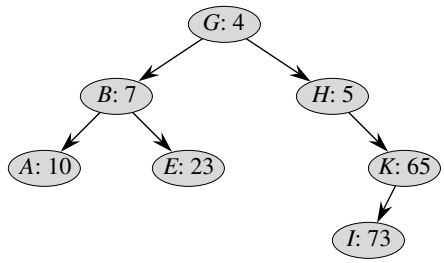


Figura 13.9 Un treap. Ogni nodo x è etichettato con $key[x] : priority[x]$. Per esempio, la radice ha chiave G e priorità 4.

13-4 Treap

Se inseriamo un insieme di n elementi in un albero binario di ricerca, l'albero risultante potrebbe essere notevolmente sbilanciato, determinando lunghi tempi di ricerca. Tuttavia, come detto nel Paragrafo 12.4, gli alberi binari di ricerca costruiti in modo casuale tendono a essere bilanciati. Di conseguenza, una strategia che, in media, costruisce un albero bilanciato per un insieme fisso di elementi consiste nel permutare in modo casuale gli elementi e poi nell'inserire gli elementi in quell'ordine nell'albero.

Che cosa accade se non abbiamo tutti gli elementi contemporaneamente? Se riceviamo gli elementi uno alla volta, possiamo ancora costruire in modo casuale un albero binario di ricerca con tali elementi?

Esamineremo una struttura dati che risponde affermativamente a questa domanda. Un ***treap*** è un albero binario di ricerca che ha un modo diverso di ordinare i nodi. La Figura 13.9 illustra un esempio. Come al solito, ogni nodo x nell'albero ha un valore chiave $key[x]$. In aggiunta, assegniamo $priority[x]$, che è un numero casuale scelto in modo indipendente per ogni nodo. Supponiamo che tutte le priorità e tutte le chiavi siano distinte. I nodi del treap sono ordinati in modo che le chiavi rispettino la proprietà degli alberi binari di ricerca e le priorità rispettino la proprietà di ordinamento del min-heap:

- Se v è un figlio sinistro di u, allora $key[v] < key[u]$.
- Se v è un figlio destro di u, allora $key[v] > key[u]$.
- Se v è un figlio di u, allora $priority[v] > priority[u]$.

Il nome "treap" (ottenuto dalle parole *tre*e e he*ap*) deriva da questa combinazione di proprietà; un treap ha le caratteristiche di un albero binario di ricerca e di un heap.

Può essere d'aiuto pensare ai treap nel modo seguente. Supponiamo di inserire in un treap i nodi $x_1, x_2, \ldots, x_n$ con le chiavi associate. Il treap risultante è l'albero che si sarebbe formato se avessimo inserito i nodi in un normale albero binario di ricerca nell'ordine dato dalle loro priorità (scelte in modo casuale), ovvero $priority[x_i] < priority[x_j]$ significa che x_i è stato inserito prima di x_j.

a. Dimostrate che, dato un insieme di nodi $x_1, x_2, \ldots, x_n$ con relative chiavi e priorità (tutte distinte), esiste un unico treap associato a questi nodi.

b. Dimostrate che l'altezza attesa di un treap è $\Theta(\lg n)$ e, quindi, il tempo per cercare un valore nel treap è $\Theta(\lg n)$.

Vediamo come inserire un nuovo nodo in un treap esistente. La prima cosa da fare è assegnare al nuovo nodo una priorità casuale. Poi chiamiamo l'algoritmo di inserimento TREAP-INSERT, il cui funzionamento è illustrato nella Figura 13.10.

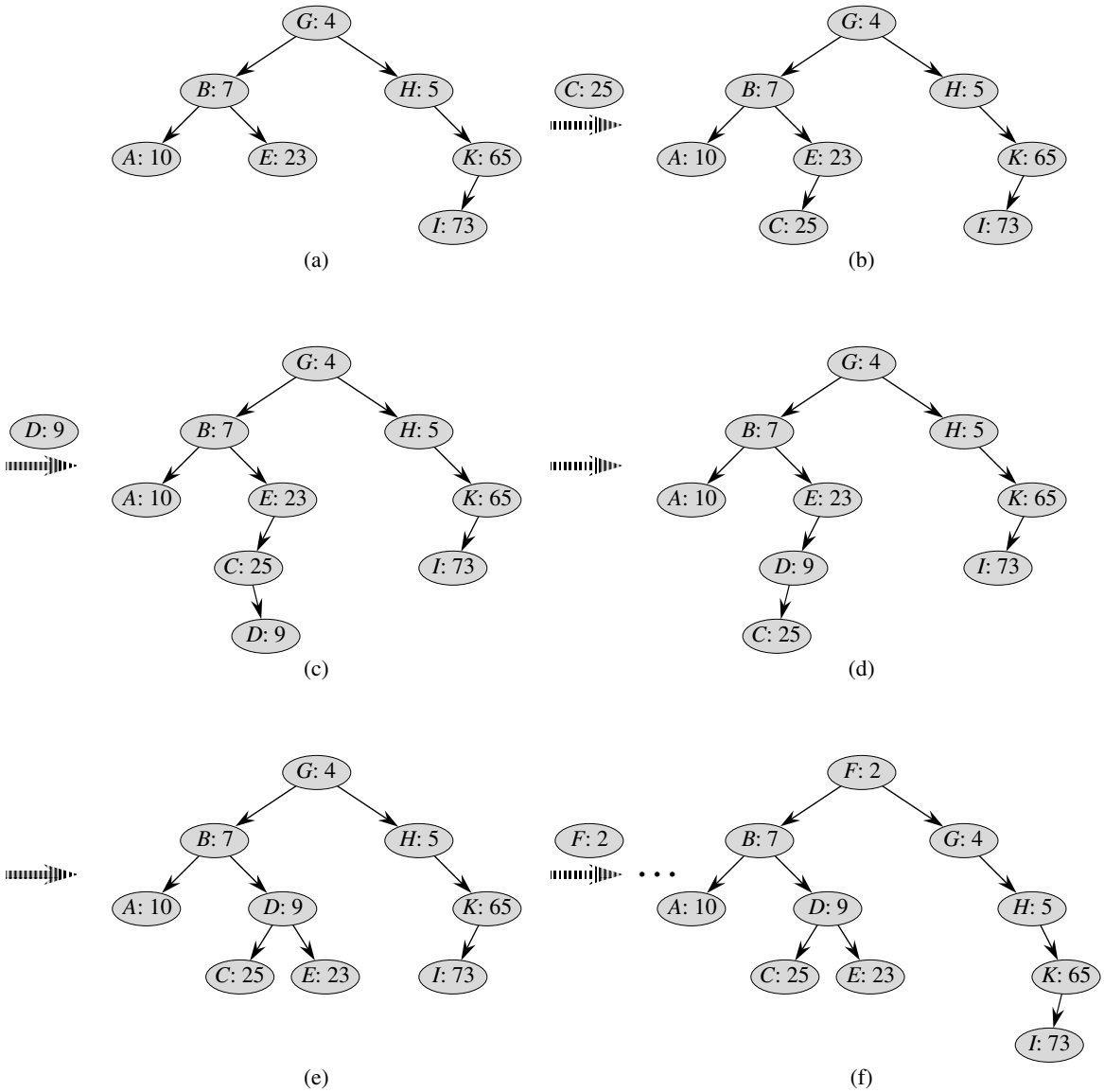


Figura 13.10 Il funzionamento di TREAP-INSERT. **(a)** Il treap originale, prima dell'inserimento del nodo. **(b)** Il treap dopo l'inserimento del nodo con chiave C e priorità 25. **(c)–(d)** Stadi intermedi durante l'inserimento del nodo con chiave D e priorità 9. **(e)** Il treap dopo l'inserimento illustrato nelle parti (c) e (d). **(f)** Il treap dopo l'inserimento del nodo con chiave F e priorità 2.

c. Spiegate come funziona TREAP-INSERT. Descrivete il processo in italiano e scrivete lo pseudocodice (*suggerimento:* eseguite la normale procedura di inserimento di un nodo in un albero binario di ricerca e poi effettuate le rotazioni per ripristinare la proprietà di ordinamento di un min-heap).

d. Dimostrate che il tempo di esecuzione atteso di TREAP-INSERT è $\Theta(\lg n)$.

TREAP-INSERT effettua una ricerca e poi una sequenza di rotazioni. Sebbene queste due operazioni abbiamo lo stesso tempo di esecuzione atteso, in pratica i loro costi sono differenti. Una ricerca legge le informazioni dal treap senza modificarle. Una rotazione, invece, cambia i puntatori dei padri e dei figli all'interno del

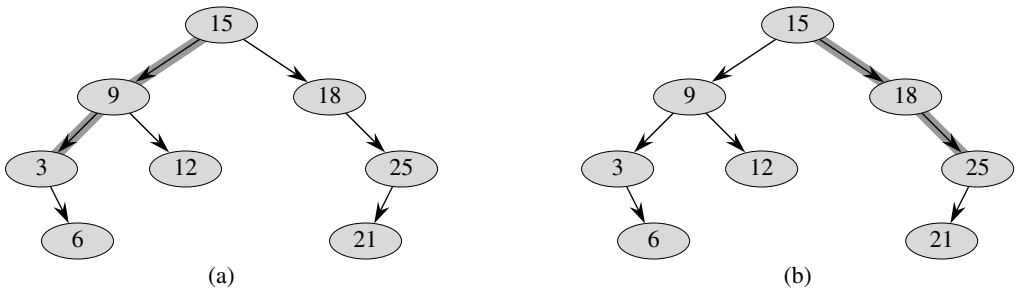


Figura 13.11 Le dorsali di un albero binario di ricerca. La dorsale sinistra è ombreggiata in **(a)**; la dorsale destra è ombreggiata in **(b)**.

treap. Nella maggior parte dei calcolatori, le operazioni di lettura sono molto più veloci di quelle di scrittura. Quindi, sarebbe preferibile che TREAP-INSERT svolgesse poche rotazioni. Dimostreremo che il numero atteso di rotazioni svolte è limitato da una costante.

Per farlo, ci servono alcune definizioni, che sono illustrate nella Figura 13.11. La ***dorsale sinistra*** di un albero binario di ricerca T è il percorso dalla radice al nodo che ha la chiave più piccola. In altre parole, la dorsale sinistra è il percorso dalla radice che è formato soltanto da archi sinistri. In modo simmetrico, la ***dorsale destra*** di T è il percorso dalla radice che è formato soltanto da archi destri. La ***lunghezza*** di una dorsale è il numero di nodi che contiene.

e. Considerate il treap T subito dopo che TREAP-INSERT ha inserito x. Sia C la lunghezza della dorsale destra del sottoalbero sinistro di x. Sia D la lunghezza della dorsale sinistra del sottoalbero destro di x. Dimostrate che il numero totale di rotazioni che sono state effettuate durante l'inserimento di x è uguale a $C + D$.

Adesso calcoliamo i valori attesi di C e D. Senza ridurre la generalità del problema, supponiamo che le chiavi siano $1, 2, \ldots, n$, in quanto le confrontiamo solamente con un'altra chiave.

Per i nodi x e y, con $y \neq x$, sia $k = key[x]$ e $i = key[y]$. Definiamo le variabili casuali indicatrici

$X_{i,k} = \mathrm{I}\{y \text{ è nella dorsale destra del sottoalbero sinistro di } x \text{ (in } T)\}$

f. Dimostrate che $X_{i,k} = 1$, se e soltanto se $priority[y] > priority[x]$, $key[y] < key[x]$, e, per ogni z tale che $key[y] < key[z] < key[x]$, si ha $priority[y] < priority[z]$.

g. Dimostrate che

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}$$

h. Dimostrate che

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}$$

i. Applicate la simmetria per dimostrare che

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1}$$

j. Concludete che il numero atteso di rotazioni effettuate per l'inserimento di un nodo in un treap è minore di 2.

Note

L'idea di bilanciare un albero di ricerca è dovuta ad Adel'son-Vel'skiĭ e Landis [2], che nel 1962 hanno introdotto una classe di alberi di ricerca bilanciati detti "alberi AVL", che sono descritti nel Problema 13-3. Nel 1970 J. E. Hopcroft (nessuna pubblicazione) introdusse un'altra classe di alberi di ricerca, detti "alberi 2-3". Il bilanciamento in un albero 2-3 è mantenuto manipolando i gradi dei nodi dell'albero. Il Capitolo 18 tratta gli alberi B, una generalizzazione degli alberi 2-3 sviluppata da Bayer e McCreight [32].

Gli alberi red-black sono stati ideati da Bayer [31] con il nome di "alberi B binari simmetrici". Guibas e Sedgewick [135] hanno studiato a lungo le loro proprietà e hanno introdotto la convenzione dei colori rosso e nero. Andersson [15] ha fornito una variante degli alberi red-black più semplice da codificare. Weiss [311] ha chiamato questa variante albero AA. Questo tipo di albero è simile a un albero red-black, con la differenza che i figli sinistri non possono essere rossi.

I treap sono stati ideati da Seidel e Aragon [271]. Sono l'implementazione standard di un dizionario in LEDA, che è una collezione bene implementata di strutture dati e algoritmi.

Esistono molte altre varianti di alberi binari bilanciati, come gli alberi bilanciati in peso [230], gli alberi k-neighbor [213] e gli alberi del capro espiatorio (scapegoat tree) [108]. Forse i più interessanti sono gli "alberi splay", introdotti da Sleator e Tarjan [281], che sono "auto-regolanti" (Tarjan [292] ha descritto molto bene questo tipi di alberi). Gli alberi splay mantengono il bilanciamento senza alcuna esplicita condizione di bilanciamento, come il colore. Piuttosto, le "operazioni splay" (che includono le rotazioni) sono svolte all'interno dell'albero ogni volta che si accede all'albero. Il costo ammortizzato (consultate il Capitolo 17) di ciascuna operazione in un albero di n nodi è $O(\lg n)$.

Le skip list [251] sono un'alternativa agli alberi binari di ricerca. Una skip list è una lista concatenata che viene ampliata con un certo numero di puntatori addizionali. Ogni operazione di dizionario viene eseguita nel tempo atteso $O(\lg n)$ in una skip list di n elementi.

Estendere le strutture dati 14

Per alcuni problemi di ingegneria informatica è sufficiente una struttura dati elementare (fra quelle trattate in qualsiasi "libro di testo") – come una lista doppiamente concatenata, una tabella hash o un albero binario di ricerca – ma per molti altri problemi occorre un pizzico di creatività. Soltanto in rare situazioni avrete bisogno di creare un tipo di struttura dati completamente nuovo. Più frequentemente, sarà sufficiente estendere una struttura dati elementare memorizzando in essa delle informazioni aggiuntive; poi, potrete programmare le nuove operazioni per la struttura dati per realizzare l'applicazione desiderata. Estendere una struttura dati non è sempre semplice, in quanto le informazioni aggiuntive devono essere aggiornate e gestite dalle ordinarie operazioni sulla struttura dati.

Questo capitolo descrive due strutture dati che sono costruite estendendo gli alberi red-black. Il Paragrafo 14.1 descrive un struttura dati che supporta le operazioni generali di statistica d'ordine su un insieme dinamico. Potremo così trovare rapidamente l'i-esimo numero più piccolo in un insieme o il rango di un elemento in un insieme ordinato di elementi. Il Paragrafo 14.2 generalizza il processo dell'estensione di una struttura dati e presenta un teorema che può semplificare l'estensione degli alberi red-black. Il Paragrafo 14.3 applica questo teorema per agevolare la progettazione di una struttura dati che gestisce un insieme dinamico di intervalli, come gli intervalli temporali. Dato un intervallo di input, potremo rapidamente trovare un intervallo nell'insieme che si sovrappone ad esso.

14.1 Statistiche d'ordine dinamiche

Il Capitolo 9 ha introdotto il concetto di statistica d'ordine. Specificatamente, l'i-esima statistica d'ordine di un insieme di n elementi, con $i \in \{1, 2, \ldots, n\}$, è semplicemente l'elemento dell'insieme con l'i-esima chiave più piccola. Abbiamo visto che qualsiasi statistica d'ordine può essere ottenuta nel tempo $O(n)$ da un insieme non ordinato. In questo paragrafo, vedremo come modificare gli alberi red-black in modo che qualsiasi statistica d'ordine possa essere determinata nel tempo $O(\lg n)$. Vedremo che anche il ***rango*** di un elemento – la posizione che occupa nella sequenza ordinata degli elementi dell'insieme – può essere determinato nel tempo $O(\lg n)$.

Nella Figura 14.1 è illustrata una struttura dati che è in grado di supportare operazioni rapide con le statistiche d'ordine. Un ***albero di statistiche d'ordine*** T è un albero red-black con un'informazione aggiuntiva memorizzata in ogni nodo. In un nodo x di un albero red-black, oltre ai campi usuali $key[x]$, $color[x]$, $p[x]$, $left[x]$ e $right[x]$, troviamo un altro campo: $size[x]$. Questo campo contiene il numero di nodi (interni) nel sottoalbero con radice in x (incluso lo stesso x), cioè la dimensione del sottoalbero. Se definiamo che la dimensione della sentinella è

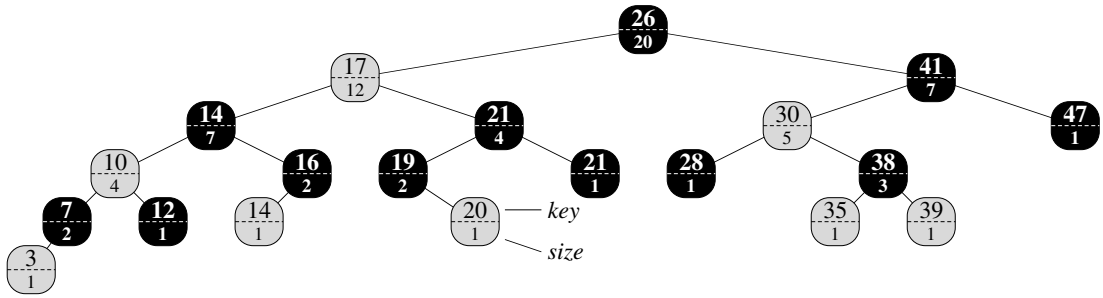


Figura 14.1 Un albero di statistiche d'ordine; è un'estensione di un albero red-black. I nodi su sfondo grigio sono rossi; quelli su sfondo nero sono nodi neri. Oltre ai suoi campi usuali, ogni nodo x ha un campo $size[x]$, che è il numero di nodi nel sottoalbero con radice in x.

0, ovvero impostiamo $size[nil[T]]$ a 0, allora abbiamo l'identità

$$size[x] = size[left[x]] + size[right[x]] + 1$$

In un albero di statistiche d'ordine non è richiesto che le chiavi siano distinte (per esempio, l'albero nella Figura 14.1 ha due chiavi con valore 14 e due chiavi con valore 21). In presenza di chiavi uguali, la precedente nozione di rango non è ben definita. Eliminiamo questa ambiguità per un albero di statistiche d'ordine definendo il rango di un elemento come la posizione in cui l'elemento sarebbe elencato in un attraversamento simmetrico dell'albero. Per esempio, nella Figura 14.1 la chiave 14 memorizzata in un nodo nero ha rango 5 e la chiave 14 memorizzata in un nodo rosso ha rango 6.

Ricerca di un elemento con un dato rango

Prima di vedere come gestire le informazioni sulle dimensioni (*size*) durante l'inserimento e la cancellazione, esaminiamo l'implementazione di due operazioni di statistica d'ordine che usano questa informazione aggiuntiva. Iniziamo con un'operazione che trova un elemento con un dato rango. La procedura OS-SELECT(x, i) restituisce un puntatore al nodo che contiene l'i-esima chiave più piccola nel sottoalbero con radice in x. Per trovare l'i-esima chiave più piccola in un albero di statistiche d'ordine T, chiamiamo OS-SELECT$(radice[T], i)$.

```
OS-SELECT(x, i)
1  r ← size[left[x]]+1
2  if i = r
3     then return x
4  elseif i < r
5     then return OS-SELECT(left[x], i)
6  else return OS-SELECT(right[x], i − r)
```

L'idea che sta alla base di OS-SELECT è simile a quella degli algoritmi di selezione descritti nel Capitolo 9. Il valore di $size[left[x]]$ è il numero di nodi che precedono x in un attraversamento simmetrico del sottoalbero con radice in x. Quindi, $size[left[x]] + 1$ è il rango di x all'interno del sottoalbero con radice in x.

La riga 1 di OS-SELECT calcola r, il rango del nodo x all'interno del sottoalbero con radice in x. Se $i = r$, allora il nodo x è l'i-esimo elemento più piccolo, quindi la riga 3 restituisce x. Se $i < r$, allora l'i-esimo elemento più piccolo è

nel sottoalbero sinistro di x, quindi la riga 5 effettua una ricorsione su $left[x]$. Se $i > r$, allora l'i-esimo elemento più piccolo è nel sottoalbero destro di x. Poiché ci sono r elementi nel sottoalbero con radice in x che precedono il sottoalbero destro di x in un attraversamento simmetrico, l'i-esimo elemento più piccolo nel sottoalbero con radice in x è l'$(i - r)$-esimo elemento più piccolo nel sottoalbero con radice in $right[x]$. Questo elemento è determinato in modo ricorsivo nella riga 6.

Per vedere come opera OS-SELECT, consideriamo la ricerca del 17-esimo elemento più piccolo nell'albero di statistiche d'ordine della Figura 14.1. Iniziamo con x come radice, la cui chiave è 26, e con $i = 17$. Poiché la dimensione del sottoalbero sinistro di 26 è 12, il suo rango è 13. Quindi, sappiamo che il nodo con rango 17 è il quarto $(17 - 13 = 4)$ elemento più piccolo nel sottoalbero destro di 26. Dopo la chiamata ricorsiva, x è il nodo con chiave 41 e $i = 4$. Poiché la dimensione del sottoalbero sinistro di 41 è 5, il suo rango all'interno del suo sottoalbero è 6. Quindi, sappiamo che il nodo con rango 4 è il quarto elemento più piccolo nel sottoalbero sinistro di 41. Dopo la chiamata ricorsiva, x è il nodo con chiave 30 e il suo rango all'interno del suo sottoalbero è 2. Quindi, effettuiamo di nuovo la ricorsione per trovare il secondo $(4 - 2 = 2)$ elemento più piccolo nel sottoalbero con radice nel nodo con chiave 38. Adesso il suo sottoalbero sinistro ha dimensione 1; questo significa che esso è il secondo elemento più piccolo. Quindi, la procedura restituisce un puntatore al nodo con chiave 38.

Poiché per ogni chiamata ricorsiva si scende di un livello nell'albero di statistiche d'ordine, il tempo totale di OS-SELECT, nel caso peggiore, è proporzionale all'altezza dell'albero. Poiché l'albero è un albero red-black, la sua altezza è $O(\lg n)$, dove n è il numero di nodi. Quindi, il tempo di esecuzione di OS-SELECT è $O(\lg n)$ per un insieme dinamico di n elementi.

Determinare il rango di un elemento

Dato un puntatore a un nodo x in un albero di statistiche d'ordine T, la procedura OS-RANK restituisce la posizione di x nell'ordinamento lineare determinato da un attraversamento simmetrico dell'albero T.

```
OS-RANK(T, x)
1  r ← size[left[x]] + 1
2  y ← x
3  while y ≠ radice[T]
4        do if y = right[p[y]]
5              then r ← r + size[left[p[y]]] + 1
6           y ← p[y]
7  return r
```

La procedura funziona nel modo seguente. Il rango di x può essere considerato come il numero di nodi che precedono x in un attraversamento simmetrico dell'albero, più 1 per x stesso. OS-RANK conserva la seguente invariante di ciclo:

> All'inizio di ogni iterazione del ciclo **while** (righe 3–6), r è il rango di $key[x]$ nel sottoalbero con radice nel nodo y.

Utilizzeremo questa invariante di ciclo per dimostrare che OS-RANK opera correttamente:

Inizializzazione: prima della prima iterazione, la riga 1 imposta r al rango di $key[x]$ all'interno del sottoalbero con radice in x. L'impostazione $y \leftarrow x$ nella riga 2 rende l'invariante vera la prima volta che viene eseguito il test nella riga 3.

Conservazione: alla fine di ogni iterazione del ciclo **while**, poniamo $y \leftarrow p[y]$. Quindi dobbiamo dimostrare che, se r è il rango di $key[x]$ nel sottoalbero con radice in y all'inizio del corpo del ciclo, allora r è il rango di $key[x]$ nel sottoalbero con radice in $p[y]$ alla fine del corpo del ciclo. In ogni iterazione del ciclo **while**, consideriamo il sottoalbero con radice in $p[y]$. Abbiamo già contato il numero di nodi nel sottoalbero con radice nel nodo y che precedono x in un attraversamento simmetrico, quindi dobbiamo aggiungere i nodi nel sottoalbero con radice nel fratello di y che precedono x in un attraversamento simmetrico, più 1 per $p[y]$, se anche questo nodo precede x. Se y è un figlio sinistro, allora né $p[y]$ né altri nodi nel sottoalbero destro di $p[y]$ precedono x, quindi lasciamo r invariato. Altrimenti, y è un figlio destro e tutti i nodi nel sottoalbero sinistro di $p[y]$ precedono x, come lo stesso $p[y]$. Quindi, nella riga 5, aggiungiamo $size[left[p[y]]] + 1$ al valore corrente di r.

Conclusione: il ciclo termina quando $y = radice[T]$, sicché il sottoalbero con radice in y è l'intero albero. Dunque, il valore di r è il rango di $key[x]$ nell'intero albero.

Per esempio, se eseguiamo OS-RANK nell'albero di statistiche d'ordine della Figura 14.1 per trovare il rango del nodo con chiave 38, otteniamo le seguente sequenza di valori per $key[y]$ e r all'inizio del ciclo **while**:

iterazione	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

La procedura restituisce il rango 17.

Poiché ogni iterazione del ciclo **while** impiega il tempo $O(1)$ e y risale di un livello nell'albero a ogni iterazione, il tempo di esecuzione di OS-RANK, nel caso peggiore, è proporzionale all'altezza dell'albero: $O(\lg n)$ in un albero di statistiche d'ordine di n nodi.

Gestione delle dimensioni dei sottoalberi

Dato il campo $size$ in ogni nodo, OS-SELECT e OS-RANK possono calcolare rapidamente le informazioni sulle statistiche d'ordine. Tuttavia, questo lavoro risulterebbe inutile se questi campi non potessero essere gestiti con efficienza dalle operazioni di base che modificano gli alberi red-black. Vediamo ora come gestire le dimensioni dei sottoalberi durante le operazioni di inserimento e cancellazione senza influire sul tempo di esecuzione asintotico di ciascuna operazione.

Come detto nel Paragrafo 13.3, l'inserimento in un albero red-black si svolge in due fasi. Nella prima fase, si discende dalla radice dell'albero, inserendo il nuovo nodo come figlio di un nodo esistente. Nella seconda fase si risale verso la radice, cambiando i colori ed effettuando qualche rotazione per conservare le proprietà degli alberi red-black.

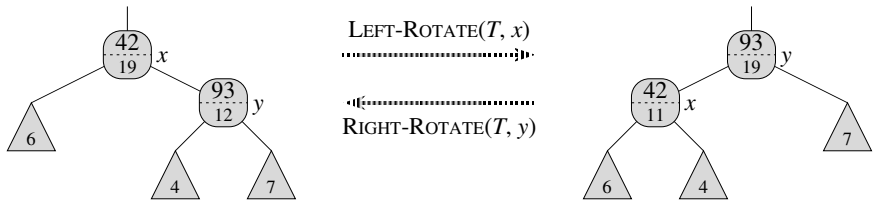


Figura 14.2
Aggiornamento delle dimensioni dei sottoalberi durante le rotazioni. Il collegamento attorno al quale viene effettuata la rotazione unisce i due nodi i cui campi $size$ devono essere aggiornati. Gli aggiornamenti sono locali, richiedendo soltanto le informazioni $size$ memorizzate in x, y e nelle radici dei sottoalberi rappresentati da triangoli.

Per gestire le dimensioni dei sottoalberi nella prima fase, incrementiamo semplicemente $size[x]$ per ogni nodo x nel percorso che va dalla radice fino alle foglie. Il nuovo nodo che viene aggiunto ha il campo $size$ pari a 1. Poiché ci sono $O(\lg n)$ nodi lungo il percorso che è stato fatto, il costo addizionale per gestire i campi $size$ è $O(\lg n)$.

Nella seconda fase, le uniche modifiche strutturali dell'albero red-black di base sono provocate dalle rotazioni, che sono al massimo due. Inoltre, una rotazione è un'operazione locale: soltanto due nodi hanno i campi $size$ invalidati. Il collegamento attorno al quale viene effettuata la rotazione unisce questi due nodi. Facendo riferimento al codice della procedura LEFT-ROTATE(T, x) (Paragrafo 13.2), aggiungiamo le seguenti righe:

13 $size[y] \leftarrow size[x]$
14 $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$

L'aggiornamento dei campi è illustrato nella Figura 14.2. La modifica di RIGHT-ROTATE è simmetrica.

Poiché vengono effettuate al massimo due rotazioni durante l'inserimento in un albero red-black, occorre soltanto un tempo addizionale $O(1)$ per aggiornare i campi $size$ nella seconda fase. Quindi, il tempo totale per completare l'inserimento in un albero di statistiche d'ordine di n nodi è $O(\lg n)$, che è asintoticamente uguale a quello di un normale albero red-black.

Anche la cancellazione da un albero red-black è formata da due fasi: la prima opera sull'albero di ricerca di base; la seconda provoca al massimo tre rotazioni, senza altre modifiche strutturali (vedere il Paragrafo 13.4). La prima fase rimuove un nodo y. Per aggiornare le dimensioni dei sottoalberi, seguiamo un percorso dal nodo y fino alla radice, riducendo il valore del campo $size$ per ogni nodo che incontriamo. Poiché questo percorso ha una lunghezza $O(\lg n)$ in un albero red-black di n nodi, il tempo aggiuntivo che viene impiegato per gestire i campi $size$ nella prima fase è $O(\lg n)$. Le $O(1)$ rotazioni nella seconda fase della cancellazione possono essere gestite come è stato fatto nell'inserimento. Quindi, le operazioni di inserimento e cancellazione, inclusa la gestione dei campi $size$, richiedono un tempo $O(\lg n)$ per un albero di statistiche d'ordine di n nodi.

Esercizi

14.1-1

Spiegate come opera OS-SELECT$(radice[T], 10)$ sull'albero red-black T della Figura 14.1.

14.1-2

Spiegate come opera OS-RANK(T, x) sull'albero red-black T della Figura 14.1 se il nodo x ha il campo $key[x] = 35$.

14.1-3
Scrivete una versione non ricorsiva di OS-SELECT.

14.1-4
Scrivete una procedura ricorsiva di OS-KEY-RANK(T, k) che riceve in input un albero di statistiche d'ordine T e una chiave k e restituisce il rango di k nell'insieme dinamico rappresentato da T. Supponete che le chiavi di T siano distinte.

14.1-5
Dato un elemento x in un albero di statistiche d'ordine di n nodi e un numero naturale i, come può essere determinato nel tempo $O(\lg n)$ l'i-esimo successore di x nell'ordinamento lineare dell'albero?

14.1-6
Notate che ogni volta che c'è un riferimento al campo $size$ di un nodo nelle procedure OS-SELECT e OS-RANK, il campo è utilizzato soltanto per calcolare il rango del nodo nel sottoalbero con radice in quel nodo. Conformemente, supponete di registrare in ogni nodo il suo rango nel sottoalbero di cui esso è la radice. Spiegate come questa informazione può essere gestita durante l'inserimento e la cancellazione (ricordiamo che queste due operazioni possono provocare delle rotazioni).

14.1-7
Spiegate come utilizzare un albero di statistiche d'ordine per contare nel tempo $O(n \lg n)$ il numero di inversioni (vedere il Problema 2-4) in un array di dimensione n.

14.1-8 $\star$
Considerate n corde in un cerchio, ciascuna definita dai suoi estremi. Descrivete un algoritmo con tempo $O(n \lg n)$ per determinare il numero di coppie di corde che si intersecano all'interno del cerchio (per esempio, se le n corde sono diametri che si intersecano al centro del cerchio, allora la soluzione corretta è $\binom{n}{2}$). Supponete che due corde non possano avere un estremo in comune.

14.2 Come estendere una struttura dati

Il processo che estende una struttura dati elementare in modo da supportare nuove funzionalità si ripete spesso nella progettazione degli algoritmi. Questo processo sarà applicato di nuovo nel prossimo paragrafo per progettare una struttura dati che supporta le operazioni sugli intervalli. In questo paragrafo, esamineremo i passaggi che formano il processo di estensione. Dimostreremo anche un teorema che, in molti casi, permette di estendere facilmente gli alberi red-black.

Il processo di estensione di una struttura dati può essere suddiviso in quattro passaggi:

1. Scegliere una struttura dati di base.
2. Determinare le informazioni aggiuntive da gestire nella struttura dati di base.
3. Verificare che le informazioni aggiuntive possono essere gestite come quelle di base modificando le operazioni sulla struttura dati di base.
4. Sviluppare nuove operazioni.

Come in tutti i metodi di progettazione, non occorre seguire alla cieca questi passi nell'ordine in cui sono elencati. Spesso la progettazione include una fase in cui si procede per tentativi e il progresso nei vari passaggi, di solito, avviene in parallelo. Per esempio, non c'è un punto in cui determiniamo le informazioni aggiuntive o sviluppiamo le nuove operazioni (passaggi 2 e 4) se non siamo in grado di gestire con efficienza le informazioni aggiuntive. Ciononostante, questo metodo in quattro passaggi rappresenta uno strumento valido per focalizzare i nostri sforzi sul processo di estensione di una struttura dati ed è anche un buon sistema per organizzare la documentazione di una struttura dati estesa.

Abbiamo seguito questi passaggi nel Paragrafo 14.1 per progettare i nostri alberi di statistiche d'ordine. Per il passaggio 1, abbiamo scelto gli alberi red-black come struttura dati di base. Un segnale sull'idoneità degli alberi red-black proviene dal loro efficiente supporto ad altre operazioni sugli insiemi dinamici con ordinamento totale, come MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR.

Per il passaggio 2, abbiamo fornito il campo $size$, dove ogni nodo x memorizza la dimensione del sottoalbero con radice in x. In generale, le informazioni aggiuntive rendono le operazioni più efficienti. Per esempio, avremmo potuto implementare le procedure OS-SELECT e OS-RANK utilizzando soltanto le chiavi memorizzate nell'albero, ma tali procedure non sarebbero state eseguite nel tempo $O(\lg n)$. A volte, le informazioni aggiuntive sono informazioni sui puntatori anziché i dati, come nell'Esercizio 14.2-1.

Per il passaggio 3, abbiamo garantito che le operazioni di inserimento e cancellazione possano gestire correttamente i campi $size$, continuando a essere eseguite nel tempo $O(\lg n)$. Teoricamente, soltanto pochi elementi della struttura dati hanno bisogno di essere aggiornati per gestire le informazioni aggiuntive. Per esempio, se registrassimo semplicemente in ogni nodo il loro rango nell'albero, le procedure OS-SELECT e OS-RANK sarebbero eseguite rapidamente, ma l'inserimento di un nuovo elemento minimo modificherebbe questa informazione in tutti i nodi dell'albero. Se, invece, memorizziamo le dimensioni dei sottoalberi, l'inserimento di un nuovo elemento modifica le informazioni soltanto in $O(\lg n)$ nodi.

Per il passaggio 4, abbiamo sviluppato le operazioni OS-SELECT e OS-RANK. Dopo tutto, l'esigenza di svolgere nuove operazioni è il motivo principale per cui ci preoccupiamo di estendere una struttura dati. A volte, anziché sviluppare nuove operazioni, utilizziamo le informazioni aggiuntive per accelerare quelle esistenti, come nell'Esercizio 14.2-1.

Estendere gli alberi red-black

Quando gli alberi red-black sono utilizzati per estendere una struttura dati, possiamo provare che certi tipi di informazioni aggiuntive possono essere sempre gestiti in maniera efficiente nelle operazioni di inserimento e cancellazione, semplificando notevolmente il passaggio 3. La dimostrazione del seguente teorema è simile al ragionamento fatto nel Paragrafo 14.1 per spiegare che il campo $size$ può essere correttamente gestito negli alberi di statistiche d'ordine.

Teorema 14.1 (Estendere un albero red-black)
Sia f un campo che estende un albero red-black T di n nodi; supponiamo che il contenuto di f per un nodo x possa essere calcolato utilizzando soltanto le

informazioni nei nodi x, $left[x]$ e $right[x]$, inclusi $f[left[x]]$ e $f[right[x]]$. Allora, è possibile gestire i valori di f in tutti i nodi di T durante l'inserimento e la cancellazione, senza influire asintoticamente sulla prestazione $O(\lg n)$ di queste operazioni.

Dimostrazione Il concetto che sta alla base della dimostrazione è che la modifica di un campo f in un nodo x si propaga soltanto negli antenati di x nell'albero. In altre parole, la modifica di $f[x]$ potrebbe richiedere l'aggiornamento di $f[p[x]]$, ma niente altro; l'aggiornamento di $f[p[x]]$ potrebbe richiedere l'aggiornamento di $f[p[p[x]]]$, ma niente altro; e così via risalendo l'albero. Quando viene aggiornato $f[radice[T]]$, nessun altro nodo dipende da questo nuovo valore, quindi il processo termina. Poiché l'altezza di un albero red-black è $O(\lg n)$, la modifica di un campo f in un nodo richiede un tempo $O(\lg n)$ per aggiornare i nodi che dipendono da tale modifica.

L'inserimento di un nodo x nell'albero T è formato da due fasi (vedere il Paragrafo 13.3). Durante la prima fase, x viene inserito come figlio di un nodo esistente $p[x]$. Il valore di $f[x]$ può essere calcolato nel tempo $O(1)$ perché, per ipotesi, dipende soltanto dalle informazioni negli altri campi dello stesso x e dalle informazioni dei figli di x, ma i figli di x sono entrambi la sentinella $nil[T]$. Una volta calcolato $f[x]$, la modifica si propaga verso l'alto nell'albero. Quindi, il tempo totale per la prima fase dell'inserimento è $O(\lg n)$. Durante la seconda fase, le uniche modifiche strutturali dell'albero derivano dalle rotazioni. Poiché in una rotazione cambiano soltanto due nodi, il tempo totale per aggiornare i campi f è $O(\lg n)$ per rotazione. Dal momento che in un inserimento ci sono al massimo due rotazioni, il tempo totale per l'inserimento è $O(\lg n)$.

Come l'inserimento, anche la cancellazione si svolge in due fasi (vedere il Paragrafo 13.4). Nella prima fase, le modifiche dell'albero si verificano se il nodo cancellato è sostituito con il suo successore e quando il nodo cancellato o il suo successore viene effettivamente rimosso. La propagazione degli aggiornamenti di f indotti da queste modifiche costa al massimo $O(\lg n)$, in quanto le modifiche cambiano localmente l'albero. La sistemazione dell'albero red-black durante la seconda fase richiede al massimo tre rotazioni, ciascuna delle quali impiega al massimo il tempo $O(\lg n)$ per propagare gli aggiornamenti di f. Quindi, come per l'inserimento, il tempo totale per la cancellazione è $O(\lg n)$. ■

In molti casi, come la gestione dei campi $size$ negli alberi di statistiche d'ordine, il costo di aggiornamento dopo una rotazione è $O(1)$, anziché $O(\lg n)$ che abbiamo ottenuto nella dimostrazione del Teorema 14.1 (un esempio è descritto nell'Esercizio 14.2-4).

Esercizi

14.2-1

Spiegate come ciascuna delle operazioni MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR possa essere eseguita nel tempo $O(1)$ nel caso peggiore in un albero di statistiche d'ordine esteso, senza influenzare le prestazioni asintotiche delle altre operazioni (*suggerimento:* aggiungete i puntatori ai nodi).

14.2-2

È possibile gestire le altezze nere dei nodi in un albero red-black come campi dei nodi dell'albero senza influire sulle prestazioni asintotiche delle altre operazioni con l'albero red-black? Spiegate come o perché no.

14.2-3
È possibile gestire in modo efficiente le profondità dei nodi in un albero red-black come campi dei nodi dell'albero? Spiegate come o perché no.

14.2-4 $\star$
Indichiamo con $\otimes$ un operatore binario associativo; sia a un campo aggiuntivo in ciascun nodo di un albero red-black. Supponete di volere includere in ogni nodo x un campo addizionale f tale che $f[x] = a[x_1] \otimes a[x_2] \otimes \cdots \otimes a[x_m]$, dove $x_1, x_2, \ldots, x_m$ è l'elenco dei nodi di un attraversamento simmetrico del sottoalbero con radice in x. Dimostrate che i campi f possono essere appropriatamente aggiornati nel tempo $O(1)$ dopo una rotazione. Modificate un po' il vostro ragionamento per dimostrare che i campi *size* negli alberi di statistiche d'ordine possono essere aggiornati nel tempo $O(1)$ in ogni rotazione.

14.2-5 $\star$
Vogliamo estendere gli alberi red-black con un'operazione RB-ENUMERATE(x, a, b) che genera in output tutte le chiavi k tali che $a \leq k \leq b$ in un albero red-black con radice in x. Spiegate come implementare RB-ENUMERATE nel tempo $\Theta(m + \lg n)$, dove m è il numero di chiavi in output e n è il numero di nodi interni dell'albero (*suggerimento:* non occorre aggiungere nuovi campi all'albero red-black).

14.3 Alberi di intervalli

In questo paragrafo estenderemo gli alberi red-black per supportare le operazioni con gli insiemi dinamici di intervalli. Un ***intervallo chiuso*** è una coppia ordinata di numeri reali $[t_1, t_2]$, con $t_1 \leq t_2$. L'intervallo $[t_1, t_2]$ rappresenta l'insieme $\{t \in \mathbf{R} : t_1 \leq t \leq t_2\}$. Gli ***intervalli semiaperti*** e ***aperti*** omettono, rispettivamente, uno o entrambi gli estremi dell'insieme. In questo paragrafo, supporremo che gli intervalli siano chiusi; estendere i risultati agli intervalli aperti e semiaperti è concettualmente semplice.

Gli intervalli sono comodi per rappresentare gli eventi che si svolgono in un periodo continuo di tempo. Per esempio, potrebbe essere necessario interrogare un database di intervalli temporali per trovare quali eventi si sono verificati durante un determinato intervallo di tempo. La struttura dati di questo paragrafo fornisce uno strumento efficiente per gestire tali database di intervalli.

Possiamo rappresentare un intervallo $[t_1, t_2]$ come un oggetto i, con i campi $low[i] = t_1$ (***estremo inferiore***) e $high[i] = t_2$ (***estremo superiore***). Diciamo che gli intervalli i e i' ***si sovrappongono*** se $i \cap i' \neq \emptyset$, ovvero se $low[i] \leq high[i']$ e $low[i'] \leq high[i]$. Due intervalli qualsiasi i e i' soddisfano la ***tricotomia degli intervalli***; ovvero una sola delle seguenti proprietà può essere vera:

a. i e i' si sovrappongono.

b. i è a sinistra di i' (cioè $high[i] < low[i']$).

c. i è a destra di i' (cioè $high[i'] < low[i]$).

La Figura 14.3 illustra le tre possibilità.

Un ***albero di intervalli*** è un albero red-black che gestisce un insieme dinamico di elementi, in cui ogni elemento x contiene un intervallo $int[x]$. Gli alberi di intervalli supportano le seguenti operazioni.

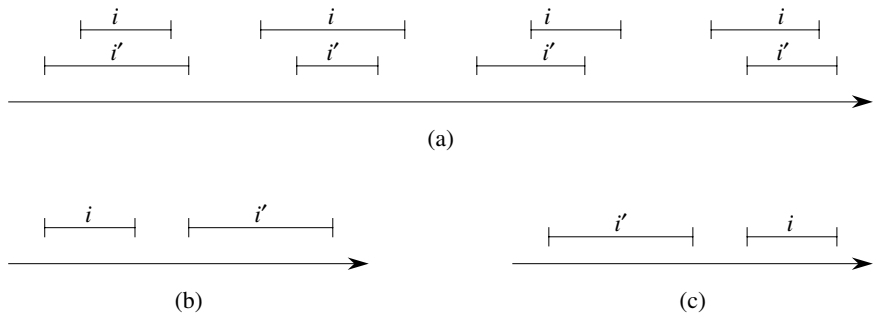


Figura 14.3 Tricotomia degli intervalli per due intervalli chiusi i e i'. **(a)** Se i e i' si sovrappongono, si hanno quattro casi; in ogni caso, $low[i] \leq high[i']$ e $low[i'] \leq high[i]$. **(b)** Gli intervalli non si sovrappongono e $high[i] < low[i']$. **(c)** Gli intervalli non si sovrappongono e $high[i'] < low[i]$.

INTERVAL-INSERT(T, x) aggiunge l'elemento x, il cui campo int si suppone contenga un intervallo, all'albero di intervalli T.

INTERVAL-DELETE(T, x) rimuove l'elemento x dall'albero di intervalli T.

INTERVAL-SEARCH(T, i) restituisce un puntatore a un elemento x nell'albero di intervalli T tale che $int[x]$ si sovrappone all'intervallo i; restituisce la sentinella $nil[T]$ se non esiste tale elemento nell'insieme.

La Figura 14.4 illustra come un albero di intervalli rappresenta un insieme di intervalli. Seguiremo i quattro passaggi del metodo descritto nel Paragrafo 14.2 mentre analizzeremo il progetto di un albero di intervalli e le operazioni che vengono svolte su di esso.

Passaggio 1: struttura dati di base

Scegliamo un albero red-black in cui ogni nodo x contiene un intervallo $int[x]$ e la chiave di x è l'estremo inferiore, $low[int[x]]$, dell'intervallo. Quindi, un attraversamento simmetrico della struttura dati elenca ordinatamente gli intervalli in funzione dell'estremo inferiore.

Passaggio 2: informazioni aggiuntive

Oltre agli intervalli, ogni nodo x contiene un valore $max[x]$, che è il massimo tra tutti gli estremi degli intervalli memorizzati nel sottoalbero con radice in x.

Passaggio 3: gestione delle informazioni

Dobbiamo verificare che le operazioni di inserimento e cancellazione possono essere svolte nel tempo $O(\lg n)$ in un albero di intervalli di n nodi. Se conosciamo l'intervallo $int[x]$ e i valori max dei figli del nodo x, possiamo determinare $max[x]$:

$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$$

Per il Teorema 14.1, le operazioni di inserimento e cancellazione vengono eseguite nel tempo $O(\lg n)$. Infatti, l'aggiornamento dei campi max dopo una rotazione può essere eseguito nel tempo $O(1)$, come dimostrano gli Esercizi 14.2-4 e 14.3-1.

Passaggio 4: sviluppare le nuove operazioni

L'unica operazione nuova da implementare è INTERVAL-SEARCH(T, i), che trova un nodo nell'albero T il cui intervallo si sovrappone all'intervallo i. Se non c'è

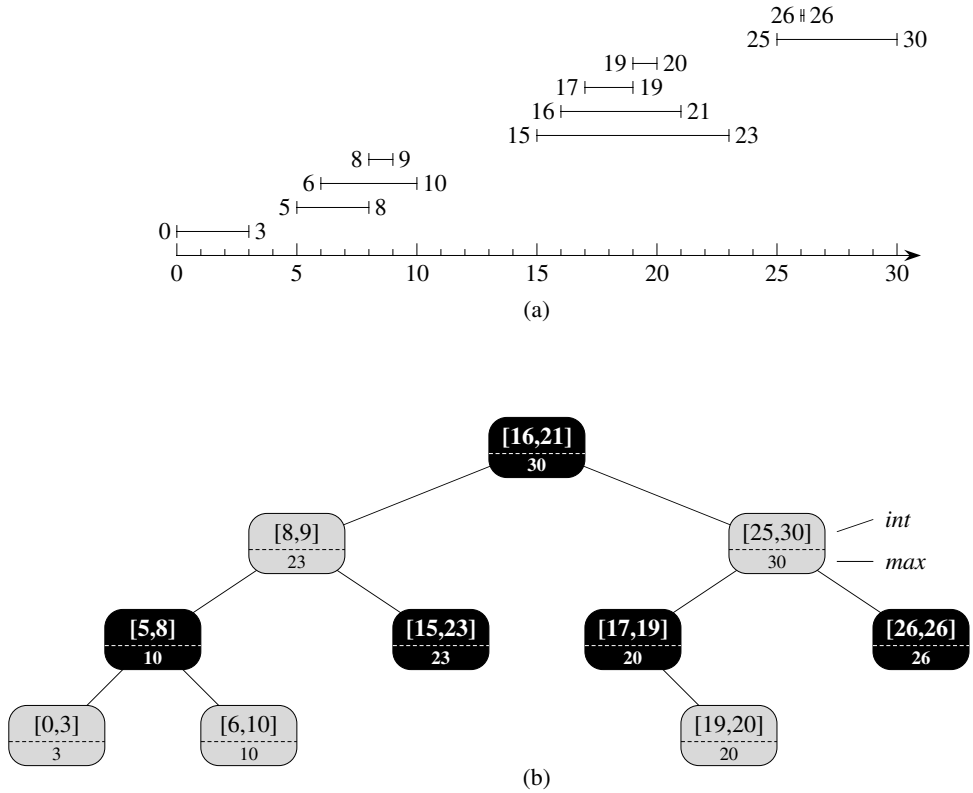


Figura 14.4 Un albero di intervalli. **(a)** Un insieme di 10 intervalli, ordinati dal basso verso l'alto in funzione dell'estremo sinistro degli intervalli. **(b)** L'albero di intervalli che rappresenta tale insieme di intervalli. Un attraversamento simmetrico dell'albero elenca ordinatamente i nodi in funzione dell'estremo sinistro.

un intervallo che si sovrappone a i nell'albero, viene restituito un puntatore alla sentinella $nil[T]$.

INTERVAL-SEARCH(T, i)

```
1  x ← radice[T]
2  while x ≠ nil[T] e i non si sovrappone a int[x]
3        do if left[x] ≠ nil[T] e max[left[x]] ≥ low[i]
4              then x ← left[x]
5              else  x ← right[x]
6  return x
```

La ricerca di un intervallo che si sovrappone a i inizia con x nella radice dell'albero e prosegue verso il basso. Termina quando viene trovato un intervallo che si sovrappone a i o quando x punta alla sentinella $nil[T]$. Poiché ogni iterazione del ciclo di base impiega il tempo $O(1)$ e poiché l'altezza di un albero red-black di n nodi è $O(\lg n)$, la procedura INTERVAL-SEARCH impiega il tempo $O(\lg n)$.

Prima di spiegare perché l'operazione INTERVAL-SEARCH è corretta, analizziamo il suo funzionamento con l'albero di intervalli illustrato nella Figura 14.4. Supponiamo di volere trovare un intervallo che si sovrappone all'intervallo $i = [22, 25]$. Iniziamo con x nella radice, che contiene $[16, 21]$ e non si sovrap-

pone a i. Poiché $max[left[x]] = 23$ è maggiore di $low[i] = 22$, il ciclo continua con x come figlio sinistro della radice – il nodo che contiene $[8, 9]$; anche questo intervallo non si sovrappone a i. Stavolta $max[left[x]] = 10$ è minore di $low[i] = 22$, quindi il ciclo continua con il figlio destro di x come il nuovo x. L'intervallo $[15, 23]$ memorizzato in questo nodo si sovrappone a i, quindi la procedura restituisce questo nodo.

Come esempio di ricerca senza successo, supponiamo di trovare un intervallo che si sovrappone a $i = [11, 14]$ nell'albero di intervalli illustrato nella Figura 14.4. Iniziamo ancora una volta con x come radice. Poiché l'intervallo $[16, 21]$ della radice non si sovrappone a i e poiché $max[left[x]] = 23$ è maggiore di $low[i] = 11$, andiamo a sinistra nel nodo che contiene $[8, 9]$. L'intervallo $[8, 9]$ non si sovrappone a i e $max[left[x]] = 10$ è minore di $low[i] = 11$, quindi andiamo a destra (notate che nessun intervallo nel sottoalbero sinistro si sovrappone a i). L'intervallo $[15, 23]$ non si sovrappone a i e suo figlio sinistro è $nil[T]$, quindi andiamo a destra, il ciclo termina e viene restituita la sentinella $nil[T]$.

Per spiegare perché la procedura INTERVAL-SEARCH è corretta, bisogna capire perché è sufficiente esaminare un solo percorso dalla radice. Il concetto di base è che in qualsiasi nodo x, se $int[x]$ non si sovrappone a i, la ricerca procede sempre in una direzione sicura: alla fine sarà trovato un intervallo che si sovrappone a quello dato, se ce n'è uno nell'albero. Il seguente teorema definisce in maniera più precisa questa proprietà.

Teorema 14.2
La procedura INTERVAL-SEARCH(T, i) restituisce un nodo il cui intervallo si sovrappone a i oppure restituisce $nil[T]$ se l'albero T non contiene alcun nodo il cui intervallo si sovrappone a i.

Dimostrazione Il ciclo **while** (righe 2–5) termina quando $x = nil[T]$ o quando i si sovrappone a $int[x]$. Nel secondo caso, è certamente corretto restituire x. Quindi, concentriamo la nostra attenzione sul primo caso, in cui il ciclo **while** termina perché $x = nil[T]$.

Utilizziamo la seguente invariante per il ciclo **while** (righe 2–5):

> Se l'albero T contiene un intervallo che si sovrappone a i, allora esiste un intervallo nel sottoalbero con radice in x.

Applichiamo questa invariante di ciclo nel modo seguente:

Inizializzazione: prima della prima iterazione, la riga 1 imposta x come radice di T, quindi l'invariante è vera.

Conservazione: in ogni iterazione del ciclo **while**, viene eseguita la riga 4 o la riga 5. Dimostreremo che l'invariante di ciclo si conserva nei due casi.

Se viene eseguita la riga 5, allora per la condizione di diramazione nella riga 3, abbiamo $left[x] = nil[T]$ o $max[left[x]] < low[i]$. Se $left[x] = nil[T]$, il sottoalbero con radice in $left[x]$ chiaramente non contiene un intervallo che si sovrappone a i; quindi, impostando x a $right[x]$, si conserva l'invariante. Supponete, allora, che $left[x] \neq nil[T]$ e $max[left[x]] < low[i]$. Come illustra la Figura 14.5(a), per ogni intervallo i' nel sottoalbero sinistro di x, abbiamo

$$\begin{aligned} high[i'] &\leq max[left[x]] \\ &< low[i] \end{aligned}$$

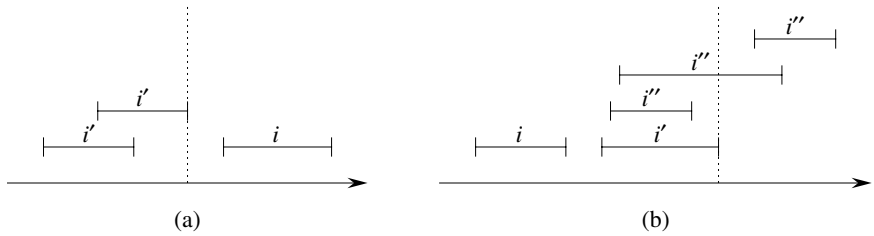


Figura 14.5 Gli intervalli nella dimostrazione del Teorema 14.2. Il valore di $max[left[x]]$ è illustrato in entrambi i casi come una linea tratteggiata. **(a)** La ricerca va a destra. Nessun intervallo i' nel sottoalbero sinistro di x può sovrapporsi a i. **(b)** La ricerca va a sinistra. Il sottoalbero sinistro di x contiene un intervallo che si sovrappone a i (caso non illustrato) oppure c'è un intervallo i' nel sottoalbero sinistro di x tale che $high[i'] = max[left[x]]$. Poiché i non si sovrappone a i', non può sovrapporsi a un intervallo i'' nel sottoalbero destro di x, perché $low[i'] \leq low[i'']$.

Per la tricotomia degli intervalli, i' e i non si sovrappongono. Allora il sottoalbero sinistro di x non contiene intervalli che si sovrappongono a i; quindi, impostando x a $right[x]$, si conserva l'invariante.

Se, invece, viene eseguita la riga 4, allora dimostreremo che è vera la versione contraria dell'invariante di ciclo. Ovvero, se non c'è un intervallo che si sovrappone a i nel sottoalbero con radice in $left[x]$, allora non c'è un intervallo che si sovrappone a i in qualsiasi punto dell'albero. Poiché viene eseguita la riga 4, allora per la condizione di diramazione nella riga 3, abbiamo $max[left[x]] \geq low[i]$. Inoltre, per la definizione del campo max, ci deve essere un intervallo i' nel sottoalbero sinistro di x tale che

$$\begin{aligned} high[i'] &= max[left[x]] \\ &\geq low[i] \end{aligned}$$

La Figura 14.5(b) illustra la situazione. Poiché i e i' non si sovrappongono e poiché non è vero che $high[i'] < low[i]$, allora per la tricotomia degli intervalli si ha $high[i] < low[i']$. Gli alberi di intervalli hanno le chiavi negli estremi inferiori degli intervalli, quindi la proprietà dell'albero di ricerca implica che, per qualsiasi intervallo i'' nel sottoalbero destro di x, si abbia

$$\begin{aligned} high[i] &< low[i'] \\ &\leq low[i''] \end{aligned}$$

Per la tricotomia degli intervalli, i e i'' non si sovrappongono. Concludiamo che, indipendentemente dal fatto che un intervallo nel sottoalbero sinistro di x si sovrapponga oppure no a i, impostando x a $left[x]$, si conserva l'invariante.

Conclusione: se il ciclo termina quando $x = nil[T]$, non c'è un intervallo che si sovrappone a i nel sottoalbero con radice in x. La versione contraria dell'invariante di ciclo implica che T non contiene un intervallo che si sovrappone a i. Quindi è corretto restituire $x = nil[T]$. ■

In conclusione, la procedura INTERVAL-SEARCH funziona correttamente.

Esercizi

14.3-1

Scrivete lo pseudocodice per LEFT-ROTATE che opera sui nodi di un albero di intervalli e aggiorna i campi max nel tempo $O(1)$.

14.3-2

Riscrivete il codice per INTERVAL-SEARCH in modo che funzioni correttamente nell'ipotesi che tutti gli intervalli siano aperti.

14.3-3
Descrivete un algoritmo efficiente che, dato un intervallo i, restituisce un intervallo che si sovrappone a i e che ha l'estremo inferiore minimo, oppure restituisce $nil[T]$, se tale intervallo non esiste.

14.3-4
Dato un albero di intervalli T e un intervallo i, spiegate come tutti gli intervalli in T che si sovrappongono a i possono essere elencati nel tempo $O(\min(n, k \lg n))$, dove k è il numero di intervalli nella lista di output (*facoltativo:* trovate una soluzione che non modifica l'albero).

14.3-5
Descrivete le modifiche da apportare alle procedure degli alberi di intervalli per supportare la nuova operazione INTERVAL-SEARCH-EXACTLY(T, i) che restituisce un puntatore a un nodo x in un albero di intervalli T tale che $low[int[x]] = low[i]$ e $high[int[x]] = high[i]$, oppure $nil[T]$ se T non contiene tale nodo. Tutte le operazioni, inclusa INTERVAL-SEARCH-EXACTLY, dovrebbero essere eseguite nel tempo $O(\lg n)$ in un albero di n nodi.

14.3-6
Spiegate come gestire un insieme dinamico Q di numeri che supporta l'operazione MIN-GAP; questa operazione fornisce il valore della differenza tra i due numeri più vicini in Q. Per esempio, se $Q = \{1, 5, 9, 15, 18, 22\}$, allora MIN-GAP$(Q)$ restituisce $18 - 15 = 3$, in quanto 15 e 18 sono i due numeri più vicini in Q. Le operazioni INSERT, DELETE, SEARCH e MIN-GAP devono essere le più efficienti possibili; analizzate il loro tempi di esecuzione.

14.3-7 $\star$
Tipicamente, i database VLSI rappresentano un circuito integrato come una lista di rettangoli. Supponete che ogni rettangolo abbia i lati paralleli agli assi x e y, in modo che possa essere rappresentato dalle coordinate x e y minime e massime. Create un algoritmo con tempo $O(n \lg n)$ per determinare se un insieme di rettangoli così rappresentati contenga due rettangoli che si sovrappongono. Il vostro algoritmo non deve elencare tutte le coppie di rettangoli che si intersecano, ma deve indicare che esiste una sovrapposizione se un rettangolo si sovrappone interamente a un altro, anche se le linee di contorno non si intersecano (*suggerimento:* fate scorrere una retta di "scansione" attraverso l'insieme dei rettangoli).

Problemi

14-1 Punto di massima sovrapposizione
Supponete di voler trovare un ***punto di massima sovrapposizione*** in un insieme di intervalli – un punto dove si sovrappone il maggior numero di intervalli del database.

a. Dimostrate che c'è sempre un punto di massima sovrapposizione che è un estremo di uno dei segmenti.

b. Progettate una struttura dati che supporta efficientemente le operazioni INTERVAL-INSERT, INTERVAL-DELETE e FIND-POM; quest'ultima operazione restituisce un punto di massima sovrapposizione. (*Suggerimento:* create

un albero red-black con tutti gli estremi. Associate il valore $+1$ a ogni estremo sinistro e il valore -1 a ogni estremo destro. Estendete i nodi dell'albero con un'informazione aggiuntiva per gestire il punto di massima sovrapposizione.)

14-2 La permutazione di Josephus

Il ***problema di Josephus*** è definito nel modo seguente. Supponete che n persone siamo disposte in cerchio; sia dato un numero intero positivo $m \leq n$. Iniziando da una persona, si procede intorno al cerchio allontanando ogni m-esima persona. Dopo avere allontanato una persona, il conteggio prosegue con le persone rimaste. Il processo continua finché non saranno allontanate tutte le n persone. L'ordine in cui le persone vengono allontanate dal cerchio definisce la (n, m)***-permutazione di Josephus*** degli interi $1, 2, \ldots, n$. Per esempio, la $(7, 3)$-permutazione di Josephus è $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Supponete che m sia una costante. Descrivete un algoritmo con tempo $O(n)$ che, dato un intero n, generi come output la (n, m)-permutazione di Josephus.

b. Supponete che m non sia una costante. Descrivete un algoritmo con tempo $O(n \lg n)$ che, dati gli interi n e m, generi in output la (n, m)-permutazione di Josephus.

Note

Nel loro libro, Preparata e Shamos [247] descrivono diversi alberi di intervalli che si trovano nella letteratura, facendo riferimento agli studi di H. Edelsbrunner (1980) ed E. M. McCreight (1981). Il libro esamina dettagliatamente un albero di intervalli per il quale, dato un database statico di n intervalli, tutti i k intervalli che si sovrappongono a un dato intervallo possono essere enumerati nel tempo $O(k + \lg n)$.

IV Progettazione avanzata e tecniche di analisi

Introduzione

Questa parte tratta tre importanti tecniche per la progettazione e l'analisi di algoritmi efficienti: la programmazione dinamica (Capitolo 15), gli algoritmi greedy (Capitolo 16) e l'analisi ammortizzata (Capitolo 17). Le parti precedenti di questo libro hanno presentato altre tecniche molto diffuse, come il metodo divide et impera, la randomizzazione e la soluzione delle ricorrenze. Le nuove tecniche presentate in questa parte sono più sofisticate, ma utili per affrontare e risolvere in modo efficiente molti problemi computazionali. I temi trattati in questa parte ricorreranno nelle parti successive del libro.

Tipicamente, la programmazione dinamica viene applicata per risolvere problemi di ottimizzazione, nei quali una soluzione ottima è il frutto di una serie di scelte. Una volta fatte le scelte, spesso si presentano sottoproblemi dello stesso tipo. La programmazione dinamica è efficace quando lo stesso sottoproblema può scaturire da più serie di scelte; il concetto chiave della programmazione dinamica consiste nel memorizzare le soluzioni di vari sottoproblemi per poterle applicare nel caso in cui si dovesse ripresentare uno di questi sottoproblemi. Il Capitolo 15 spiega come questo semplice concetto, a volte, consente di trasformare gli algoritmi con tempi esponenziali in algoritmi con tempi polinomiali.

Analogamente agli algoritmi della programmazione dinamica, gli algoritmi greedy tipicamente sono utilizzati per risolvere problemi di ottimizzazione nei quali occorre fare una serie di scelte per arrivare a una soluzione ottima. L'idea che sta alla base di un algoritmo greedy è quella di effettuare le singole scelte in modo che siano localmente ottime. Un semplice esempio è il problema del resto in monete: per minimizzare il numero di monete necessarie per formare un determinato resto, è sufficiente selezionare ripetutamente la moneta di taglio più grande che non supera l'importo ancora dovuto. Ci sono vari problemi come questo per i quali un algoritmo greedy fornisce una soluzione ottima più rapidamente di quanto sarebbe possibile con un metodo di programmazione dinamica. Tuttavia, non sempre è facile dire se una tecnica greedy sarà efficace. Il Capitolo 16 introduce la teoria dei matroidi, che può essere utile in molti casi per valutare la qualità di una tecnica greedy.

L'analisi ammortizzata è uno strumento per analizzare gli algoritmi che svolgono una sequenza di operazioni simili. Anziché limitare il costo della sequenza delle operazioni limitando il costo effettivo di ciascuna operazione separatamente,

l'analisi ammortizzata può essere utilizzata per fornire un limite del costo effettivo dell'intera sequenza di operazioni. Un motivo per cui questa idea può risultare efficace è che è poco probabile che, in una sequenza di operazioni, ogni singola operazione venga eseguita nel tempo del suo caso peggiore. Alcune operazioni potranno essere costose, ma molte altre saranno più economiche. L'analisi ammortizzata, tuttavia, non è un semplice strumento di analisi; essa permette anche di ragionare sul progetto degli algoritmi, in quanto la progettazione di un algoritmo e l'analisi del suo tempo di esecuzione sono strettamente interconnesse. Il Capitolo 17 introduce tre metodi per svolgere l'analisi ammortizzata degli algoritmi.

Programmazione dinamica 15

La programmazione dinamica, come il metodo divide et impera, risolve i problemi combinando le soluzioni dei sottoproblemi (in questo contesto, con il termine "programmazione" facciamo riferimento a un metodo tabulare, non alla scrittura del codice per un calcolatore). Come detto nel Capitolo 2, gli algoritmi divide et impera suddividono un problema in sottoproblemi indipendenti, risolvono in modo ricorsivo i sottoproblemi e, poi, combinano le loro soluzioni per risolvere il problema originale. La programmazione dinamica, invece, può essere applicata quando i sottoproblemi non sono indipendenti, ovvero quando i sottoproblemi hanno in comune dei sottosottoproblemi. In questo contesto, un algoritmo divide et impera svolge molto più lavoro del necessario, risolvendo ripetutamente i sottosottoproblemi comuni. Un algoritmo di programmazione dinamica risolve ciascun sottosottoproblema una sola volta e salva la sua soluzione in una tabella, evitando così il lavoro di ricalcolare la soluzione ogni volta che si presenta il sottosottoproblema.

La programmazione dinamica, tipicamente, si applica ai ***problemi di ottimizzazione***. Per questi problemi ci possono essere molte soluzioni possibili. Ogni soluzione ha un valore e si vuole trovare una soluzione con il valore ottimo (minimo o massimo). Precisiamo che abbiamo detto *una* soluzione ottima del problema, non *la* soluzione ottima, perché ci possono essere varie soluzioni che permettono di ottenere il valore ottimo.

Il processo di sviluppo di un algoritmo di programmazione dinamica può essere suddiviso in una sequenza di quattro fasi.

1. Caratterizzare la struttura di una soluzione ottima.
2. Definire in modo ricorsivo il valore di una soluzione ottima.
3. Calcolare il valore di una soluzione ottima secondo uno schema bottom-up (dal basso verso l'alto).
4. Costruire una soluzione ottima dalle informazioni calcolate.

Le fasi 1–3 formano la base per risolvere un problema applicando la programmazione dinamica. La fase 4 può essere omessa se è richiesto soltanto il valore di una soluzione ottima. Quando eseguiamo la fase 4, a volte inseriamo delle informazioni aggiuntive durante il calcolo della fase 3 per semplificare la costruzione di una soluzione ottima.

I prossimi paragrafi applicano il metodo della programmazione dinamica per risolvere alcuni problemi di ottimizzazione. Il Paragrafo 15.1 esamina un problema nella programmazione di due linee di assemblaggio di automobili, dove dopo ogni stazione, l'auto in costruzione può restare nella stessa linea o spostarsi nell'altra. Il Paragrafo 15.2 spiega come moltiplicare una sequenza di matrici in modo da svolgere complessivamente il minor numero di prodotti scalari. Dopo questi esempi

di programmazione dinamica, il Paragrafo 15.3 descrive due caratteristiche chiave che un problema deve avere per essere risolto con la tecnica della programmazione dinamica. Il Paragrafo 15.4 spiega come trovare la più lunga sottosequenza comune a due sequenze. Infine, il Paragrafo 15.5 applica la programmazione dinamica per costruire alberi binari di ricerca che hanno prestazioni ottime per una determinata distribuzione di chiavi.

15.1 Programmazione delle linee di assemblaggio

Il primo esempio di programmazione dinamica risolve un problema di produzione. La Colonel Motors Corporation produce automobili in uno stabilimento che ha due linee di assemblaggio, illustrate nella Figura 15.1. Lo chassis di un'automobile entra in una linea di assemblaggio, riceve nuovi componenti in un certo numero di stazioni e, infine, l'auto completa esce dalla linea di assemblaggio. Ogni linea di assemblaggio ha n stazioni, numerate con $j = 1, 2, \ldots, n$. Indichiamo con $S_{i,j}$ la j-esima stazione nella linea i (dove i è 1 o 2). La j-esima stazione nella linea 1 ($S_{1,j}$) svolge la stessa funzione della j-esima stazione nella linea 2 ($S_{2,j}$). Le stazioni, però, sono state costruite in periodi diversi e con tecnologie differenti, quindi il tempo richiesto in ogni stazione varia, anche fra stazioni che occupano la stessa posizione nelle due linee differenti. Indichiamo con $a_{i,j}$ il tempo di assemblaggio richiesto nella stazione $S_{i,j}$. Come illustra la Figura 15.1, uno chassis entra nella stazione 1 di una linea di assemblaggio e passa da una stazione alla successiva. C'è anche un tempo di ingresso e_i per lo chassis che entra nella linea di assemblaggio i e un tempo di uscita x_i per l'auto completa che esce dalla linea di assemblaggio i.

Di norma, una volta che uno chassis entra in una linea di assemblaggio, resta in questa linea. Il tempo per passare da una stazione alla successiva all'interno della stessa linea di assemblaggio è trascurabile. Ogni tanto, arriva un ordine urgente di un cliente che vuole l'automobile il più presto possibile. Per un ordine urgente, lo chassis attraversa ancora le n stazioni in sequenza, ma il responsabile della produzione può passare un'auto non ancora completata da una linea di assemblaggio all'altra, all'uscita da una stazione qualsiasi. Il tempo per trasferire uno chassis da una linea di assemblaggio i, dopo avere attraversato la stazione $S_{i,j}$ è $t_{i,j}$, dove $i = 1, 2$ e $j = 1, 2, \ldots, n-1$ (perché dopo l'n-esima stazione, l'assemblaggio è completo). Il problema è determinare quali stazioni scegliere dalla linea 1 e dalla linea 2 per minimizzare il tempo totale per completare l'assemblaggio di un'auto. Nell'esempio della Figura 15.2(a), il tempo più breve si ottiene scegliendo le stazioni 1, 3 e 6 dalla linea 1 e le stazioni 2, 4 e 5 dalla linea 2.

Non è possibile applicare la tecnica ovvia a "forza bruta" per minimizzare il tempo di attraversamento dello stabilimento quando ci sono molte stazioni. Se avessimo l'elenco delle stazioni da utilizzare nella linea 1 e nella linea 2, sarebbe semplice calcolare nel tempo $\Theta(n)$ quanto tempo impiega uno chassis per attraversare lo stabilimento. Purtroppo, ci sono 2^n possibili modi di scegliere le stazioni, che possiamo determinare considerando l'insieme delle stazioni utilizzate nella linea 1 come un sottoinsieme di $\{1, 2, \ldots, n\}$ e notando che ci sono 2^n di tali sottoinsiemi. Quindi, per determinare il percorso più rapido per attraversare lo stabilimento, elencando tutti i modi possibili di scegliere le stazioni e calcolando il tempo richiesto da ciascuno di essi, occorrerebbe il tempo $\Omega(2^n)$, che sarebbe inaccettabile con n grande.

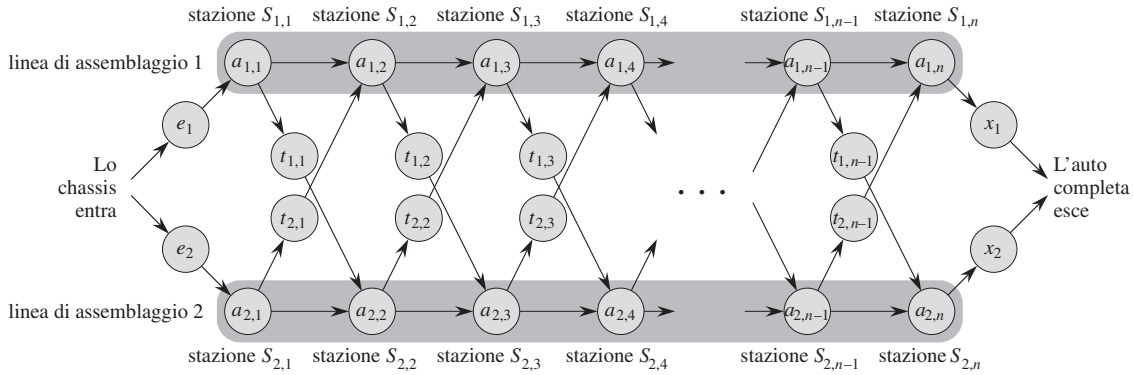


Figura 15.1 Un problema di produzione: trovare il percorso più rapido all'interno di uno stabilimento. Ci sono due linee di assemblaggio, ciascuna con n stazioni; la j-esima stazione nella linea i è indicata con $S_{i,j}$ e il tempo di assemblaggio in quella stazione è $a_{i,j}$. Lo chassis di un'automobile entra nello stabilimento e va nella linea i (dove $i = 1$ o 2), impiegando il tempo e_i. Dopo avere attraversato la j-esima stazione in una linea, lo chassis passa alla $(j+1)$-esima stazione in una delle due linee. Non c'è un costo di trasferimento se lo chassis resta nella stessa linea, altrimenti occorre un tempo $t_{i,j}$ per passare nell'altra linea dopo la stazione $S_{i,j}$. Una volta che l'auto è uscita dalla n-esima stazione di una linea, occorre il tempo x_i per essere completata e uscire dallo stabilimento. Il problema è determinare quali stazioni scegliere nella linea 1 e nella linea 2 per minimizzare il tempo totale impiegato da un'auto per attraversare lo stabilimento.

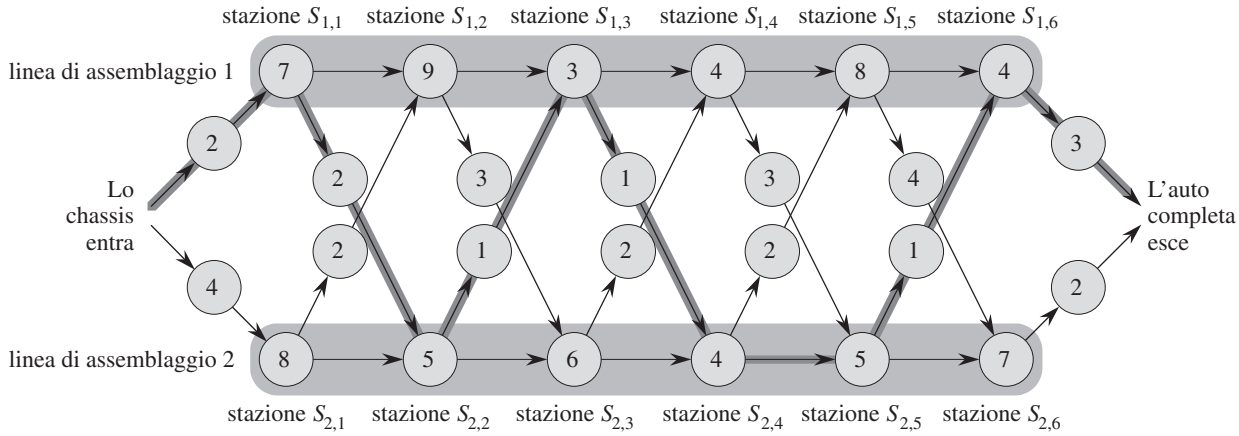
Fase 1: la struttura del percorso più rapido

La prima fase del paradigma della programmazione dinamica consiste nel caratterizzare la struttura di una soluzione ottima. Per il problema della programmazione delle linee di assemblaggio, possiamo svolgere tale compito nel modo seguente. Consideriamo il percorso più veloce possibile che può seguire uno chassis dal punto iniziale fino alla stazione $S_{1,j}$. Se $j = 1$, c'è un solo percorso che lo chassis può seguire; quindi, è facile determinare quanto tempo impiega per arrivare alla stazione $S_{1,j}$. Per $j = 2, 3, \ldots, n$, invece, ci sono due scelte: lo chassis può provenire dalla stazione $S_{1,j-1}$ e poi arrivare direttamente alla stazione $S_{1,j}$; il tempo per andare dalla stazione $j-1$ alla stazione j nella stessa linea di assemblaggio è trascurabile. In alternativa, lo chassis può provenire dalla stazione $S_{2,j-1}$ e, poi, essere trasferito alla stazione $S_{1,j}$; il tempo di trasferimento è $t_{2,j-1}$. Analizziamo queste due possibilità separatamente, sebbene abbiano molti punti in comune.

In primo luogo, supponiamo che il percorso più rapido per arrivare alla stazione $S_{1,j}$ passi per la stazione $S_{1,j-1}$. L'osservazione chiave è che lo chassis deve avere seguito il percorso più rapido dal punto iniziale fino alla stazione $S_{1,j-1}$. Perché? Se ci fosse un percorso più rapido per raggiungere la stazione $S_{1,j-1}$, potremmo seguire questo percorso per ottenere un percorso più rapido fino alla stazione $S_{1,j}$: una contraddizione.

Analogamente, supponiamo ora che il percorso più rapido per arrivare alla stazione $S_{1,j}$ passi per la stazione $S_{2,j-1}$. Notiamo adesso che lo chassis deve avere seguito il percorso più rapido dal punto iniziale fino alla stazione $S_{2,j-1}$. Il ragionamento è lo stesso: se ci fosse un percorso più rapido per raggiungere la stazione $S_{2,j-1}$, potremmo seguire questo percorso per ottenere un percorso più rapido fino alla stazione $S_{1,j}$, che sarebbe una contraddizione.

Più in generale, possiamo dire che per la programmazione delle linee di assemblaggio, una soluzione ottima di un problema (trovare il percorso più rapido per



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Figura 15.2 **(a)** Un'istanza del problema delle linee di assemblaggio con i costi indicati e_i, $a_{i,j}$, $t_{i,j}$ e x_i. Il percorso più rapido per attraversare lo stabilimento è messo in evidenza da uno sfondo grigio scuro. **(b)** I valori di $f_i[j]$, f^*, $l_i[j]$ e l^* per l'istanza illustrata in (a).

arrivare alla stazione $S_{i,j}$) contiene al suo interno una soluzione ottima di sottoproblemi (trovare il percorso più rapido per raggiungere $S_{1,j-1}$ o $S_{2,j-1}$). Faremo riferimento a questa proprietà con il termine ***sottostruttura ottima***, che è una delle caratteristiche peculiari dell'applicabilità della programmazione dinamica, come vedremo nel Paragrafo 15.3.

Utilizziamo la sottostruttura ottima per dimostrare che possiamo costruire una soluzione ottima di un problema dalle soluzioni ottime dei sottoproblemi. Per la programmazione delle linee di assemblaggio, facciamo il seguente ragionamento. Se consideriamo il percorso più rapido per arrivare alla stazione $S_{1,j}$, esso deve passare per la stazione $j-1$ nella linea 1 o nella linea 2. Quindi, il percorso più rapido per arrivare alla stazione $S_{1,j}$ può essere

- il percorso più rapido per raggiungere la stazione $S_{1,j-1}$ e da qui arrivare direttamente alla stazione $S_{1,j}$, oppure
- il percorso più rapido per raggiungere la stazione $S_{2,j-1}$, un trasferimento dalla linea 2 alla linea 1 per poi arrivare stazione $S_{1,j}$.

Seguendo un ragionamento simmetrico, il percorso più rapido per arrivare alla stazione $S_{2,j}$ può essere

- il percorso più rapido per raggiungere la stazione $S_{2,j-1}$ e da qui arrivare direttamente alla stazione $S_{2,j}$, oppure
- il percorso più rapido per raggiungere la stazione $S_{1,j-1}$, un trasferimento dalla linea 1 alla linea 2 per poi arrivare stazione $S_{2,j}$.

Per risolvere il problema di trovare il percorso più rapido per arrivare alla stazione j in una delle linee, risolviamo i sottoproblemi di trovare i percorsi più rapidi fino alla stazione $j-1$ in entrambe le linee.

Quindi, possiamo costruire una soluzione ottima di un'istanza del problema della programmazione delle linee di assemblaggio sfruttando le soluzioni ottime dei sottoproblemi.

Fase 2: una soluzione ricorsiva

La seconda fase del paradigma della programmazione dinamica consiste nel definire il valore di una soluzione ottima ricorsivamente in funzione delle soluzioni ottime dei sottoproblemi. Per il problema della programmazione delle linee di assemblaggio, scegliamo come nostri sottoproblemi il problema di trovare il percorso più rapido fino alla stazione j in entrambe le linee di assemblaggio, con $j = 1, 2, \ldots, n$. Indichiamo con $f_i[j]$ il tempo più piccolo possibile che impiega uno chassis dal punto iniziale fino alla stazione $S_{i,j}$.

L'obiettivo finale è determinare il tempo minimo che impiega uno chassis per attraversare lo stabilimento; indichiamo con f^* questo tempo. Lo chassis deve arrivare fino alla stazione n nella linea 1 o nella linea 2 e poi uscire dallo stabilimento. Poiché il più rapido di questi percorsi è il percorso più rapido per l'intero stabilimento, abbiamo

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2) \tag{15.1}$$

È anche facile ragionare su $f_1[1]$ e $f_2[1]$. Per raggiungere la stazione 1 in una linea, uno chassis va direttamente in questa stazione; quindi

$$f_1[1] = e_1 + a_{1,1} \tag{15.2}$$
$$f_2[1] = e_2 + a_{2,1} \tag{15.3}$$

Vediamo adesso come calcolare $f_i[j]$ per $j = 2, 3, \ldots, n$ (e $i = 1, 2$). Concentrandoci su $f_1[j]$, ricordiamo che il percorso più rapido per raggiungere la stazione $S_{1,j}$ può essere il percorso più rapido fino alla stazione $S_{1,j-1}$ e da qui direttamente alla stazione $S_{1,j}$, oppure il percorso più rapido per raggiungere la stazione $S_{2,j-1}$, un trasferimento dalla linea 2 alla linea 1 per poi arrivare alla stazione $S_{1,j}$. Nel primo caso, abbiamo $f_1[j] = f_1[j-1] + a_{1,j}$ e, nel secondo caso, $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$. Quindi

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) \tag{15.4}$$

per $j = 2, 3, \ldots, n$. Simmetricamente, si ha

$$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) \tag{15.5}$$

per $j = 2, 3, \ldots, n$. Combinando le equazioni (15.2)–(15.5), otteniamo le seguenti equazioni ricorsive

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{se } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{se } j \geq 2 \end{cases} \tag{15.6}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{se } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{se } j \geq 2 \end{cases} \tag{15.7}$$

La Figura 15.2(b) illustra i valori $f_i[j]$ per l'esempio della parte (a), ottenuti con le equazioni (15.6) e (15.7), insieme con il valore di f^*.

I valori di $f_i[j]$ sono i valori delle soluzioni ottime dei sottoproblemi. Per seguire più facilmente il processo di costruzione di una soluzione ottima, definiamo $l_i[j]$ come il numero della linea di assemblaggio, 1 o 2, la cui stazione $j-1$ è utilizzata nel percorso più rapido per arrivare alla stazione $S_{i,j}$. Qui, $i = 1, 2$ e $j = 2, 3, \ldots, n$ (evitiamo di definire $l_i[1]$ perché nessuna stazione precede la stazione 1 nelle due linee di assemblaggio). Definiamo l^* come la linea la cui stazione n è utilizzata nel percorso più rapido dell'intero stabilimento.

I valori di $l_i[j]$ ci aiutano a tracciare il percorso più rapido. Utilizzando i valori di l^* e $l_i[j]$ illustrati nella Figura 15.2(b), possiamo tracciare il percorso più rapido illustrato nella parte (a) nel modo seguente: partendo da $l^* = 1$, utilizziamo la stazione $S_{1,6}$; adesso guardiamo $l_1[6]$, che è 2, quindi utilizziamo la stazione $S_{2,5}$; continuando, guardiamo $l_2[5] = 2$ (utilizziamo la stazione $S_{2,4}$), $l_2[4] = 1$ (stazione $S_{1,3}$), $l_1[3] = 2$ (stazione $S_{2,2}$) e $l_2[2] = 1$ (stazione $S_{1,1}$).

Fase 3: calcolo dei tempi minimi

A questo punto, dovrebbe essere semplice scrivere un algoritmo ricorsivo basato sull'equazione (15.1) e le ricorrenze (15.6) e (15.7) per calcolare il percorso più rapido che attraversa lo stabilimento. C'è un problema con questo algoritmo ricorsivo: il suo tempo di esecuzione è esponenziale in n. Per capire perché, indichiamo con $r_i(j)$ il numero di riferimenti fatti a $f_i[j]$ in un algoritmo ricorsivo. Dall'equazione (15.1), abbiamo

$$r_1(n) = r_2(n) = 1 \tag{15.8}$$

Dalle ricorrenze (15.6) e (15.7), abbiamo

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1) \tag{15.9}$$

per $j = 1, 2, \ldots, n-1$. Come chiede di dimostrare l'Esercizio 15.1-2, $r_i(j) = 2^{n-j}$. Quindi, il solo $f_1[1]$ ha 2^{n-1} riferimenti! Come chiede di dimostrare l'Esercizio 15.1-3, il numero totale di riferimenti a tutti i valori di $f_i[j]$ è $\Theta(2^n)$.

Possiamo fare di meglio se calcoliamo i valori di $f_i[j]$ in un ordine diverso da quello ricorsivo. Osservate che per $j \geq 2$, ogni valore di $f_i[j]$ dipende soltanto dai valori di $f_1[j-1]$ e $f_2[j-1]$. Calcolando i valori di $f_i[j]$ in funzione dei numeri *crescenti* delle stazioni j – da sinistra a destra nella Figura 15.2(b) – possiamo calcolare, nel tempo $\Theta(n)$, il percorso più rapido che attraversa lo stabilimento e il tempo richiesto. La procedura FASTEST-WAY riceve come input i valori $a_{i,j}$, $t_{i,j}$, e_i e x_i, come pure n, il numero di stazioni in ciascuna linea di assemblaggio.

La procedura FASTEST-WAY opera nel modo seguente. Le righe 1–2 calcolano $f_1[1]$ e $f_2[1]$ utilizzando le equazioni (15.2) e (15.3). Poi il ciclo **for** (righe 3–13) calcola $f_i[j]$ e $l_i[j]$ per $i = 1, 2$ e $j = 2, 3, \ldots, n$. Le righe 4–8 calcolano $f_1[j]$ e $l_1[j]$ utilizzando l'equazione (15.4); le righe 9–13 calcolano $f_2[j]$ e $l_2[j]$ utilizzando l'equazione (15.5). Infine, le righe 14–18 calcolano f^* e l^* utilizzando l'equazione (15.1). Poiché le righe 1–2 e 14–18 impiegano un tempo costante e ciascuna delle $n-1$ iterazioni del ciclo **for** (righe 3–13) impiega un tempo costante, l'intera procedura impiega il tempo $\Theta(n)$.

Il processo di calcolo dei valori di $f_i[j]$ e $l_i[j]$ può essere visto come il riempimento delle posizioni di una tabella. Facendo riferimento alla Figura 15.2(b), riempiamo le tabelle che contengono i valori di $f_i[j]$ e $l_i[j]$ da sinistra a destra (e dall'alto verso il basso all'interno di una colonna). Per riempire una posizione

$f_i[j]$, occorrono i valori di $f_1[j-1]$ e $f_2[j-1]$ e, sapendo che li abbiamo già calcolati e memorizzati, determiniamo questi valori ricercandoli semplicemente nella tabella.

```
FASTEST-WAY(a, t, e, x, n)
 1  f1[1] ← e1 + a1,1
 2  f2[1] ← e2 + a2,1
 3  for j ← 2 to n
 4       do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
 5            then f1[j] ← f1[j − 1] + a1,j
 6                 l1[j] ← 1
 7            else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
 8                 l1[j] ← 2
 9          if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10            then f2[j] ← f2[j − 1] + a2,j
11                 l2[j] ← 2
12            else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                 l2[j] ← 1
14  if f1[n] + x1 ≤ f2[n] + x2
15     then f* ← f1[n] + x1
16          l* ← 1
17     else f* ← f2[n] + x2
18          l* ← 2
```

Fase 4: costruzione del percorso più rapido

Avendo calcolato i valori di $f_i[j]$, f^*, $l_i[j]$ e l^*, dobbiamo costruire la sequenza delle stazioni utilizzate nel percorso più rapido che attraversa lo stabilimento. L'esempio della precedente Figura 15.2 illustra come fare. La seguente procedura elenca le stazioni utilizzate in ordine decrescente. L'Esercizio 15.1-1 chiede di modificare questa procedura per elencare le stazioni in ordine crescente.

```
PRINT-STATIONS(l, n)
1  i ← l*
2  stampa "linea " i ", stazione " n
3  for j ← n downto 2
4       do i ← li[j]
5          stampa "linea " i ", stazione " j − 1
```

Per l'esempio della Figura 15.2, PRINT-STATIONS produrrebbe il seguente output

```
linea 1, stazione 6
linea 2, stazione 5
linea 2, stazione 4
linea 1, stazione 3
linea 2, stazione 2
linea 1, stazione 1
```

Esercizi

15.1-1

Spiegate come modificare la procedura PRINT-STATIONS per visualizzare le stazioni in ordine crescente (*suggerimento:* applicate la ricorsione).

15.1-2
Utilizzate le equazioni (15.8) e (15.9) e il metodo di sostituzione per dimostrare che $r_i(j)$, il numero di riferimenti fatti a $f_i[j]$ in un algoritmo ricorsivo, è uguale a 2^{n-j}.

15.1-3
Utilizzando il risultato dell'Esercizio 15.1-2, dimostrate che il numero totale di riferimenti a tutti i valori $f_i[j]$, o $\sum_{i=1}^{2}\sum_{j=1}^{n} r_i(j)$, è esattamente $2^{n+1} - 2$.

15.1-4
Le tabelle con i valori di $f_i[j]$ e $l_i[j]$ contengono, in totale, $4n - 2$ elementi. Spiegate come ridurre lo spazio in memoria a un totale di $2n+2$ elementi, continuando a calcolare f^* e a elencare tutte le stazioni del percorso più rapido che attraversa lo stabilimento.

15.1-5
Il professor Canty ritiene che possano esistere dei valori e_i, $a_{i,j}$ e $t_{i,j}$ per i quali FASTEST-WAY produce i valori $l_i[j]$ tali che $l_1[j] = 2$ e $l_2[j] = 1$ per qualche stazione j. Supponendo che tutti i costi di trasferimento $t_{i,j}$ siano non negativi, dimostrate che il professore si sbaglia.

15.2 Moltiplicare di una sequenza di matrici

Il prossimo esempio di programmazione dinamica è un algoritmo che risolve il problema della moltiplicazione di una sequenza di matrici. Data una sequenza (catena) di n matrici $\langle A_1, A_2, \ldots, A_n\rangle$, vogliamo calcolare il prodotto

$$A_1 A_2 \cdots A_n \tag{15.10}$$

Possiamo calcolare l'espressione (15.10) utilizzando come subroutine l'algoritmo standard per moltiplicare una coppia di matrici, dopo che abbiamo posto le opportune parentesi per eliminare qualsiasi ambiguità sul modo in cui devono essere moltiplicate le matrici. Un prodotto di matrici è ***completamente parentesizzato*** se include matrici singole e prodotti di coppie di matrici, tutti racchiusi fra parentesi. La moltiplicazione delle matrici è associativa, quindi tutte le parentesizzazioni forniscono lo stesso prodotto. Per esempio, se la sequenza delle matrici è $\langle A_1, A_2, A_3, A_4\rangle$, il prodotto $A_1A_2A_3A_4$ può essere completamente parentesizzato in cinque modi distinti:

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$

Il modo in cui parentesizziamo una sequenza di matrici può avere un impatto notevole sul costo per calcolare del prodotto. Consideriamo prima il costo per moltiplicare due matrici. L'algoritmo standard è dato dal seguente pseudocodice. Gli attributi *rows* e *columns* sono i numeri delle righe e delle colonne di una matrice.

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "dimensioni non compatibili"
3     else  for i ← 1 to rows[A]
4                do for j ← 1 to columns[B]
5                       do C[i, j] ← 0
6                          for k ← 1 to columns[A]
7                              do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8           return C
```

Possiamo moltiplicare due matrici A e B soltanto se sono ***compatibili***: il numero di colonne di A deve essere uguale al numero di righe di B. Se A è una matrice $p \times q$ e B è una matrice $q \times r$, la matrice risultante C è una matrice $p \times r$. Il tempo per calcolare C è dominato dal numero di prodotti scalari nella riga 7, che è pqr. In seguito esprimeremo i costi in funzione del numero di prodotti scalari.

Per spiegare come il costo per moltiplicare le matrici dipenda dallo schema di parentesizzazione, consideriamo il problema di moltiplicare una sequenza di tre matrici $\langle A_1, A_2, A_3\rangle$. Supponiamo che le dimensioni delle matrici siano, rispettivamente, 10×100, 100×5 e 5×50. Se moltiplichiamo secondo lo schema di parentesizzazione $((A_1A_2)A_3)$, eseguiamo $10 \cdot 100 \cdot 5 = 5000$ prodotti scalari per calcolare la matrice 10×5 risultante dal prodotto delle matrici A_1A_2, più altri $10 \cdot 5 \cdot 50 = 2500$ prodotti scalari per moltiplicare questa matrice per A_3, per un totale di 7500 prodotti scalari. Se, invece, moltiplichiamo secondo lo schema di parentesizzazione $(A_1(A_2A_3))$, eseguiamo $100 \cdot 5 \cdot 50 = 25\,000$ prodotti scalari per calcolare la matrice 100×50 risultante dal prodotto A_2A_3, più altri $10 \cdot 100 \cdot 50 = 50\,000$ prodotti scalari per moltiplicare A_1 per questa matrice, per un totale di 75 000 prodotti scalari. Quindi, il calcolo della moltiplicazione delle matrici è 10 volte più rapido con il primo schema di parentesizzazione.

Il ***problema della moltiplicazione di una sequenza di matrici*** può essere definito in questo modo: data una sequenza di n matrici $\langle A_1, A_2, \ldots, A_n\rangle$, dove la matrice A_i ha dimensioni $p_{i-1} \times p_i$ per $i = 1, 2, \ldots, n$, determinare lo schema di parentesizzazione completa del prodotto $A_1A_2 \cdots A_n$ che minimizza il numero di prodotti scalari.

È importante notare che, nel problema della moltiplicazione di una sequenza di matrici, non vengono effettivamente moltiplicate le matrici. Il nostro obbiettivo è soltanto quello di determinare un ordine di moltiplicazione delle matrici che ha il costo minimo. Tipicamente, il tempo impiegato per determinare quest'ordine ottimo è più che ripagato dal tempo risparmiato successivamente per eseguire effettivamente i prodotti delle matrici (per esempio, eseguire soltanto 7500 prodotti scalari, anziché 75 000).

Contare il numero di parentesizzazioni

Prima di risolvere il problema della moltiplicazione di una sequenza di matrici con la programmazione dinamica, vogliamo dimostrare che controllare tutti i possibili schemi di parentesizzazione non ci consente di ottenere un algoritmo efficiente. Indichiamo con $P(n)$ il numero di parentesizzazioni alternative di una sequenza di n matrici. Quando $n = 1$, c'è una sola matrice e, quindi, un solo schema di parentesizzazione. Quando $n \geq 2$, un prodotto di matrici completa-

mente parentesizzato è il prodotto di due sottoprodotti di matrici completamente parentesizzati e la suddivisione fra i due sottoprodotti può avvenire fra la k-esima e la $(k+1)$-esima matrice per qualsiasi $k = 1, 2, \ldots, n-1$. Quindi, otteniamo la ricorrenza

$$P(n) = \begin{cases} 1 & \text{se } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{se } n \geq 2 \end{cases} \tag{15.11}$$

Il Problema 12-4 richiedeva di dimostrare che la soluzione di una ricorrenza simile è la sequenza dei ***numeri catalani***, che cresce come $\Omega(4^n/n^{3/2})$. Un esercizio più semplice (vedere l'Esercizio 15.2-3) consiste nel dimostrare che la soluzione della ricorrenza (15.11) è $\Omega(2^n)$. Il numero di soluzioni è quindi esponenziale in n; pertanto la tecnica a forza bruta di ricercare tutti i possibili schemi di parentesizzazione è una strategia inadeguata per determinare la parentesizzazione ottima di una sequenza di matrici.

Fase 1: struttura di una parentesizzazione ottima

La prima fase del paradigma della programmazione dinamica consiste nel trovare la sottostruttura ottima e poi utilizzare questa sottostruttura per costruire una soluzione ottima del problema dalle soluzioni ottime dei sottoproblemi. Per il problema della moltiplicazione di una sequenza di matrici, possiamo svolgere tale compito nel modo seguente. Per comodità, adottiamo la notazione $A_{i..j}$, dove $i \leq j$, per la matrice che si ottiene calcolando il prodotto $A_i A_{i+1} \cdots A_j$. Notate che, se il problema non è banale, cioè $i < j$, allora qualsiasi parentesizzazione del prodotto $A_i A_{i+1} \cdots A_j$ deve suddividere il prodotto fra A_k e A_{k+1} per qualche intero k nell'intervallo $i \leq k < j$; ovvero, per qualche valore di k, prima calcoliamo le matrici $A_{i..k}$ e $A_{k+1..j}$ e, poi, le moltiplichiamo per ottenere il prodotto finale $A_{i..j}$. Il costo di questa parentesizzazione è, quindi, il costo per calcolare la matrice $A_{i..k}$, più il costo per calcolare la matrice $A_{k+1..j}$, più il costo per moltiplicare queste due matrici.

La sottostruttura ottima di questo problema è la seguente. Supponiamo che una parentesizzazione ottima di $A_i A_{i+1} \cdots A_j$ suddivida il prodotto fra A_k e A_{k+1}. Allora la parentesizzazione della prima sottosequenza $A_i A_{i+1} \cdots A_k$ ("prefisso") all'interno di questa parentesizzazione ottima di $A_i A_{i+1} \cdots A_j$ deve essere una parentesizzazione ottima di $A_i A_{i+1} \cdots A_k$. Perché? Se ci fosse un modo meno costoso di parentesizzare $A_i A_{i+1} \cdots A_k$, sostituendo questa parentesizzazione in quella ottima di $A_i A_{i+1} \cdots A_j$ otterremmo un'altra parentesizzazione di $A_i A_{i+1} \cdots A_j$ il cui costo sarebbe minore di quella ottima: una contraddizione. Un'osservazione analoga vale per la parentesizzazione della sottosequenza $A_{k+1} A_{k+2} \cdots A_j$ nella parentesizzazione ottima di $A_i A_{i+1} \cdots A_j$: deve essere una parentesizzazione ottima di $A_{k+1} A_{k+2} \cdots A_j$.

Adesso utilizziamo la nostra sottostruttura ottima per dimostrare che possiamo costruire una soluzione ottima del problema dalle soluzioni ottime dei sottoproblemi. Abbiamo visto che qualsiasi soluzione di un'istanza non banale del problema della moltiplicazione di una sequenza di matrici richiede che il prodotto venga suddiviso in due sottoprodotti; inoltre qualsiasi soluzione ottima contiene al suo interno soluzioni ottime delle istanze dei sottoproblemi. Quindi, possiamo costruire una soluzione ottima di un'istanza del problema della moltiplicazione di una sequenza di matrici suddividendo il problema in due sottoproblemi (con

parentesizzazione ottima di $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$), trovando le soluzioni ottime delle istanze dei sottoproblemi e, infine, combinando le soluzioni ottime dei sottoproblemi. Quando cerchiamo il punto esatto in cui suddividere il prodotto, dobbiamo considerare tutti i possibili punti per avere la certezza di avere esaminato il punto ottimo.

Fase 2: una soluzione ricorsiva

Adesso definiamo il costo di una soluzione ottima ricorsivamente in funzione delle soluzioni ottime dei sottoproblemi. Per il problema della moltiplicazione di una sequenza di matrici, scegliamo come nostri sottoproblemi i problemi di determinare il costo minimo di una parentesizzazione di $A_i A_{i+1} \cdots A_j$ per $1 \leq i \leq j \leq n$. Sia $m[i, j]$ il numero minimo di prodotti scalari richiesti per calcolare la matrice $A_{i..j}$; per tutto il problema, il costo del metodo più economico per calcolare $A_{1..n}$ sarà quindi $m[1, n]$.

Possiamo definire $m[i, j]$ ricorsivamente in questo modo. Se $i = j$, il problema è banale; la sequenza è formata da una matrice $A_{i..i} = A_i$, quindi non occorre eseguire alcun prodotto scalare. Allora, $m[i, i] = 0$ per $i = 1, 2, \ldots, n$. Per calcolare $m[i, j]$ quando $i < j$, sfruttiamo la struttura di una soluzione ottima ottenuta nella fase 1. Supponiamo che la parentesizzazione ottima suddivida il prodotto $A_i A_{i+1} \cdots A_j$ fra A_k e A_{k+1}, dove $i \leq k < j$. Quindi, $m[i, j]$ è uguale al costo minimo per calcolare i sottoprodotti $A_{i..k}$ e $A_{k+1..j}$, più il costo per moltiplicare queste due matrici. Ricordando che ogni matrice A_i è $p_{i-1} \times p_i$, il calcolo del prodotto delle matrici $A_{i..k} A_{k+1..j}$ richiede $p_{i-1} p_k p_j$ prodotti scalari. Quindi, otteniamo

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

Questa equazione ricorsiva suppone che sia noto il valore di k, che invece non conosciamo. Notiamo, tuttavia, che ci sono soltanto $j - i$ valori possibili per k, ovvero $k = i, i + 1, \ldots, j - 1$. Poiché la parentesizzazione ottima deve utilizzare uno di questi valori di k, dobbiamo semplicemente controllarli tutti per trovare il migliore. Quindi, la nostra definizione ricorsiva per il costo minimo di parentesizzazione del prodotto $A_i A_{i+1} \cdots A_j$ diventa

$$m[i, j] = \begin{cases} 0 & \text{se } i = j \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{se } i < j \end{cases} \tag{15.12}$$

I valori $m[i, j]$ sono i costi delle soluzioni ottime dei sottoproblemi. Per seguire più facilmente il processo di costruzione di una soluzione ottima, definiamo $s[i, j]$ come un valore di k in cui è possibile suddividere il prodotto $A_i A_{i+1} \cdots A_j$ per ottenere una parentesizzazione ottima. Ovvero, $s[i, j]$ è uguale a un valore k tale che $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Fase 3: calcolo dei costi ottimi

A questo punto, è semplice scrivere un algoritmo ricorsivo basato sulla ricorrenza (15.12) per calcolare il costo minimo $m[1, n]$ del prodotto $A_1 A_2 \cdots A_n$. Tuttavia, come vedremo nel Paragrafo 15.3, questo algoritmo ha un tempo esponenziale, che non è migliore della tecnica a forza bruta di controllare tutti i possibili schemi di parentesizzazione del prodotto.

L'osservazione importante che possiamo fare a questo punto è che abbiamo un numero relativamente piccolo di sottoproblemi: un problema per ogni possibile scelta di i e j, con $1 \leq i \leq j \leq n$, per un totale di $\binom{n}{2} + n = \Theta(n^2)$. Un algoritmo ricorsivo può incontrare ciascun sottoproblema più volte nelle varie diramazioni del suo albero di ricorsione. Questa proprietà dei sottoproblemi che si ripresentano è la seconda caratteristica peculiare dell'applicabilità della programmazione dinamica (la prima è la sottostruttura ottima).

Anziché calcolare la soluzione della ricorrenza (15.12) ricorsivamente, passiamo alla quarta fase del paradigma della programmazione dinamica e calcoliamo il costo ottimale applicando un metodo tabulare bottom-up (dal basso verso l'alto). Il seguente pseudocodice suppone che la matrice A_i abbia dimensioni $p_{i-1} \times p_i$ per $i = 1, 2, \ldots, n$. L'input è una sequenza $p = \langle p_0, p_1, \ldots, p_n \rangle$, dove $length[p] = n + 1$. La procedura usa una tabella ausiliaria $m[1\,..\,n, 1\,..\,n]$ per memorizzare i costi $m[i, j]$ e una tabella ausiliaria $s[1\,..\,n, 1\,..\,n]$ che registra l'indice k cui corrisponde il costo ottimo nel calcolo di $m[i, j]$. Utilizzeremo la tabella s per costruire una soluzione ottima.

Per implementare correttamente il metodo bottom-up, dobbiamo determinare quali posizioni nella tabella sono utilizzate nel calcolo di $m[i, j]$. L'Equazione (15.12) indica che il costo $m[i, j]$ per calcolare il prodotto di $j - i + 1$ matrici dipende soltanto dai costi per calcolare il prodotto di una sequenza di meno di $j - i + 1$ matrici. Ovvero, per $k = i, i+1, \ldots, j-1$, la matrice $A_{i..k}$ è un prodotto di $k-i+1 < j-i+1$ matrici e la matrice $A_{k+1..j}$ è un prodotto di $j-k < j-i+1$ matrici. Quindi, l'algoritmo dovrebbe riempire la tabella m secondo una modalità che corrisponde a risolvere il problema della parentesizzazione di sequenze di matrici di lunghezza crescente.

L'algoritmo prima calcola $m[i, i] \leftarrow 0$ per $i = 1, 2, \ldots, n$ (i costi minimi per le sequenze di lunghezza 1) nelle righe 2–3. Poi, usa la ricorrenza (15.12) per calcolare $m[i, i + 1]$ per $i = 1, 2, \ldots, n - 1$ (i costi minimi per le sequenze di lunghezza $l = 2$) durante la prima esecuzione del ciclo (righe 4–12). La seconda volta che esegue il ciclo, l'algoritmo calcola $m[i, i + 2]$ per $i = 1, 2, \ldots, n - 2$ (i costi minimi per le sequenze di lunghezza $l = 3$) e così via. A ogni passo, il costo $m[i, j]$ calcolato nelle righe 9–12 dipende soltanto dalle posizioni $m[i, k]$ e $m[k + 1, j]$ già calcolate.

MATRIX-CHAIN-ORDER(p)

```
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n            ▷ l è la lunghezza della sequenza.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13  return m e s
```

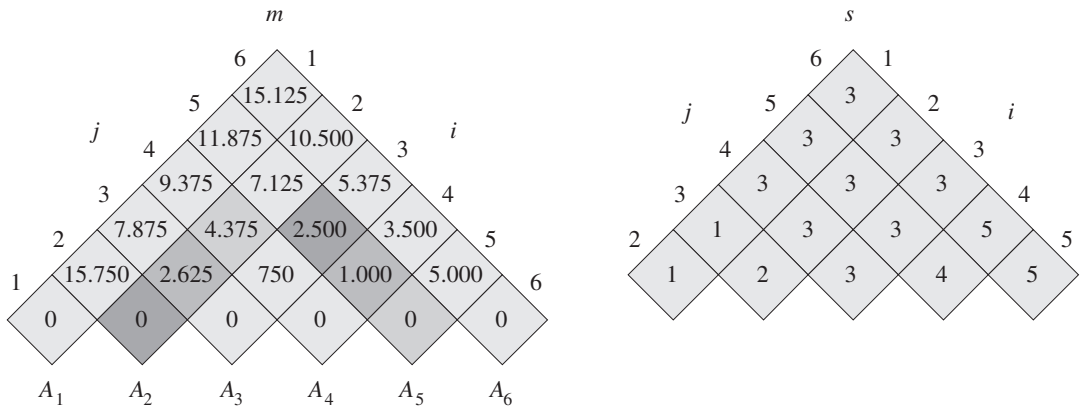


Figura 15.3 Le tabelle m e s calcolate da MATRIX-CHAIN-ORDER per $n = 6$ matrici con le seguenti dimensioni:

matrice	dimensione
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Le tabelle sono ruotate in modo che la diagonale principale sia orizzontale. Soltanto la diagonale principale e il triangolo superiore sono utilizzati nella tabella m. Soltanto il triangolo superiore è utilizzato nella tabella s. Il numero minimo di prodotti scalari per moltiplicare le 6 matrici è $m[1,6] = 15125$. Fra le posizioni con sfondo più scuro, le coppie che hanno la stessa gradazione di grigio sono utilizzate insieme nella riga 9 durante i calcoli

$$\begin{aligned} m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11375 \end{cases} \\ &= 7125 \end{aligned}$$

La Figura 15.3 illustra questa procedura con una sequenza di $n = 6$ matrici. Poiché abbiamo definito $m[i, j]$ soltanto per $i \leq j$, viene utilizzata solamente la porzione della tabella m che si trova subito sopra la diagonale principale. La figura ruota la tabella per rappresentare orizzontalmente la diagonale principale. La sequenza delle matrici è elencata sotto la figura. Utilizzando questo schema, il costo minimo $m[i, j]$ per moltiplicare una sottosequenza di matrici $A_i A_{i+1} \cdots A_j$ può essere trovato nel punto di intersezione delle linee con direzione nord-est da A_i e con direzione nord-ovest da A_j. Ogni riga orizzontale nella tabella contiene le posizioni per sequenze di matrici della stessa lunghezza. MATRIX-CHAIN-ORDER calcola le righe dal basso verso l'alto e da sinistra a destra all'interno di ogni riga. Una posizione $m[i, j]$ è calcolata utilizzando i prodotti $p_{i-1} p_k p_j$ per $k = i, i + 1, \ldots, j - 1$ e tutte le posizioni con direzione sud-ovest e sud-est da $m[i, j]$.

Da un semplice esame della struttura annidata dei cicli della procedura MATRIX-CHAIN-ORDER si deduce che il tempo di esecuzione dell'algoritmo è pari a $O(n^3)$. I cicli hanno tre livelli di annidamento e ogni indice di ciclo (l, i e k) assu-

me al massimo $n-1$ valori. L'Esercizio 15.2-4 chiede di dimostrare che il tempo di esecuzione di questo algoritmo, in effetti, è anche $\Omega(n^3)$. L'algoritmo richiede lo spazio $\Theta(n^2)$ per memorizzare le tabelle m e s. Quindi, la procedura MATRIX-CHAIN-ORDER è molto più efficiente del metodo con tempo esponenziale che elenca tutte le possibili parentesizzazioni controllandole una per una.

Fase 4: costruire una soluzione ottima

La procedura MATRIX-CHAIN-ORDER determina il numero ottimo di prodotti scalari richiesti per moltiplicare una sequenza di matrici, ma non mostra direttamente come moltiplicare le matrici. Non è difficile costruire una soluzione ottima dalle informazioni calcolate che sono memorizzate nella tabella $s[1\,..\,n, 1\,..\,n]$. Ogni posizione $s[i, j]$ registra quel valore di k per il quale la parentesizzazione ottima di $A_i A_{i+1} \cdots A_j$ suddivide il prodotto fra A_k e A_{k+1}. Quindi, sappiamo che il prodotto finale delle matrici nel calcolo ottimale di $A_{1..n}$ è $A_{1..s[1,n]} A_{s[1,n]+1..n}$. I primi prodotti possono essere calcolati ricorsivamente, perché $s[1, s[1, n]]$ determina l'ultimo prodotto nel calcolo di $A_{1..s[1,n]}$ e $s[s[1, n] + 1, n]$ determina l'ultimo prodotto nel calcolo di $A_{s[1,n]+1..n}$. La seguente procedura ricorsiva produce una parentesizzazione ottima di $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, dati gli indici i e j e la tabella s calcolata da MATRIX-CHAIN-ORDER. La chiamata iniziale di PRINT-OPTIMAL-PARENS$(s, 1, n)$ produce una parentesizzazione ottima di $\langle A_1, A_2, \ldots, A_n \rangle$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2     then stampa "A"_i
3     else stampa "("
4          PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5          PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6          stampa ")"
```

Nell'esempio della Figura 15.3, la chiamata PRINT-OPTIMAL-PARENS$(s, 1, 6)$ produce la parentesizzazione $((A_1(A_2A_3))((A_4A_5)A_6))$.

Esercizi

15.2-1

Trovate una parentesizzazione ottima del prodotto di una sequenza di matrici le cui dimensioni sono $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2-2

Create un algoritmo ricorsivo MATRIX-CHAIN-MULTIPLY(A, s, i, j) che calcola effettivamente il prodotto di una sequenza di matrici, dati gli indici i e j, la sequenza delle matrici $\langle A_1, A_2, \ldots, A_n \rangle$ e la tabella s calcolata da MATRIX-CHAIN-ORDER (la chiamata iniziale sarà MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$).

15.2-3

Utilizzate il metodo di sostituzione per dimostrare che la soluzione della ricorrenza (15.11) è $\Omega(2^n)$.

15.2-4

Sia $R(i, j)$ il numero di riferimenti alla posizione $m[i, j]$ della tabella m mentre vengono calcolate le altre posizioni della tabella in una chiamata di MATRIX-

CHAIN-ORDER. Dimostrate che il numero totale di riferimenti per l'intera tabella è il seguente

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3 - n}{3}$$

(*Suggerimento:* potrebbe essere utile l'equazione (A.3).)

15.2-5

Dimostrate che una parentesizzazione completa di un'espressione di n elementi ha esattamente $n - 1$ coppie di parentesi.

15.3 Elementi della programmazione dinamica

Sebbene abbiamo appena descritto due esempi applicativi del metodo della programmazione dinamica, tuttavia qualcuno potrebbe chiedersi quando si applica questo metodo. Da un punto di vista ingegneristico, quando dovremmo cercare di risolvere un problema con la programmazione dinamica? In questo paragrafo, esamineremo i due ingredienti chiave che deve avere un problema di ottimizzazione affinché possa essere applicata la programmazione dinamica: la sottostruttura ottima e i sottoproblemi ripetitivi. Analizzeremo anche una variante della programmazione dinamica, detta *tecnica di memoization*,[1] che sfrutta la proprietà dei problemi ripetitivi.

Sottostruttura ottima

La prima fase del processo di risoluzione di un problema di ottimizzazione mediante la programmazione dinamica consiste nel caratterizzare la struttura di una soluzione ottima. Ricordiamo che un problema presenta una ***sottostruttura ottima*** se una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi. Quando un problema presenta una sottostruttura ottima, ciò potrebbe essere un buon indizio dell'applicabilità della programmazione dinamica (ma potrebbe anche indicare che è possibile applicare una strategia greedy, come vedremo nel Capitolo 16). Nella programmazione dinamica, costruiamo una soluzione ottima del problema dalle soluzioni ottime dei sottoproblemi. Di conseguenza, dobbiamo essere sicuri che l'insieme dei sottoproblemi considerati includa quelli utilizzati in una soluzione ottima.

Abbiamo identificato una sottostruttura ottima in entrambi i problemi esaminati in questo capitolo. Nel Paragrafo 15.1, abbiamo osservato che il percorso più rapido per arrivare alla stazione j nelle due linee di assemblaggio conteneva al suo interno il percorso più rapido per raggiungere la stazione $j-1$ in una linea. Nel Paragrafo 15.2, abbiamo osservato che una parentesizzazione ottima di $A_i A_{i+1} \cdots A_j$ che suddivide il prodotto fra A_k e A_{k+1} contiene al suo interno soluzioni ottime dei problemi di parentesizzazione di $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$.

Vedrete voi stessi che bisogna adottare uno schema comune per trovare una sottostruttura ottima:

[1] Non è un errore ortografico. Il termine originale è proprio *memoization*, non *memorization*, e deriva da *memo* (promemoria) perché la tecnica consiste nel registrare un valore in modo che possa essere riutilizzato in futuro.

1. Dimostrate che una soluzione del problema consiste nel fare una scelta (per esempio, scegliere la stazione precedente in una linea di assemblaggio o un indice in corrispondenza del quale suddividere la sequenza delle matrici). Questa scelta lascia uno o più sottoproblemi da risolvere.
2. Per un dato problema, supponete di conoscere la scelta che porta a una soluzione ottima. Non vi interessa sapere come sia stata determinata questa scelta. Semplicemente, supponete di conoscere tale scelta.
3. Fatta la scelta, determinate quali sottoproblemi considerare e come meglio caratterizzare lo spazio risultante dei sottoproblemi.
4. Dimostrate che le soluzioni dei sottoproblemi che avete utilizzato all'interno della soluzione ottima del problema devono essere necessariamente ottime, adottando una tecnica "taglia e incolla". Per fare questo, prima supponete che ciascuna delle soluzioni dei sottoproblemi non sia ottima e, poi, arrivate a una contraddizione. In particolare, "tagliando" la soluzione non ottima di un sottoproblema e "incollando" quella ottima, dimostrate che potete ottenere una soluzione migliore del problema originale, contraddicendo l'ipotesi di avere già una soluzione ottima. Se ci sono più sottoproblemi, tipicamente, essi sono così simili che la tecnica "taglia e incolla" applicata a un sottoproblema può essere adattata con una piccola modifica agli altri sottoproblemi.

Per caratterizzare lo spazio dei sottoproblemi, una buona regola consiste nel cercare di mantenere tale spazio quanto più semplice possibile, per poi espanderlo, se necessario. Per esempio, lo spazio dei sottoproblemi che abbiamo considerato nella programmazione delle linee di assemblaggio era il percorso più rapido dall'entrata nello stabilimento fino alle stazioni $S_{1,j}$ e $S_{2,j}$. Questo spazio di sottoproblemi ha funzionato bene e non c'è stato bisogno di provare uno spazio più generale. Supponiamo, invece, di limitare lo spazio dei sottoproblemi per la moltiplicazione di una sequenza di matrici a quei prodotti matriciali della forma $A_1 A_2 \cdots A_j$. Come nel caso esaminato in precedenza, una parentesizzazione ottima deve suddividere questo prodotto fra A_k e A_{k+1} per qualche $1 \leq k < j$. A meno che non garantiamo che k sia sempre uguale a $j - 1$, troveremo che i sottoproblemi hanno la forma $A_1 A_2 \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$ e che quest'ultimo sottoproblema non ha la forma $A_1 A_2 \cdots A_j$. Ecco perché, per questo problema, è stato necessario consentire ai sottoproblemi di variare in "entrambe le estremità", ovvero consentire a i e j di variare nel sottoproblema $A_i A_{i+1} \cdots A_j$.

La sottostruttura ottima varia nei domini dei problemi in due modi:

1. per il numero di sottoproblemi che sono utilizzati in una soluzione ottima del problema originale;
2. per il numero di scelte che possiamo fare per determinare quale sottoproblema (o quali sottoproblemi) utilizzare in una soluzione ottima.

Nella programmazione delle linee di assemblaggio, una soluzione ottima usa un solo sottoproblema, ma dobbiamo considerare due scelte per determinare una soluzione ottima. Per trovare il percorso più rapido fino alla stazione $S_{i,j}$, utilizziamo o il percorso più rapido fino a $S_{1,j-1}$ o il percorso più rapido fino a $S_{2,j-1}$; qualunque percorso scegliamo, esso rappresenta il sottoproblema da risolvere. La moltiplicazione di una sequenza di matrici per la sottosequenza $A_i A_{i+1} \cdots A_j$ è un esempio di due sottoproblemi e $j - i$ scelte. Per una data matrice A_k, in corrispondenza della quale suddividiamo il prodotto, abbiamo due sottoproblemi

– parentesizzazione di $A_i A_{i+1} \cdots A_k$ e parentesizzazione di $A_{k+1} A_{k+2} \cdots A_j$ – e dobbiamo trovare le soluzioni ottime per *entrambi* i sottoproblemi. Una volta trovate queste soluzioni, scegliamo l'indice k fra $j - i$ candidati.

Informalmente, il tempo di esecuzione di un algoritmo di programmazione dinamica dipende dal prodotto di due fattori: il numero di sottoproblemi da risolvere complessivamente e il numero di scelte da considerare per ogni sottoproblema. Nella programmazione delle linee di assemblaggio avevamo $\Theta(n)$ sottoproblemi complessivamente e due sole scelte da esaminare per ogni sottoproblema, con un tempo di esecuzione pari a $\Theta(n)$. Nella moltiplicazione di una sequenza di matrici c'erano $\Theta(n^2)$ sottoproblemi complessivamente e per ciascuno di essi avevamo al massimo $n - 1$ scelte, con un tempo di esecuzione pari a $O(n^3)$.

La programmazione dinamica usa la sottostruttura ottima secondo uno schema *bottom-up* (dal basso verso l'alto); ovvero, prima vengono trovate le soluzioni ottime dei sottoproblemi e, dopo avere risolto i sottoproblemi, viene trovata una soluzione ottima del problema. Trovare una soluzione ottima del problema significa scegliere uno dei sottoproblemi da utilizzare per risolvere il problema. Il costo della soluzione del problema, di solito, è pari ai costi per risolvere i sottoproblemi più un costo che è direttamente imputabile alla scelta stessa. Per esempio, nella programmazione delle linee di assemblaggio, prima risolviamo i sottoproblemi per trovare il percorso più rapido fino alle stazioni $S_{1,j-1}$ e $S_{2,j-1}$; poi scegliamo una di queste stazioni come quella che precede la stazione $S_{i,j}$. Il costo imputabile alla scelta varia a seconda che cambiamo linea di assemblaggio fra le stazioni $j - 1$ e j; questo costo è pari a $a_{i,j}$ se restiamo nella stessa linea, è pari a $t_{i',j-1} + a_{i,j}$, dove $i' \neq i$, se cambiamo linea. Nella moltiplicazione di una sequenza di matrici, prima determiniamo le parentesizzazioni ottime delle sottosequenze di $A_i A_{i+1} \cdots A_j$; poi scegliamo la matrice A_k, in corrispondenza della quale suddividere il prodotto. Il costo imputabile alla scelta è il termine $p_{i-1} p_k p_j$.

Nel Capitolo 16 esamineremo gli "algoritmi greedy", che hanno molte affinità con la programmazione dinamica. In particolare, i problemi ai quali si applicano gli algoritmi greedy hanno una sottostruttura ottima. Una differenza importante fra gli algoritmi greedy e la programmazione dinamica è che negli algoritmi greedy utilizziamo la sottostruttura ottima secondo uno schema *top-down* (dall'alto verso il basso). Anziché trovare prima le soluzioni ottime dei sottoproblemi e poi fare una scelta, gli algoritmi greedy prima fanno una scelta – quella che sembra ottima in quel momento – e poi risolvono un sottoproblema risultante.

Finezze

Occorre prestare attenzione a non pensare di applicare la sottostruttura ottima quando non è possibile farlo. Consideriamo i seguenti due problemi in cui sono dati un grafo orientato $G = (V, A)$ e i vertici $u, v \in V$.

Cammino minimo in un grafo non pesato:[2] trovare un cammino da u a v formato dal minor numero di archi. Tale cammino deve essere semplice, perché eliminando un ciclo da un cammino si ottiene un cammino con un minor numero di archi.

[2]Utilizziamo il termine "non pesato" per distinguere questo problema da quello della ricerca dei cammini minimi con archi pesati, che esamineremo nei Capitoli 24 e 25. Possiamo utilizzare la tecnica della ricerca in ampiezza (breadth-first search) descritta nel Capitolo 22 per risolvere il problema del cammino minimo in un grafo non pesato.

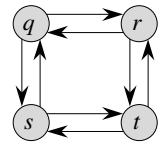


Figura 15.4 Il grafo orientato illustra che il problema di trovare un cammino semplice massimo in un grafo orientato non pesato non ha una sottostruttura ottima. Il percorso $q \to r \to t$ è un cammino semplice massimo da q a t, ma il sottocammino $q \to r$ non è un cammino semplice massimo da q a r, né il sottocammino $r \to t$ è un cammino semplice massimo da r a t.

Cammino semplice massimo in un grafo non pesato: trovare un cammino semplice da u a v che è formato dal maggior numero di archi. Dobbiamo richiedere che il cammino sia semplice, perché altrimenti potremmo attraversare un ciclo un numero indefinito di volte per creare cammini con un numero arbitrariamente grande di archi.

Il problema del cammino minimo (o percorso più breve) in un grafo non pesato presenta una sottostruttura ottima, nel modo seguente. Supponiamo che $u \neq v$, in modo che il problema non sia banale; allora qualsiasi percorso p da u a v deve contenere un vertice intermedio w (notate che w può essere u o v). Quindi possiamo scomporre il cammino $u \overset{p}{\rightsquigarrow} v$ nei sottocammini $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$. Chiaramente, il numero di archi in p è uguale alla somma del numero di archi in p_1 e del numero di archi in p_2. Noi asseriamo che, se p è un cammino ottimo (cioè minimo) da u a v, allora p_1 deve essere un cammino minimo da u a w. Perché? Applichiamo la tecnica "taglia e incolla": se ci fosse un altro cammino, p'_1, da u a w con un numero minore di archi rispetto a p_1, allora potremmo tagliare p_1 e incollare p'_1 per ottenere un cammino $u \overset{p'_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ cha ha meno archi di p, contraddicendo l'ipotesi che p sia un cammino ottimo. In maniera simmetrica, p_2 deve essere un cammino minimo da w a v. Quindi, possiamo trovare un cammino minimo da u a v considerando tutti i vertici intermedi w, trovando un cammino minimo da u a w e un cammino minimo da w a v, e scegliendo un vertice intermedio w che produce il cammino minimo totale. Nel Paragrafo 25.2 utilizzeremo una variante di questa osservazione della sottostruttura ottima per trovare un cammino minimo fra ogni coppia di vertici in un grafo orientato pesato.

Saremmo tentati di supporre che anche il problema di trovare un percorso semplice più lungo in un grafo non pesato presenti una sottostruttura ottima. Dopo tutto, se scomponiamo un cammino semplice più lungo $u \overset{p}{\rightsquigarrow} v$ nei sottocammini $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, perché p_1 non dovrebbe essere un cammino semplice massimo da u a w, e p_2 non dovrebbe essere un cammino semplice massimo da w a v? La risposta è no! La Figura 15.4 illustra un esempio. Consideriamo il cammino $q \to r \to t$, che è un cammino semplice massimo da q a t. Il percorso $q \to r$ è un cammino semplice massimo da q a r? No, perché il percorso $q \to s \to t \to r$ è un cammino semplice che è più lungo. Il percorso $r \to t$ è un cammino semplice massimo da r a t? Ancora no, perché il percorso $r \to q \to s \to t$ è un cammino semplice che è più lungo.

Questo esempio dimostra che per i cammini semplici massimi, non soltanto manca una sottostruttura ottima, ma non è possibile assemblare una soluzione "valida" del problema dalle soluzioni dei sottoproblemi. Se combiniamo i cammini semplici massimi $q \to s \to t \to r$ e $r \to q \to s \to t$, otteniamo il cammino $q \to s \to t \to r \to q \to s \to t$, che non è semplice. Il problema di trovare un cammino semplice massimo in un grafo non pesato sembra davvero non avere alcuna sorta di sottostruttura ottima. Non è stato ancora trovato un algoritmo efficiente di programmazione dinamica per questo problema. In effetti, si tratta di

un problema NP-completo, che – come vedremo nel Capitolo 34 – significa che è improbabile che possa essere risolto in un tempo polinomiale.

Che cosa rende la sottostruttura di un cammino semplice massimo così diversa da quella di un cammino minimo? Sebbene siano utilizzati due sottoproblemi in una soluzione di un problema per entrambi i cammini massimi e minimi, i sottoproblemi per trovare il cammino semplice massimo non sono ***indipendenti***, mentre lo sono per i cammini minimi. Che cosa significa che i sottoproblemi sono indipendenti? Significa che la soluzione di un sottoproblema non influisce sulla soluzione di un altro sottoproblema dello stesso problema. Per l'esempio della Figura 15.4, abbiamo il problema di trovare un cammino semplice massimo da q a t con due sottoproblemi: trovare i cammini semplici massimi da q a r e da r a t. Per il primo di questi sottoproblemi, scegliamo il cammino $q \rightarrow s \rightarrow t \rightarrow r$; quindi abbiamo utilizzato anche i vertici s e t. Non possiamo più utilizzare questi vertici nel secondo sottoproblema, perché la combinazione delle due soluzioni dei sottoproblemi produrrebbe un cammino che non è semplice. Se non possiamo più utilizzare il vertice t, allora il secondo sottoproblema è irrisolvibile, perché t appartiene al cammino che cerchiamo e non è il vertice in cui stiamo congiungendo le soluzioni dei sottoproblemi (questo vertice è r). Il nostro utilizzo dei vertici s e t nella soluzione di un sottoproblema ci impedisce di utilizzarli nella soluzione dell'altro sottoproblema. Tuttavia, dobbiamo utilizzare almeno uno di questi vertici per risolvere l'altro sottoproblema e dobbiamo utilizzarli entrambi per trovare la soluzione ottima di questo sottoproblema. Quindi, diciamo che questi sottoproblemi non sono indipendenti. In altre parole, il nostro utilizzo delle risorse nel risolvere un sottoproblema (le risorse sono i vertici) le ha rese indisponibili per l'altro sottoproblema.

Perché, allora, i sottoproblemi per trovare un cammino minimo sono indipendenti? La risposta è che, per natura, i sottoproblemi non condividono le risorse. Noi asseriamo che se un vertice w si trova in un percorso minimo p da u a v, allora possiamo congiungere *qualsiasi* percorso minimo $u \overset{p_1}{\rightsquigarrow} w$ con *qualsiasi* cammino minimo $w \overset{p_2}{\rightsquigarrow} v$ per produrre un cammino minimo da u a v. Abbiamo la certezza che, tranne w, nessun vertice può apparire in entrambi i percorsi p_1 e p_2. Perché? Supponiamo che qualche vertice $x \neq w$ appaia in entrambi i cammini p_1 e p_2, in modo che possiamo scomporre p_1 in $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ e p_2 in $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. Per la sottostruttura ottima di questo problema, il cammino p ha tanti archi quanti ne hanno i due cammini p_1 e p_2; diciamo che p ha e archi. Adesso costruiamo un cammino $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ da u a v. Questo cammino ha al massimo $e - 2$ archi, che contraddice l'ipotesi che p sia una cammino minimo. Quindi, abbiamo la certezza che i sottoproblemi per il problema del cammino minimo sono indipendenti.

Entrambi i problemi esaminati nei Paragrafi 15.1 e 15.2 hanno sottoproblemi indipendenti. Nella moltiplicazione di una sequenza di matrici, i sottoproblemi sono i prodotti delle sottosequenze $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$. Queste sottosequenze sono disgiunte, quindi nessuna matrice potrebbe essere inclusa in nessuna di esse. Nella programmazione delle linee di assemblaggio, per trovare il percorso più rapido fino alla stazione $S_{i,j}$, abbiamo esaminato i percorsi più rapidi fino alle stazioni $S_{1,j-1}$ e $S_{2,j-1}$. Poiché la nostra soluzione del percorso più rapido fino alla stazione $S_{i,j}$ includerà soltanto una di queste soluzioni del sottoproblema, questo sottoproblema è automaticamente indipendente da tutti gli altri utilizzati nella soluzione.

Sottoproblemi ripetitivi

Il secondo ingrediente che un problema di ottimizzazione deve avere affinché la programmazione dinamica possa essere applicata è che lo spazio dei sottoproblemi deve essere "piccolo", nel senso che un algoritmo ricorsivo per il problema risolve ripetutamente gli stessi sottoproblemi, anziché generare sempre nuovi sottoproblemi. Tipicamente, il numero totale di sottoproblemi distinti è un polinomio nella dimensione dell'input. Quando un algoritmo ricorsivo rivisita più volte lo stesso problema, diciamo che il problema di ottimizzazione ha dei ***sottoproblemi ripetitivi***.[3] D'altra parte, un problema per il quale è appropriato un approccio divide et impera, di solito, genera problemi nuovi di zecca a ogni passaggio della ricorsione. Gli algoritmi di programmazione dinamica tipicamente sfruttano i sottoproblemi ripetitivi risolvendo ciascun sottoproblema una sola volta e, poi, memorizzando la soluzione in una tabella dove può essere ricercata quando serve, impiegando un tempo costante per la ricerca.

Nel Paragrafo 15.1 abbiamo visto che una soluzione ricorsiva per la programmazione delle linee di assemblaggio effettua 2^{n-j} riferimenti a $f_i[j]$ per $j = 1, 2, \ldots, n$. La nostra soluzione tabulare abbassa il tempo esponenziale di un algoritmo ricorsivo a un tempo lineare.

Per illustrare più dettagliatamente la proprietà dei sottoproblemi ripetitivi, riesaminiamo il problema della moltiplicazione di una sequenza di matrici. Facendo riferimento alla precedente Figura 15.3, notiamo che MATRIX-CHAIN-ORDER cerca ripetutamente la soluzione dei sottoproblemi nelle righe inferiori quando risolve i sottoproblemi nelle righe superiori. Per esempio, la posizione $m[3, 4]$ ha 4 riferimenti: durante il calcolo di $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ e $m[3, 6]$. Se il valore di $m[3, 4]$ fosse ricalcolato ogni volta, anziché soltanto cercato, il tempo di esecuzione aumenterebbe enormemente. Per vedere questo, consideriamo la seguente procedura ricorsiva (inefficiente) che calcola $m[i, j]$, il numero minimo di prodotti scalari richiesti per calcolare il prodotto di una sequenza di matrici $A_{i..j} = A_i A_{i+1} \cdots A_j$. La procedura si basa direttamente sulla ricorrenza (15.12).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5        do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                  + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
                  + p_{i−1} p_k p_j
6           if q < m[i, j]
7              then m[i, j] ← q
8  return m[i, j]
```

[3] Potrebbe sembrare strano che la programmazione dinamica si affidi a sottoproblemi che sono indipendenti e ripetitivi. Sebbene questi requisiti possano apparire contraddittori, tuttavia esprimono due concetti differenti, anziché due punti sullo stesso asse. Due sottoproblemi dello stesso problema sono indipendenti se non condividono le stesse risorse. Due sottoproblemi sono ripetitivi se sono veramente lo stesso sottoproblema che si presenta come un sottoproblema di problemi differenti.

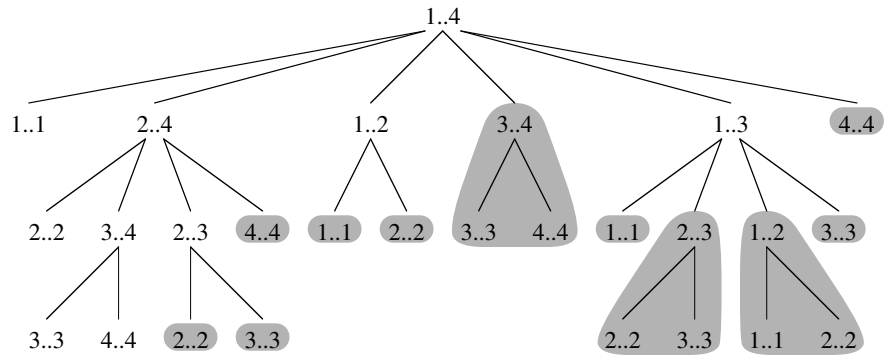


Figura 15.5 L'albero di ricorsione per il calcolo di RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Ogni nodo contiene i parametri i e j. I calcoli svolti in un sottoalbero con sfondo grigio sono sostituiti da una singola ricerca in tabella nella chiamata MEMOIZED-MATRIX-CHAIN$(p, 1, 4)$.

La Figura 15.5 illustra l'albero di ricorsione prodotto dalla chiamata RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Ogni nodo è etichettato dai valori dei parametri i e j. Notate che alcune coppie di valori si presentano più volte.

In effetti, possiamo dimostrare che il tempo per calcolare $m[1, n]$ con questa procedura ricorsiva è almeno esponenziale in n. Sia $T(n)$ il tempo impiegato da RECURSIVE-MATRIX-CHAIN per calcolare una parentesizzazione ottima di una sequenza di n matrici. Se supponiamo che per ogni esecuzione delle righe 1–2 e delle righe 6–7 occorre almeno un'unità di tempo, allora l'ispezione della procedura genera la ricorrenza

$$\begin{aligned} T(1) &\geq 1 \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1 \end{aligned}$$

Notate che per $i = 1, 2, \ldots, n-1$, ogni termine $T(i)$ appare una volta come $T(k)$ e una volta come $T(n-k)$; raccogliendo gli $n-1$ 1 nella sommatoria insieme con l'1 iniziale, possiamo riscrivere la ricorrenza in questo modo

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n \tag{15.13}$$

Dimostreremo che $T(n) = \Omega(2^n)$ utilizzando il metodo di sostituzione. Più precisamente, dimostreremo che $T(n) \geq 2^{n-1}$ per ogni $n \geq 1$. La base è semplice, in quanto $T(1) \geq 1 = 2^0$. Induttivamente, per $n \geq 2$ abbiamo

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\geq 2^{n-1} \end{aligned}$$

che completa la dimostrazione. Quindi, la quantità totale di lavoro svolto dalla chiamata RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ è almeno esponenziale in n.

Confrontate questo algoritmo ricorsivo top-down con l'algoritmo bottom-up di programmazione dinamica. Quest'ultimo è più efficiente perché sfrutta la proprietà dei sottoproblemi ripetitivi. Ci sono soltanto $\Theta(n^2)$ sottoproblemi differenti e l'algoritmo di programmazione dinamica risolve ciascun sottoproblema una sola volta. L'algoritmo ricorsivo, invece, deve risolvere ripetutamente ciascun sottoproblema, ogni volta che un sottoproblema si ripresenta nell'albero di ricorsione. Quando un albero di ricorsione per la soluzione ricorsiva naturale di un problema contiene più volte lo stesso sottoproblema e il numero totale di sottoproblemi differenti è piccolo, è una buona idea verificare se può essere applicata la programmazione dinamica.

Ricostruire una soluzione ottima

Spesso memorizziamo in una tabella la scelta fatta in ciascun sottoproblema, per evitare di dovere ricostruire questa informazione dai costi che abbiamo memorizzato. Nella programmazione delle linee di assemblaggio, abbiamo memorizzato in $l_i[j]$ la stazione che precede $S_{i,j}$ in un percorso più rapido fino alla stazione $S_{i,j}$. In alternativa, avendo riempito l'intera tabella $f_i[j]$, potremmo determinare quale stazione precede $S_{1,j}$ in un percorso più rapido fino a $S_{1,j}$ con un piccolo lavoro extra. Se $f_1[j] = f_1[j-1] + a_{1,j}$, allora la stazione $S_{1,j-1}$ precede $S_{1,j}$ in un percorso più rapido fino a $S_{1,j}$. Altrimenti, deve verificarsi il caso che $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ e, quindi, $S_{2,j-1}$ precede $S_{1,j}$. Per la programmazione delle linee di assemblaggio, la ricostruzione delle stazioni precedenti richiede soltanto un tempo $O(1)$ per stazione, anche senza la tabella $l_i[j]$.

Per la moltiplicazione di una sequenza di matrici, invece, la tabella $s[i,j]$ permette di risparmiare una notevole quantità di lavoro per ricostruire una soluzione ottima. Supponiamo di non avere utilizzato la tabella $s[i,j]$, avendo riempito soltanto la tabella $m[i,j]$ che contiene i costi ottimi dei sottoproblemi. Ci sono $j-i$ scelte per determinare quali sottoproblemi utilizzare in una soluzione ottima per parentesizzare $A_iA_{i+1}\cdots A_j$; $j-i$ non è una costante. Quindi, occorrerebbe il tempo $\Theta(j-i) = \omega(1)$ per ricostruire i sottoproblemi scelti per una soluzione di un determinato problema. Memorizzando in $s[i,j]$ l'indice della matrice in corrispondenza della quale suddividiamo il prodotto $A_iA_{i+1}\cdots A_j$, possiamo ricostruire ciascuna scelta nel tempo $O(1)$.

Memoization

La tecnica di memoization è una variante della programmazione dinamica che, pur conservando la strategia top-down, spesso offre la stessa efficienza dell'usuale approccio della programmazione dinamica. Il concetto che sta alla base di questa tecnica consiste nel *memoizzare* il naturale, ma inefficiente, algoritmo ricorsivo. Come nell'ordinaria programmazione dinamica, viene utilizzata una tabella con le soluzioni dei sottoproblemi, ma la struttura di controllo per riempire la tabella è più simile all'algoritmo ricorsivo.

Un algoritmo ricorsivo memoizzato utilizza una posizione di una tabella per la soluzione di ciascun sottoproblema. Ogni posizione della tabella inizialmente contiene un valore speciale per indicare che la posizione non è stata ancora riempita. La prima volta che si presenta il sottoproblema durante l'esecuzione dell'algoritmo ricorsivo, viene calcolata la soluzione del sottoproblema che, poi, viene memorizzata nella tabella. Successivamente, ogni volta che si ripresenta questo sot-

toproblema, l'algoritmo ricerca e restituisce il corrispondente valore memorizzato nella tabella.[4]

Riportiamo qui di seguito una versione memoizzata di RECURSIVE-MATRIX-CHAIN:

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3       do for j ← i to n
4             do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2     then return m[i, j]
3  if i = j
4     then m[i, j] ← 0
5     else  for k ← i to j − 1
6             do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                if q < m[i, j]
8                   then m[i, j] ← q
9  return m[i, j]
```

MEMOIZED-MATRIX-CHAIN, come MATRIX-CHAIN-ORDER, utilizza una tabella $m[1\,..\,n, 1\,..\,n]$ per i valori calcolati di $m[i, j]$, il numero minimo di prodotti scalari richiesti per calcolare la matrice $A_{i..j}$. Inizialmente, ogni posizione della tabella contiene il valore ∞ per indicare che non è stato ancora inserito un valore in una posizione. Quando viene eseguita la chiamata LOOKUP-CHAIN(p, i, j), se $m[i, j] < \infty$ nella riga 1, la procedura restituisce semplicemente il costo precedentemente calcolato $m[i, j]$ (riga 2). Altrimenti, il costo viene calcolato come in RECURSIVE-MATRIX-CHAIN, memorizzato in $m[i, j]$ e poi restituito (è comodo utilizzare il valore ∞ per una posizione non ancora riempita della tabella, perché lo stesso valore è utilizzato per inizializzare $m[i, j]$ nella riga 3 di RECURSIVE-MATRIX-CHAIN). Quindi, LOOKUP-CHAIN(p, i, j) restituisce sempre il valore di $m[i, j]$, ma lo calcola soltanto se è la prima volta che viene chiamata la procedura LOOKUP-CHAIN con i parametri i e j.

La Figura 15.5 illustra come la procedura MEMOIZED-MATRIX-CHAIN consente di risparmiare tempo rispetto a RECURSIVE-MATRIX-CHAIN. I sottoalberi con sfondo grigio rappresentano i valori che vengono cercati, anziché calcolati.

Come l'algoritmo di programmazione dinamica MATRIX-CHAIN-ORDER, la procedura MEMOIZED-MATRIX-CHAIN viene eseguita nel tempo $O(n^3)$. Ciascuna delle $\Theta(n^2)$ posizioni della tabella viene inizializzata una volta nella riga 4 di MEMOIZED-MATRIX-CHAIN. Possiamo classificare le chiamate di LOOKUP-CHAIN in due tipi:

[4] Questo approccio presuppone che sia noto l'insieme di tutti i parametri dei sottoproblemi e che sia definita la relazione fra le posizioni della tabella e i sottoproblemi. Un altro approccio consiste nel memoizzare utilizzando l'hashing con i parametri dei sottoproblemi come chiavi.

1. le chiamate in cui $m[i, j] = \infty$; vengono eseguite le righe 3–9.
2. le chiamate in cui $m[i, j] < \infty$; LOOKUP-CHAIN termina nella riga 2.

Ci sono $\Theta(n^2)$ chiamate del primo tipo, una per ogni posizione della tabella. Tutte le chiamate del secondo tipo sono fatte come chiamate ricorsive da chiamate del primo tipo. Ogni volta che una chiamata di LOOKUP-CHAIN fa delle chiamate ricorsive, ne fa $O(n)$. Quindi, in totale ci sono $O(n^3)$ chiamate del secondo tipo. Ogni chiamata del secondo tipo impiega un tempo $O(1)$; ogni chiamata del primo tipo impiega un tempo $O(n)$ più il tempo speso nelle sue chiamate ricorsive. Il tempo totale è, dunque, $O(n^3)$. In conclusione, il processo di memoization trasforma un algoritmo con tempo $\Omega(2^n)$ in un algoritmo con tempo $O(n^3)$.

In sintesi, il problema della moltiplicazione di una sequenza di matrici può essere risolto nel tempo $O(n^3)$ sia da un algoritmo memoizzato top-down sia da un algoritmo bottom-up di programmazione dinamica. Entrambi i metodi sfruttano la proprietà dei sottoproblemi ripetitivi. Ci sono soltanto $\Theta(n^2)$ sottoproblemi differenti in totale e ciascuno di questi metodi calcola la soluzione di ogni sottoproblema una sola volta. Senza il processo di memoization, l'algoritmo ricorsivo naturale viene eseguito in un tempo esponenziale, perché i sottoproblemi risolti vengono ripetutamente risolti.

In generale, se tutti i sottoproblemi devono essere risolti almeno una volta, un algoritmo bottom-up di programmazione dinamica, di solito, supera le prestazioni di un algoritmo memoizzato top-down per un fattore costante, perché non ci sono costi per la ricorsione e i costi per la gestione della tabella sono minori. Inoltre, ci sono problemi per i quali è possibile sfruttare il normale schema di accessi alla tabella nell'algoritmo di programmazione dinamica per ridurre ulteriormente le esigenze di tempo o spazio. In alternativa, se alcuni sottoproblemi nello spazio dei sottoproblemi non richiedono affatto di essere risolti, la soluzione memoizzata ha il vantaggio di risolvere soltanto quei sottoproblemi che devono essere sicuramente risolti.

Esercizi

15.3-1

Qual è il metodo più efficiente per determinare il numero ottimo di prodotti matriciali nel problema della moltiplicazione di una sequenza di matrici: enumerare tutti gli schemi di parentesizzazione della moltiplicazione e calcolare il numero di prodotti per ogni schema oppure eseguire RECURSIVE-MATRIX-CHAIN? Spiegate la vostra risposta.

15.3-2

Disegnate l'albero di ricorsione per la procedura MERGE-SORT descritta nel Paragrafo 2.3.1 per un array di 16 elementi. Spiegate perché il processo di memoization non è efficace per accelerare un buon algoritmo divide et impera come MERGE-SORT.

15.3-3

Considerate una variante del problema della moltiplicazione di una sequenza di matrici in cui l'obiettivo è la parentesizzazione della sequenza delle matrici per massimizzare, anziché minimizzare, il numero dei prodotti scalari. Questo problema presenta una sottostruttura ottima?

15.3-4
Spiegate in che modo la programmazione delle linee di assemblaggio include dei sottoproblemi ripetitivi.

15.3-5
Come detto in precedenza, nella programmazione dinamica prima risolviamo i sottoproblemi e poi scegliamo quali sottoproblemi utilizzare in una soluzione ottima del problema. Il professor Capulet sostiene che non sempre è necessario risolvere tutti i sottoproblemi per trovare una soluzione ottima; ritiene che una soluzione ottima del problema della moltiplicazione di una sequenza di matrici può essere trovata scegliendo, *prima* di risolvere i sottoproblemi, la matrice A_k in corrispondenza della quale suddividere il sottoprodotto $A_i A_{i+1} \cdots A_j$ (scegliendo k per minimizzare la quantità $p_{i-1} p_k p_j$). Trovare un'istanza del problema della moltiplicazione di una sequenza di matrici per la quale questo approccio greedy produce una soluzione non ottima.

15.4 La più lunga sottosequenza comune (LCS)

Nelle applicazioni biologiche spesso si confronta il DNA di due (o più) organismi differenti. La struttura del DNA è formata da una stringa di molecole chiamate ***basi***; le possibili basi sono l'adenina, la citosina, la guanina e la timina. Rappresentando ciascuna di queste basi con le loro lettere iniziali, la struttura del DNA può essere espressa coma una stringa di un insieme finito $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ (la stringa è definita nell'Appendice C). Per esempio, il DNA di un organismo potrebbe essere $S_1 = \mathtt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$, mentre il DNA di un altro organismo potrebbe essere $S_2 = \mathtt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. Fra gli scopi del confronto di due molecole di DNA c'è quello di determinare il grado di somiglianza delle due molecole, misurando in qualche modo quanto è stretta la correlazione fra i due organismi. La somiglianza può essere definita in vari modi. Per esempio, potremmo dire che due molecole di DNA si somigliano se una è una sottostringa dell'altra (il Capitolo 32 descrive gli algoritmi che risolvono questo problema). Nell'esempio in esame, S_1 non è una sottostringa di S_2 né S_2 è una sottostringa di S_1. In alternativa, potremmo dire che due molecole di DNA si somigliano se il numero di modifiche richieste per trasformare una molecola nell'altra è piccolo (vedere il Problema 15-3). Un altro modo per misurare la somiglianza delle stringhe S_1 e S_2 consiste nel trovare una terza stringa S_3 le cui basi si presentano in ciascuna delle stringhe S_1 e S_2; queste basi devono presentarsi nello stesso ordine, senza essere necessariamente consecutive. Quanto più è lunga S_3, tanto più S_1 ed S_2 si somigliano. Nel nostro esempio, la più lunga stringa S_3 è $\mathtt{GTCGTCGGAAGCCGGCCGAA}$.

Formalizziamo quest'ultimo concetto di somiglianza come il problema della più lunga sottosequenza comune. Una sottosequenza di una data sequenza è la sequenza data con l'esclusione di zero o più elementi. Formalmente, data una sequenza $X = \langle x_1, x_2, \ldots, x_m \rangle$, un'altra sequenza $Z = \langle z_1, z_2, \ldots, z_k \rangle$ è una ***sottosequenza*** di X se esiste una sequenza strettamente crescente $\langle i_1, i_2, \ldots, i_k \rangle$ di indici di X tale che per ogni $j = 1, 2, \ldots, k$, si ha $x_{i_j} = z_j$. Per esempio, $Z = \langle B, C, D, B \rangle$ è una sottosequenza di $X = \langle A, B, C, B, D, A, B \rangle$ con la corrispondente sequenza di indici $\langle 2, 3, 5, 7 \rangle$.

Date due sequenze X e Y, diciamo che una sequenza Z è una ***sottosequenza comune*** di X e Y se Z è una sottosequenza di entrambe le sequenze X e Y. Per esempio, se $X = \langle A, B, C, B, D, A, B\rangle$ e $Y = \langle B, D, C, A, B, A\rangle$, la sequenza $\langle B, C, A\rangle$ è una sottosequenza comune di X e Y. Tuttavia, la sequenza $\langle B, C, A\rangle$ non è la *più lunga* sottosequenza comune (Longest Common Subsequence o LCS) di X e Y, perché ha lunghezza 3 e la sequenza $\langle B, C, B, A\rangle$, che è anche comune a X e Y, ha lunghezza 4. La sequenza $\langle B, C, B, A\rangle$ è una LCS di X e Y, come pure la sequenza $\langle B, D, A, B\rangle$, perché non esiste una sottosequenza comune di lunghezza 5 o più.

Nel ***problema della più lunga sottosequenza comune*** sono date due sequenze $X = \langle x_1, x_2, \ldots, x_m\rangle$ e $Y = \langle y_1, y_2, \ldots, y_n\rangle$ e si vuole trovare una sottosequenza di lunghezza massima che è comune a X e Y. Questo paragrafo dimostra che il problema della LCS può essere risolto in modo efficiente applicando la programmazione dinamica.

Fase 1: caratterizzare la più lunga sottosequenza comune

Una tecnica a forza bruta per risolvere il problema della più lunga sottosequenza comune consiste nell'enumerare tutte le sottosequenze di X e controllare le singole sottosequenze per vedere se sono anche sottosequenze di Y, tenendo traccia della più lunga sottosequenza trovata. Ogni sottosequenza di X corrisponde a un sottoinsieme degli indici $\{1, 2, \ldots, m\}$ di X. Ci sono 2^m sottosequenze di X, quindi questo approccio richiede un tempo esponenziale, rendendolo poco conveniente per le lunghe sequenze.

Tuttavia, il problema della LCS gode della proprietà della sottostruttura ottima, come dimostra il seguente teorema. Come vedremo, le classi naturali di sottoproblemi corrispondono a coppie di "prefissi" delle due sequenze di input. Più precisamente, data una sequenza $X = \langle x_1, x_2, \ldots, x_m\rangle$, definiamo $X_i = \langle x_1, x_2, \ldots, x_i\rangle$ l'i-esimo ***prefisso*** di X, per $i = 0, 1, \ldots, m$. Per esempio, se $X = \langle A, B, C, B, D, A, B\rangle$, allora $X_4 = \langle A, B, C, B\rangle$ e X_0 è la sequenza vuota.

Teorema 15.1 (Sottostruttura ottima di una LCS)
Siano $X = \langle x_1, x_2, \ldots, x_m\rangle$ e $Y = \langle y_1, y_2, \ldots, y_n\rangle$ le sequenze; sia $Z = \langle z_1, z_2, \ldots, z_k\rangle$ una qualsiasi LCS di X e Y.

1. Se $x_m = y_n$, allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, allora $z_k \neq x_m$ implica che Z è una LCS di X_{m-1} e Y.
3. Se $x_m \neq y_n$, allora $z_k \neq y_n$ implica che Z è una LCS di X e Y_{n-1}.

Dimostrazione (1) Se $z_k \neq x_m$, allora potremmo accodare $x_m = y_n$ a Z per ottenere una sottosequenza comune di X e Y di lunghezza $k+1$, contraddicendo l'ipotesi che Z è la *più lunga* sottosequenza comune di X e Y. Quindi, deve essere $z_k = x_m = y_n$. Ora, il prefisso Z_{k-1} è una sottosequenza comune di X_{m-1} e Y_{n-1} di lunghezza $k-1$. Vogliamo dimostrare che questo prefisso è una LCS. Supponiamo per assurdo che ci sia una sottosequenza comune W di X_{m-1} e Y_{n-1} di lunghezza maggiore di $k-1$. Allora, accodando $x_m = y_n$ a W si ottiene una sottosequenza comune di X e Y la cui lunghezza è maggiore di k, che è una contraddizione.

(2) Se $z_k \neq x_m$, allora Z è una sottosequenza comune di X_{m-1} e Y. Se esistesse una sottosequenza comune W di X_{m-1} e Y di lunghezza maggiore di k, allora W sarebbe anche una sottosequenza comune di X_m e Y, contraddicendo l'ipotesi che Z è una LCS di X e Y.

(3) La dimostrazione è simmetrica a quella del punto (2). ■

La caratterizzazione del Teorema 15.1 dimostra che una LCS di due sequenze contiene al suo interno una LCS di prefissi delle due sequenze. Quindi, il problema della più lunga sottosequenza comune gode della proprietà della sottostruttura ottima. Una soluzione ricorsiva gode anche della proprietà dei sottoproblemi ripetitivi, come vedremo qui di seguito.

Fase 2: una soluzione ricorsiva

Il Teorema 15.1 implica che ci sono uno o due sottoproblemi da esaminare per trovare una LCS di $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Se $x_m = y_n$, dobbiamo trovare una LCS di X_{m-1} e Y_{n-1}. Accodando $x_m = y_n$ a questa LCS, si ottiene una LCS di X e Y. Se $x_m \neq y_n$, allora dobbiamo risolvere due sottoproblemi: trovare una LCS di X_{m-1} e Y e trovare una LCS di X e Y_{n-1}. La più lunga di queste due LCS è una LCS di X e Y. Poiché questi casi esauriscono tutte le possibilità, sappiamo che una delle soluzioni ottime dei sottoproblemi deve essere utilizzata all'interno di una LCS di X e Y.

Possiamo facilmente vedere la proprietà dei sottoproblemi ripetitivi nel problema della più lunga sottosequenza comune. Per trovare una LCS di X e Y, potrebbe essere necessario trovare sia una LCS di X e Y_{n-1} sia una LCS di X_{m-1} e Y. Tuttavia, ciascuno di questi sottoproblemi ha il sottosottoproblema di trovare la LCS di X_{m-1} e Y_{n-1}. Molti altri sottoproblemi condividono sottosottoproblemi.

Come nel problema della moltiplicazione di una sequenza di matrici, la nostra soluzione ricorsiva del problema della più lunga sottosequenza comune richiede la definizione di una ricorrenza per il valore di una soluzione ottima. Definiamo $c[i, j]$ come la lunghezza di una LCS delle sequenze X_i e Y_j. Se $i = 0$ o $j = 0$, una delle sequenze ha lunghezza 0, quindi la LCS ha lunghezza 0. La sottostruttura ottima del problema della LCS consente di scrivere la formula ricorsiva

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases} \tag{15.14}$$

È importante notare che in questa formulazione ricorsiva una condizione del problema riduce il numero di sottoproblemi che possiamo considerare. Quando $x_i = y_j$, possiamo e dobbiamo considerare il sottoproblema di trovare la LCS di X_{i-1} e Y_{j-1}. Altrimenti, consideriamo i due sottoproblemi di trovare la LCS di X_i e Y_{j-1} e la LCS di X_{i-1} e Y_j. Nei precedenti algoritmi di programmazione dinamica che abbiamo esaminato – per la programmazione delle linee di assemblaggio e per la moltiplicazione di una sequenza di matrici – nessun sottoproblema è stato escluso a causa delle condizioni del problema. Trovare la LCS non è l'unico algoritmo di programmazione dinamica che esclude i sottoproblemi in base alle condizioni del problema. Per esempio, anche il problema della distanza di editing (vedere il Problema 15-3) ha questa caratteristica.

Fase 3: calcolare la lunghezza di una LCS

Utilizzando l'equazione (15.14) potremmo scrivere facilmente un algoritmo ricorsivo con tempo esponenziale per calcolare la lunghezza di una LCS di due sequenze. Tuttavia, poiché ci sono soltanto $\Theta(mn)$ sottoproblemi distinti, possiamo utilizzare la programmazione dinamica per calcolare le soluzioni con un metodo bottom-up. La procedura LCS-LENGTH riceve come input due sequenze $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ e memorizza i valori $c[i, j]$ in una tabella $c[0\,..\,m, 0\,..\,n]$, le cui posizioni sono calcolate secondo l'ordine delle righe (cioè, vengono inseriti i valori nella prima riga di c da sinistra a destra, poi vengono inseriti i valori nella seconda riga e così via).

La procedura utilizza anche la tabella $b[1\,..\,m, 1\,..\,n]$ per semplificare la costruzione di una soluzione ottima. Intuitivamente, $b[i, j]$ punta alla posizione della tabella che corrisponde alla soluzione ottima del sottoproblema che è stata scelta per calcolare $c[i, j]$. La procedura restituisce le tabelle b e c; la posizione $c[m, n]$ contiene la lunghezza di una LCS di X e Y.

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                        then c[i, j] ← c[i − 1, j]
14                             b[i, j] ← "↑"
15                        else c[i, j] ← c[i, j − 1]
16                             b[i, j] ← "←"
17  return c e b
```

La Figura 15.6 illustra le tabelle prodotte da LCS-LENGTH con le sequenze $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. Il tempo di esecuzione della procedura è $O(mn)$, perché il calcolo di ogni posizione della tabella richiede un tempo $O(1)$.

Fase 4: costruire una LCS

La tabella b restituita dalla procedura LCS-LENGTH può essere utilizzata per costruire rapidamente una LCS delle sequenze $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Iniziamo semplicemente da $b[m, n]$ e attraversiamo la tabella seguendo le frecce. Ogni volta che incontriamo una freccia "↖" nella posizione $b[i, j]$, significa che $x_i = y_j$ è un elemento della LCS. Gli elementi della LCS si incontrano in senso inverso con questo metodo. La seguente procedura ricorsiva stampa una LCS di X e Y nell'ordine corretto (diretto). La chiamata iniziale è PRINT-LCS$(b, X, length[X], length[Y])$.

i	j	0	1	2	3	4	5	6
		y_j	*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	*B*	0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	*C*	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	*B*	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	*D*	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	*A*	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	*B*	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figura 15.6 Le tabelle c e b calcolate da LCS-LENGTH con le sequenze $X = \langle A, B, C, B, D, A, B\rangle$ e $Y = \langle B, D, C, A, B, A\rangle$. La casella nella riga i e colonna j contiene il valore di $c[i, j]$ e la freccia appropriata al valore di $b[i, j]$. Il valore 4 in $c[7, 6]$ – l'angolo inferiore destro della tabella – è la lunghezza di una LCS $\langle B, C, B, A\rangle$ di X e Y. Per $i, j > 0$, la posizione $c[i, j]$ dipende soltanto da $x_i = y_j$ e dai valori $c[i-1, j]$, $c[i, j-1]$ e $c[i-1, j-1]$, che sono calcolati prima di $c[i, j]$. Per ricostruire gli elementi di una LCS, basta seguire le frecce $b[i, j]$ partendo dall'angolo inferiore destro della tabella; il percorso è indicato dallo sfondo grigio. Ogni freccia "↖" sul percorso corrisponde a una posizione (evidenziata) per la quale $x_i = y_j$ è un membro di una LCS.

PRINT-LCS(b, X, i, j)
1 **if** $i = 0$ o $j = 0$
2 **then return**
3 **if** $b[i, j] =$ "↖"
4 **then** PRINT-LCS$(b, X, i-1, j-1)$
5 stampa x_i
6 **elseif** $b[i, j] =$ "↑"
7 **then** PRINT-LCS$(b, X, i-1, j)$
8 **else** PRINT-LCS$(b, X, i, j-1)$

Per la tabella b illustrata nella Figura 15.6, questa procedura stampa "$BCBA$". La procedura impiega un tempo $O(m + n)$, perché almeno uno dei valori i e j diminuisce in ogni stadio della ricorsione.

Migliorare il codice

Dopo avere sviluppato un algoritmo, spesso ci accorgiamo che è possibile migliorare il tempo di esecuzione e la quantità di memoria utilizzata dall'algoritmo. Questo è particolarmente vero per gli algoritmi più semplici della programmazione dinamica. Alcune modifiche possono semplificare il codice dell'algoritmo e migliorare i fattori costanti, ma non producono miglioramenti asintotici delle prestazioni. Altre modifiche possono portare a sostanziali risparmi asintotici di tempo e memoria.

Per esempio, potremmo eliminare completamente la tabella b. Ogni posizione $c[i, j]$ dipende soltanto da altre tre posizioni della tabella c: $c[i-1, j-1]$, $c[i-1, j]$ e $c[i, j-1]$. Dato il valore di $c[i, j]$, possiamo determinare nel tempo $O(1)$ quale di questi tre valori è stato utilizzato per calcolare $c[i, j]$, senza ispezionare la tabella b. Quindi, possiamo ricostruire una LCS nel tempo $O(m + n)$ utilizzando una procedura simile a PRINT-LCS (l'Esercizio 15.4-2 chiede di scrivere lo pseudocodice). Sebbene questo metodo permetta di risparmiare uno spazio $\Theta(mn)$ in memoria, tuttavia lo spazio ausiliario richiesto per calcolare una LCS non diminuisce asintoticamente, perché occorre comunque uno spazio $\Theta(mn)$ per la tabella c.

Possiamo, però, ridurre il fabbisogno asintotico di memoria per LCS-LENGTH, perché questa procedura usa soltanto due righe alla volta della tabella c: la riga da calcolare e la riga precedente (in effetti, possiamo utilizzare uno spazio soltanto un po' più grande di quello richiesto da una riga di c per calcolare la lunghezza di una LCS. Vedere l'Esercizio 15.4-4). Questo miglioramento funziona se occorre calcolare soltanto la lunghezza di una LCS; se vogliamo ricostruire gli elementi di una LCS, la tabella più piccola non può contenere le informazioni necessarie per rifare il percorso inverso nel tempo $O(m + n)$.

Esercizi

15.4-1
Trovate una LCS delle sequenze $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ e $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$.

15.4-2
Spiegate come ricostruire nel tempo $O(m + n)$ una LCS dalla tabella completa c e dalle sequenze originali $X = \langle x_1, x_2, \ldots, x_m\rangle$ e $Y = \langle y_1, y_2, \ldots, y_n\rangle$, senza utilizzare la tabella b.

15.4-3
Create una versione memoizzata della procedura LCS-LENGTH che viene eseguita nel tempo $O(mn)$.

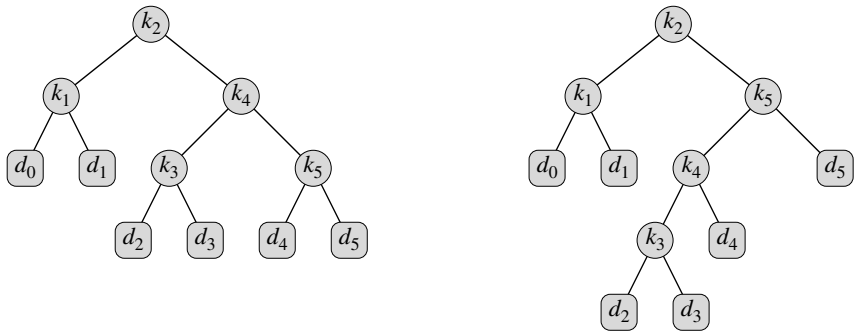
15.4-4
Spiegate come calcolare la lunghezza di una LCS utilizzando soltanto $2 \cdot \min(m, n)$ posizioni nella tabella c più uno spazio $O(1)$ aggiuntivo. Risolvete lo stesso problema utilizzando $\min(m, n)$ posizioni più uno spazio $O(1)$ aggiuntivo.

15.4-5
Create un algoritmo con tempo $O(n^2)$ per trovare la più lunga sottosequenza monotonicamente crescente di una sequenza di n numeri.

15.4-6 $\star$
Create un algoritmo con tempo $O(n \lg n)$ per trovare la più lunga sottosequenza monotonicamente crescente di una sequenza di n numeri (*suggerimento:* notate che l'ultimo elemento di una sottosequenza candidata di lunghezza i è grande almeno quanto l'ultimo elemento di una sottosequenza candidata di lunghezza $i - 1$. Memorizzate le sottosequenze candidate collegandole tramite una sequenza di input).

15.5 Alberi binari di ricerca ottimi

Supponete di progettare un programma di traduzione di un testo dall'inglese in italiano. Per ogni ricorrenza di una parola inglese nel testo, bisogna cercare l'equivalente parola italiana. Un modo per fare queste operazioni di ricerca è costruire un albero binario di ricerca con n parole inglesi come chiavi e le corrispondenti parole italiane come dati satelliti. Poiché ogni parola del testo dovrà essere cercata nell'albero, bisogna ridurre al minimo il tempo totale impiegato nelle ricerche. Potremmo garantire un tempo di ricerca $O(\lg n)$ per ogni ricorrenza, utilizzando un albero red-black o qualsiasi altro albero binario di ricerca bilanciato. Le parole, però, si presentano con frequenze differenti e potrebbe accadere che una parola frequentemente utilizzata, come "the" si trovi lontana dalla radice, mentre una parola raramente utilizzata, come "mycophagist", si trovi vicino alla radice.



(a) (b)

Figura 15.7 Due alberi binari di ricerca per un insieme di $n = 5$ chiavi con le seguenti probabilità:

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

(a) Un albero binario di ricerca con un costo atteso di ricerca pari a 2,80. **(b)** Un albero binario di ricerca con un costo atteso di ricerca pari a 2,75. Questo albero è ottimo.

Tale organizzazione rallenterebbe la traduzione del testo, in quanto il numero di nodi visitati durante la ricerca di una chiave in un albero binario di ricerca è pari a uno più la profondità del nodo che contiene la chiave. Noi vogliamo che le parole più frequenti nel testo occupino posizioni più vicine alla radice.[5] Inoltre, potrebbero esserci parole nel testo inglese per le quali non esistono le corrispondenti parole italiane; queste parole potrebbero non apparire affatto nell'albero binario di ricerca. Come possiamo organizzare un albero binario di ricerca per minimizzare il numero di nodi visitati in tutte le ricerche, conoscendo il numero di volte che si presenta ogni singola parola?

Ciò di cui abbiamo bisogno è un ***albero binario di ricerca ottimo***. Formalmente, data una sequenza $K = \langle k_1, k_2, \ldots, k_n \rangle$ di n chiavi distinte e ordinate (con $k_1 < k_2 < \cdots < k_n$), vogliamo costruire un albero binario di ricerca da queste chiavi. Per ogni chiave k_i, abbiamo una probabilità p_i che una ricerca riguarderà k_i. Alcune ricerche potrebbero riguardare valori che non si trovano in K, quindi abbiamo anche $n + 1$ *chiavi fittizie* (o *chiavi dummy*) $d_0, d_1, d_2, \ldots, d_n$ che rappresentano valori che non appartengono a K. In particolare, d_0 rappresenta tutti i valori minori di k_1, d_n rappresenta tutti i valori maggiori di k_n e, per $i = 1, 2, \ldots, n-1$, la chiave fittizia d_i rappresenta tutti i valori fra k_i e k_{i+1}. Per ogni chiave fittizia d_i, abbiamo una probabilità q_i che una ricerca corrisponderà a d_i. La Figura 15.7 illustra due alberi binari di ricerca di un insieme di $n = 5$ chiavi. Ogni chiave k_i è un nodo interno; ogni chiave fittizia d_i è una foglia. Una ricerca può riuscire (viene trovata una chiave k_i) o fallire (viene trovata una chiave fittizia d_i), quindi abbiamo

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \tag{15.15}$$

[5] Se il testo da tradurre riguarda i funghi commestibili, sarebbe preferibile avere la parola "mycophagist" vicino alla radice.

Poiché conosciamo le probabilità delle ricerche per ogni chiave e per ogni chiave fittizia, possiamo determinare il costo atteso di una ricerca in un determinato albero binario di ricerca T. Supponiamo che il costo effettivo di una ricerca sia il numero di nodi esaminati, ovvero la profondità del nodo trovato dalla ricerca in T, più 1. Allora, il costo atteso di una ricerca in T è

$$\begin{aligned}\mathrm{E}\left[\text{costo di una ricerca in } T\right] &= \sum_{i=1}^{n}(\text{profondità}_T(k_i)+1)\cdot p_i + \\ &\quad \sum_{i=0}^{n}(\text{profondità}_T(d_i)+1)\cdot q_i \\ &= 1+\sum_{i=1}^{n}\text{profondità}_T(k_i)\cdot p_i + \\ &\quad \sum_{i=0}^{n}\text{profondità}_T(d_i)\cdot q_i \qquad (15.16)\end{aligned}$$

dove profondità_T indica la profondità di un nodo nell'albero T. L'ultima uguaglianza deriva dall'equazione (15.15). Nella Figura 15.7(a) possiamo calcolare il costo atteso di ricerca nodo per nodo:

nodo	profondità	probabilità	contributo
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40
Totale			2,80

Per un dato insieme di probabilità, il nostro obiettivo è costruire un albero binario di ricerca il cui costo atteso di ricerca è minimo. Questo albero è detto ***albero binario di ricerca ottimo***. La Figura 15.7(b) illustra un albero binario di ricerca ottimo per le probabilità elencate nella didascalia della figura; il suo costo atteso è 2,75. Questo esempio dimostra che un albero binario di ricerca ottimo non è necessariamente un albero la cui altezza totale è minima, né possiamo necessariamente costruire un albero binario di ricerca ottimo ponendo sempre nella radice la chiave con la probabilità massima. Qui, la chiave k_5 ha la probabilità di ricerca più grande di qualsiasi chiave e la radice dell'albero binario di ricerca ottimo è ancora k_2 (il costo minimo atteso di qualsiasi albero binario di ricerca con k_5 nella radice è 2,85).

Come nella moltiplicazione di una sequenza di matrici, il controllo completo di tutte le possibilità non riesce a produrre un algoritmo efficiente. Possiamo etichettare i nodi di qualsiasi albero binario di n nodi con le chiavi $k_1, k_2, \ldots, k_n$ per costruire un albero binario di ricerca e, poi, aggiungere le chiavi fittizie come foglie. Nel Problema 12-4 abbiamo visto che il numero di alberi binari con n nodi è $\Omega(4^n/n^{3/2})$; quindi, in una ricerca completa, dovremmo esaminare un numero

esponenziale di alberi binari di ricerca. Nessuna sorpresa, quindi, se risolveremo questo problema con la programmazione dinamica.

Fase 1: la struttura di un albero binario di ricerca ottimo

Per caratterizzare la sottostruttura ottima degli alberi binari di ricerca ottimi, iniziamo con una osservazione sui sottoalberi. Consideriamo un sottoalbero qualsiasi di un albero binario di ricerca; le sue chiavi devono essere in un intervallo contiguo $k_i, \ldots, k_j$, per qualche $1 \leq i \leq j \leq n$. Inoltre, un sottoalbero che contiene le chiavi $k_i, \ldots, k_j$ deve anche avere come foglie le chiavi fittizie $d_{i-1}, \ldots, d_j$.

Adesso possiamo definire la sottostruttura ottima: se un albero binario di ricerca ottimo T ha un sottoalbero T' che contiene le chiavi $k_i, \ldots, k_j$, allora questo sottoalbero T' deve essere ottimo anche per il sottoproblema con chiavi $k_i, \ldots, k_j$ e chiavi fittizie $d_{i-1}, \ldots, d_j$. È possibile applicare la tecnica taglia e incolla. Se ci fosse un sottoalbero T'' il cui costo atteso è minore di quello di T', allora potremmo tagliare T' da T e incollare T'', ottenendo un albero binario di ricerca con un costo atteso minore di quello di T, ma questo sarebbe in contraddizione con l'ipotesi che T è un albero binario di ricerca ottimo.

Bisogna utilizzare la sottostruttura ottima per dimostrare che è possibile costruire una soluzione ottima del problema delle soluzioni ottime dei sottoproblemi. Date le chiavi $k_i, \ldots, k_j$, una di queste chiavi, per esempio k_r ($i \leq r \leq j$), sarà la radice di un sottoalbero ottimo che contiene queste chiavi. Il sottoalbero sinistro della radice k_r conterrà le chiavi $k_i, \ldots, k_{r-1}$ (e le chiavi fittizie $d_{i-1}, \ldots, d_{r-1}$) e il sottoalbero destro conterrà le chiavi $k_{r+1}, \ldots, k_j$ (e le chiavi fittizie $d_r, \ldots, d_j$). Finché esaminiamo tutte le radici candidate k_r, con $i \leq r \leq j$, e determiniamo tutti gli alberi binari di ricerca ottimi che contengono $k_i, \ldots, k_{r-1}$ e quelli che contengono $k_{r+1}, \ldots, k_j$, avremo la garanzia di trovare un albero binario di ricerca ottimo.

C'è un dettaglio importante da esaminare sui sottoalberi "vuoti". Supponiamo di scegliere k_i come radice di un sottoalbero con chiavi $k_i, \ldots, k_j$. In base al precedente ragionamento, il sottoalbero sinistro di k_i contiene le chiavi $k_i, \ldots, k_{i-1}$. È naturale dedurre che questa sequenza non contiene chiavi. Ricordiamo, però, che i sottoalberi contengono anche le chiavi fittizie. Adottiamo la convenzione che un sottoalbero che contiene le chiavi $k_i, \ldots, k_{i-1}$ non ha chiavi reali, ma contiene l'unica chiave fittizia d_{i-1}. In modo simmetrico, se selezioniamo k_j come radice, allora il sottoalbero destro di k_j contiene le chiavi $k_{j+1}, \ldots, k_j$; questo sottoalbero destro non contiene chiavi reali, ma contiene la chiave fittizia d_j.

Fase 2: una soluzione ricorsiva

A questo punto possiamo definire il valore di una soluzione ottima in modo ricorsivo. Il nostro dominio dei sottoproblemi è trovare un albero binario di ricerca ottimo che contiene le chiavi $k_i, \ldots, k_j$, dove $i \geq 1$, $j \leq n$ e $j \geq i - 1$ (quando $j = i - 1$, non ci sono chiavi reali e c'è l'unica chiave fittizia d_{i-1}). Definiamo $e[i, j]$ come il costo atteso per cercare un albero binario di ricerca ottimo che contiene le chiavi $k_i, \ldots, k_j$. In ultima analisi, vogliamo calcolare $e[1, n]$.

Il caso semplice si verifica quando $j = i - 1$; c'è una sola chiave fittizia: d_{i-1}. Il costo atteso di ricerca è $e[i, i - 1] = q_{i-1}$.

Quando $j \geq i$, bisogna scegliere una radice k_r fra $k_i, \ldots, k_j$ e poi creare il suo sottoalbero sinistro con un albero binario di ricerca ottimo con le chiavi

$k_i, \ldots, k_{r-1}$ e il suo sottoalbero destro con un albero binario di ricerca ottimo con le chiavi $k_{r+1}, \ldots, k_j$. Che cosa accade al costo atteso di ricerca di un sottoalbero quando questo diventa un sottoalbero di un nodo? La profondità di ogni nodo nel sottoalbero aumenta di 1. Per l'equazione (15.16), il costo atteso di ricerca di questo sottoalbero aumenta della somma di tutte le probabilità nel sottoalbero. Per un sottoalbero con chiavi $k_i, \ldots, k_j$, indichiamo questa somma di probabilità con la seguente espressione

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \tag{15.17}$$

Quindi, se k_r è la radice di un sottoalbero ottimo che contiene le chiavi $k_i, \ldots, k_j$, abbiamo

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

Osservando che

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j)$$

possiamo riscrivere $e[i, j]$ in questo modo

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \tag{15.18}$$

L'equazione ricorsiva (15.18) suppone che sia noto il nodo k_r da utilizzare come radice. Se scegliamo la radice che ha il costo atteso di ricerca minimo, otteniamo la formula ricorsiva finale:

$$e[i, j] = \begin{cases} q_{i-1} & \text{se } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{se } i \le j \end{cases} \tag{15.19}$$

I valori $e[i, j]$ rappresentano i costi attesi di ricerca negli alberi binari di ricerca ottimi. Per seguire più facilmente la struttura degli alberi binari di ricerca ottimi, definiamo $root[i, j]$, per $1 \le i \le j \le n$, come l'indice r per il quale k_r è la radice di un albero binario di ricerca ottimo che contiene le chiavi $k_i, \ldots, k_j$. Anche se spiegheremo come calcolare i valori di $root[i, j]$, lasciamo al lettore il compito di costruire l'albero binario di ricerca ottimo da questi valori (vedere l'Esercizio 15.5-1).

Fase 3: calcolare il costo atteso di ricerca di un albero binario di ricerca ottimo

A questo punto, qualcuno avrà notato qualche analogia fra la caratterizzazione degli alberi binari di ricerca ottimi e la caratterizzazione della moltiplicazione di una sequenza di matrici. Per entrambi i domini dei problemi, i sottoproblemi sono formati da sottointervalli di indici contigui. Una implementazione ricorsiva diretta dell'equazione (15.19) potrebbe risultare inefficiente come l'algoritmo ricorsivo diretto della moltiplicazione di una sequenza di matrici. Invece, memorizziamo i valori $e[i, j]$ in una tabella $e[1 \,..\, n+1, 0 \,..\, n]$. Il primo indice deve avere il valore $n + 1$ (anziché n) perché, per ottenere un sottoalbero che contiene soltanto la chiave fittizia d_n, dobbiamo calcolare e memorizzare $e[n+1, n]$. Il secondo indice deve iniziare da 0 perché, per ottenere un sottoalbero che contiene soltanto la chiave fittizia d_0, dobbiamo calcolare e memorizzare $e[1, 0]$. Utilizzeremo soltanto

le posizioni $e[i, j]$ per le quali $j \geq i - 1$. Utilizzeremo anche una tabella $root[i, j]$ per memorizzare la radice del sottoalbero che contiene le chiavi $k_i, \ldots, k_j$. Questa tabella usa soltanto le posizioni per le quali $1 \leq i \leq j \leq n$. Per migliorare l'efficienza, utilizzeremo un'altra tabella. Anziché ricominciare da zero il calcolo di $w(i, j)$ ogni volta che calcoliamo $e[i, j]$ – occorrerebbero $\Theta(j - i)$ addizioni – memorizziamo questi valori in una tabella $w[1\,..\,n + 1, 0\,..\,n]$. Per il caso base, calcoliamo $w[i, i - 1] = q_{i-1}$ per $1 \leq i \leq n + 1$. Per $j \geq i$, calcoliamo

$$w[i, j] = w[i, j - 1] + p_j + q_j \tag{15.20}$$

Quindi, possiamo calcolare ciascuno dei $\Theta(n^2)$ valori di $w[i, j]$ nel tempo $\Theta(1)$. Il seguente pseudocodice riceve come input le probabilità $p_1, \ldots, p_n$ e $q_0, \ldots, q_n$ e la dimensione n e restituisce le tabelle e e $root$.

```
OPTIMAL-BST(p, q, n)
 1  for i ← 1 to n + 1
 2       do e[i, i − 1] ← q_{i−1}
 3          w[i, i − 1] ← q_{i−1}
 4  for l ← 1 to n
 5       do for i ← 1 to n − l + 1
 6             do j ← i + l − 1
 7                e[i, j] ← ∞
 8                w[i, j] ← w[i, j − 1] + p_j + q_j
 9                for r ← i to j
10                   do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
11                      if t < e[i, j]
12                        then e[i, j] ← t
13                             root[i, j] ← r
14  return e e root
```

Dalla precedente descrizione e per l'analogia con la procedura MATRIX-CHAIN-ORDER (Paragrafo 15.2), il funzionamento di OPTIMAL-BST dovrebbe essere abbastanza chiaro. Il primo ciclo **for** (righe 1–3) inizializza i valori di $e[i, i - 1]$ e $w[i, i-1]$. Il ciclo **for** (righe 4–13) usa le ricorrenze (15.19) e (15.20) per calcolare $e[i, j]$ e $w[i, j]$ per ogni $1 \leq i \leq j \leq n$. Nella prima iterazione, quando $l = 1$, il ciclo calcola $e[i, i]$ e $w[i, i]$ per $i = 1, 2, \ldots, n$. Nella seconda iterazione, con $l = 2$, il ciclo calcola $e[i, i + 1]$ e $w[i, i + 1]$ per $i = 1, 2, \ldots, n - 1$, e così via. Il ciclo **for** più interno (righe 9–13) prova ciascun indice r candidato per determinare quale chiave k_r utilizzare come radice di un albero binario di ricerca ottimo che contiene le chiavi $k_i, \ldots, k_j$. Questo ciclo **for** salva il valore corrente dell'indice r nella posizione $root[i, j]$, ogni volta che trova una chiave migliore da utilizzare come radice.

La Figura 15.8 illustra le tabelle $e[i, j]$, $w[i, j]$ e $root[i, j]$ calcolate dalla procedura OPTIMAL-BST con la distribuzione delle chiavi indicata nella Figura 15.7. Come nell'esempio della moltiplicazione di una sequenza di matrici, le tabelle sono ruotate per rappresentare orizzontalmente le diagonali. La procedura OPTIMAL-BST calcola le righe dal basso verso l'alto e da sinistra a destra all'interno di ogni riga.

La procedura OPTIMAL-BST impiega un tempo $\Theta(n^3)$, esattamente come MATRIX-CHAIN-ORDER. È facile capire che il tempo di esecuzione è $O(n^3)$,

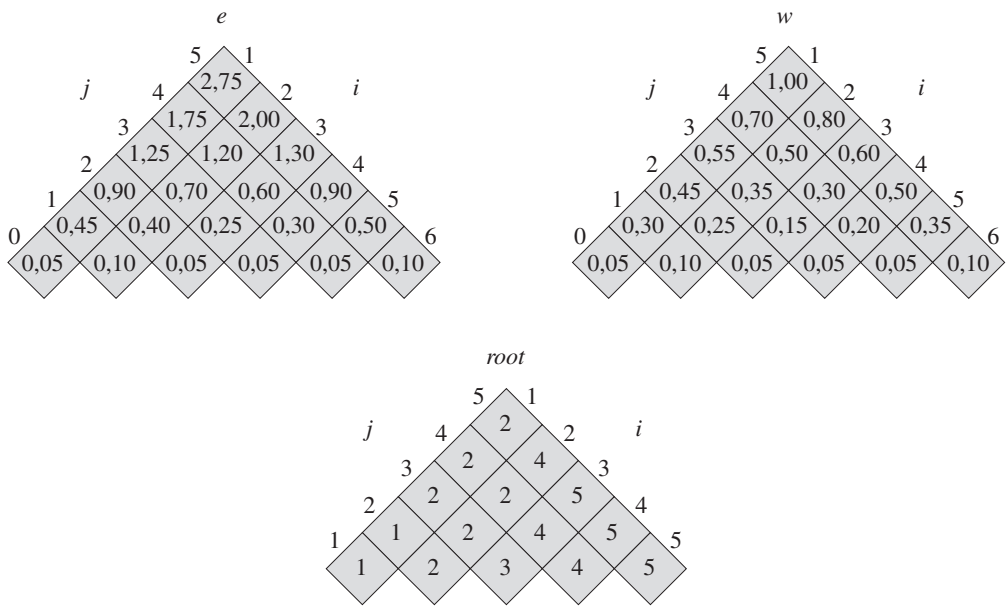


Figura 15.8 Le tabelle $e[i, j]$, $w[i, j]$ e $root[i, j]$ calcolate da OPTIMAL-BST con la distribuzione delle chiavi indicata nella Figura 15.7. Le tabelle sono ruotate in modo che le diagonali siano orizzontali.

perché i cicli **for** hanno tre livelli di annidamento e ogni indice di ciclo assume al massimo n valori. Gli indici di ciclo in OPTIMAL-BST non hanno esattamente gli stessi limiti di quelli di MATRIX-CHAIN-ORDER, ma restano al più minori di 1 in tutte le direzioni. Quindi, come MATRIX-CHAIN-ORDER, la procedura OPTIMAL-BST impiega il tempo $\Omega(n^3)$.

Esercizi

15.5-1

Scrivete lo pseudocodice per la procedura CONSTRUCT-OPTIMAL-BST$(root)$ che, data la tabella $root$, genera in output la struttura di un albero binario di ricerca ottimo. Per l'esempio illustrato nella Figura 15.8, la vostra procedura dovrebbe visualizzare la struttura

k_2 è la radice
k_1 è il figlio sinistro di k_2
d_0 è il figlio sinistro di k_1
d_1 è il figlio destro di k_1
k_5 è il figlio destro di k_2
k_4 è il figlio sinistro di k_5
k_3 è il figlio sinistro di k_4
d_2 è il figlio sinistro di k_3
d_3 è il figlio destro di k_3
d_4 è il figlio destro di k_4
d_5 è il figlio destro di k_5

che corrisponde all'albero binario di ricerca ottimo illustrato nella Figura 15.7(b).

15.5-2
Determinate il costo e la struttura di un albero binario di ricerca ottimo per un insieme di $n = 7$ chiavi con le seguenti probabilità:

i	0	1	2	3	4	5	6	7
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,14
q_i	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05

15.5-3
Anziché mantenere la tabella $w[i, j]$, supponiamo di calcolare il valore di $w(i, j)$ direttamente dall'equazione (15.17) nella riga 8 di OPTIMAL-BST e di utilizzare il valore risultante nella riga 10. Come influisce tutto questo sul tempo di esecuzione asintotico di OPTIMAL-BST?

15.5-4 $\star$
Knuth [184] ha dimostrato che ci sono sempre delle radici di sottoalberi ottimi tali che $root[i, j-1] \leq root[i, j] \leq root[i+1, j]$ per ogni $1 \leq i < j \leq n$. Utilizzate questo fatto per modificare la procedura OPTIMAL-BST in modo che sia eseguita nel tempo $\Theta(n^2)$.

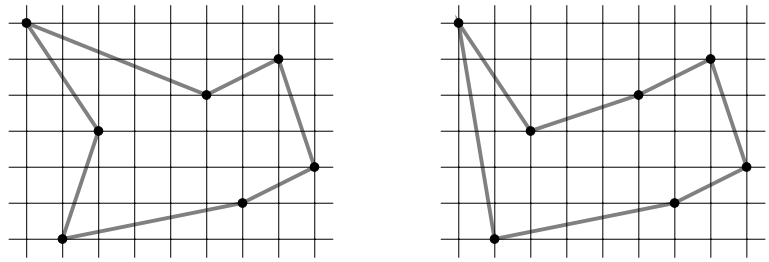
Problemi

15-1 Problema del commesso viaggiatore euclideo e bitonico
Il ***problema del commesso viaggiatore euclideo*** consiste nel determinare il cammino chiuso minimo che collega un dato insieme di n punti del piano. La Figura 15.9(a) illustra la soluzione di un problema con un insieme di 7 punti. Il problema generale è NP-completo, pertanto la sua soluzione richiede un tempo più che polinomiale (vedere il Capitolo 34).

J. L. Bentley ritiene che il problema possa essere semplificato se si considerano soltanto i ***cammini bitonici***, ovvero quei percorsi che iniziano dal punto più a sinistra, vanno sempre da sinistra a destra fino a raggiungere il punto più a destra e poi sempre da destra a sinistra fino a ritornare al punto di partenza. La Figura 15.9(b) illustra il cammino bitonico minimo per lo stesso insieme di 7 punti. In questo caso, è possibile definire un algoritmo con tempo polinomiale.

Descrivete un algoritmo con tempo $O(n^2)$ per determinare un cammino bitonico ottimo. Supponete che due punti non possano avere la stessa coordinata x (*suggerimento:* fate una scansione dei punti da sinistra a destra, mantenendo le possibilità ottime per le due parti del cammino).

15-2 Una stampa accurata
Considerate il problema di stampare in modo accurato un paragrafo di testo con una stampante. Il testo di input è una sequenza di n parole di lunghezza $l_1, l_2, \ldots, l_n$; la lunghezza di una parola è espressa dal numero di caratteri. Vogliamo stampare questo paragrafo in modo accurato su un certo numero di righe, ciascuna delle quali contiene al massimo M caratteri. Il nostro criterio di "stampa accurata" è il seguente. Se una data riga contiene le parole da i a j, con $i \leq j$, e lasciamo esattamente uno spazio fra le parole, il numero di spazi extra alla fine della riga è $M - j + i - \sum_{k=i}^{j} l_k$, che deve essere non negativo, in modo che le parole possano adattarsi alla riga. Vogliamo rendere minima la sommatoria, per



(a) (b)

Figura 15.9 Sette punti di un piano illustrati su una griglia con quadrati di lato unitario. **(a)** Il cammino chiuso minimo, con lunghezza approssimativamente pari a $24,89$. Questo cammino non è bitonico. **(b)** Il cammino bitonico minimo per lo stesso insieme di punti. La sua lunghezza è approssimativamente pari a $25,58$.

tutte le righe tranne l'ultima, dei cubi dei numeri di spazi extra che restano alla fine di ogni riga. Descrivete un algoritmo di programmazione dinamica per stampare in modo accurato un paragrafo di n parole. Analizzate il tempo di esecuzione e la quantità di memoria richiesta dal vostro algoritmo.

15-3 Distanza di editing

Per trasformare una stringa di input $x[1\,..\,m]$ in una stringa di output $y[1\,..\,n]$, possiamo svolgere varie operazioni. Date le stringhe x e y, il nostro obiettivo è effettuare una serie di trasformazioni che cambiano x in y. Utilizziamo un array z per memorizzare i risultati intermedi (supponendo che l'array abbia una dimensione sufficiente a contenere tutti i caratteri che servono). Inizialmente, z è vuoto e, alla fine, dovremmo avere $z[j] = y[j]$ per $j = 1, 2, \ldots, n$. Memorizziamo gli indici correnti i in x e j in z; le operazioni possono modificare z e questi indici. Inizialmente, $i = j = 1$. Ci è stato richiesto di esaminare i singoli caratteri di x durante la trasformazione; questo significa che, alla fine della sequenza delle operazioni, dobbiamo avere $i = m + 1$.

Ci sono sei operazioni di trasformazione:

Copia un carattere da x a z, impostando $z[j] \leftarrow x[i]$ e poi incrementando i e j. Questa operazione esamina $x[i]$.

Sostituisci un carattere di x con un altro carattere c, impostando $z[j] \leftarrow c$ e poi incrementando i e j. Questa operazione esamina $x[i]$.

Cancella un carattere di x, incrementando i, senza modificare j. Questa operazione esamina $x[i]$.

Inserisci il carattere c in z, impostando $z[j] \leftarrow c$ e poi incrementando j, senza modificare i. Questa operazione non esamina i caratteri di x.

Scambia i prossimi due caratteri copiandoli da x a z, ma in ordine inverso; per fare questo, impostiamo prima $z[j] \leftarrow x[i+1]$ e $z[j+1] \leftarrow x[i]$ e, poi, $i \leftarrow i + 2$ e $j \leftarrow j + 2$. Questa operazione esamina $x[i]$ e $x[i+1]$.

Distruggi la parte restante di x, impostando $i \leftarrow m + 1$. Questa operazione esamina tutti i caratteri di x che non sono stati ancora esaminati. Se questa operazione viene svolta, deve essere l'ultima.

Per esempio, un modo per trasformare la stringa di input `algorithm` nella stringa di output `altruistic` consiste nell'utilizzare questa sequenza di operazioni (i caratteri sottolineati sono $x[i]$ e $z[j]$ dopo ogni operazione):

Operazione	x	z
stringhe iniziali	a`lgorithm`	_
copia	a`l`gorithm	a_
copia	alg`o`rithm	al_
sostituisci con `t`	alg`o`rithm	alt_
cancella	algo`r`ithm	alt_
copia	algor`i`thm	altr_
inserisci `u`	algor`i`thm	altru_
inserisci `i`	algor`i`thm	altrui_
inserisci `s`	algor`i`thm	altruis_
scambia	algorit`hm`	altruisti_
inserisci `c`	algorit`h`m	altruistic_
distruggi	algorithm_	altruistic_

Notate che ci sono molte altre sequenze di operazioni che possono trasformare `algorithm` in `altruistic`.

Ciascuna delle operazioni di trasformazione ha un costo associato. Il costo di un'operazione dipende dalla specifica applicazione, ma supponiamo che il costo di ciascuna operazione sia una costante nota. Supponiamo inoltre che i singoli costi delle operazioni di copia e sostituzione siano minori dei costi combinati delle operazioni di cancellazione e inserimento, altrimenti le operazioni di copia e sostituzione non sarebbero utilizzate. Il costo di una data sequenza di operazioni di trasformazione è la somma dei costi delle singole operazioni nella sequenza. Per la precedente sequenza, il costo per trasformare `algorithm` in `altruistic` è

$$(3 \cdot \text{costo}(\text{copia})) + \text{costo}(\text{sostituisci}) + \text{costo}(\text{cancella}) + (4 \cdot \text{costo}(\text{inserisci})) + \text{costo}(\text{scambia}) + \text{costo}(\text{distruggi})$$

a. Date due sequenze $x[1\,..\,m]$ e $y[1\,..\,n]$ e un insieme di costi di trasformazione, la ***distanza di editing*** tra x e y è il costo della sequenza di operazioni più economica che trasforma x in y. Descrivete un algoritmo di programmazione dinamica che trova la distanza di editing tra $x[1\,..\,m]$ e $y[1\,..\,n]$ e stampa una sequenza di operazioni ottima. Analizzate il tempo di esecuzione e la quantità di memoria richiesta dal vostro algoritmo.

Il problema della distanza di editing è una generalizzazione del problema dell'allineamento di due sequenze di DNA (vedere, per esempio, Setubal e Meidanis [272, Paragrafo 3.2]). Ci sono vari metodi per misurare la somiglianza di due sequenze di DNA. Uno di questi metodi consiste nell'allineare due sequenze x e y inserendo degli spazi in posizioni arbitrarie delle due sequenze (incluse le due estremità), in modo che le sequenze risultanti x' e y' abbiano la stessa lunghezza, ma non abbiano uno spazio nella stessa posizione (ovvero, per nessuna posizione j, gli elementi $x'[j]$ e $y'[j]$ possono essere entrambi uno spazio). Poi assegniamo un "punteggio" a ogni posizione. La posizione j riceve i seguenti punteggi:

- $+1$ se $x'[j] = y'[j]$ e nessuno dei due elementi è uno spazio
- -1 se $x'[j] \neq y'[j]$ e nessuno dei due elementi è uno spazio
- -2 se $x'[j]$ o $y'[j]$ è uno spazio

Il punteggio dell'allineamento è la somma dei punteggi delle singole posizioni. Per esempio, date le sequenze $x =$ `GATCGGCAT` e $y =$ `CAATGTGAATC`, un allineamento è

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

Un segno più (+) sotto una posizione indica un punteggio $+1$ per quella posizione; un segno meno (-) indica un punteggio -1 e l'asterisco (*) indica un punteggio -2; quindi, questo allineamento ha un punteggio totale pari a $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Spiegate come trasformare il problema di trovare un allineamento ottimo in un problema di ricerca della distanza di editing, utilizzando un sottoinsieme delle operazioni copia, sostituisci, cancella, inserisci, scambia e distruggi.

15-4 Programmare una festa aziendale

Il professor Stewart è un consulente del presidente di una grande azienda che sta programmando una festa per i dipendenti. L'azienda ha una struttura gerarchica, nella quale ogni direttore forma un albero che ha la radice nel presidente. L'ufficio del personale ha classificato ogni dipendente con un grado di giovialità, espresso da un numero reale. Affinché la festa possa essere piacevole per tutti i partecipanti, il presidente non vuole che alla festa siano presenti un dipendente con il suo diretto superiore.

Il professor Stewart ha l'albero che descrive la struttura gerarchica dell'azienda mediante una rappresentazione figlio-sinistro fratello-destro (vedere il Paragrafo 10.4). Ogni nodo dell'albero contiene, oltre ai puntatori, il nome di un dipendente e il suo grado di giovialità. Descrivete un algoritmo che prepara la lista degli invitati, massimizzando la somma dei gradi di giovialità degli ospiti. Analizzate il tempo di esecuzione dell'algoritmo.

15-5 Algoritmo di Viterbi

Possiamo applicare la programmazione dinamica a un grafo orientato $G = (V, A)$ per il riconoscimento della voce. Ogni arco $(u, v) \in A$ è etichettato con un suono $\sigma(u, v)$ che appartiene a un insieme finito Σ di suoni. Il grafo etichettato è un modello formale di una persona che parla una lingua molto semplice. Ogni percorso nel grafo che inizia da un vertice distinto $v_0 \in V$ corrisponde a una possibile sequenza di suoni prodotti dal modello. L'etichetta di un grafo orientato è definita come la concatenazione delle etichette degli archi in quel percorso.

a. Supponete di avere un grafo etichettato G con un vertice distinto v_0 e una sequenza $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ di caratteri dell'insieme Σ. Descrivete un algoritmo efficiente che restituisce un percorso in G che inizia da v_0 e ha s come sua etichetta, se tale percorso esiste. Altrimenti, l'algoritmo dovrà restituire PERCORSO-INESISTENTE. Analizzate il tempo di esecuzione dell'algoritmo (*suggerimento:* potrebbe essere utile consultare il Capitolo 22).

Adesso, supponete che a ogni arco $(u, v) \in A$ sia associata una probabilità non negativa $p(u, v)$ di attraversare l'arco (u, v) a partire dal vertice u, producendo così il corrispondente suono. La somma delle probabilità degli archi che escono da un vertice qualsiasi è pari a 1. La probabilità di un percorso è definita come il prodotto delle probabilità dei suoi archi. Possiamo considerare la probabilità di un percorso che inizia da v_0 come la probabilità che un "attraversamento casuale" che inizia da v_0 segua il percorso specificato, dove la scelta di quale arco prendere

in un vertice u viene fatta in maniera probabilistica, secondo le probabilità degli archi disponibili che partono da u.

b. Estendete l'algoritmo descritto nel punto (a) in modo che, se viene restituito un percorso, questo sia il *più probabile percorso* che parte da v_0 e che ha l'etichetta s. Analizzate il tempo di esecuzione dell'algoritmo.

15-6 Mosse sulla scacchiera

Supponete di avere una scacchiera con $n \times n$ caselle e un pezzo che dovrete muovere dall'estremità inferiore all'estremità superiore della scacchiera, rispettando le seguenti regole. In ogni mossa il pezzo può andare in una delle seguenti caselle:

1. la casella immediatamente in alto
2. la casella immediatamente in alto a sinistra (a meno che il pezzo non si trovi già nell'ultima colonna a sinistra)
3. la casella immediatamente in alto a destra (a meno che il pezzo non si trovi già nell'ultima colonna a destra)

Ogni volta che spostate il pezzo dalla casella x alla casella y, riceverete $p(x, y)$ dollari. Riceverete tale importo per tutte le coppie (x, y) per le quali è valida la mossa da x a y. Non bisogna supporre che $p(x, y)$ sia positivo.

Descrivete un algoritmo che calcola l'insieme delle mosse che dovrà fare il pezzo partendo da una casella dell'estremità inferiore della scacchiera fino a raggiungere una casella dell'estremità superiore, guadagnando il maggior numero possibile di dollari. L'algoritmo è libero di scegliere qualsiasi casella nell'estremità inferiore della scacchiera come punto di partenza e qualsiasi casella nell'estremità superiore come punto di arrivo. Qual è il tempo di esecuzione del vostro algoritmo?

15-7 Massimizzare i profitti

Supponete di avere una macchina e un insieme di n lavori $a_1, a_2, \ldots, a_n$ da svolgere con questa macchina. Ogni lavoro a_j ha un tempo di elaborazione t_j, un profitto p_j e data di scadenza d_j. La macchina può elaborare un solo lavoro alla volta; inoltre un generico lavoro a_j deve essere eseguito ininterrottamente per t_j unità di tempo consecutive. Se il lavoro a_j viene completato entro la sua scadenza d_j, avrete un profitto p_j, ma se viene completato in ritardo, avrete un profitto 0. Descrivete un algoritmo che programma i lavori in modo da massimizzarne i profitti, supponendo che tutti i tempi di elaborazione siano numeri interi compresi tra 1 e n. Qual è il tempo di esecuzione del vostro algoritmo?

Note

Lo studio sistematico della programmazione dinamica è stato avviato da R. Bellman nel 1955. In questo ambito e anche nella programmazione lineare, la parola "programmazione" fa riferimento all'impiego di un metodo di risoluzione dei problemi basato sulle tabelle. Sebbene fossero già note alcune tecniche di ottimizzazione che incorporavano elementi della programmazione dinamica, Bellman ha introdotto in questa materia una solida teoria matematica [34].

continua

Hu e Shing [159, 160] hanno creato un algoritmo con tempo $O(n \lg n)$ per risolvere il problema della moltiplicazione di una sequenza di matrici.

L'algoritmo con tempo $O(mn)$ per risolvere il problema della più lunga sottosequenza comune ha suscitato l'interesse di molti studiosi. Knuth [63] si è domandato se esistessero algoritmi subquadratici per il problema della LCS. Masek e Paterson [212] hanno risposto affermativamente a questa domanda, fornendo un algoritmo che viene eseguito nel tempo $O(mn/\lg n)$, dove $n \leq m$ e le sequenze sono estratte da un insieme di dimensione limitata. Nel caso speciale in cui nessun elemento appaia più di una volta in una sequenza di input, Szymanski [288] ha dimostrato che il problema può essere risolto nel tempo $O((n+m)\lg(n+m))$. Molti di questi risultati possono essere estesi al problema del calcolo della distanza di editing delle stringhe (Problema 15-3).

Un vecchio articolo di Gilbert e Moore [114] sulla codifica binaria a lunghezza variabile descrive delle applicazioni per costruire alberi binari di ricerca ottimi per il caso in cui tutte le probabilità p_i siano 0; questo articolo contiene un algoritmo con tempo $O(n^3)$. Aho, Hopcroft e Ullman [5] hanno sviluppato l'algoritmo descritto nel Paragrafo 15.5. L'Esercizio 15.5-4 è tratto da un articolo di Knuth [184]. Hu e Tucker [161] hanno ideato un algoritmo per il caso in cui tutte le probabilità p_i siano 0, che impiega un tempo $O(n^2)$ e uno spazio $O(n)$; successivamente, Knuth [185] ha ridotto il tempo a $O(n \lg n)$.